МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН

ПАВЛОДАРСКИЙ ТЕХНИКО-ЭКОНОМИЧЕСКИЙ КОЛЛЕДЖ

ПЛАНЫ УРОКОВ ПО ДИСЦИПЛИНЕ «ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ»

Разработала: преподаватель

спец. дисциплин Салий Н.А.

2005

СОДЕРЖАНИЕ

Раздел №12. Работа с окнами редактирования и документами

1. Организация работы пользователя с текстовыми и графическими документами

2. Обработка документов в компоненте RichEdit

3. Работа с произвольными типами файловых документов

4. Просмотр и редактирование любых документов с помощью WebBrowser

Раздел №13. Взаимодействие с приложениями Microsoft Office для обработки документов

5. Работа с сервером Microsoft Word

6. Работа с сервером Microsoft Excel

Раздел №14. Графики и диаграммы

7. Графики и диаграммы. Основные свойства и методы компонента TChart.

Раздел №15. Настройка меню и инструментальных панелей

8. Настройка меню и инструментальных панелей

9. Хранение информации в реестре. Работа с файлами INI

Раздел №16. Работа с файлами и каталогами

10. Работа с файлами и каталогами.

Раздел №17. Delphi и механизмы Windows

11. SystemTray. Шрифты приложения

Раздел №18. Некоторые приемы программирования приложений Windows

12. Оконные компоненты и формы.

13. Клавиатура и курсор мыши

14. Динамически присоединяемые библиотеки DLL

Раздел №19. Динамические переменные

15. Указательный (ссылочный) тип. Динамические переменные.

16. Динамические объекты – линейные списки

17. Динамические объекты сложной структуры

18. Контрольная работа

РАЗДЕЛ №12. РАБОТА С ОКНАМИ РЕДАКТИРОВАНИЯ И ДОКУМЕНТАМИ

УРОК №12.1 ОРГАНИЗАЦИЯ РАБОТЫ ПОЛЬЗОВАТЕЛЯ С ТЕКСТОВЫМИ И ГРАФИЧЕСКИМИ ДОКУМЕНТАМИ

Цель:

Образовательная: объяснить принципы работы с текстовыми файлами: загрузка, сохранение и др. Рассмотреть компоненты RichEdit, OpenDialog, SaveDialog. Объяснить особенности открытия и сохранения документов в RichEdit, а также ввод и вывод графических файлов.

Воспитательная: продолжить формирование у учащихся основ научного мировоззрения. Воспитание любви к профессии и предмету

Развивающая: развитие познавательных способностей (мышления, воображения и т.д.).

Тип занятия: урок формирования новых знаний

Форма организации учебного процесса: лекция с элементами беседы

ХОД УРОКА

1. **Организационный момент** (требования к уроку, план работы)
2. **Изложение нового материала.**
3. **Подведение итогов урока. Обобщение знаний.**
4. **Домашнее задание.** Повторить изученный материал.

КОНСПЕКТ ПОД ЗАПИСЬ

Организация файлового ввода/вывода в окнах редактирования текстов

Окно Memo позволяет читать и редактировать файлы «только текст».

Окно RichEdit (стр.Win32) позволяет читать и редактировать как файлы «только текст», так и файлы в обогащенном формате RTF (раздельно форматировать различные фрагменты текста). Эти документы могут содержать только текст и не могут включать изображения.

И Memo, и RichEdit содержат текст в свойстве Lines типа TStrings.

Тип строк **TStrings** имеет методы **LoadFromFile** и **SaveToFile** загрузки из файла и сохранения в файле.

RichEdit1.Lines.LoadFromFile(SFile); - загрузит в окно редактирования текст из файла SFile,
RichEdit1.Lines.SaveToFile(SFile); - сохранит текст в файле SFile.

Аналогичным образом можно работать со списками компонентов **ListBox, ComboBox, CheckListBox, RadioGroup** (свойства **Items)** и ряда других.

Над файлами выполняют следующие операции:

1. **Open** - открытие файла, указанного пользователем,
2. **Save** - сохранение отредактированного документа в том же файле,
3. **Save As** - сохранения документа в файле, указанном пользователем,
4. **New** — создание пользователем нового документа.

Они реализуются по следующей схеме.

Вводится глобальная строковая переменная, хранящая имя файла:

```
var SFile: AnsiString = 'Неизвестный';
```

Открытие и сохранение в выбранном пользователем файле осуществляется вызовом стандартных диалогов Windows открытия и сохранения файла (компоненты **OpenDialog и SaveDialog)**.

Процедура **Open:**

```
if OpenDialog1.Execute then
begin
  SFile:=OpenDialog1.FileName;
  RichEdit1.Lines.LoadFromFile(SFile);
 end;
```

Процедура **SaveAs**:

```
SaveDialog1.FileName := SFile;
if SaveDialog1.Execute then
begin
 SFile := SaveDialog1.FileName;
 RichEdit1.Lines.SaveToFile(SFile);
end;
```

Процедура **Save**:

```
if  (SFile = 'Неизвестный') then <вызов процедуры SaveAs>
                              else  RichEdit1.Lines.SaveToFile(SFile);
```

Процедура **New**:

```
SFile := 'Неизвестный';
RichEdit1.Lines.Clear;
```

При закрытии приложения необходимо проверять, не было ли редактирования документа, и если было, то спросить у пользователя подтверждения закрытия.

Для этого используется обработчик события формы **OnCloseQuery**:

```
if RichEdit1.Modified then
CanClose := (Application.MessageBox('Текст документа не сохранен'+#13'Действительно хотите закончить работу?', 'Подтвердите завершение работы', MB_ICONQUESTION+MB_YESNO) = IDYES);
```

Свойство Modified, свидетельствующее об изменении документа пользователем, становится равным true автоматически при редактировании документа.

Однако устанавливать это значение равным false надо программно. Так что в процедурах открытия и сохранения файлов следует вставить оператор: RichEdit1.Modified := false;

Тонкости настройки диалогов

Компоненты-диалоги OpenDialog и SaveDialog (стр.Dialogs). Свойства:

Свойство	**Для чего используется**	**Пример**
Filter	фильтры типов файлов, отображаемых в диалогах	два фильтра: • текстовых файлов с расширениями .txt и .rtf • всех файлов с любыми расширениями. Filter = текстовые файлы (*.txt; *.rtf)|*.txt; *.rtf | все файлы |*.*
DefaultExt	расширение файлов по умолчанию	Если этого не сделать, пользователю придется писать его полное имя вместе с расширением. Это свойство задается без точки, предшествующей расширению. Например, "txt" или "rtf".
FilterIndex	номер фильтра, отображаемого при открытии диалога	В предыдущем примере FilterIndex = 1, тогда отображается сначала фильтр текстовых файлов.

Реакция на несуществующий файл при открытии: установка в true опций OfFileMustExist и OfPathMustExist в свойстве Options диалога.

Ввод/вывод графических файлов

При работе с графическими документами удобнее всего использовать компонент **Image,** в котором автоматически решаются многие проблемы отображения графики, в частности, перерисовки испорченных изображений.

Для файлового ввода/вывода можно использовать методы **LoadFromFile** и **SaveToFile** свойства **Picture** этого компонента.

В качестве диалогов открытия и сохранения для многих форматов графических файлов удобно использовать компоненты **OpenPictureDialog** и **SavePictureDialog.**

Загрузка графического файла в компонент Image1 может осуществляться оператором:

```
if OpenPictureDialog1.Execute
        then Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
```

При этом в свойстве **AutoSize** = **true,** чтобы размер компонента автоматически подстраивался под размер изображения загруженного файла.

Компонент Image не может загружать изображения с расширением .jpg. Для решения проблемы достаточно включить в модуль оператор

```
uses Jpeg;
```

УРОК 12.2 ОБРАБОТКА ДОКУМЕНТОВ В КОМПОНЕНТЕ RICHEDIT

Цель:

Образовательная: рассмотреть возможности компонента RichEdit для просмотра и редактирования любых документов. Изучить его основные свойства, методы и события.

Воспитательная: продолжить формирование у учащихся основ научного мировоззрения. Воспитание любви к профессии и предмету

Развивающая: развитие познавательных способностей (мышления, воображения и т.д.).

Тип занятия: урок формирования новых знаний

Форма организации учебного процесса: лекция

ХОД УРОКА

1. **Организационный момент** (требования к уроку, план работы)
2. **Изучение нового материала.**
3. **Задание 1:** в примере из лекции 1 создать панель инструментов, состоящую из следующих элементов:
    a. кнопки «Полужирный шрифт», «Курсив», «Подчеркивание», «Обычный шрифт»
    b. элемент для изменения размера шрифта
    c. элемент для изменения цвета шрифта. Попробовать использовать компонент ColorDialog со страницы Dialogs. `if ColorDialog1.Execute then RichEdit1.SelAttributes.Color:=ColorDialog1.Color;`
    d. кнопки «По левому краю», «По правому краю», «По центру»
    e. кнопки «Список», «Выключить список»
    f. кнопка «Печать»
4. **Задание 2.** Создать панель, с помощью которой можно создать договор из примеров 5-7.
5. **Задание 3.** Организовать поиск слова, введенного пользователем в компонент Edit.
6. **Подведение итогов урока. Обобщение знаний.**
7. **Домашнее задание.** Повторить изученный материал.

КОНСПЕКТ ПОД ЗАПИСЬ

Форматирование шрифта

Свойство SelAttributes задает атрибуты форматирования**.**

Если в окне имеется выделенный фрагмент текста, то **SelAttributes** определяет формат этого фрагмента, иначе определяет формат того текста, который будет вводиться, начиная с текущей позиции курсора.

Его подсвойства:

1. Color - цвет,
2. Name - имя шрифта,
3. Size и Height - определяют размер шрифта,
4. Style - стиль: fsBold — полужирный, fsltalic — курсив, fsUnderline — подчеркнутый, fsStrikeOut — зачеркнутый,
5. Charset — набор символов.

Пример №1:

```
RichEdit1.SelAttributes.Style:=RichEdit1.SelAttributes.Style +[fsBold]
RichEdit1.SelAttributes.Size:=14;
RichEdit1.SelAttributes.Color:=clRed;
```

Свойство DefAttributes содержит атрибуты по умолчанию. Эти атрибуты действуют до того момента, когда изменяются атрибуты в свойстве **SelAttributes.** Но значения атрибутов в **DefAttributes** сохраняются и в любой момент эти значения могут быть присвоены атрибутам свойства **SelAttributes** следующим оператором:

```
RichEdit1.SelAttributes := RichEdit1.DefAttributes;
```

В компоненте **ActionList** введены стандартные действия для операций форматирования. Для стандартных действий не надо задавать обработчики действий.

Форматирование абзацев

Свойство Paragraph отвечает **з**а выравнивание, отступы и т.д. в пределах текущего абзаца**.** Подсвойства:

Alignment	Определяет выравнивание текста. Может принимать значения: • taLeftJustify (влево), • taCenter (по центру) • taRightJustify (вправо).
Numbering	Управляет вставкой маркеров, как в списках. Может принимать значения • **nsNone** — отсутствие маркеров, • **nsBullet** — маркеры ставятся.
FirstIndent	Число пикселов отступа красной строки.
LefUndent	Отступ в пикселах от левого поля.
RightIndent	Отступ в пикселах от правого поля.
TabCount	Количество позиций табуляции.
Tab	Значения позиций табуляции в пикселах.

Значения подсвойств относятся к тому абзацу, в котором находится курсор.

Пример №2: Выравнивание по левому краю

```
RichEdit1.Paragraph.Alignment := taLeftJustify;
```

Пример №3: Отображение текущего абзаца как список, т.е. с маркерами:

```
RichEdit1.Paragraph.Numbering:=nsBullet;
```

Пример №4. Уничтожение списка в текущем абзаце

```
RichEdit1.Paragraph.Numbering:=nsNone;
```

Свойство WantTabs позволяет задавать позицию табуляции клавишей Tab. Если = **true** то клавиша Tab выполняет отступ в тексте. При этом свойство **Tab[i]** указывает позиции табуляции в пунктах: **Tab[0]** соответствует первой позиции, **Tab[1]** — второй и т.д.

Свойство TabCount указывает число позиций табуляции, определенных для данного абзаца. Единицей измерения позиций табуляции является пункт.

Программное формирование текста в RichEdit

Свойство Lines содержит список всех строк текста, записанного в **RichEdit.** Значение **Lines[i]** — это текст i-ой строки (индексы начинаются с 0). Число строк определяется свойством **Lines.Count.**

Свойство Text содержит те же строки, но они объединены в одну, причем разделителями служат символы "#13#10" — переход на новую строку и возврат каретки.

Добавление новой строки: RichEdit1.Lines.Add(‘строка’);

Пример №5: программа должна формировать стандартную форму некоего договора, в которую предмет договора, его условия и атрибуты лица, заключающего договор, черпаются из окон редактирования:

```
RichEdit1.Clear;
RichEdit1.SelAttributes.Name:= 'Courier New';
RichEdit1.SelAttributes.Size:= 14;
RichEdit1.Paragraph.Alignment:= taCenter;
RichEdit1.SelAttributes.Style:=RichEdit1.SelAttributes.Style +  [fsBold];
RichEdit1.Lines.Add('ДОГОВОР');
RichEdit1.SelAttributes.Style:=RichEdit1.SelAttributes.Style - [fsBold];
RichEdit1.Lines.Add(Edit1.Text);
RichEdit1.SelAttributes.Size:= 12;
RichEdit1.Paragraph.Alignment := taLeftJustify;
RichEdit1.Paragraph.FirstIndent:= 15;
RichEdit1.Paragraph.LeftIndent:= -15;
RichEdit1.Lines.Add('Зицпредседатель конторы "Рога и копыта" Фунт и '+ Edit2.Text
+'заключили настоящий договор ...');
```

Часто требуется форматировать отдельные слова или фрагменты, не завершающиеся концом строки. В этих случаях занесение очередных фрагментов текста удобно осуществлять методом **SetSelTextBuf.** В этот метод передается указатель на строку с нулевым символом в конце, содержащую вставляемый текст. А позиция вставки определяется свойством **SelStart** компонента **RichEdit.**

Если в тексте есть выделенный фрагмент	***Если выделенного текста нет***
SelStart указывает на начало выделения	**SelStart** указывает на текущую позицию, в которой расположен курсор
SelLength определяет длину выделенного текста в символах	**SelLength** = 0

Так что при наличии выделенного текста метод **SetSelTextBuf** заменяется его на текст, вставляемый методом **SetSelTextBuf.** А если выделения нет, текст вставляется в текущую позицию курсора.

Пример №6: добавление строки, в которой фамилии выделены жирным шрифтом:

```
RichEdit1.Lines.Add('Зицпредседатель конторы "Рога и копыта" ');
RichEdit1.SelStart:= RichEdit1.SelStart-2; //обход двух последних символов #13#10
RichEdit1.SelAttributes.Style:=RichEdit1.SelAttributes.Style +  [fsBold];
RichEdit1.SetSelTextBuf('Фунт');
RichEdit1.SelAttributes.Style:=RichEdit1.SelAttributes.Style - [fsBold];
RichEdit1.SetSelTextBuf(' и ');
RichEdit1.SelAttributes.Style:=RichEdit1.SelAttributes.Style +  [fsBold];
RichEdit1.SetSelTextBuf(PChar(Edit2.Text));
RichEdit1.SelAttributes.Style:=RichEdit1.SelAttributes.Style -  [fsBold];
RichEdit1.SetSelTextBuf (' заключили настоящий договор ...');
```

Пример №7: второй вариант выделения шрифтом отдельных слов

```
RichEdit1.SelStart:=Pos(‘Фунт′, RichEdit1.Text)-1;
RichEdit1.SelLength:=Length('Фунт');
RichEdit1.SelAttributes.Style:=RichEdit1.SelAttributes.Style + [fsBold];
RichEdit1.SelStart:=Pos(Edit2.Text, RichEdit1.Text)-1;
RichEdit1.SelLength:=Length(Edit2.Text);
RichEdit1.SelAttributes.Style:=RichEdit1.SelAttributes.Style + [fsBold];
```

Печать документа

Печать документа, загруженного в окно **RichEdit,** осуществляется методом **Print.**

Пример №8. обеспечивает печать текста компонента **RichEdit1,** причем задание на печать получает имя «Печать RichEdit1».

```
RichEdit1.Print('Печать RichEdit1');
```

Перенос строк и разбиение текста на страницы производится автоматически.

Для предоставления возможности установить характеристики печати, можно использовать диалог **PrintDialog.** Вызов диалога **PrintDialog**:

```
var i:integer;
if (PrintDialog1.Execute) then
  for i:=1  to PrintDialog1.Copies do RichEdit1.Print('Печать RichEdit1');
```

Поиск текста

Организация поиска с помощью метода **FindText** компонента **RichEdit.**

Общий вид:

```
function FindText(const SearchStr: string; StartPos,  Length:  Integer; Options:
                    TSearchTypes):  Integer;
```

1. **SearchStr** - текст искомого фрагмента.
2. Поиск ведется начиная с позиции, указанной параметром **StartPos,**
3. на протяжении **Length** символов.
4. Параметр **Options** является множеством опций, которое может содержать опции stWholeWord — поиск только целого слова, и stMatchCase — поиск с учетом регистра.

УРОК 12.3 РАБОТА С ПРОИЗВОЛЬНЫМИ ТИПАМИ ФАЙЛОВЫХ ДОКУМЕНТОВ

Цель:

Образовательная: рассмотреть основные функции для работы с произвольными типа файлов: ShellExecute и ShellExecuteEx, а также способ автоматизации работы с этими функциями - стандартное действие FileRun компоненты ActionList.

Воспитательная: продолжить формирование у учащихся основ научного мировоззрения. Воспитание любви к профессии и предмету

Развивающая: развитие познавательных способностей (мышления, воображения и т.д.).

Тип занятия: урок формирования новых знаний

Форма организации учебного процесса: лекция с элементами беседы

ХОД УРОКА

1. **Организационный момент** (требования к уроку, план работы)
2. **Изложение нового материала.**
3. **Подведение итогов урока. Обобщение знаний.**
4. **Домашнее задание.** Повторить изученный материал.

КОНСПЕКТ ПОД ЗАПИСЬ

Функция ShellExecute

Может открывать и печатать документ любого зарегистрированного в Windows типа. Под термином «открыть файл документа» понимается выполнение связанного с ним приложения и загрузка в него этого документа.

Для использования функции **ShellExecute** в оператор uses надо добавить модуль *ShellAPI.*

Общий вид:

```
function ShellExecute(Wnd: HWnd; Operation,  FileName,  Parameters,  Directory: PChar; ShowCmd: Integer): THandle;
```

Параметр	Значение
Wnd	дескриптор родительского окна, в котором отображаются сообщения запускаемого приложения. Обычно можно указать **Handle.**
Operation	указывает на строку с нулевым символом в конце, которая определяет выполняемую операцию. ➢ «open» — открыть файл, ➢ «print» — напечатать файл, ➢ «explore» — открыть папку. ➢ «edit» — редактировать (в Windows 2000/XP). Если параметр **Operation** равен **nil,** то по умолчанию выполняется операция «open».
FileName	указывает на строку с нулевым символом в конце, которая определяет имя открываемого файла или имя открываемой папки.
Parameters	указывает на строку с нулевым символом в конце, которая определяет передаваемые в приложение параметры. Если **FileName** указывает на строку, определяющую открываемый документ или папку, то этот параметр задается равным **nil.**
Directory	указывает на строку с нулевым символом в конце, которая определяет каталог по умолчанию.
ShowCmd	определяет режим открытия указанного файла.

Параметр **ShowCmd** может принимать следующие значения:

SW_MINIMIZE	Минимизирует указанное окно и активизирует следующее в Z-последовательности окно верхнего уровня в списке системы
SW MAXIMIZE	Максимизирует указанное окно
SW_SHOW	Активизирует и отображает окно в его текущей позиции и с текущими размерами
SW_SHOWMAXIMIZED	Активизирует и отображает окно в развернутом виде
SW_SHOWMINIMIZED	Активизирует и отображает окно в свернутом виде
SW SHOWNORMAL	Активизирует и отображает окно.

Функция возвращает дескриптор открытого приложения. Если возвращаемое значение меньше или равно 32, это указывает на ошибку.

Пример №1: открывает любой файл выбранный в окне открытия файла

```
uses ShellAPI;
…
if OpenDialog1.Execute then
  ShellExecute(Handle, 'open', PChar(OpenDialog1.FileName), nil,   nil,   SW_SHOWNORMAL);
```

Пример №2: печать документа открытого с помощью диалога открытия, печать происходит с помощью программы в которой открывается файл:

```
ShellExecute(Handle, 'print',  PChar(OpenDialog1.FileName), nil, nil, SW_SHOWNORMAL);
```

Функция ShellExecuteEx

Обладает более широкими возможностями, большинство из которых, правда, относятся к выполнению приложений, а не к просмотру файлов документов.

Общий вид:

```
function ShellExecuteEx(lpExecInfo: PShellExecuteInfo):BOOL;
```

Функция возвращает **false** в случае ошибки.

Параметр функции - указатель на запись типа **TShellExecuteInfo.**

Запись имеет следующие поля:

cbSize: DWORD	размер записи в байтах
Wnd: HWND	дескриптор окна приложения
lpVerb: PAnsiChar	вид операции («open», «edit», «print»)
lpFile: PAnsiChar	имя файла с путем
hInstApp: HINST	дескриптор стартовавшего приложения или идентификатор ошибки 1. SE_ERR_FNF - Файл не найден 2. SE_ERR_PNF - Путь не найден 3. SE_ERR_ACCESSDENIED - Доступ к файлу запрещен 4. E_ERR_OOM - Не хватает памяти 5. SE_ERR_SHARE - Файл захвачен другим пользователем 6. SE_ERR_NOASSOC - Нет приложения, связанного с файлом
nShow: Integer	режим отображения открываемого окна при запуске выполняемого файла. При работе с файлами документов значение должно=0.

Пример №3.

```
uses ShellAPI;
var SInfo: TShellExecuteInfo;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if OpenDialog1.Execute then
  begin
      SInfo.Wnd := Handle;
      SInfo.nShow := 0;
      SInfo.lpVerb := 'open';
      SInfo.lpFile := PChar(OpenDialog1.FileName);
      SInfo.cbSize := sizeof(SInfo);
      ShellExecuteEx (@Sinfo);
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
      SInfo.lpVerb:='print';
      ShellExecuteEx(@SInfo);
end;
```

Стандартное действие класса TFileRun

С помощью компонента **ActionList** можно настроить стандартное действие класса **TFileRun,** с помощью которого настраивать все параметры функций ShellExecute и ShellExecuteEx.

1. Установите на форму компонент **ActionList стр. Standart.**
2. Добавьте новое действие **FileRun1** (дважды щелкните по компоненту, в открывшемся окне в контекстном меню выберите команду New Standart Action и в появившемся окне выберите пункт FileRun).
3. Установите на форму кнопку и в ее свойстве **Action** укажите действием FileRun1.
4. В свойство **Operation** объекта **FileRun1** запишите требуемую операцию, например, **edit.**
5. Свойство **FileName** может содержать имя файла документа.
6. Если установить в **true** свойство **Browse,** то при выполнении действия пользователю будет сначала показано окно диалога.
7. Свойство **ShowCmd** действия **FileRun** определяет, каким образом будет показано пользователю окно, содержащее документ. Обычно можно согласиться со значением **scShowNormal,** установленным по умолчанию.

УРОК 12.4 ПРОСМОТР И РЕДАКТИРОВАНИЕ ЛЮБЫХ ДОКУМЕНТОВ С ПОМОЩЬЮ WEBBROWSER

Цель:

Образовательная: рассмотреть возможности компонента WebBrowser для просмотра и редактирования любых документов. Изучить его основные свойства, методы и события.

Воспитательная: продолжить формирование у учащихся основ научного мировоззрения. Воспитание любви к профессии и предмету

Развивающая: развитие познавательных способностей (мышления, воображения и т.д.).

Тип занятия: урок формирования новых знаний

Форма организации учебного процесса: лекция

ХОД УРОКА

1. **Организационный момент** (требования к уроку, план работы)
2. **Изучение нового материала.**
3. **Задание:** Составить программу, которая с помощью главного меню программы позволяла бы открыть любую Web-страницу и просмотреть ее с помощью компонента WebBrowser. Реализовать панель инструментов, с помощью которой можно перемещаться между страницами (вперед и назад), переходить на домашнюю страницу и печатать документ.
4. **Подведение итогов урока. Обобщение знаний.**
5. **Домашнее задание.** Повторить изученный материал.

КОНСПЕКТ ПОД ЗАПИСЬ

Компонент WebBrowser (стр. Internet) - универсальное средство загрузки и просмотра практически любых типов файлов.

```
if OpenDialog1.Execute then WebBrowser1.Navigate(OpenDialog1.FileName) ;
```

Метод ExecWB компонента WebBrowser **-** для выполнения действий сохранения, печати и предварительного просмотра.

Общий вид:

```
procedure ExecWB(cmdID: OLECMDID; cmdexecopt: OLECMDEXECOPT);
```

1. **cmdID** определяет выполняемую операцию. Он может принимать множество значений:

OLECMDID_SAVEAS	сохранить как ...
0LECMDID_PRINT	вывести на печать
OLECMDID_PRINTPREVIEW	предварительный просмотр

2. **cmdexecopt** способ выполнения команды. Значение 1 указывает, что команда будет выполняться после того, как пользователь введет какие-то значения в предложенном ему диалоге.

Свойства, методы и события компонента WebBrowser

Свойства	Значение
LocationName	Указывает краткое имя открытого ресурса. Например: «html1.html». Если же ресурс указан своим **URL,** то **LocationName** — совпадает с URL. Например: «http://www.bmompress.ru/db.exe».
LocationURL	Полный путь к файлу. Например: «file:///d:/tests/Internet/Html1.html».
FullName	Полное имя выполняемого модуля. Например: «D:\EXAMPL\BROUSER1.EXE»
Name	Имя модуля, управляющего процессом просмотра
Path	Путь к выполняемому модулю. Например: «D:\EXAMPL\»
Offline	определяет, может ли браузер соединяться с Интернет. Если установить **Offline = true,** то соединение будет невозможно и браузер сможет читать только файлы локального сервера.
Silent	определяет, будет ли браузер отображать диалоговые окна с сообщениями о невозможности соединиться, найти файл и т.п. Если **Silent** = **true,** то эти диалоги не вызываются.

Метод Navigate обеспечивает навигацию по документам и загрузку в браузер указанного ресурса. Общий вид:

```
procedure Navigate(const URL: WideString; var Flags: OleVariant; var TargetFrameName:
                   OleVariant; var PostData: OleVariant; var Headers:  OleVariant);
```

1. **URL** — это адрес загружаемого ресурса
2. **Flags** является множеством флагов, определяющих характер загрузки ресурса в браузер:

Константа	Значение	Описание
navOpenInNewWindow	**1**	Открывает ресурс в новом окне
navNoHistory	**2**	Блокирует добавление ресурса в список истории.
navNoReadFromCache	**4**	Запрещает чтение из диска ранее прочитанной той же страницы
navNoWriteToCache	**8**	Запрещает запись результатов
navAllowAutosearch	**16**	Если поиск ресурса закончился неудачей, браузер пытается перейти к соответствующим файлам **.com, .edu** и др.

Надо учитывать, что тип параметра **Flags** — **OleVariant.** Поэтому нельзя непосредственно включать в вызов **Navigate** указанные в таблицы значения флагов. Задавать флаги надо следующим образом:

```
var Flags: OleVariant;
begin
  Flags := navNoReadFromCache + navAllowAutosearch;
  WebBrowser1.Navigate(Edit1.Text,Flags);
end;
```

3. **TargetFrameName** определяет фрейм, в котором будет отображаться страница. Если **NULL** то отображается в данном окне.
4. **PostData** содержит данные, которые передаются на сервер, если метод **Navigate** используется для посылки серверу сообщения HTTP **POST.** Если значение **PostData** равно **NULL,** то метод генерирует сообщение HTTP **GET.**
5. **Headers** содержит данные заголовка.

Методы компонента

GoBack	Перемещается к предыдущему документу в списке истории. **Пример**: WebBrowser1.GoBack;
GoForward	Перемещается к последующему документу в списке истории
GoHome	загружает домашнюю страницу, которая обычно является начальной при открытии браузера.
GoSearch	загружает страницу поиска информации в страницах
Refresh	осуществляет повторную загрузку текущего документа
Stop	прерывает текущую операцию навигации или загрузки документа, а также останавливает любые фоновые операции, выполняемые на данной странице, в частности, воспроизведение аудио-, видео-фрагментов

Событие OnBeforeNavigate2 возникает перед выполнением любого из методов навигации, а также при щелчке пользователя на ссылке страницы Web**.**

Общий вид:

```
procedure TForm1.WebBrowser1BeforeNavigate2(Sender: TObject; const: IDispatch; var URL,
            Flags, TargetFrameName,  PostData, Headers: OleVariant; var Cancel: WordBool);
```

1. Параметр **pDisp** определяет интерфейс браузера.
2. Параметры **URL, Flags, TargetFrameName, PostData** и **Headers** аналогичны рассмотренным выше параметрам метода **Navigate.** Значения этих параметров можно изменить в обработчике события, изменив, например, адрес загружаемого документа. Правда, в настоящий момент флаги **Flags** изменять не разрешается.
3. Параметр **Cancel** определяет, будет ли просматриваться данный ресурс в браузере. Если задать **true** параметру **Cancel,** то просмотр будет прерван.

Событие OnProgressChange возникает во время загрузки очередного документа**.**

Общий вид:

```
procedure TForm1.WebBrowser1ProgressChange(Sender: TObject; Progress,  ProgressMax:
                                          Integer);
```

1. параметры **Progress** — объем загруженного документа
2. **ProgressMax** — полный объем документа.
3. Когда загрузка завершена, значение **Progress** становится равным -1.

Событие OnDocumentComplete наступает после завершения загрузки документа**.**

РАЗДЕЛ №13. ВЗАИМОДЕЙСТВИЕ С ПРИЛОЖЕНИЯМИ MS OFFICE ДЛЯ ОБРАБОТКИ ДОКУМЕНТОВ

УРОК №13.1. РАБОТА С СЕРВЕРОМ MICROSOFT WORD

Цель:

Образовательная: изучить основные принципы соединения приложения с серверами Microsoft Office, в частности с сервером Word. Рассмотреть возможности управления файлами, настройки параметров документа, вызова диалогов сервера, форматирование текста и изменение содержимого документа.

Воспитательная: продолжить формирование у учащихся основ научного мировоззрения. Воспитание любви к профессии и предмету

Развивающая: развитие познавательных способностей (мышления, воображения и т.д.).

Тип занятия: урок формирования новых знаний

Форма организации учебного процесса: лекция

ХОД УРОКА

1. **Организационный момент** (требования к уроку, план работы)
2. **Изложение нового материала.**
3. **Задание:** Составить программу, которая открывает текстовый редактор Microsoft Word и позволяет выполнить простейшие действия над открытым документом (в программе должны присутствовать кнопки управления шрифтом выделенного фрагмента текста, управления абзацем и др.).
4. **Подведение итогов урока. Обобщение знаний.**
5. **Домашнее задание.** Повторить изученный материал.

КОНСПЕКТ ПОД ЗАПИСЬ

Компоненты — серверы в Delphi

При работе с сервером Word в ваше приложение должен быть включен оператор

```
uses ... Word97;
```

или оператор

```
uses ... Word2000;
```

или оператор

```
uses ... WordXP;
```

Если требуется создать универсальное приложение, которое работало бы с любым офисом, то надо подключить два или три модуля. Например:

```
uses  ... Word97, Word2000;
```

Тот модуль, который подключается последним будет работать по умолчанию.

<u>Определение версии офиса:</u>

```
If (StrToInt(Copy(WordApplication1.Version, 0, Pos('.' ,WordApplication1.Version)-1)))< 9
  then ...  //для Word 97
  else .. . . //для Word 2000 и старше
```

<u>Два режима использования серверов:</u>

- перенос на форму со страницы Servers соответствующих компонентов **WordApplication** и **ExcelApplication**
- программное создание объектов классов **TWordApplication** и **TExcelApplication**.

Например, для сервера Word это делается следующим образом:

```
var WordApplication1: TWordApplication;
…
WordApplication1:= TWordApplication.Create(self);
... // операторы задания свойств
```

Свойства

AutoConnect	определяет, должен ли сервер автоматически загружаться с началом выполнения приложения. Если = **true,** то соединение с сервером произойдет в момент начала выполнения приложения. Если = **false** (по умолчанию), то соединение с сервером можно установить вызовом метода Connect: WordApplication1.Connect;
ConnectKind	определяет, как именно осуществляется соединение с сервером. Это свойство может принимать следующие значения:

ckRunningOrNew (по умолчанию)	Подсоединиться к выполняющемуся серверу или создать новый экземпляр сервера.
ckNewInstance	Всегда создавать новый экземпляр сервера.
ckRunningInstance	Только подсоединиться к выполняющемуся серверу.
ckRemote	Подсоединиться к удаленному серверу. Эта опция должна сочетаться с заданием свойства **RemoteMachineName** (должно указывать компьютер, на котором выполняется удаленный сервер).
ckAttachToInterface	Не подсоединяться к серверу. Вместо этого Приложение обеспечивает интерфейс методом **ConnectTo** (об этом методе будет сказано позднее). Опция **ckAttachToInterface** не может использоваться совместно с установкой в **true** свойства **AutoConnect.**

Отображение сервера на экране: WordApplication1.Visible:=true;

Разрыв соединения с сервером: метод **Disconnect**

Автоматическое завершение сервера: если **AutoQuit** = **true,** то при завершении приложения автоматически вызовется метод, завершающий сервер.

Пример работы с сервером Word:

```
var WordApplication1:TWordApplication;
procedure TForm1.FormCreate(Sender: TObject);
begin
        WordApplication1:=TWordApplication.Create(self);
```

```
        WordApplication1.ConnectKind:=ckNewInstance;
        WordApplication1.AutoQuit:=true;
        WordApplication1.Visible:=true; // если надо сделать видимым
end;

procedure TForm1.FormDestroy(Sender:  TObject);
begin
        WordApplication1.Free;
end;
```

Свойства и методы сервера Word

У свойства Options есть подсвойства **CheckSpellingAsYouType** и **CheckGrammarAsYouType** — указывающие, должен ли Word автоматически проверять синтаксис и грамматику и отмечать в тексте ошибки.

Для их отключения в сервере введите в приложение операторы:

```
WordApplication1.Options.CheckSpellingAsYouType := false;
WordApplication1.Options.CheckGrammarAsYouType := false;
```

Свойства:

ActiveDocument	активный документ
Documents	собрание всех документов, открытых в Word в данный момент. Каждый документ представлен в этом собрании как объект **Document,** имеющий в свою очередь собственные свойства и методы.
Documents.Count	Общее число открытых документов

Documents. Add	Создание нового документа **Document**
	Параметры:
	Для Office 97:
	• **Template** указывает шаблон, который используется при создании документа
	• **NewTemplate** - открывается ли документ как шаблон (= **true),** или как обычный документ.
	Для Office 2000 (еще 2 необязательных параметра):
	• **DocumentType**— тип документа
	  ○ **wdNewBlankDocument** — новый пустой документ (по умолч.)
	  ○ **wdNewEmailMessage** — новое электронное сообщение
	  ○ **wdNewFrameset** — новый фрейм
	  ○ **wdNewWebPage** — новая веб-страница
	• **Visible -** должен ли быть виден этот документ (true – по умолч.)
	Если какие-то аргументы не являются обязательными, то все равно они должны фигурировать в вызове метода. Только вместо их значений может быть указана **EmptyParam**. Эта переменная объявлена в модулях System и OleCtrls.
	Пример для Office 97:
	WordApplication1.Documents.Add(EmptyParam, EmptyParam);
	Пример для Office 2000/XP:
	WordApplication1.Documents.Add(EmptyParam, EmptyParam, EmptyParam, EmptyParam);

Selection	ссылка на объект **Selection** — выделенный фрагмент текста или, если нет выделения, просто текущая позиция курсора в документе.
Selection. InsertBefore	вставляет текст указанный в качестве параметра в скобках до объекта Selection. WordApplication1.Selection.InsertAfter(Edit1.text);
Selection. InsertAfter	вставляет текст после объекта Selection
Selection. Font	Шрифт выделенного фрагмента Подсвойства: • **Name** — имя шрифта • **Bold** — жирный • **Italic** — курсив • **Size** — размер • **Underline** — подчеркнутый

<table>
<tr><td></td><td>

StrikeThrough — перечеркнутый
DoubleStrikeThrough — перечеркнутый двойной линией
Shadow — с тенью
Emboss — приподнятый
Engrave — утопленный
Hidden — невидимый
Subscript — нижний индекс
Superscript — верхний индекс

WordApplication1.Selection.Font.Bold:=1;
</td></tr>
<tr><td>Selection. ParagraphFormat</td><td>
формат абзаца. Подсвойство: Alignment — выравнивание

wdAlignParagraphLeft— влево
wdAlignParagraphCenter — по центру
wdAlignParagraphRight— вправо
wdAlignParagraphJustify — по ширине

WordApplication1.Selection.ParagraphFormat.Alignment:=
wdAlignParagraphCenter;
</td></tr>
<tr><td>Selection. Paste</td><td>
Вставляет в документ информацию из буфера обмена
WordApplication1.Selection.Paste;
</td></tr>
<tr><td>Selection. Collapse</td><td>
свертывает выделение к его начальной или конечной точке, т.е. снимает выделение, перемещая курсор к его началу или концу
Аргумент:

Direction определяет, куда перемещается курсор:

wdCollapseStart - к начальной точке
wdCollapseEnd - к конечной

Пример:
WordApplication1.Selection.Collapse(EmptyParam);
уберет выделение и переместит курсор в позицию перед бывшим выделением
Пример 2:
var Direction:OleVariant;
Direction:=wdCollapseEnd;
WordApplication1.Selection.Collapse(Direction);
убирает выделение и перемещает курсор на позицию, следующую за бывшим выделением.
</td></tr>
</table>

Объект Document

Объект Range может создаваться специальным методом **Range,** в котором в качестве начала и конца указываются определенные позиции символов или параграфы.

Например создаются объект **MyRange,** начинающийся с десятого, и кончающийся пятидесятым символом документа:

```
var MyRange,b,e: OleVariant;
…
b:=10;
e:=50;
MyRange := WordDocument1.Range(b,e);
```

Форматирование жирным шрифтом фрагмент текста: MyRange.Font.Bold := 1;

Форматирование всего текста жирным шрифтом:

```
WordDocument1.Content.Font.Bold:=1;
```

Свойство Dialogs - это собрание встроенных диалогов Word. Доступ осуществляется через выражение вида:

```
WordApplication1.Dialogs.Item(WdWordDialog)
```

где константа **WdWordDialog** может принимать одно из предопределенных значений:

wdDialogEditFind	Найти фрагмент текста
wdDialogEditPasteSpecial	Специальная вставка из буфера обмена
wdDialogEditReplace	Заменить фрагмент текста
wdDialogFileFind	Найти файл
wdDialogFileNew	Новый файл
wdDialogFileOpen	Открыть файл

wdDialogFilePageSetup	Параметры страницы
wdDialogFilePrint	Печать файла
wdDialogFilePrintSetup	Установить принтер
wdDialogFileSaveAs	Сохранить файл как
wdDialogFileSummaryInfo	Свойства документа (Статистика)
wdDialogFormatFont	Шрифт
wdDialogFormatParagraph	Абзац
wdDialogInsertDatabase	Вставить базу данных
wdDialogInsertFile	Вставить файл
wdDialogInsertPageNumbers	Вставить номера страниц

Метод Show - открывает пользователю соответствующее диалоговое окно и выполняет те команды, которые указал в нем пользователь. Аргумент **TimeOut** — время в миллисекундах, после которого диалог автоматически закроется. Если в качестве **TimeOut** передать **EmptyParam,** то диалог закрывается только пользователем.

Пример вызова диалога открытия файла:

```
var Dial: OleVariant;
  …
WordApplication1.Visible := true;
Dial:=wdDialogFileOpen;
WordApplication1.Dialogs.Item(Dial).Show(EmptyParam);
```

Вызов диалога методом **Show** возвращает целое значение, позволяющее определить, какой кнопкой пользователь закрыл диалог:

-2	Кнопкой Закрыть
-1	Кнопкой ОК
0	Кнопкой Отмена или клавишей Esc
>0	Одной из командных кнопок: 1 — первой, 2 — второй и т.д.

Сохранение документа: метод SaveAs.

```
var FileName: OleVariant;
FileName := 'My.doc';
WordDocument1.SaveAs(FileName) ;
```

Печать документа без отображения диалога печати: метод Printout:

Предварительный просмотр документа: метод PrintPreview

УРОК №13.2. РАБОТА С СЕРВЕРОМ MICROSOFT EXCEL

Цель:

Образовательная: изучить основные принципы соединения приложения с сервером Excel. Рассмотреть возможности управления рабочими книгами, листами, ячейками. Использование автоматизации OLE при работе с Excel, Word, Outlook, Access.

Воспитательная: продолжить формирование у учащихся основ научного мировоззрения. Воспитание любви к профессии и предмету

Развивающая: развитие познавательных способностей (мышления, воображения и т.д.).

Тип занятия: урок формирования новых знаний

Форма организации учебного процесса: лекция

ХОД УРОКА

1. **Организационный момент** (требования к уроку, план работы)
2. **Изложение нового материала.**
3. **Задание:** Составить программу, которая открывает табличный процессор Microsoft Excel и позволяет выполнить простейшие действия над открытым документом.
4. **Подведение итогов урока. Обобщение знаний.**
5. **Домашнее задание.** Повторить изученный материал.

КОНСПЕКТ ПОД ЗАПИСЬ

Схема организации работы с Excel может иметь следующий вид:

```
uses ... OleServer, Excel97, Excel2000;
…
var ExcelApplication1: TExcelApplication; //объект сервера
    ExcelWorkbook1: ExcelWorkbook;     //объект книги
    ExcelWorksheet1: ExcelWorkSheet;   // объект страницы книги
…
procedure TForm1.FormCreate(Sender: TObject);
begin
  ExcelApplication1:= TExcelApplication.Create(Self);
  ExcelApplication1.ConnectKind := ckNewInstance;
  ExcelApplication1.AutoQuit := true;
  ExcelApplication1.Visible[LOCALE_USER_DEFAULT]   := true;
  ExcelWorkbook1  := ExcelApplication1.Workbooks.Add(EmptyParam,  0) ;
  ExcelWorksheet1  := ExcelApplication1.ActiveWorkbook.ActiveSheet as ExcelWorksheet;
end;
```

У свойства **Visible** имеется параметр **LCID** — идентификатора локализации. Его значение следует задавать **LOCALE_USER_DEFAULT** или **LOCALE_SYSTEM_DEFAULT** — идентификатором локализации текущего пользователя или системы.

ОПЕРАЦИИ НАД КНИГАМИ

Создание новой книги:

```
ExcelApplication1.Workbooks.Add(EmptyParam,  LOCALE_USER_DEFAULT);
```

Открытие книги:

```
if OpenDialog1.Execute
  then ExcelWorkbook1 := ExcelApplication1.Workbooks.Add(OpenDialog1.FileName,
                                                          LQCALE_USER_DEFAULT);
```

Число листов в книгах по умолчанию: свойство SheetsInNewWorkbook:

```
ExcelApplication1.SheetsInNewWorkbook[LOCALE_USER_DEFAULT] := 1;
```

Число открытых книг: подсвойство Count свойства Workbooks**.**

Сохранение книги: метод Save

```
ExcelApplication1.ActiveWorkbook.Save(LOCALE_USER_DEFAULT);
```

Сохранить все открытые книги:

```
var i: integer;
for i:=1 to ExcelApplication1.Workbooks.Count do
          ExcelApplication1. Workbooks[i].Save(LOCALE_USER_DEFAULT);
```

Закрыть все книги: метод Close

```
ExcelApplication1.Workbooks.Close(LOCALE_USER_DEFAULT);
```

Закрыть конкретную книгу:

```
procedure Close(SaveChanges: OleVariant; Filename: OleVariant; RouteWorkbook: OleVariant; lcid: Integer);
```

Параметры:

1. **SaveChanges** указывает, что должно происходить, если в закрываемой книге сделаны какие-то изменения, и они еще не сохранены.
   a. = **true,** то изменения сохраняются в файле, указанном в параметре **Filename.**
   b. **= false,** то изменения не сохраняются, независимо от значения **Filename.**
   c. **= EmptyParam,** то пользователю предлагается вопрос, надо ли сохранять изменения в файле.
2. **RouteWorkbook** используется в тех случаях, когда книга предназначена для пересылки какому-то адресату. В обычных случаях = **EmptyParam.**

Например, оператор закрывает первую книгу, а если в ней есть несохраненные изменения, то пользователю задается вопрос об их сохранении.

```
ExcelApplication1.Workbooks[1].Close(EmptyParam, EmptyParam, EmptyParam,
                                     LOCALE_USER_DEFAULT);
```

Печать книги или ее отдельных листов: метод **Printout**

Для файла *Excel2000* и *ExcelXP:*

```
procedure Printout(From: OleVariant; To_: OleVariant; Copies: OleVariant; Preview:
                                  OleVariant; ActivePrinter: OleVariant; PrintToFile: OleVariant;Collate:
                                  OleVariant; PrToFileName: OleVariant; lcid: Integer);
```

Параметры

1. **From** - номер страницы, с которой начинается печать. Если =**EmptyParam,** то печатаются страницы, начиная с первой.
2. **To_** указывает номер последней печатаемой страницы. Если =**EmptyParam,** то последней печатаемой страницей является последняя страница книги.
3. **Copies** определяет число копий.
4. **Preview** указывает, должно ли перед печатью открываться окно предварительного просмотра (при значении **false** или **EmptyParam** — не должно).
5. **ActivePrinter** может задавать имя принтера (по умолчанию — текущий принтер).
6. **PrintToFile** указывает, должна ли печать осуществляться в файл.
   a. Если =**false** или **EmptyParam** — не должна.
   b. Если =**true** параметр **PrToFileName** указывает имя файла.
7. **Collate** указывает, надо ли напечатанные страницы разбирать по копиям (если =**false** или **EmptyParam** — не надо).
8. Параметр **lcid** = **LOCALE_USER_DEFAULT.**

ОПЕРАЦИИ С ЛИСТАМИ КНИГИ

Коллекция листов содержится в свойстве **Worksheets** объекта книги. К листу можно обращаться по индексу или по имени. Например, следующие операторы открывают и активизируют первый лист книги, представленной объектом **ExcelWorkbook1,** передают указатель на этот лист в переменную **ExcelWorksheet1** и активизируют лист, т.е. выдвигают его на первый план в окне Excel:

```
ExcelWorksheet1  := ExcelWorkbook1.Worksheets[1] as ExcelWorksheet;
ExcelWorksheet1.Activate(LOCALE_USER_DEFAULT);
```

Добавить новый лист: метод **Add** объекта Worksheets:

```
function   Add(Before: OleVariant; After: OleVariant; Count: OleVariant; Type_: OleVariant;
                 lcid:  Integer):  IDispatch;
```

Параметры:

1. **Before** или **After** — это объект листа, перед которым или после которого осуществляется вставка. Обычно достаточно задать только один из этих параметров, а другой сделать равным **EmptyParam.** Если оба =**EmptyParam,** то новые листы вставляются перед текущим активным листом.
2. **Count** - число вставляемых листов. Если =**EmptyParam,** то вставляется один лист.
3. **Type_** определяет тип вставки. Если =**EmptyParam** вставляется новый пустой лист.

Вставить один новый лист перед активным листом активной книги:

```
ExcelWorksheet1:= ExcelApplication1.Worksheets.Add( EmptyParam, EmptyParam,
                 EmptyParam, EmptyParam,  LOCALE_USER_DEFAULT)  as ExcelWorksheet;
```

Вставить два новых листа после третьего листа активной книги:

```
var After, Num: OleVariant;
…
After := ExcelApplication1 .Worksheets [3] ;
Num := 2;
ExcelWorksheet1  := ExcelApplication1.Worksheets.Add(EmptyParam,After,Num, EmptyParam,
LOCALE_USER_DEFAULT)   as ExcelWorksheet;
```

Изменение имени листа: свойство **Name:** ExcelWorksheet1.Name := 'Счет-фактура';
Удаление листа: метод **Delete:** ExcelWorksheet1.Delete(LOCALE_USER_DEFAULT);
Предварительный просмотр листа: метод **PrintPreview:**

```
procedure PrintPreview(EnableChanges: OleVariant; lcid:  Integer);
```

Параметр **EnableChanges** указывает возможность внесения изменений при просмотре. Например, следующий оператор обеспечивает предварительный просмотр активного листа книги:

```
(ExcelApplication1.ActiveSheet as ExcelWorksheet).PrintPreview(true, LOCALE_USER_DEFAULT);
```

РАБОТА С ЯЧЕЙКАМИ

Работа с ячейками сводится к тому, что на листе выделяется множество связанных друг с другом ячеек **Range**.

Свойство **Value** позволяет читать или изменять данные из объекта **Range.**

Метод **Range** создает объект **Range.** В качестве начала и конца указываются левая верхняя и правая нижняя ячейки. Если требуется только одна ячейка, вместо второй можно задать **EmptyParam.**

Например, следующие операторы создают объект **MyRange,** выделяющий всего одну ячейку B2, читают и изменяют значение этой ячейки.

```
var MyRange, V: OleVariant;
…
MyRange := ExcelWorksheet1.Range['B2', EmptyParam];
// просмотр содержимого ячейки B2
V := MyRange.Value;
ShowMessage(V);
// занесение в ячейку значения из окна Edit1
MyRange.Value := Edit1.Text;
```

Свойство **Font** объекта **Range** задает шрифт текста выделенного диапазона ячеек. (подсвойства Bold, Italic, Underline, Color, Size, Name).

Например,

```
MyRange.Font.Bold := true;
MyRange.Font.Color := clRed;
```

ИСПОЛЬЗОВАНИЕ АВТОМАТИЗАЦИИ OLE ПРИ РАБОТЕ С EXCEL, WORD, OUTLOOK, ACCESS

Обращение к Word, Excel и иным компонентам Microsoft Office как к объектам автоматизации.

Ниже рассмотрены основные операции, используемые при работе с Excel, независимо от версии Microsoft Office. Приложение должно содержать оператор:

```
uses ComObj,  Excel97;
```

Этот оператор подключает модуль *ComObj,* который необходим для работы с объектами OLE. А модуль *Excel97* не является обязательным. Он потребуется нам только потому, что содержит объявления констант **xlDialogOpen** и **xlDialogPrint,** используемых в дальнейших примерах. Если эти константы не требуются, модуль *Excel97* можно не подключать.

Для работы с Excel и входящими в Excel объектами можно объявить следующие глобальные переменные:

```
var
Excel: OleVariant; // будет связана с экземпляром Excel
WorkBook: OLEVariant; // будет связываться с рабочей книгой
Worksheet: OLEVariant; // будет связываться с листом
MyRange: OleVariant; // будет связываться с диапазоном выделенных ячеек.
V: OleVariant; // значения, полученные из ячеек таблицы Excel или записываемые в них.
```

Запуск нового экземпляра Excel и доступ к нему осуществляется оператором:

```
Excel := CreateOleObject('Excel.Application');
```

Функция **CreateOleObject** - создает объект класса, указанного ее аргументом, и возвращает ссылку на идентификатор интерфейса типа **IDispatch,** используемый для связи с объектом.

Подключиться к уже выполняющемуся на компьютере экземпляру Excel:

```
Excel  := GetActiveOleObject('Excel.Application');
```

Функция **GetActiveOleObject** соединяется с уже выполняющимся экземпляром программы.

Сделать Excel видимым: Excel.Visible := true;

Завершение выполнения Excel: Excel.Quit;

Открытие новой рабочей книги: WorkBook := Excel.WorkBooks.Add;

Число листов по умолчанию: Свойство **SheetsInNewWorkbook.**

Открытие рабочей книги, хранящейся в указанном файле:

```
if OpenDialog1.Execute then WorkBook := Excel.WorkBooks.Open(OpenDialog1.FileName);
```

Закрытие рабочей книги: WorkBook.Close;

Соединение переменной **Worksheet** с листом, имеющим имя «Лист1» и этот лист становится активным:

```
Worksheet := WorkBook.Worksheets.Item['Лист1'];
Worksheet.Activate;
```

Пример: соединение переменной **Worksheet** с активной книгой, выделение на активном листе ячейки A1, и занесение в эту ячейку значения, заданного в окне Edit2 полужирным красным шрифтом:

```
Worksheet := WorkBook.ActiveSheet;
MyRange := Worksheet.Range['A1'];
MyRange.Value := Edit2.Text;
MyRange.Font.Bold := true;
MyRange.Font.Color := clRed;
```

Пример: чтение данных из указанной ячейки:

```
Worksheet := WorkBook.ActiveSheet;
MyRange := Worksheet.Range['A1'];
V := MyRange.Value;
ShowMessage(V);
```

Вызов диалога открытия файла рабочей книги: Excel.Dialogs[xlDialogOpen].Show;

Вызов диалога печати: Excel.Dialogs[xlDialogPrint].Show;

Предварительный просмотр активного листа: Excel.ActiveSheet.PrintPreview;

Работа с помощью автоматизации OLE с почтовой программой.

Пример автоматического создания и отправки почтового сообщения:

```
uses ComObj , Outlook8 {для констант} ;
…
var Outlook: OleVariant; Letter: OleVariant;
…
Outlook := CreateOleObject('Outlook.Application');
Letter := Outlook.CreateItem(olMailItem);
Letter.Recipients.Add('aaa@aa.ru');
Letter.CC := 'aaa@cc.ru';
Letter.Body := Memo1.Text;
Letter.Subject := 'Тест';
Letter.Display;
Letter.Send; Outlook.Quit;
```

Первый выполняемый оператор связывает переменную **Outlook** с почтовой программой. Второй оператор методом **CreateItem** создает новый объект почтового сообщения. О том, что создается именно почтовое сообщение, свидетельствует параметр **olMailItem.** Далее в список адресатов сообщения (свойство **Recipients)** добавляется адрес "aaa@aa.ru". В свойство CC заносится адрес получателя копии письма "aaa@cc.ru". В свойство **Body** заносится текст сообщения из окна редактирования **Memo1.** В свойство **Subject** заносится тема сообщения. Метод **Display** делает подготовленное послание видимым. В принципе, для отправки сообщения это не требуется и может понадобиться только для проверки и внесения каких-то исправлений. Метод **Send** отправляет сообщение. Метод **Quit** закрывает почтовую программу.

РАЗДЕЛ №14. ГРАФИКИ И ДИАГРАММЫ

УРОК №14.1 ГРАФИКИ И ДИАГРАММЫ. ОСНОВНЫЕ СВОЙСТВА И МЕТОДЫ КОМПОНЕНТА CHART

Цель:

Образовательная: рассмотреть основные свойства, методы и события компонента Chart, пример построения графика функции с помощью компонента Chart. Изучить возможности динамического изменения серии диаграммы, ее добавления и удаления. А также рассмотреть два основных типа диаграмм и принципы работы с ними: TBarSeries и TPieSeries.

Воспитательная: продолжить формирование у учащихся основ научного мировоззрения. Воспитание любви к профессии и предмету

Развивающая: развитие познавательных способностей (мышления, воображения и т.д.).

Тип занятия: урок формирования новых знаний

Форма организации учебного процесса: лекция

ХОД УРОКА

1. **Организационный момент** (требования к уроку, план работы)
2. **Изложение нового материала.**
3. **Задание:** Составить программу, выводящую на экран график синуса и косинуса.
4. **Подведение итогов урока. Обобщение знаний.**
5. **Домашнее задание.** Повторить изученный материал.

КОНСПЕКТ ПОД ЗАПИСЬ

Компонент **Chart** позволяет строить различные диаграммы и графики, которые выглядят очень эффектно, с трехмерными эффектами.

Простейшее приложение с графиками

Компонент **Chart** предоставляет поле, на котором строятся графики и диаграммы, управляет координатными осями и формой отображения. Отображаемые данные содержатся в объектах **Series** типа **TChartSeries.** Для каждого компонента **Chart** можно указать несколько серий. Если должны отображаться графики, то каждая серия соответствует одной кривой на графике. Если должны отображаться диаграммы, то для некоторых видов диаграмм можно наложить друг на друга несколько различных серий, для других (например, для круговых диаграмм) это, вероятно, будет выглядеть некрасиво. Однако и в этом случае можно задать для одного компонента **Chart** несколько серий одинаковых данных с разным типом диаграммы. Тогда, делая в каждый момент времени активной одну из них, вы можете предоставить пользователю выбор типа диаграммы, отображающей интересующие его данные.

Простейшее приложение, отображающее графики синуса и косинуса.

1. Разместите на форме компонент **Chart,** задайте в его свойстве **Align** = **alClient** и сделайте двойной щелчок на этом компоненте.
2. Перед вами откроется окно Редактора Диаграмм **Chart**. Оно имеет две основные страницы:
   2.1. Chart — задание общих настроек компонента,
   2.2. Series — настройка свойств отдельных серий.
3. на закладке Series страницы Chart нажмите кнопку Add.
4. В открывшемся окне выберите тип диаграммы Line — обычный кусочно-линейный график.
5. В обработчике события формы **OnCreate** напишите код программы:

```
procedure TForm1.FormCreate(Sender: TObject);
var i: integer;
begin
  for i:=0 to 100 do begin
          Series1.AddXY(0.02*Pi*i,sin (0.02*Pi*i), '',clRed);
          Series2.AddXY(0.02*Pi*i,cos(0.02*Pi*i), '',clBlue);
  End;
end;
```

Основные свойства Chart

Свойство	Описание
AllowPanning	Управляет возможностью пользователя прокручивать график. • **pmNone** — прокручивание запрещено, • **pmHorizontal** — разрешена горизонтальная прокрутка, • **pmVertical** — разрешена вертикальная прокрутка, • **pmBoth** — разрешена прокрутка в обоих направлениях
AllowZoom	Разрешает или запрещает возможность пользователя изменять масштаб графика
AnimatedZoom	Определяет скачкообразное, или постепенное изменение масштаба
AnimatedZoomSteps	Число шагов постепенного изменения масштаба
AxisVisible	Определяет видимость координатных осей
BackImage	Изображение на фоне поля графика
BackWall, BottomWall, LeftWall	Это объекты, описывающие при трехмерном изображении соответственно заднюю, нижнюю и левую стенки
BottomAxis, DepthAxis, LeftAxis, RightAxis, TopAxis	Координатные оси
Foot, Title	Объекты, описывающие подпись и заголовок графика
Legend	Легенда (список обозначений)
MarginLeft, MarginTop, MarginRight, MarginBottom	Левое, верхнее, правое и нижнее поля графика
Series	Индексированный список серий
View3d	Обеспечивает трехмерный, или плоский характер изображения

Основные методы Chart

ChartPreview	Вызов диалога предварительного просмотра перед печатью
ChartXCenter, ChartYCenter	Возвращают координаты соответственно середины горизонтальной и вертикальной осей
MaxXValue, MaxYValue, MinXValue, MinYValue	Возвращают минимальные и максимальные значения координатных осей
ZoomRect,ZoomPercent, UndoZoom	Методы изменения масштаба

События Chart

OnAfterDraw	Наступает после прорисовки изображений всех серий.
OnClickAxis	Наступает при щелчке на оси.
OnClickLegend	Наступает при щелчке на легенде.
OnClickSeries	Наступает при щелчке на одной из точек серии.

Вызов окна предварительного просмотра:

```
uses teeprevi;
…
ChartPreview(Form1, Chart1);
```

Печать в альбомной (горизонтальной) или книжной ориентации: методы PrintLandscape и PrintPortrait

Добавление точки в диаграмму

Если тип серии предполагает численные значения аргумента X и функции Y, добавление может осуществляться методом **AddXY.** Функция возвращает индекс новой точки в массиве **Values**.

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

Параметры:

1. **AXValue** и **AYValue** задают значения X и Y.
2. **AXLabel** задает текст метки, соответствующей добавляемой точке.
3. **AColor** задает цвет точки.

Примеры:

```
Series1.AddXY(O,5,'начало'#13'бизнеса',clRed);
Series1.AddXY(1,6,'',clRed);
Series 1.AddXY(3,8);
```

Для занесения точки в серию типа, который не имеет значений X, используется функция **AddY:**

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

Параметры:

1. **AYValue** указывает значение Y,
2. **AXLabel** — метку точки,
3. **AColor** — цвет.

Удаление точки осуществляется методом **Delete,** который удаляет точку с индексом **ValueIndex.**

Метод **Clear** полностью очищает список точек серии.

Метод **Count** возвращает текущее число точек, имеющихся в списке серии.

Метод **VisibleCount** возвращает число точек, видимых в данный момент.

Динамическое создание серии

```
Var tmpSeries: TChartSeries;
…
TmpSeries:= TBarSeries.Create (Chart1);  //создание новой серии
TmpSeries.ParentChart:= Chart1;  //включение серии в список серий компонента Chart1
TmpSeries.Free; //удаление серии
```

Классы диаграмм TBarSeries и THorizBarSeries

Классы **TBarSeries** и **THorizBarSeries** создают вертикальные и горизонтальные диаграммы. Различие заключается в том, что вертикальная и горизонтальная оси меняются местами.

Диаграммы рассматриваемых типов не имеют числовой оси аргумента. Поэтому добавление в них новых точек осуществляется методом **Add.**

Пример:

```
Series1.Add(155, 'Цех 1', GetDefaultColor(1));
Series1.Add(251, 'Цех 2', GetDefaultColor(2));
Series1.Add(203, 'Цех 3', GetDefaultColor(3));
Series1.Add(404, 'Цех 4', GetDefaultColor(4));
Series2.Add(140, 'Цех 1', GetDefaultColor(1));
Series2.Add(20,  'Цех 2', GetDefaultColor(2));
Series2.Add(253, 'Цех 3', GetDefaultColor(3));
Series2.Add(200, 'Цех 4', GetDefaultColor(4));
```

Первые аргументы функций **Add** задают значения в точках (155, 251 и т.п.). Задаваемые метки («Цех1», «Цех 2» и т.п.) являются как бы делениями оси аргументов, идентифицирующими положение течек. В данном примере для каждого цеха задается свой цвет. И для обеих серий последовательность цветов повторяется. Это осуществляется с помощью функции **GetDefaultColor.**

По оси Z в трехмерном отображении отложены названия **(Title)** серий. В приведенном примере это «январь» и «февраль».

Круговые диаграммы — класс TPieSeries

Этот тип диаграмм не имеет ни одной шкалы, так как значения точек отображаются углами соответствующих секторов.

Добавление точек в диаграмму производится методом **AddPie,** можно добавлять точки и методом **Add.**

Пример:

```
Series9.AddPie(140, 'Цех 1',  clRed);
Series9.AddPie(20,  'Цех 2', clYellow);
Series9.AddPie(253, 'Цех 3', clBlue);
Series9.AddPie(200, 'Цех 4', clGreen);
```

Первый аргумент содержит численное значение выпуска продукции соответствующим цехом. Второй аргумент — метка, которая фигурирует и в маркерах, и в легенде. Третий аргумент указывает цвет, которым отображается соответствующий сегмент.

РАЗДЕЛ №15. НАСТРОЙКА МЕНЮ И ИНСТРУМЕНТАЛЬНЫХ ПАНЕЛЕЙ

УРОК №15.1. НАСТРОЙКА МЕНЮ И ИНСТРУМЕНТАЛЬНЫХ ПАНЕЛЕЙ

Цель:

Образовательная: рассмотреть компоненты и их свойства, позволяющие пользователю создавать и настраивать меню и инструментальные панели, переставлять их элементы, добавлять и удалять разделы и быстрые кнопки.

Воспитательная: продолжить формирование у учащихся основ научного мировоззрения. Воспитание любви к профессии и предмету

Развивающая: развитие познавательных способностей (мышления, воображения и т.д.).

Тип занятия: урок формирования новых знаний

Форма организации учебного процесса: лекция

ХОД УРОКА

1. **Организационный момент** (требования к уроку, план работы)
2. **Изложение нового материала.**
3. **Подведение итогов урока. Обобщение знаний.**
4. **Домашнее задание.** Повторить изученный материал.

КОНСПЕКТ ПОД ЗАПИСЬ

Для создания главного меню используется компонент **MainMenu** (стр. Standart).
Для создания контекстного меню используется компонент **PopupMenu** (стр. Standart).

Компонент MainMenu

Невизуальный компонент. Основное свойство компонента — **Items.** Его заполнение производится с помощью Конструктора Меню, вызываемого двойным щелчком на компоненте **MainMenu** или нажатием кнопки с многоточием рядом со свойством Items в окне Инспектора Объектов.

При работе в конструкторе меню новые разделы можно вводить, помещая курсор в рамку из точек, обозначающую место расположения нового раздела.

Введите название пункта меню в свойстве **Caption**. Если

В качестве значения Caption ввести символ «-», то вместо раздела в меню появится разделитель.

В редакторе меню с помощью команды **Create Submenu** можно ввести подменю в выделенный раздел.

Горячие клавиши

Свойство ShortCut определяет горячие клавиши. Чтобы определить «горячие» клавиши, надо открыть выпадающий список свойства Shortcut и выбрать нужную комбинацию клавиш.

Свойства пунктов меню

Checked	**=true,** указывает, что в пункте меню будет отображаться маркер флажка, показывающий, что данный раздел выбран
AutoCheck	**=true,** то при каждом выборе пользователем данного раздела маркер будет автоматически переключаться, указывая то на выбранное состояние, то на отсутствие выбора.
RadioItem	**=true,** определяет, что данный раздел должен работать в режиме радиокнопки совместно с другими разделами, имеющими то же значение свойства **GroupIndex.**
GroupIndex	По умолчанию **GroupIndex =** 0. Если >0, то, если имеется несколько пунктов с одинаковым значением **GroupIndex и с RadioItem = true,** то в них могут появляться маркеры флажков, причем только в одном из них**.** Если задать программно в одном из этих пунктов **Checked = true,** то в остальных пунктах **Checked** автоматически сбросится в **false.**
Enabled	Доступ к пункту меню
Visible	Видимость пункты меню
Bitmap	Изображение для пункта меню
ImageIndex	Индекс картинки, расположенной в компоненте ImageList (хранилище рисунков). При этом в свойстве Images компонента MainMenu необходимо указать имя компонента ImageList.

Контекстное меню – компонент PopupMenu

Контекстное меню также формируется с помощью конструктора меню.

У компонента, который должен иметь контекстное меню, необходимо установить свойство **PopupMenu** равным имени компонента PopupMenu.

Программное управление меню

При создании меню во время проектирования формируется свойство **Items.**

Например,

1. **MainMenu1.Items[0].Caption** — надпись головного раздела первого меню**.**
2. **MainMenu1.Items[0].Items[3].Caption** — надпись четвертого (индексы начинаются с 0) раздела первого меню.
3. **MainMenu1.Items[0].Items[3].Items[1].Caption** — надпись второго раздела подменю, относящегося к четвертому раздела первого меню.

Свойство **Count** — число разделов.

<u>Пример №1</u>: отображение в компоненте **Memo1** надписей всех головных разделов меню **MainMenu1**

```
for i:=0 to MainMenu1.Items.Count - 1 do
Memo1.Lines.Add(MainMenu1.Items[i].Caption);
```

Пример №2: обеспечит отображение в **Memo1** надписей каждого раздела раскрывающегося меню в виде строк: «имя_головного_раздела->имя_раздела»

```
for i:=0 to MainMenu1.Items.Count - 1 do
begin
  for j:=0 to MainMenu1.Items[i].Count - 1 do
      memo1.Lines.Add(MainMenu1.Items[i].Caption+'->'+
      MainMenu1.Items[i].Items[j].Caption);
end;
```

Свойство **MenuIndex** объекта типа **TMenuItem** определяет индекс раздела в меню. Его изменение приводит к перемещению раздела в меню.

Очистка меню и удаление всех его элементов	метод **Clear**.
Удаление раздела с индексом Index	Delete(Index: Integer);
Добавление пункта Item в меню	Add(Item: TMenuItem);
Вставка пункта Item в позицию Index	Insert(Index: Integer; Item:TMenuItem);
Вставка разделителя перед элементом AItem	function **InsertNewLineBefore**(AItem: TMenuItem): Integer;
Вставка разделителя после элемента AItem	Function **InsertNewLineAfter**(AItem: TMenuItem): Integer;

Панели и компоненты внешнего оформления

Панели являются контейнерами, служащими для объединения других управляющих элементов. Они могут выполнять как чисто декоративные функции, зрительно объединяя компоненты, связанные друг с другом по назначению, так и функции управления, организуя совместную работу своих дочерних компонентов.

Компонент	Страница	Описание
GroupBox	Standart	Является контейнером, объединяющим группу связанных компонентов управления.
Panel	Standart	
ScrollBox	Additional	Используется для создания контейнера отображения с прокруткой.
Splitter	Additional	Разделитель панелей. Используется для создания панелей с изменяемыми размерами.
ControlBar	Additional	Контейнер инструментальных панелей
TabControl	Win32	Панель с закладками
PageControl	Win32	Позволяет создавать страницы управляемые закладками - многостраничное окно
StatusBar	Win32	Полоса состояния приложения
ToolBar	Win32	Инструментальная панель, с помощью контекстного меню которой можно создавать кнопки.

Панель Panel. Свойства

BevelInner	Стиль внутренней части панели
BevelOuter	Стиль внешней части панели
BevelWidth	Ширина внешней части панели
BorderStyle	Стиль бордюра
BorderWidth	Ширина бордюра

Компонент Splitter

Установите на форму две панели и компонент Splitter. У первой панели установите Align=alLeft, у Splitter свойство Align=AlLeft, у правой панели Align=AlClient. В результате с помощью компонента Splitter можно будет изменять размеры панелей.

Свойства:

Beveled	вид разделителя
MinSize	устанавливает минимальный размер панелей по обе стороны от разделителя

Компонент PageControl

С помощью команды **NewPage** контекстного меню компонента можно добавлять новые страницы. Каждая создаваемая страница - это отдельная панель, на которую можно устанавливать различные компоненты.

Свойства страницы:

Name	Имя страницы
Caption	Надпись на ярлыке страницы
PageIndex	Индекс страницы
ImageIndex	Индекс изображения на ярлыке страницы

Свойства PageControl

Style	Стиль отображения компонента: • tsTabs - закладки • tsButtons - кнопки • tsFlatButtons – плоские кнопки
MultiLine	=true, то закладки будут отображаться в несколько рядов, если все они не помещаются в один ряд
TabPosition	Место расположения закладок: • tpBottom - снизу • tbLeft - слева • tpRight - справа • tpTop – сверху
ActivePage	Имя активной страницы
PageCount	Количество страниц

Компонент TabControl в отличие от компонента PageControl представляет собой одну страницу с несколькими закладками.

Компоненты ControlBar и ToolBar

Плавающие панели

1. Установите на форму компонент ControlBar и на него 4 компонента ToolBar (стр. Win32).
2. У контейнера **ControlBar** свойство **AutoDrag**=**true.**
3. У панелей **ToolBar** свойства **DragKind = dkDock.**
4. У компонента **ControlBar** установите свойство **DockSite=true**, в результате плавающую панель контейнер сможет принять назад.
5. У компонента **ControlBar** установлено свойство **AutoSize=true,** чтобы он мог изменять свои размеры при перемещении внутри него панелей **ToolBar.**

На панели ToolBar можно устанавливать обычные кнопки, но есть более простой способ создания кнопок на панели. Для этого необходимо выполнить команду контекстного меню New Button, в результате на панели появится кнопка Toolbutton.

Свойства кнопки ToolButton

Style	Определяет вид и поведение кнопки • tbsButton – кнопка • tbsCheck – после щелчка пользователя на кнопке она остается нажатой, повторный щелчок возвращает ее исходное состояние. При этом если ряд кнопок имеет свойство Grouped=true, то эти кнопки образуют группу, из которой только одна кнопка может находиться в нажатом состоянии. • tbsDropDown – кнопка в виде выпадающего списка (выпадающее меню) • tbsSeparator – разделитель • tbsDivider – разделитель в виде вертикальной линии.
ImageIndex	Индекс изображения
Wrap	=true, то все кнопки расположенные после данной переносятся в другой ряд

УРОК №15.2. ХРАНЕНИЕ ИНФОРМАЦИИ В РЕЕСТРЕ. РАБОТА С INI-ФАЙЛАМИ

Цель:

Образовательная: рассмотреть компоненты и их свойства, позволяющие пользователю создавать и настраивать меню и инструментальные панели, переставлять их элементы, добавлять и удалять разделы и быстрые кнопки.

Воспитательная: продолжить формирование у учащихся основ научного мировоззрения. Воспитание любви к профессии и предмету

Развивающая: развитие познавательных способностей (мышления, воображения и т.д.).

Тип занятия: урок формирования новых знаний

Форма организации учебного процесса: лекция

ХОД УРОКА

1. **Организационный момент** (требования к уроку, план работы)
2. **Изложение нового материала.**
3. **Подведение итогов урока. Обобщение знаний.**
4. **Домашнее задание.** Повторить изученный материал.

КОНСПЕКТ ПОД ЗАПИСЬ

Системный реестр и файлы INI

Вся информация, которая связана с регистрацией приложения, файлов, форматов и т.п., должна заноситься в системный реестр.

Реестр (Registry) — это база данных для хранения информации о системной конфигурации аппаратуры, о Windows и о приложениях Windows.

Многие программы кроме этого используют файлы настройки INI, в которых хранится вся информация.

Все главное, связанное с размещением программ на дисках, с форматами, с взаимодействием программ друг с другом и системой, действительно должно храниться в реестре. Но забивать реестр второстепенными данными, необходимыми только для одной из сотен или тысяч программ, установленных на компьютере, вряд ли целесообразно. Логичнее хранить это именно в файлах INI.

Реестр

Реестр имеет иерархическую организацию, которая, содержит много уровней ключей, субключей и параметров. Информация хранится в виде иерархического дерева, каждый узел которого называется ключом. Ключ может содержать субключи и значения параметров.

Реестр делит все свои данные на две категории:

- характеризующие компьютер (все, связанное с техническими средствами, а также с установленными приложениями и их конфигурацией)
- характеризующие пользователя (установки по умолчанию для экрана, пользовательские конфигурации, информацию о выбранных пользователем принтерах, установки сети).

Все субключи относятся к пяти основным ключам реестра. Три из них:

определяют характеристики компьютера	**Hkey_Local_Machine**	Информация о компьютере, включая конфигурацию установленной аппаратуры и программного обеспечения
	Hkey_Current_Config	Информация о текущем оборудовании
	Hkey_Classes_Root	Информация об OLE, Drag&Drop, клавишах быстрого доступа и пользовательском интерфейсе
определяют характеристики пользователя	**Hkey_Users**	Информация о пользователях, включая установки экрана и приложений
	Hkey_Current_User	Информация о пользователе, зарегистрированном в данный момент

Если ваше приложение предусматривает возможность последующего удаления его с компьютера средствами Windows, то программа установки этого приложения должна создать в ключе «HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Uninstall\» свой подключ и в нем два параметра.

1. **DisplayName** — имя, под которым ваша программа будет видна пользователю в окне Установка и удаление программ программы «Панель управления».
2. **UninstallString -** это командная строка с полным путем к вашей программе удаления и всеми необходимыми опциями.

Работа с реестром, установка и удаление программ

Для использования в программе объекта класса **TRegistry** для работы с реестром надо подключить модуль

```
uses registry;
```

Все ключи в объекте класса **TRegistry** создаются как субключи определенного корневого ключа, записанного в свойстве **RootKey.** По умолчанию **RootKey** = **HKEY_CURRENT_USER.** В каждый момент объект типа **TRegistry** имеет доступ только к одному текущему ключу в иерархии, начинающейся с ключа **RootKey.** Текущий ключ определяется свойством только для чтения **CurrentKey.** Но это значение вам ничего не скажет — это просто некоторое целое значение. А вот свойство **CurrentPath** (тоже только для чтения) содержит строку, включающую имя текущего ключа и путь к нему по дереву.

Изменить текущий ключ	OpenKey(const Key: String; CanCreate: Boolean): Boolean Этот метод открывает ключ Key, делая его текущим для объекта. Параметр **Key** — строка полного пути по дереву ключей к открываемому ключу. Если Key — пустая строка, то текущим делается корневой ключ,

	указанный свойством **RootKey.** Параметр **CanCreate** указывает, должен ли создаваться ключ **Key,** если его нет в реестре. Ключ открывается или создается с доступом **KEY_ALL_ACCESS.** Создаваемый ключ сохраняется в реестре при последующих запусках системы.
Запись значений параметров в ключ	Методы **WriteInteger, WriteFloat, WriteBool, WriteString** и др Все они заносят значение **Value** в параметр с именем **Name.** procedure WriteInteger (const Name: string; Value: Integer); procedure WriteString(const Name, Value: String);

Чтение значений параметров из ключа	методы **ReadInteger, ReadFloat, ReadBool, ReadString** и др.
	function ReadInteger(const Name: String): Integer;
	function ReadString(const Name: String): String;

Пример: программа, иллюстрирующая установку и удаление программы.

Перенесите на форму две кнопки и напишите на них «Установка» и «Удаление». Пусть при нажатии пользователем этой кнопки данная программа будет регистрироваться в Windows. При этом кроме ключа и параметров для удаления, пусть создается ключ «HKEY_LOCAL_MACHINE\Software\Мой проект», в котором в качестве параметров заносится имя и путь программы и координаты левого верхнего угла ее окна.

А запуск этой же программы с опцией "-u" в командной строке должен удалять программу с компьютера. Правда, саму программу мы удалять не будем. Ограничимся только удалением из реестра ее ключей.

```
uses registry;

var Reg : TRegistry;

procedure TForm1.FormCreate(Sender: TObject);
begin
 // Создание объекта Reg типа TRegistry
 Reg := TRegistry.Create;
 Reg.RootKey:=HKEY_LOCAL_MACHINE;
   if Reg.KeyExists('\Software\Мой проект')
        then begin
           Reg.OpenKey('\Software\Мой проект',true);
           Left := Reg.ReadInteger('Left');
           Top := Reg.ReadInteger('Top');
        end;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
//открывается ключ
  Reg.OpenKey('\Software\Мой проект',true);
//заносится два параметра Left и Top, в которые записываются текущие координаты левого верхнего угла приложения
  Reg.WriteInteger('Left', Left);
  Reg.WriteInteger('Top', Top);
  // Удаление из памяти объекта Reg
  Reg.Free;
end;

//Кнопка установки программы
procedure TForm1.Button1Click(Sender: TObject);
begin
  //создает ключ
 Reg.OpenKey('\Software\Microsoft\Windows\CurrentVersion\Uninstall\Мой проект',true);
  //в ключе создается параметр DisplayName и заносится полное имя программы
 Reg.WriteString('DisplayName','Мой прекрасный проект');
  //создается параметр UninstallString с польным именем программы
```

```
 Reg.WriteString('UninstallString',ParamStr(0));
  //создается второй ключ
 Reg.OpenKey('\Software\Мой проект',true);
  //заносится параметр
 Reg.WriteString('Программа',ParamStr(0));
end;

//Кнопка удаления программы
procedure TForm1.Button2Click(Sender: TObject);
begin
  // Удаление ключей из реестра
  Reg.DeleteKey('\Software\Microsoft\Windows\CurrentVersion\Uninstall\Мой проект');
  Reg.DeleteKey('\Software\Мой проект');
  ShowMessage('Удаление завершено');
  Application.Terminate;
end;
```

Работа с файлами INI

Файлы настройки INI - это текстовые файлы с расширением *.ini,* предназначенные для хранения информации о настройках различных приложений. Информация в файле логически группируется в разделы, каждый из которых начинается оператором заголовка, заключенным в квадратные скобки. Например, **[Desktop].** В строках, следующих за заголовком, содержится информация, относящаяся к данному разделу, в форме: <ключ>=<значение>

Файлы INI, как правило, хранятся в каталоге Windows, который можно найти с помощью функции **GetWindowsDirectory.**

Компонент TIniFile

В Delphi работу с файлами **INI** проще всего осуществлять с помощью создания в приложении объекта типа **TIniFile.** Этот тип описан в модуле *inifiles,* который надо подключать к приложению оператором **uses**.

Создается объект типа **TIniFile**	методом **Create(<имя файла>),** в который передается имя файла **.ini,** с которым он связывается. Файл должен существовать до обращения к методу **Create.**
Запись значений ключей	**WriteString, WriteInteger, WriteFloat, WriteBool** и др. procedure WriteString(const Section,Ident, Value: string); procedure WriteInteger (const Section, Ident: string; Value: Longint); • **Section** — раздел файла, • **Ident** — ключ этого раздела, • **Value** — значение ключа.
Чтение значений ключей	**ReadString, ReadInteger, ReadFloat, ReadBool** и др. function ReadString(const Section,Ident, Default: string): string; function ReadInteger(const Section, Ident: string; Default: Longint):Longint Методы возвращают значение ключа **Ident** раздела **Section.** Параметр **Default** определяет значение, возвращаемое в случае, если в файле не указано значение соответствующего ключа.
Проверка наличия значения ключа	Метод **ValueExists,** в который передаются имена раздела и ключа.
Удаление из файла значения указанного ключа в указанном разделе	Метод **DeleteKey**
Проверка наличия в файле необходимого раздела	Метод **SectionExists**
удаление из файла указанного раздела вместе со всеми его ключами.	Метод **EraseSection**

Пример сохранения позиции формы (Top и Left) и чтения из Ini-файла данных по требованию.

```
unit Unit1;

…
uses inifiles;

var
  Form1: TForm1;
  sIniFile: TIniFile;

implementation
{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
sIniFile := TIniFile.Create(ChangeFileExt(ParamStr(0),'.ini'));
sIniFile.WriteString('Position', 'Top', inttostr(form1.top));
sIniFile.WriteString('Position', 'Left', inttostr(form1.Left));
sIniFile.Free;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
sIniFile := TIniFile.Create(ChangeFileExt(ParamStr(0),'.ini'));
form1.Top:=strtoint(sIniFile.ReadString('Position', 'Top', '0'));
form1.Left:=strtoint(sIniFile.ReadString('Position', 'Left', '0'));
sIniFile.Free;
end;
```

Задание: изменить пример таким образом, чтобы сохранение позиции формы и ее размеров происходило в момент закрытия приложения и чтение данных из ini-файла, а также установка параметров происходила в момент запуска программы.

РАЗДЕЛ №16. РАБОТА С ФАЙЛАМИ И КАТАЛОГАМИ

УРОК №16.1 РАБОТА С ФАЙЛАМИ И КАТАЛОГАМИ

Цель:

Образовательная: рассмотреть как организуются диалоги выбора папок, как осуществляется поиск файлов по дереву каталогов, как организовать поиск файлов с помощью программы Проводник, как манипулировать с файлами и каталогами с помощью функции ShFileOperation.

Воспитательная: продолжить формирование у учащихся основ научного мировоззрения. Воспитание любви к профессии и предмету

Развивающая: развитие познавательных способностей (мышления, воображения и т.д.).

Тип занятия: урок формирования новых знаний

Форма организации учебного процесса: лекция

ХОД УРОКА

1. **Организационный момент** (требования к уроку, план работы)
2. **Изложение нового материала.**
3. **Задание 1:** Составить программу, формирующую список файлов расположенных в указанной папке, и всех папках, вложенных в нее.
4. **Задание 2:** Составить программу, которая позволяет скопировать или переместить указанный пользователем файл в указанную папку, либо удалить указанный файл.
5. **Подведение итогов урока. Обобщение знаний.**
6. **Домашнее задание.** Повторить изученный материал.

КОНСПЕКТ ПОД ЗАПИСЬ

Диалоги выбора папок

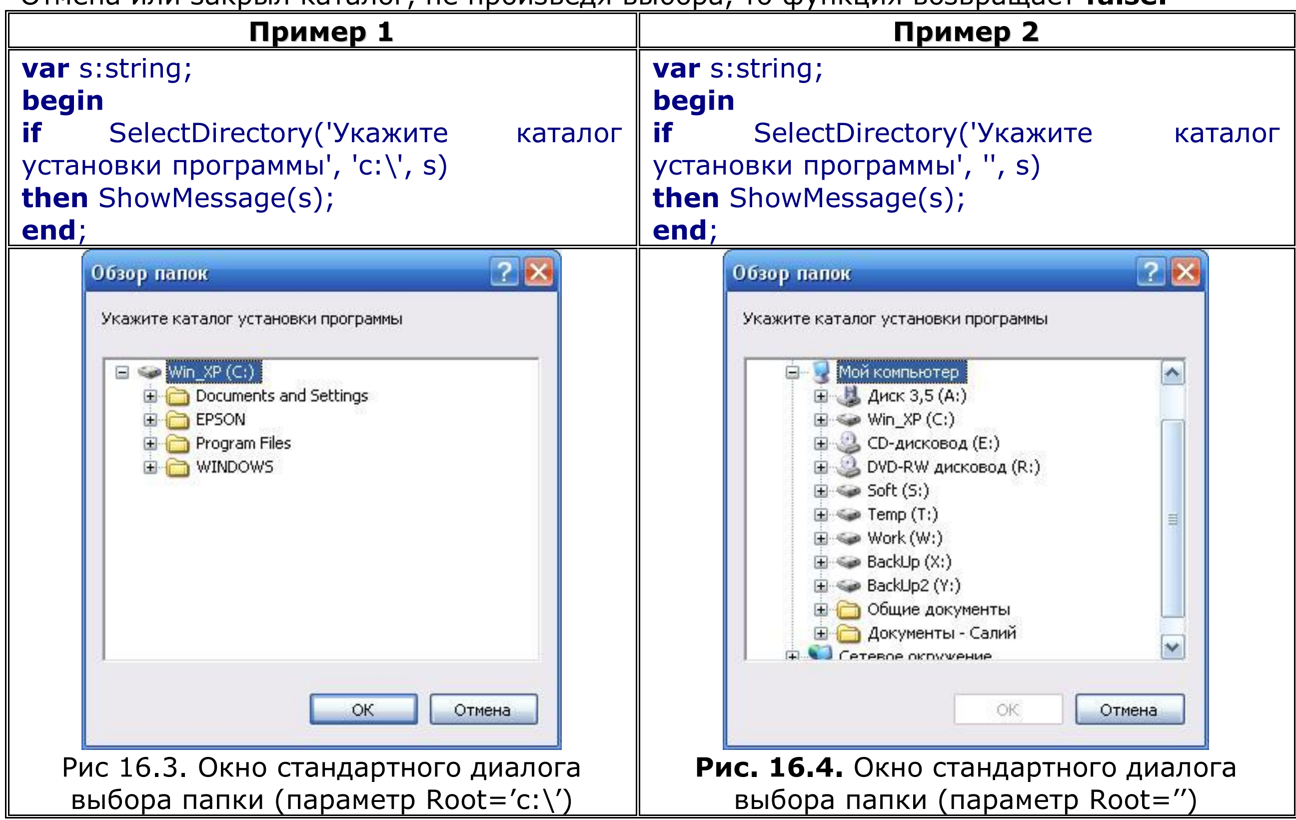
Нередко в приложениях надо предоставить пользователю возможность выбрать в стандартном диалоге папку. Для ее использования подключите модуль *FileCtrl*.

Функция SelectDirectory предоставляет пользователю возможность вызвать стандартный диалог Windows и, работая с ним, выбрать каталог.

Первая форма функции: вызывает стандартный диалог Windows для поиска каталога.

Параметры:

- **Caption** содержит строку, отображаемую в диалоге как указание пользователю.
- **Root** задает корневой каталог, внутри которого пользователь может выбирать подкаталоги (рис.16.3). За пределы каталога **Root** пользователь выйти не может. Если указать вместо **Root** пустую строку или отсутствующий на компьютере каталог, то в диалоговом окне отобразится дерево всех папок (рис. 16.4) и пользователь имеет возможность выбрать на любом диске любой каталог.
- Выходной параметр **Directory** содержит результат выбора пользователя. Функция возвращает **true,** если пользователь выбрал каталог и нажал ОК. Если пользователь нажал Отмена или закрыл каталог, не произведя выбора, то функция возвращает **false.**



Пример 1

```
var s:string;
begin
if SelectDirectory('Укажите каталог установки программы', 'c:\', s)
then ShowMessage(s);
end;
```

Пример 2

```
var s:string;
begin
if SelectDirectory('Укажите каталог установки программы', '', s)
then ShowMessage(s);
end;
```


Рис 16.3. Окно стандартного диалога выбора папки (параметр Root='c:\')

Рис. 16.4. Окно стандартного диалога выбора папки (параметр Root='')

Вторая форма функции: более гибкий диалог (см. рис. 16.5). Возвращаемое значение, как и в первой форме функции, указывает, выбрал ли пользователь каталог.

Параметры:

- **Directory** содержит выбранный пользователем каталог. Если перед вызовом **SelectDirectory** задано начальное значение **Directory,** то именно этот каталог будет раскрыт в окне диалога в первый момент времени.
- **HelpCtx** является ссылкой на контекстную справку, содержащую подсказку по действиям пользователя.
- **Options** является множеством следующих опций:

sdAllowCreate	В диалоговом окне отображается окошко редактирования Directory Name (см. рис. 16.5), в котором пользователь может написать каталог, который отсутствует. Эта опция не создает сам каталог. Это задача приложения, которое прочтет имя каталога и при необходимости создаст его
sdPerformCreate	Применяется только в сочетании с **sdAllowCreate** и обеспечивает создание каталога, если указанный пользователем каталог отсутствует
sdPrompt	Применяется только в сочетании с **sdAllowCreate.** Если пользователь указал несуществующий каталог, ему предлагается вопрос, надо ли его

	создавать.

Если множество **Options** пустое, то пользователь не может указать каталог, которого не существует.

Пример:

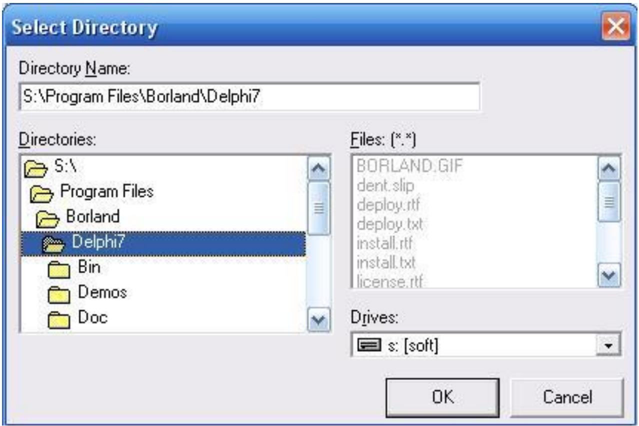


Рис. 16.5. Диалог при выборе папки вторым вариантом функции SelectDirectory

```
var Dir:string;
begin
Dir:='S:\Program Files\Borland\Delphi7';
if SelectDirectory(Dir, [sdAllowCreate, sdPerformCreate,  sdPrompt],0)
then ShowMessage(dir);
end;
```

Вызываемое диалоговое окно не будет иметь кнопки Help, поскольку идентификатор контекстной справки задан равным нулю.

Поиск файлов по дереву каталогов

Для поиска файлов, удовлетворяющих указанному шаблону и имеющих указанные атрибуты, используются функции **FindFirst, FindNext** и процедура **FindClose.** Они объявлены в модуле *SysUtils* следующим образом:

```
function FindFirst(const Path: string; Attr:  Integer; var F: TSearchRec): Integer;
function FindNext(var F: TSearchRec): Integer;
procedure FindClose(var F: TSearchRec);
```

Все они используют для работы запись типа **TSearchRec:**

```
type
TSearchRec = record
	Time: Integer; // Время создания файла
	Size: Integer; // Размер файла в байтах
	Attr: Integer; // Атрибуты файла
	Name: TFileName; // Имя файла
	ExcludeAttr: Integer; // Для внутреннего использования
	FindHandle: THandle; // Для внутреннего использования
	FindData: TWin32FindData; // Дополнительные сведения
end;
```

Начинается поиск вызовом функции **FindFirst.** Параметр **Path** определяет путь и шаблон искомых файлов. Например, если **Path** — "c:\test*.*", то будут искаться все файлы в каталоге *c:\test*. А если **Path** = "c:\test*.tmp", то в каталоге *c:\test* будут искаться файлы с расширением *tmp.* Параметр **Attr** определяет типы файлов, которые будут включены в поиск дополнительно к обычным файлам. Параметр может включать следующие флаги:

faReadOnly	файл только для чтения
faHidden	невидимый файл
faSysFile	системный файл
faVolumeID	идентификатор диска
faDirectory	каталог
faArchive	архивный файл
faAnyFile	любой файл

Флаги могут объединяться. Например, если **Attr = faReadOnly + faHidden,** то к обычным файлам будут добавлены невидимые файлы и файлы только для чтения. Если **Attr = faAnyFile,** то будут искаться все файлы и каталоги. А если **Attr** = 0, то будут искаться только обычные файлы.

Функция **FindFirst** возвращает 0, если файл, удовлетворяющий условиям поиска, найден. В противном случае возвращается код ошибки Windows.

Если файл найден, то сведения о нем заносятся в поля записи типа **TSearchRec**, определяемой параметром F. В поле **Name** этой записи можно найти имя файла вместе с его расширением. Например, «Test.txt». В поле **Time** заносится дата и время создания файла. Это время в формате DOS. Его можно перевести во время типа **TDateTime** функцией **FileDateToDateTime,** а если требуется перевести его в строку, то к значению типа **TDateTime** можно затем применить функцию **DateTimeToStr.** Таким образом, выражение вида

```
DateTimeToStr (FileDateToDateTime (F.Time));
```

вернет дату и время создания файла в виде строки.

Поле **Attr** записи F содержит атрибуты файла. Определить тип найденного файла можно комбинированием соответствующего флага с полем **Attr** по операции **and.** Если файл имеет данный атрибут, то результат этой операции будет больше 0. Например, чтобы узнать, является ли найденный файл системным, надо записать выражение

```
(F.Attr and faSysFile > 0)
```

Это выражение вернет **true,** если файл системный.

Таким образом, вызов **FindFirst** может найти первый файл, удовлетворяющий условиям поиска, или убедиться, что ни одного такого файла нет. Продолжение поиска осуществляется вызовом функции **FindNext** и передачей в нее в качестве параметра F той же записи, которая передавалась в **FindFirst.** Если **FindNext** вернет 0, значит, нашелся еще один файл, удовлетворяющий условиям поиска. Информация об этом файле занесется в ту же запись F, после чего можно снова вызывать **FindNext** для поиска следующего файла. Если **FindNext** вернет ненулевое значение, значит больше нет файлов, удовлетворяющих условиям поиска. В этом случае надо вызвать процедуру **FindClose** с тем же параметром F. Эта процедура завершает поиск и освобождает ресурсы, выделенные для него.

Пример: поиск всех файлов в указанной папке и во всех вложенных в нее папках.

```
procedure ViewFind(dir:string);
var SR: TSearchRec;
   ires: integer;
begin
 ChDir(dir);
 ires := FindFirst('*.*', faAnyFile, SR);
 while ires = 0 do
 begin
  if (SR.Name <> '.') and (SR.Name <> '..')
    then begin
      if (SR.Attr = faDirectory) then Viewfind(SR.Name)
      else form1.memo1.Lines.add(SR.Name);
    end;
  ires := FindNext(SR);
 end;
 FindClose(SR);
 ChDir('..');
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
viewfind('W:\1\');
end;
```

Поиск файлов с помощью программы Проводник

Можно дать возможность пользователю воспользоваться диалогом поиска файлов и папок, имеющимся в программе Проводник. Вызывается этот диалог с помощью технологии **DDE**. Вы можете создать в приложении временный объект класса **TDDEClientConv,** связать его с программой Проводник *(explorer.exe)* как с сервером **DDE** и выполнить требуемый макрос. Все это может выглядеть следующим образом:

```
uses DdeMan;
…
with TDDEClientConv.Create(Self) do
begin
      ServiceApplication := 'explorer.exe';
      SetLink('Folders', 'AppProperties');
      OpenLink;
      ExecuteMacro('[FindFolder(, C:\T)]',  False);
      CloseLink;
      Free;
end;
```

В данном примере в качестве начального каталога поиска задается каталог *C:\T.*

Манипуляции с файлами и каталогами с помощью функции ShFileOperation

Очень широкие возможности по копированию, перемещению, переименованию и удалению файлов и каталогов предоставляет функция API Windows **ShFileOperation.** В модуле *ShelLAPI* она объявлена следующим образом:

```
function SHFileOperation(const lpFileOp: TSHFileOpStruct): Integer; stdcall;
```

В функцию передается единственный параметр **lpFileOp** — указатель на запись, содержащую всю информацию о требуемой операции с файлами и каталогами. Тип этой записи объявлен следующим образом:

```
TSHFileOpStruct = TSHFileOpStructA;
TSHFileOpStructA = _SHFILEOPSTRUCTA;

_SHFILEOPSTRUCTA= packed record
     Wnd:  HWND;
     wFunc: UINT;
     pFrom: PAnsiChar;
     pTo: PAnsiChar;
     fFlags: FILEOP_FLAGS;
     fAnyOperationsAborted: BOOL;
     hNameMappings: Pointer;
     lpszProgressTitle: PAnsiChar;
end;
```

Основное поле, определяющее производимую операцию — **wFunc.** В это поле можно записать одно из следующих значений:

FO_COPY	Копирование файлов, указанных в поле **pFrom,** в файл или каталог, указанный полем **pTo.**
FO_DELETE	Удаление файлов, указанных в поле **pFrom.** Поле **pTo** игнорируется.
FO_MOVE	Перемещение файлов, указанных в поле **pFrom,** в файл или каталог, указанный полем **pTo.**
FO_RENAME	Переименование файлов, указанных в поле **pFrom.**

Поле **pFrom** указывает буфер, содержащий имя или имена файлов, с которыми производится заданная операция. Отдельные имена должны разделяться нулевыми символами. Завершаться список имен должен двумя нулевыми символами.

Поле **pTo** указывает буфер, содержащий имя файла или каталога, являющегося приемником. Если в поле **fFlags** указан флаг **FOF_MULTIDESTFILES,** то буфер может содержать список имен файлов. Отдельные имена должны разделяться нулевыми символами. Завершаться список имен должен двумя нулевыми символами.

Поле **fFlags** может содержать комбинацию следующих флагов:

FOF_ALLOWUNDO	Сохранить по возможности информацию для отмены операции. Например, при удалении файлов они перемещаются в корзину, из которой их потом можно восстановить.
FOF_CONFIRMMOUSE	Не реализован.
FOF_FILESONLY	Осуществлять операцию только при задании шаблона **"* *"**
FOF_ALLOWUNDO	Сохранить по возможности информацию для отмены операции. Например, при удалении файлов они перемещаются в корзину, из которой их потом можно восстановить.

FOF_MULTIDESTFILES	Указывает, что поле **pTo** содержит не папку, куда надо доставить все файлы-источники, а множество имен приемников — по одному на каждый файл-источник.
FOF_NOCONFIRMATION	Отвечать «да для всех» на все вопросы, которые могут задавать диалоги в процессе выполнения операции: об удалении файлов, о создании новых папок и т.д.
FOF_NOCONFIRMMKDIR	Не делать запрос о создании нового каталога, если требуемый каталог отсутствует.
FOF_RENAMEONCOLLISION	Если при операциях копирования, переноса или переименования файлы-приемники уже существуют, давать им имена типа «Копия ...», не запрашивая у пользователя подтверждения.
FOF_SILENT	Не показывать окно с отображением хода процесса.
FOF_SIMPLEPROGRESS	Показывать окно с отображением хода процесса, но не отображать в нем имена файлов.
FOF_WANTMAPPINGHANDLE	Заполнить поле **hNameMappings.** Указанный в нем дескриптор должен быть в дальнейшем освобожден функцией SHFreeNameMappings.

В поле **fAnyOperationsAborted** записи **TSHFileOpStruct** функция **ShFileOperation** возвращает **true,** если выполнение операции было прервано пользователем. Поле **hNameMappings** содержит указатель на массив записей, содержащий прежние и новые полные имена всех файлов, над которыми проводились операции. Поле **lpszProgressTitle** содержит указатель текста, помещается в окно отображения процесса при включенном флаге **FOF_SIMPLEPROGRESS.**

Пример копирования: копирует из файла, указанного в Edit1 в каталог указанный в Edit2.

```
Uses ShellApi;
…
procedure TForm1.Button1Click(Sender: TObject);
var s:TSHFileOpStruct;
begin
with s do
begin
  Wnd := 0;
  pFrom := PChar(Edit1.Text);
  pTo := PChar(Edit2.Text);
   wFunc := FO_COPY;
   fFlags:= FOF_ALLOWUNDO;
end;
SHFileOperation(S);
end;
```

В **pFrom** может быть задан шаблон. Например, "c:\test*.txt" или "c:\test*.*". Первый из них означает копирование всех файлов с расширением .txt из папки *c:\test,* а второй — копирование всех файлов папки. При задании в **pFrom** шаблона подразумевается, что в **pTo** задан каталог.

В pFrom можно задавать имена нескольких файлов или нескольких шаблонов, разделяя их нулевыми символами. Например, текст "*.txt" #0 "*.exe" #0 #0, занесенный в **pFrom,** обеспечит копирование из текущего каталога всех файлов с расширениями .txt и .exe в каталог, указанный в **pTo.**

С помощью шаблона можно скопировать все файлы указанного каталога. Однако в **pFrom** можно задать не файл, не шаблон, а имя каталога. Например, "c:\test".В этом случае и в **pTo** тоже должен быть задан каталог. Например, "d:\t". Тогда каталог, указанный в **pFrom,** будет скопирован как вложенный в каталог, указанный в **pTo.**

Перемещение файлов (режим **FO_MOVE)** осуществляется аналогично. Отличие только в том, что файлы-источники удаляются.

При удаление файлов (режим FO_DELETE) содержимое поля pTo безразлично, а в поле pFrom можно, как и раньше, задавать имена папок, файлов и шаблоны.

И при перемещении, и при удалении можно устанавливать флаг **FOF_ALLOWUNDO.** Это

обеспечит удаление файлов в корзину, так что в дальнейшем пользователь сможет отменить это удаление, если оно сделано по ошибке.

Переименование файлов осуществляется в режиме FO_RENAME. Шаблоны при этом не используются. Если в поле **pFrom** записано несколько файлов, то в поле **pTo** должен быть записан аналогичный по числу элементов список новых имен. Пользуясь этой возможностью, несложно обеспечить и применение шаблонов при переименовании. Но это должно быть сделано внешним образом, чтобы представить шаблоны списками файлов.

Операции, выполняемые функцией **ShFileOperation,** сопровождаются по умолчанию стандартными диалоговыми окнами Windows, сообщающими о проводимой операции, именах файлов, времени, оставшемся до окончания операции, отображающими диаграмму процесса. Если установить флаг FOF_SIMPLEPROGRESS, то эти окна не будут отображать имен файлов. Вместо этого будет отображена строка, указанная в поле **lpszProgressTitle,** так что вы скроете от пользователя детали проводимой операции. А если установлен флаг **FOF_SILENT,** то окна отображения хода процесса вообще не появляются. Работать в таком режиме имеет смысл, если вы не хотите дать возможность пользователю прервать операцию. К тому же, при операциях, производимых над большим числом не очень больших файлов, окно отображения хода процесса очень заметно увеличивает время выполнения операции.

РАЗДЕЛ №17. DELPHI И МЕХАНИЗМЫ WINDOWS

УРОК №17.1 SystemTray. Шрифты приложения

Цель:

Образовательная: рассмотреть приемы работы с областью SystemTray, способы создания и удаления значка приложения в этой области; научить получать список установленных в системе шрифтов и связывать его с компонентом RichEdit. Рассмотреть процедуру создания логического пользовательского шрифта и научиться выводить текст под различными углами.

Воспитательная: продолжить формирование у учащихся основ научного мировоззрения. Воспитание любви к профессии и предмету

Развивающая: развитие познавательных способностей (мышления, воображения и т.д.).

Тип занятия: урок формирования новых знаний

Форма организации учебного процесса: лекция

ХОД УРОКА

1. **Организационный момент** (требования к уроку, план работы)
2. **Изложение нового материала.**
3. **Задание:** Составить программу, которая создает значок в области System Tray и позволяет показать приложение с помощью меню этого значка.
4. **Подведение итогов урока. Обобщение знаний.**
5. **Домашнее задание.** Повторить изученный материал.

КОНСПЕКТ ПОД ЗАПИСЬ

Пиктограммы в области System Tray

Область System Tray представляет собой часть полосы задач, которая размещается в правом нижнем углу экрана и содержит такие значки, как часы и ряд других. Она отличается от полосы задач тем, что в ней расположены не свернутые окна приложений, а только пиктограммы. В области System Tray целесообразно размещать пиктограммы, связанные с приложениями, которые постоянно должны выполняться в фоновом режиме.

Связь приложения с пиктограммой, размещенной в System Tray, осуществляется функцией **Shell_NotifyIcon,** объявленной в модуле *ShellAPI* следующим образом:

```
function Shell_NotifyIcon(dwMessage: DWORD; lpData: PNotifylconData): BOOL; stdcall;
```

Параметр dwMessage указывает выполняемую операцию и может принимать значения:

NIM_ADD	добавить пиктограмму в область System Tray
NIM_DELETE	удалить пиктограмму из области System Tray
NIM_MODIFY	изменить изображение пиктограммы, текст ее ярлычка или идентификатор ее сообщения

Параметр **lpData** является указателем на запись типа **TNotifyIconData,** содержащую всю информацию о выполняемых операциях. Функция возвращает ненулевое значение при успешном выполнении и 0 в случае ошибки.

Тип **TNotifyIconData** имеет несколько вариантов, основной из которых объявлен следующим образом:

```
type _NOTIFYICONDATA = = _NOTIFYICONDATAA;
TNotifyIconDataA = _NOTIFYICONDATAA;
TNotifyIconData = TNotifyIconDataA;

_NOTIFYICONDATAA = record
        cbSize: DWORD;
        Wnd: HWND;
        uID: UINT;
        uFlags: UINT;
        uCallbackMessage: UINT;
        hIcon: HICON;
        szTip: array [0..63] of AnsiChar;
  end;
```

Параметры:

1. Поле cbSize указывает размер записи.
2. Поле Wnd определяет дескриптор окна: которое связано с пиктограммой и получает сообщения о перемещениях над ней мыши.
3. Поле uID содержит номер размещаемой пиктограммы. Одно приложение может разместить в System Tray несколько пиктограмм и ссылаться на них по этому номеру.
4. Поле uCallbackMessage задает идентификатор сообщения, которое будет посылаться при попадании курсора мыши в область пиктограммы.
5. Поле hIcon задает дескриптор пиктограммы.
6. Поле szTip содержит текст всплывающего ярлычка.
7. Поле uFlags включает флаги, указывающие, какие поля записи принимаются во внимание:

NIF_ICON	учитывается поле hIcon
NIF_MESSAGE	учитывается поле uCallbackMessage
NIF TIP	учитывается поле szTip

Если вы разместили пиктограмму в области System Tray, надо обеспечить в приложении обработку сообщений о манипуляциях мышью в области этой пиктограммы. Идентификатор этого сообщения вы задаете в поле uCallbackMessage записи типа TNotifyIconData. В поле wParam пришедшего сообщения вы можете прочитать номер пиктограммы, который ранее задавали в поле uID записи типа TNotifyIconData. А в поле lParam пришедшего сообщения вы можете прочитать идентификатор сообщения, полученного от мыши, например, WM_LBUTTONDBLCLK — двойной щелчок левой кнопкой.

Функция Shell_NotifyIcon позволяет управлять значком в области System Tray. Но надо еще принять меры к тому, чтобы приложение не отображалось в полосе задач. Иначе значок в области System Tray теряет смысл.

<u>Пример</u>: При выполнении программы в области System Tray размещается его пиктограмма, а окно приложения пользователю не видно и доступно только из всплывающего меню пиктограммы или после двойного щелчка на значке. На форме приложения расположен компонент всплывающего меню PopupMenu1 с тремя разделами: Развернуть приложение (имя раздела в программе — MRestore) — делает видимой главную форму, Закрыть приложение (имя MClose) — завершает выполнение приложения, и Удалить значок (имя MDelete) — удаляет значок из System Tray и одновременно делает видимой главную форму, так как иначе приложение после этого было бы невозможно закрыть.

Задана пиктограмма приложения. Для этого выполнена команда Project \ Options и на странице Application кнопкой Load Icon проведена загрузка пиктограммы. Это та пиктограмма, которая появится в области System Tray.

На форме размещены три кнопки. Кнопка Создать значок (имя BCreate) доступна только в случае, если пиктограмма удалена из области System Tray. Щелчок на этой кнопке размещает пиктограмму в System Tray. Кнопка Удалить значок (BDelete) удаляет значок из области System Tray. Кнопка Свернуть (BHide) делает форму невидимой. Кнопка доступна, только если в области System Tray размещена пиктограмма.

Код приложения

```
unit USysTray;

interface

…
const MyTrayIcon = WM_USER + 1;
type
  TForm1 = class(TForm)
  private
   procedure MTIcon(var a: TMessage); message MyTrayIcon;
  …
  end;

var
  Form1: TForm1;

implementation
{$R *.dfm}
uses ShellAPI;

var NID: TNotifyIconData;

procedure TForm1.FormCreate(Sender: TObject);
begin
 Application.ShowMainForm := false;
 with NID do
  begin
   Wnd := Handle;
   uFlags := NIF_ICON or NIF_MESSAGE or NIF_TIP;
   uCallbackMessage := MyTrayIcon;
   hIcon := Application.Icon.Handle;
   szTip := 'Мое приложение';
  end;
  Shell_NotifyIcon(NIM_ADD, @NID);
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  Shell_NotifyIcon(NIM_DELETE, @NID);
end;

procedure TForm1.MTIcon(var a: TMessage);
var P: TPoint;
begin
```

```
  case a.lParam of
   WM_LBUTTONDBLCLK: begin
                      Show;
                      SetForegroundWindow(Handle);
                     end;
   WM_RBUTTONDOWN:   begin
                      GetCursorPos(P);
                      PopupMenu1.Popup(P.X, P.Y);
                     end;
  end;
end;

procedure TForm1.MRestoreClick(Sender: TObject);
begin
 Show;
end;

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

procedure TForm1.BHideClick(Sender: TObject);
begin
 Visible := false;
end;
```

Пояснение к примеру

В интерфейсной части модуля объявлена константа **MyTrayIcon** — идентификатор вводимого в приложении сообщения Windows. В объявлении кода формы введено объявление обработчика этого сообщения — процедуры **MTIcon.** В разделе реализации оператор uses подключает модуль *ShellAPI,* содержащий объявление функции **Shell_NotifyIcon.** Введена также глобальная переменная **NID** типа **TNotifyIconData,** которая содержит информацию, необходимую для этой функции.

Процедура **FormCreate** — обработчик события формы **OnCreate.** Первый оператор этой процедуры делает форму невидимой. Затем заполняется запись **NID.** Указывается идентификатор сообщения, связанного с пиктограммой — **MyTrayIcon.** В качестве пиктограммы задается пиктограмма приложения — **Application.Icon.Handle.** Задается также текст ярлычка: «Мое приложение». После заполнения записи вызывается функция **Shell_NotifyIcon,** размещающая пиктограмму в области System Tray.

При завершении приложения в процедуре **FormDestroy** пиктограмма удаляется из области System Tray. Если этого не сделать, то она там так и останется до перезагрузки Windows или до реорганизации System Tray. Причем пиктограмма уже не будет связана с приложением и, значит, будет только вводить в заблуждение пользователя.

Процедура **MTIcon** является обработчиком сообщения MyTrayIcon, поступающего в приложение при любых манипуляциях с мышью в области пиктограммы. Параметр **lParam** этого сообщения указывает на соответствующее сообщение мыши. В данном случае при двойном щелчке левой кнопкой (сообщение **WM_LBUTTONDBLCLK)** выполняется метод **Show,** относящийся к данной форме, так что форма становится видимой. Но этого мало, так как если перед этим активным было какое-то другое окно, оно может перекрыть открывшееся окно формы и пользователь его не увидит. Поэтому к окну формы применяется функция **SetForegroundWindow,** активизирующая его и перемещающая на верх Z-последовательности, так что оно становится безусловно видимым. При нажатии правой клавиши мыши (сообщение **WM_RBUTTONDOWN)** пользователю показывается контекстное меню. Позиция меню привязывается к координатам курсора, полученным функцией **GetCursorPos**.

Шрифты

Нередко в приложении желательно обеспечить пользователю возможность изменять имя используемого шрифта, не вызывая стандартный диалог выбора шрифта. Например, в текстовом редакторе желательно иметь на инструментальной панели выпадающий список, содержащий имена всех шрифтов, установленных в Windows. В этом списке должен автоматически отображаться шрифт, использованный в текущей позиции текста окна

редактирования **RichEdit.** А при смене шрифта в выпадающем списке этот шрифт должен присваиваться выделенному тексту или текущей позиции в тексте.

Получить список установленных в Windows шрифтов проще всего с помощью свойства **Fonts** объекта **Screen.** Это свойство типа **TStrings** содержит список шрифтов, доступных на данном компьютере (свойство только для чтения). Так что описанный выше выпадающий список шрифтов можно реализовать следующим образом. В обработчик события **OnCreate** формы вставьте операторы:

```
ComboBox1.Items:= Screen.Fonts;
ComboBox1.ItemIndex := ComboBox1.Items.IndexOf(RichEdit1.DefAttributes.Name) ;
```

Эти операторы обеспечивают загрузку списка **ComboBox1** именами шрифтов и устанавливают индекс списка на шрифт, используемый в окне **RichEdit1** по умолчанию.

В обработчики событий **OnKeyUp** и **OnMouseUp** окна **RichEdit1** вставьте оператор

```
ComboBox1.ItemIndex := ComboBox1.Items.IndexOf(RichEdit1.SelAttributes.Name) ;
```

Это обеспечит отображение в списке текущего шрифта.

Для обеспечения изменения текущего шрифта в окне **RichEdit1** и возвращения фокуса этому окну, необходимо в обработчик события **OnCloseUp** списка **ComboBox1** внести следующие операторы:

```
RichEdit1.SelAttributes.Name := ComboBox1.Items[ComboBox1.ItemIndex];
RichEdit1.SetFocus;
```

Создание собственного шрифта

Создать логический шрифт можно функцией **CreateFont.** Этот шрифт может использоваться для вывода текста на канву любого устройства. Если в системе не зарегистрирован шрифт с заданными характеристиками, система подбирает шрифт, наиболее близкий к заданному.

Функция **CreateFont** объявлена в модуле *Windows* следующим образом:

```
function CreateFont(nHeight, nWidth, nEscapement, nOrientaion, fnWeight:  Integer; fdwItalic,
                  fdwUnderline, fdwStrikeOut, fdwCharSet, fdwOutputPrecision,
                  fdwClipPrecision, fdwQuality, fdwPitchAndFamily: DWORD; lpszFace:
                  PChar): HFONT; stdcall;
```

Параметры:

1. **nHeight** указывает высоту в логических единицах символов шрифта (если задается отрицательное значение) или ячеек, в которые вписываются символы (если задается положительное значение). Если задано значение 0, берется высота по умолчанию. Обычно целесообразно задавать отрицательное значение. Это то же значение, которое задается в свойстве **Height** класса **TFont.** И соотношение его с размером шрифта определяется так же, как в **TFont: -11** соответствует размеру 8, -13 соответствует размеру 10 и т.д. Если задано ненулевое значение, система подбирает наиболее крупный шрифт с высотой, не превышающей заданную.
2. **nWidth** указывает среднюю ширину символов. Если задано значение 0, система сама выбирает подходящую ширину символов.
3. **nEscapement** указывает в десятых долях градуса угол наклона надписей, которые будут выполняться данным шрифтом.
4. **nOrientaion** определяет угол наклона символов.
5. **fnWeight** определяет толщину символов. Значение этого параметра можно задавать в пределах от 0 до 1000. Значение 0 соответствует толщине по умолчанию. Другие значения можно задавать целыми числами или следующими константами:

Константа	Значение
FW_DONTCARE	0
FW_THIN	100
FW EXTRALIGHT	200
FW ULTRALIGHT	200
FW LIGHT	300
FW_NORMAL	400
FW REGULAR	400
FW MEDIUM	500

FW_SEMIBOLD	600
FW_DEMIBOLD	600
FW_BOLD	700
FW_EXTRABOLD	800
FW_ULTRABOLD	800
FW_HEAVY	900
FW_BLACK	900

6. Параметры **fdwItalic, fdwUnderline** и **fdwStrikeOut** управляют модификациями шрифта: курсив, подчеркнутый, зачеркнутый. Если параметр равен 0, соответствующая модификация выключена, при значении 1 — включена.
7. Параметр **fdwCharSet** задает множество символов. Обычно используется значение **DEFAULT_CHARSET** — множество символов по умолчанию. Можно также задавать значение **RUSSIAN_CHARSET,** если есть уверенность, что множество символов кириллицы имеется в требуемом шрифте. Для вывода текста, записанного символами MS DOS, следует задавать значение **OEM_CHARSET.**
8. Параметр **fdwOutputPrecision** определяет поведение системы при поиске шрифта, наиболее соответствующего заданному. Обычно используется одно из следующих значений этого параметра:

Значение	Описание
OUT_DEFAULT_PRECIS	Поведение по умолчанию.
OUT_DEVICE_PRECIS	Если имеется несколько шрифтов с одинаковым именем, выбирается шрифт устройства.
OUT_OUTLINE_PRECIS	Шрифт ищется среди шрифтов типа TrueType и других эскизных типов.
OUT_RASTER_PRECIS	Если имеется несколько шрифтов с одинаковым именем, выбирается растровый шрифт.
OUT_TT_ONLY_PRECIS	Идет поиск только шрифтов TrueType. Если нет установленных шрифтов TrueType, реализуется поведение по умолчанию.
OUT_TT_PRECIS	Если имеется несколько шрифтов с одинаковым именем, выбирается шрифт TrueType.

9. Параметр **fdwClipPrecision** определяет, как производится усечение символов, частично не помещающихся в заданную область. Обычно используется значение **CLIP_DEFAULT_PRECIS** — усечение по умолчанию.
10. Значение параметра **fdwQuality** обеспечивает критерий компромисса между качеством отображения символов и точностью соответствия шрифта заданным требованиям. При значении **DEFAULT_QUALITY** приоритет отдается точности соответствия заданным требованиям. При значении **PROOF_QUALITY** на первое место выдвигается качество отображения символов. Значение **DRAFT_QUALITY** — промежуточный вариант.
11. Параметр **fdwPitchAndFamily** определяет шаг размещения символов, расстояние между ними. Точнее, это определяется двумя младшими разрядами. Они могут задаваться константами **DEFAULT_PITCH** — по умолчанию, **FIXED_PITCH** — фиксированный шаг, **VARIABLE_PITCH** — изменяющийся шаг. Старшие разряды определяют некоторые характеристики семейства шрифтов. Эти тонкости вы можете посмотреть во встроенной справке Delphi.
12. Параметр **lpszFace** является указателем на строку с нулевым символом в конце, определяющую имя шрифта. Длина строки не должна превышать 32 символов.

Логический шрифт создается на основе какого-то из шрифтов, зарегистрированных в системе. Задаваемые параметры служат только ориентиром при выборе подходящего шрифта. Если шрифт удалось подобрать, функция **CreateFont** возвращает его дескриптор. В дальнейшем этот дескриптор можно передавать канве устройства, на которое выводится текст. Если шрифт подобрать не удалось, возвращается 0.

После того, как созданный логический шрифт использован, его надо удалить из памяти функцией **DeleteObject,** в которую передается дескриптор шрифта.

Пример: На рис. 17.1 показаны некоторые варианты вывода строк с различными шрифтами, различающимися, прежде всего, углами наклона. На форме только Image1.

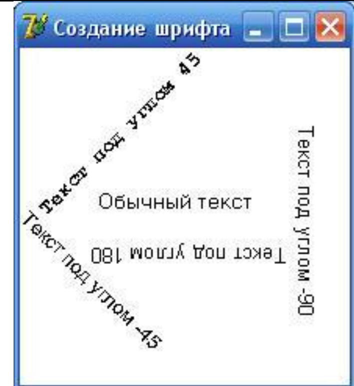


Рис. 17.1. Пример логических шрифтов

```
procedure TForm1.FormCreate(Sender: TObject);
var H1, H2, H3, H4: HFONT;
begin
H1:=CreateFont(-13, 0, 450, 450, FW_BOLD, 0, 0, 0, RUSSIAN_CHARSET, OUT_DEFAULT_PRECIS,
        CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH, 'Courier New');
H2:=CreateFont(-13, 0, -450, -450, FW_NORMAL, 0, 0, 0, DEFAULT_CHARSET, OUT_DEFAULT_PRECIS,
        CLIP_DEFAULT_PRECIS, PROOF_QUALITY, FIXED_PITCH, 'Arial');
H3:=CreateFont(-13, 0, 1800, 1800, FW_NORMAL, 0, 0, 0, DEFAULT_CHARSET, OUT_DEFAULT_PRECIS,
        CLIP_DEFAULT_PRECIS,  PROOF_QUALITY, DEFAULT_PITCH, nil);
H4:=CreateFont(-13, 0, -900, -900, FW_NORMAL, 0, 0, 0, DEFAULT_CHARSET, OUT_DEFAULT_PRECIS,
        CLIP_DEFAULT_PRECIS,  PROOF_QUALITY, DEFAULT_PITCH,  nil);
Image1.Canvas.FillRect(Image1.Canvas.ClipRect);
Image1.Canvas.Font.Size:=10;
Image1.Canvas.TextOut(50,90,'Обычный текст');
Image1.Canvas.Font.Handle:=H1;
Image1.Canvas.TextOut(5,95,'Текст под углом 45');
Image1.Canvas.Font.Handle:=H2;
Image1.Canvas.TextOut(10,100,'Текст под углом -45');
Image1.Canvas.Font.Handle:=H3;
Image1.Canvas.TextOut(170,140,'Текст под углом 180');
Image1.Canvas.Font.Handle:=H4;
Image1.Canvas.TextOut(190,50,'Текст под углом -90');
DeleteObject(H1);
DeleteObject(H2);
DeleteObject(H3);
DeleteObject (H4);
end;
```

Задание: изменить код программы таким образом, чтобы выводился текст следующим образом:

РАЗДЕЛ №18. НЕКОТОРЫЕ ПРИЕМЫ ПРОГРАММИРОВАНИЯ ПРИЛОЖЕНИЙ WINDOWS

УРОК №18.1 ОКОННЫЕ КОМПОНЕНТЫ И ФОРМЫ

Цель:

Образовательная: рассмотреть приемы работы с системным меню формы, как добавить свой раздел в системное меню приложения и как обработать реакцию на его выбор. Также рассмотреть каким образом осуществляется буксировка компонентов на форме и формы (различные способы).

Воспитательная: продолжить формирование у учащихся основ научного мировоззрения. Воспитание любви к профессии и предмету

Развивающая: развитие познавательных способностей (мышления, воображения и т.д.).

Тип занятия: урок формирования новых знаний

Форма организации учебного процесса: лекция

ХОД УРОКА

1. **Организационный момент** (требования к уроку, план работы)
2. **Изложение нового материала.**
3. **Подведение итогов урока. Обобщение знаний.**
4. **Домашнее задание.** Повторить изученный материал.

КОНСПЕКТ ПОД ЗАПИСЬ

Системные меню форм

Под системным меню понимается то стандартное меню, которое всплывает в любом окне Windows при щелчке на квадратике в его левом верхнем углу. Это меню содержит обычно разделы Восстановить, Переместить, Размер, Свернуть и т.п.

Если вам требуется как-то переопределить реакцию вашего приложения на стандартные разделы системного меню, это можно сделать, введя собственный обработчик сообщения **WM_SYSCOMMAND.**

Например:

```
type TForm1 = class(TForm)
private
{ Private declarations }
procedure WMSysCommand(var Msg: TWMSysCommand);message WM_SYSCOMMAND;
…
end;
…
procedure TForm1.WMSysCommand(var Msg: TWMSysCommand);
begin
  case Msg.CmdType of
        SC_CLOSE: ShowMessage('SC_CLOSE') ;
        SC_MAXIMIZE: ShowMessage('SC_MAXIMIZE');
        SC_MINIMIZE: ShowMessage('SC_MINIMIZE');
        SC_MOVE: ShowMessage('SC_MOVE');
        SC_RESTORE: ShowMessage('SC_RESTORE');
        SC_SIZE: ShowMessage('SC_SIZE');
  end;
  inherited;
end;
```

В этом примере приложение просто извещает вас о пришедшей команде: **SC_CLOSE** — Закрыть, **SC_MAXIMIZE** — Развернуть, **SC_MINIMIZE** — Свернуть, **SC_MOVE** — Переместить, **SC_RESTORE** — Восстановить, **SC_SIZE** — Размер. После этого извещения вызывается ключевым словом **inherited** стандартная обработка команды. Но вы, конечно, можете убрать все вызовы **ShowMessage** и вызов стандартного обработчика, и ввести собственную обработку того или иного сообщения. Например, если вы замените приведенную выше строку для команды **SC_MOVE** строкой

```
SC_MOVE: begin Left := 0; Top := 0; exit; end;
```

то команда меню Переместить будет перемещать окно приложения в верхний левый угол экрана.

Если вы хотите изменить что-то в системном меню вашего приложения, вы можете получить к нему доступ с помощью функции **GetSystemMenu.** Ее первый параметр — дескриптор окна, а значение второго параметра определяет режим работы функции: **false** — получение копии системного меню, **true** — установка системного меню по умолчанию с удалением существующего меню. Например, операторы

```
var MySystemMenu:  HMENU;
…
MySystemMenu := GetSystemMenu(Handle,  false);
```

дают копию **MySystemMenu** системного меню, после чего вы можете добавить в системное меню новые разделы или удалить некоторые из существующих в нем разделов.

Удаление раздела осуществляется функцией **DeleteMenu.** Первым аргументом в нее передается идентификатор удаляемого раздела или его индекс (индексы начинаются с 0). А второй аргумент определяет, как именно идентифицируется раздел первым аргументом: **MF_BYCOMMAND** — идентификатором раздела, **MF_BYPOSITION** — индексом. Например, оператор

```
DeleteMenu(MySystemMenu, SC_CLOSE, MF_BYCOMMAND);
```

удалит из системного меню вашего приложения раздел Закрыть и сделает недоступной соответствующую системную кнопку. Впрочем, горячие клавиши Alt-F4 все-таки будут закрывать приложение. Оператор

```
GetSystemMenu(Handle, true);
```

восстановит системное меню по умолчанию.

Добавить новые разделы в системное меню можно функцией **AppendMenu**.

Функция **AppendMenu** вставляет раздел в конец меню, указанного ее первым параметром.

Второй параметр задает флаги, определяющие вид вводимого раздела. В частности, флаг **MF_STRING** означает раздел с текстовой надписью, флаг **MF_SEPARATOR** — разделитель.

Третий параметр задает идентификатор вводимого раздела.

А последний параметр — строка текста, связанная с разделом.

Пример:

Добавление в конец системного меню разделителя и раздела с идентификатором **SC_MyItem1,** который будет выдавать простое сообщение.

Это может быть реализовано следующим образом (предполагается, что описанная ранее переменная **MySystemMenu** и процедура **WMSysCommand** уже введены в приложении). Объявите константы, являющиеся идентификаторами новых разделов:

```
Const
        SC_MyItem1 = WM_USER + 1;
```

Включение в меню этого раздела осуществляется операторами:

```
AppendMenu(MySystemMenu, MF_SEPARATOR, 0, '');
AppendMenu(MySystemMenu, MF_STRING, SC_MyItem1, 'Сообщение');
```

Первый оператор добавляет в конец меню разделитель, а следующий оператор добавляет новый раздел SC_MyItem1.

Рассмотренный ранее обработчик сообщения WM_SYSCOMMAND может быть дополнен следующим образом:

```
procedure TForm1.WMSysCommand(var Msg: TWMSysCommand);
begin
  case Msg.CmdType of
   SC_CLOSE: ...
     …
   SC_MyItem1: ShowMessage(‘Сообщение!!!’);
  end;
```

Листинг примера:

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
    procedure WMSysCommand(var Msg: TWMSysCommand);message WM_SYSCOMMAND;
  end;
 Const
        SC_MyItem1 = WM_USER + 1;
var
  Form1: TForm1;


implementation

{$R *.dfm}

{ TForm1 }

procedure TForm1.WMSysCommand(var Msg: TWMSysCommand);
begin
```

```
 case Msg.CmdType of
 SC_CLOSE: ShowMessage('SC_CLOSE') ;
 SC_MAXIMIZE: ShowMessage('SC_MAXIMIZE');
 SC_MINIMIZE: ShowMessage('SC_MINIMIZE');
 SC_MOVE: ShowMessage('SC_MOVE');
 SC_RESTORE: ShowMessage('SC_RESTORE');
 SC_SIZE: ShowMessage('SC_SIZE');
 SC_MyItem1: ShowMessage('Сообщение!!!');
 end;
 inherited;
end;

procedure TForm1.FormCreate(Sender: TObject);
var MySystemMenu:  HMENU;
begin
  MySystemMenu := GetSystemMenu(Handle,  false);
  AppendMenu(MySystemMenu, MF_SEPARATOR, 0, '');
AppendMenu(MySystemMenu, MF_STRING, SC_MyItem1, 'Сообщение');

end;

end.
```

Буксировка компонентов и форм

Иногда надо предоставить пользователю возможность буксировать компоненты по площади окна формы. Например, вы хотите разрешить пользователю перестраивать в каких-то пределах компоновку окна. Бывают также случаи, когда требуется перемещать по экрану форму, буксируя ее не за заголовок окна, а за любую ее точку.

Для реализации буксировки компонента в его обработчике события **OnMouseDown** включите следующий код:

```
ReleaseCapture ();
(Sender as TControl).Perform(WM_SYSCOMMAND, $F012, 0);
```

Если вы сошлетесь на подобный обработчик в событии формы, то пользователь сможет перемещать ее по экрану, взявшись за любую ее точку, а не только за заголовок окна.

Такая безусловная буксировка может создать сложности при работе, например, в окнах редактирования, поскольку в них буксировка мышью используется при работе с текстом. Так что, вероятно, разумнее осуществлять буксировку, если одновременно нажата какая-то вспомогательная клавиша, например, Alt. В этом случае обработчик события **OnMouseDown** может иметь вид:

```
if (ssAlt in Shift)
then begin
        ReleaseCapture();
        (Sender as TControl).Perform(WM_SYSCOMMAND, $F012, 0);
end;
```

В случае буксировки формы можно также проанализировать координаты курсора X и Y, чтобы буксировать только за определенную область формы.

Буксировка с помощью техники Drag&Dock

Для осуществления буксировки надо установить в форме свойство **DockSite** равным **true,** а в перемещаемых компонентах установить свойства **DragKind** в **dkDock** и свойства **DragMode** в **dmAutomatic.**

Рассмотренный метод универсальнее предыдущего, так как разрешает буксировать неоконные компоненты. Недостатком рассмотренного метода является то, что если вы разрешили перетаскивать кнопки, то они перестанут работать, так как нажатие мыши над ними будет восприниматься как начало перетаскивания. Неприятности могут возникнуть также в окнах редактирования и в других компонентах, в которых имеются стандартные, реакции на нажатие кнопки мыши. Это легко предотвратить, если в подобных компонентах разрешать буксировку только при нажатии какой-то вспомогательной клавиши. Например, если буксировка, предположим, кнопок **TButton,** должна начинаться, только если в этот момент нажата клавиша Alt, то для этих кнопок надо задать следующие обработчики событий **OnKeyDown** и **OnKeyUp:**

```
procedure TForm2.Button1KeyDown(Sender: TObject; var Key: Word; Shift: TShiftState);
```

```
begin
  if (Key = VK_MENU)
  then (Sender as TButton).DragMode := dmAutomatic;
end;
procedure TForm2.Button1KeyUp(Sender: TObject; var Key: Word; Shift: TShiftState);
begin
  if (Key = VK_MENU) then (Sender as TButton).DragMode := dmManual;
end;
```

При нажатии клавиши Alt в кнопке задается свойство **DragMode = dmAutomatic,** а при отпускании этой клавиши свойство **DragMode** устанавливается в **dmManual.** Так что если в момент нажатия кнопки мыши клавиша Alt не нажата, буксировки не будет.

Мигание заголовка окна и пиктограммы приложения

Иногда требуется, чтобы свернутое приложение известило пользователя о том, что оно завершило какую-то работу. Наиболее мягким способом такого извещения является мигание пиктограммы приложения в полосе задач. Организовать мигание можно функцией **FlashWindow:**

```
function FlashWindow(hWnd: HWND; bInvert: BOOL): BOOL;
```

Параметр **hWnd** является дескриптором окна, а параметр **bInvert** определяет режим переключения окна. При значении **bInvert = true** заголовок окна при каждом вызове **FlashWindow** один раз мигнет и останется в подсвеченном состоянии. При **bInvert** = **false** окно переходит в подсвеченное состояние. Если первым параметром функции указан дескриптор приложения, то мигание относится не к заголовку окна, а к пиктограмме приложения в полосе задач. Функция возвращает предыдущее состояние окна: **true,** если окно было активно.

Таким образом, для реализации мигания поместите на форму приложения компонент **Timer,** задайте в нем значение **Interval** равным, например, 2000 и значение **Enabled** = **false.** Напишите обработчик события **OnTimer** таймера:

```
FlashWindow(Application.Handle, true);
```

Этот оператор обеспечивает мигание пиктограммы в полосе задач. Если вы хотите, кроме того, обеспечить мигание заголовка окна, вы можете добавить в обработчик оператор:

```
FlashWindow(Form1.Handle, true);
```

Впрочем, это вряд ли целесообразно.

Когда в приложении произошло событие, о котором вы хотите известить пользователя, выполните оператор:

```
if not Application.Active
then Timer1.Enabled := true;
```

Это обеспечит мигание в случае, если в данный момент приложение неактивно (для активного приложения вряд ли стоит включать мигание). А если вы хотите, чтобы при активации мигающего окна мигание прекратилось, введите в приложение компонент **ApplicationEvents** и в обработчике его события **OnActivate** напишите оператор:

```
Timer1.Enabled := false;
```

УРОК №18.2 КЛАВИАТУРА И КУРСОР МЫШИ

Цель:

Образовательная:

Воспитательная: продолжить формирование у учащихся основ научного мировоззрения. Воспитание любви к профессии и предмету

Развивающая: развитие познавательных способностей (мышления, воображения и т.д.).

Тип занятия: урок формирования новых знаний

Форма организации учебного процесса: лекция

ХОД УРОКА

1. **Организационный момент** (требования к уроку, план работы)
2. **Изложение нового материала.**
3. **Подведение итогов урока. Обобщение знаний.**
4. **Домашнее задание.** Повторить изученный материал.

КОНСПЕКТ ПОД ЗАПИСЬ

Перехват событий клавиатуры

Свойство формы **KeyPreview = true**, тогда события клавиатуры сначала получает и обрабатывает форма, если = false, то события клавиатуры поступают непосредственно к тому элементу управления, который активен для ввода.

Установка свойства формы **KeyPreview = true** позволяет контролировать обработку нажатий клавиш клавиатуры. Исключение составляют клавиши Tab, Backspace, клавиши со стрелками и им подобные.

События клавиатуры

onKeyDown	Событие наступает при нажатии пользователем любой клавиши. Можно распознать нажатые клавиши, включая функциональные, и кнопки мыши, но нельзя распознать символ нажатой клавиши.
onKeyPress	Событие наступает при нажатии пользователем клавиши символа. Можно распознать только нажатую клавишу символа, различить символ в верхнем регистре и нижнем регистре, различить символы кириллицы и латинские, но нельзя распознать функциональные клавиши и кнопки.
onKeyUp	Событие наступает при отпускании пользователем любой клавиши. Можно распознавать нажатые клавиши, включая функциональные, и кнопки мыши, но нельзя распознать символ отпускаемой клавиши.

Заголовок обработчика события onKeyDown формы имеет вид:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;  Shift: TShiftState);
```

Параметр **Shift** определяет, какие вспомогательные клавиши нажаты на клавиатуре в момент нажатия кнопки мыши. Может принимать следующие значения: ssShift (клавиша Shift), ssAlt (клавиша Alt), ssCtrl (клавиша Ctrl), ssLeft (левая кнопка мыши), ssRight (правая кнопка мыши), ssMiddle (средняя кнопка мыши), ssDouble (двойной щелчок мыши)

Параметр **Key** определяет нажатую в момент события клавишу клавиатуры. Для не алфавитно-цифровых клавиш используют виртуальные коды.

Виртуальные коды клавиш:

Код	*Значение*	*Клавиша*
Vk_Back	8	Backspace
Vk_Tab	9	Tab
vk_Return	13	Enter
vk_Shift	16	Shift
vk_Control	17	Ctrl
vk_Menu	18	Alt
vk_Capital	20	Caps Lock
vk_Escape	27	esc
vk_Space	32	Пробел
vk_Prior	33	Page Up
vk_Next	34	Page Down
vk_End	35	End
vk_Home	36	Home
vk_Left	37	Курсор влево
vk_Up	38	Курсор вверх
vk_Right	39	Курсор вправо
vk_Down	40	Курсор вниз
vk_Insert	45	Insert

vk_Delete	46	Delete
vk_0..vk_9	48..57	0..9
vk_A..vk_Z	65..90	A. .Z
vk_LWin	91	Левая Windows
vk_RWin	92	Правая Windows
vk_NumpadO.. vk_Numpad9	96..105	[O]..[9]
vk_Multiply	106	[*]
vk_Add	107	[+]
vk_Subtract	109	[-]
Vk_Decimal	110	[Del]
vk_Divide	111	[/]
vkF1..vkF12	112..123	Fl..F12
vk_Numlock	144	Num Lock
vk_Scroll	145	Scroll Lock

Параметр key является целым числом, определяющим клавишу, а не символ. Проверить нажатую клавишу можно, сравнивая Key с кодом клавиши, приведенном в таблице. Например, реакцию на нажатие клавиши Enter можно оформить оператором:

If key=13 then …

Если не известен числовой код клавиши, то можно использовать его код, записанный в

первом столбце таблицы. Например, пример, приведенный выше, можно записать так:

If key=Vk_return then …

Для клавиш символов и цифр можно проводить проверку используя функцию Ord (при этом в скобках необходимо указывать только латинские буквы в верхнем регистре):

If key=ord('Y') then …

Комбинацию клавиш можно распознавать с помощью сложных условий. Например, комбинация Alt-X, распознается следующим образом:

If (key=ord('X'))and(ssAlt in Shift) then …

Событие OKeyPress

В этот обработчик события также передается параметр Key, определяющий нажатую клавишу символа. Но тип параметра не число, а символ – Char. Обработчик передает не виртуальный код клавиши, а символ, по которому можно определить, прописная или строчная буква, латинская или кириллица. Например, если необходимо распознать, что была нажата клавиша Д в нижнем или верхнем регистре, необходимо использовать следующий условный оператор:

If (key='Д')or(key='д') then …

Или

If (key in ['Д','д']) then …

Например, если необходимо, чтобы пользователь вводил в окно редактирования Edit1 только целые числа, без знака, разделенные запятыми или пробелами, то в обработчике события OnKeyPress компонента Edit1 необходимо использовать следующий оператор:

If not(key in ['0'..'9', ',' , ' ']) then begin key:=#0; beep; end;

Эмуляция нажатия клавиши

Эмуляция нажатия клавиши может быть выполнена посылкой соответствующему окну сообщения **WM_CHAR.** Его первым параметром **(wParam)** указывается код символа. Второй параметр **(lParam)** содержит дополнительную информацию, которая при эмуляции не используется.

Например, следующий код заносит посимвольно в окно **Memo1** строку S, выдерживая задержки между посылкой отдельных символов. Подобная анимация иногда полезна во всяких демонстрационных или обучающих приложениях.

```
procedure TForm1.Button1Click(Sender: TObject);
var i: word; s: string;
begin
        s := 'Привет'#13'и наилучшие пожелания !!!';
        for i:=1 to Length (s) do
        begin
              Memo1 .Perform (WM_CHAR, Ord(s[i]), 0);
              Sleep (300);
        end;
end;
```

Установка языка

Для установки языка ввода символов (русский, английский и т.п.) служит функция **LoadKeyboardLayout.**

Таким образом, оператор

```
LoadKeyboardLayout('00000419',KLF_ACTIVATE);
```

устанавливает русский язык, а оператор

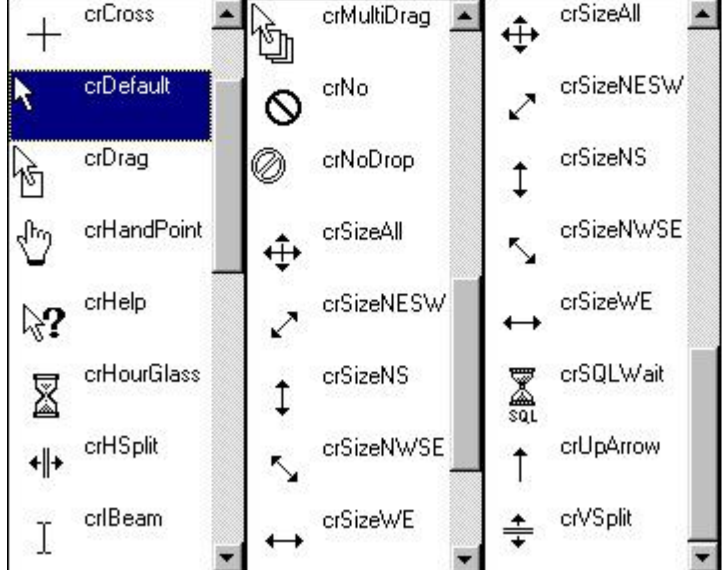
```
LoadKeyboardLayout(' 00000409',KLF_ACTIVATE) ;
```

устанавливает английский язык.

Английский язык соответствует коду "00000409", русский — коду "00000419". Флаг **KLF_ACTIVATE** активизирует указанную раскладку.

Курсор мыши

При перемещении указателя мыши по экрану он может менять свою форму в зависимости от свойства *Cursor* компонента, над которым он расположен в данный момент. В Delphi
предопределены стандартные указатели.



В практике программирования часто возникает необходимость изменения формы указателя для всех окон программы. Например, при выполнении достаточно длительного по времени процесса указатель мыши часто принимает вид crHourGlass, а после завершения процесса - восстанавливает свой первоначальный вид. Чтобы изменить форму указателя для всех окон программы одновременно, используется свойство cursor у глобального объекта screen, который автоматически создается для каждой программы:

```
Screen.Cursor := crHourGlass;
..... //Делаем длительную работу
Screen.Cursor := crDefault; // Восстанавливаем начальную  форму указателя
```

Получение и задание координат курсора мыши

Получить экранные координаты курсора мыши можно функцией **GetCursorPos** (модуль *Windows):*

```
function GetCursorPos(var lpPoint: TPoint): BOOL; stdcall;
```

Функция заносит в запись **lpPoint** значения координат. Например, операторы

```
var P: TPoint;
…
GetCursorPos(P);
ShowMessage(inttostr(p.x)+','+inttostr(p.Y)); //вывод координат курсора
```

позволяют найти экранные координаты курсора как значения полей P.X — координата X, P.Y — координата Y. Методы **ScreenToClient:**

Задать экранные координаты курсора мыши можно функцией **SetCursorPos:**

```
function SetCursorPos(X, Y: Integer): BOOL; stdcall;
```

Параметры X и Y — соответствующие координаты курсора. Например, оператор

Примеры	
SetCursorPos(0, 0);	переместит курсор в верхний левый угол экрана.
SetCursorPos(Form2.Left, Form2.Top);	переместит курсор к левому верхнему углу формы **Form2.**

var P: TPoint;
begin
  p:= Memo1.ClientToScreen(Point(0, 0)) ;
  SetCursorPos(P.x + 5, P.Y + 5);
  Sleep(1000);
  Memo1.Perform(WM_RBUTTONDOWN, 0, MakeLParam(5, 5));
  Memo1.Perform(WM_RBUTTONUP, 0, MakeLParam(5, 5)) ;
end;
	устанавливают курсор внутри клиентской области окна **Memo1,** расположенного на форме **Form2,** и с секундной задержкой имитируют щелчок правой кнопкой мыши, вызывающий вызов контекстного меню окна **Memo1**

Управление видимостью курсора мыши

Управление видимостью курсора мыши в пределах окна приложения осуществляется функцией **ShowCursor:**

```
function ShowCursor(bShow: BOOL): Integer; stdcall;
```

Значение **bShow = false** делает курсор невидимым, а значение **true** делает невидимый курсор видимым.

Точнее, так происходит при однократном вызове функции **ShowCursor.** В более общем случае надо учитывать, что в Windows подсчитывается число ссылок на курсор. Если число ссылок неотрицательно — курсор виден. Начальное значение числа ссылок равно 0. Каждый вызов **ShowCursor** с параметром **true** увеличивает число ссылок на 1, а каждый вызов **ShowCursor** с параметром **false** уменьшает на 1 число ссылок. Как только число ссылок станет отрицательным, курсор станет невидимым. Так что если вам требуется сделать курсор невидимым, независимо от числа ранее выполненных команд, делающих его видимым, это можно сделать, например, оператором:

```
while (ShowCursor(false) > 0) do ;
```

Функция **ShowCursor** возвращает число ссылок, полученное в результате вызова функции.

Обработка событий мыши

Все действия пользователя при взаимодействии с приложением сводятся к перемещению мыши, нажатию кнопок мыши и нажатию клавиш клавиатуры. Рассмотрим обработку в приложении событий, связанных с этими манипуляциями пользователя.

События мыши

В компонентах Delphi определен ряд событий, связанных с мышью. Это события:

Событие	Описание
onClick	Щелчок мыши на компоненте и некоторые другие действия пользователя.
onDblClick	Двойной щелчок мыши на компоненте
onMouseDown	Нажатие клавиши мыши над компонентом.
onMouseMove	Перемещение курсора мыши над компонентом
onMouseUp	Отпускание ранее нажатой кнопки мыши над компонентом.

В обработчики событий onMouseDown и onMouseUp передаются параметры, позволяющие распознать нажатую кнопку, нажатые при этом вспомогательные клавиши, а также определить координаты курсора мыши.

Заголовок обработчика события onMouseDown может иметь следующий вид:

```
procedure TForm1.Edit1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

В обработчик передаются параметры:

1. Button – определяет нажатую в данный момент кнопку мыши. Может иметь следующие значения: mbLeft – левая кнопка, mbRight - правая, mbMiddle –средняя.
2. X и Y – координаты курсора в области компонента.

Примеры:

Например, если необходимо, чтобы обработчик реагировал только на нажатие левой кнопки мыши, необходимо использовать следующий условный оператор:

```
If button <> mbLeft then exit; //если не левая кнопка мыши, то выйти из процедуры
```

Если необходимо выполнить какие-либо действия при нажатой левой кнопке мыши и клавиши Alt, то необходимо использовать следующий оператор в обработчике события:

```
If (button = mbLeft)and(ssAlt in Shift) then …
```

УРОК №18.3 ДИНАМИЧЕСКИ ПРИСОЕДИНЯЕМЫЕ БИБЛИОТКИ DLL

Цель:

Образовательная:

Воспитательная: продолжить формирование у учащихся основ научного мировоззрения. Воспитание любви к профессии и предмету

Развивающая: развитие познавательных способностей (мышления, воображения и т.д.).

Тип занятия: урок формирования новых знаний

Форма организации учебного процесса: лекция

ХОД УРОКА

1. **Организационный момент** (требования к уроку, план работы)
2. **Изложение нового материала.**
3. **Подведение итогов урока. Обобщение знаний.**
4. **Домашнее задание.** Повторить изученный материал.

КОНСПЕКТ ПОД ЗАПИСЬ

Динамически присоединяемая библиотека DLL – это специального вида исполняемый файл с расширением .dll, используемый для хранения функций и ресурсов отдельно от исполняемого файла. Обычно, когда вы пишете программу и создаете функции, ресурсы и т.п., все они компонуются в ваш исполняемый файл.

Создание DLL

1. Выполните команду File | New | Other
2. В диалоговом окне на странице New выберите пиктограмму DLL Wizard (Мастера DLL)

При компиляции проекта создастся файл библиотеки с расширением *.dll.*

После текстов всех функций и процедур в коде DLL располагается предложение **exports.** В нем перечисляются те процедуры и функции, которые экспортируются DLL, т.е. те, которые сможет вызывать внешнее приложение.

Если в **exports** указано просто имя процедуры или функции, то именно по этому имени внешнее приложение сможет ее вызывать.

Внешнее приложение может вызывать функции и процедуры DLL и по индексам. Индексы присваиваются автоматически, начиная с последнего имени, перечисленного в **exports.** В нашем примере функция **Code** получит индекс 1, а процедура **DoBeep** — индекс 2.

Пример: Пример, разместим в библиотеке процедуру DoBeep и функцию Code. Процедура DoBeep будет воспроизводить стандартный звук. В функцию Code передается строка и ключ, и она возвращает строку, зашифрованную или расшифрованную этим ключом.

Выполнение примера:

1. Создайте с помощью мастера Dll библиотеку.
2. Разместите в созданной библиотеке процедуру DoBeep и функцию Code как указано в листинге ниже. После этого введите раздел Exports и перечислите экспортируемые процедуру и функцию.

```
library MyDLL;

uses
  SysUtils {для функции Beep};

procedure DoBeep; stdcall;
begin
        Beep;
end;

function Code(S: PChar; Key: integer): PChar; stdcall;
var i: integer; ss: string;
begin
 ss := S;
 for i:=1 to Length(S) do ss[i] := char(Ord(ss[i]) xor Key);
 Code := PChar(ss);
end;

exports
 DoBeep,
 Code;
end.
```

3. Сохраните созданный проект в отдельной папке, под именем **MyDLL.** Скомпилируйте библиотеку сочетанием клавиш Ctrl+F9.
4. Создайте обычное приложение командой File – New Application.
5. Сохраните его в той же папке, что и библиотеку Dll.
6. Установите на форму компонент Edit и две кнопки. В первой задайте свойство Caption = «Звуковой сигнал», второй – «Шифровать (расшифровать)».
7. В обработчике события onClick для первой кнопки напишите оператор *DoBeep*.
8. В обработчике события onClick для второй кнопки напишите оператор
   ```
   Edit1.Text := Code(PChar(Edit1.Text), 5);
   ```
9. После раздела Uses поместите следующие операторы (для того, чтобы указать, что функция и процедура, используемые в программе, берутся из DLL-библиотеки):
   ```
   procedure DoBeep; stdcall; external 'MyDLL.DLL';
   ```

```
function Code(S: PChar; Key: integer): PChar; stdcall; external 'MyDLL.DLL';
```

10. Сохраните проект и запустите программу. Проверьте работу кнопок.

Пояснение к примеру:

Процедура DoBeep: просто вызывает стандартную функцию Beep.

Функция Code: она принимает строку S и целое значение ключа Key. Затем в цикле каждый символ исходной строки заменяется символом, получаемым операцией исключающего ИЛИ xor над индексом данного символа и ключом Key. В результате строка оказывается зашифрованной. Если повторно вызвать ту же функцию для зашифрованной строки с тем же ключом, то произойдет дешифровка и функция вернет первоначальную строку. По такому принципу обычно проходит любая шифровка текстов.

Функцию **Code** было бы проще реализовать, если бы тип параметра S и возвращаемого значения был определен как **string.**

Теперь обратите внимание на спецификатор **stdcall,** указанный после заголовков экспортируемых из DLL процедур и функций. Этот спецификатор задает определенные соглашения при передаче параметров, в частности, передачу параметров в последовательности справа налево. Подобная передача принята в API Windows и в ряде языков программирования, которые могут использоваться для создания приложений, которые будут вызывать вашу DLL. Так что целесообразно всегда использовать спецификатор **stdcall,** поскольку в противном случае будет принят по умолчанию спецификатор **register.** Это обеспечит наиболее быстрый обмен параметрами, но такую библиотеку **DLL** можно будет использовать только в приложениях, написанных на Pascal.

Пример №2. Создание формы в DLL-библиотеке, запрашивающей имя пользователя.

1. Откройте файл MyDLL.dpr.
2. Выполните команду File | New | Form.
3. Перенесите на форму Label, окно редактирования Edit1 и кнопку Button
4. Установите для кнопки свойство ModalResult = mrOk.
5. Сохраните модуль формы, задав ему имя *UMyDialog.*
6. Код DLL измените следующим образом (добавьте операторы, выделенные жирным шрифтом):

```
library MyDLL;
uses
  SysUtils {для функции Beep};
  Forms {для переменной Application},
  UMyDialog in 'UMyDialog.pas' {Form1};

procedure DoBeep; stdcall;
begin
     Beep;
end;

function Code(S: PChar; Key: integer): PChar; stdcall;
var i: integer; ss: string;
begin
 ss := S;
 for i:=1 to Length(S) do ss[i] := char(Ord(ss[i]) xor Key);
 Code := PChar(ss);
end;

function MyDialog(User: PChar): PChar; stdcall;
var Form: TForm1;
begin
 Form := TForm1.Create(Application); //создание формы
 Form.Edit1.Text := User;             //заносим в Edit переменную User
 Form.ShowModal;                      //показываем форму
 Result := PChar(Form.Edit1.Text);   //запоминаем результат, введенный пользователем
 Form.Free;                            //удаляем из памяти форму
end;

exports
 DoBeep,
 Code,
```

```
 MyDialog;
end.
```

7. Сохраните библиотеку и скомпилируйте ее.
8. Откройте приложение, использующее библиотеку.
9. Добавьте на форму третью кнопку, и дайте ей название «Регистрация».
10. В обработчик нажатия кнопки введите следующий оператор:

```
Edit1.Text := MyDialog('Пользователь');
```

11. После раздела uses добавьте оператор (чтобы указать, что функция из библиотеки):

```
function MyDialog(User: PChar): PChar; stdcall;  external 'MyDLL.DLL';
```

12. Сохраните программу и запустите ее на выполнение. Проверьте работу кнопки «Регистрация».

Пояснение к примеру

В приведенном коде введена функция **MyDialog,** вызывающая диалог и возвращающая строку, которую пользователь указал в окне **Edit1.** В качестве параметра в функцию передается строка **User** — начальное значение имени пользователя. В функцию введена локальная переменная **Form.** Первый выполняемый оператор создает экземпляр формы диалога. В окно редактирования этой формы заносится строка **User.** Затем методом **ShowModal** форма показывается пользователю как модальная. После завершения работы пользователя с этой формой текст окна редактирования заносится в значение, возвращаемое функцией, и форма удаляется из памяти методом **Free.**

Таким образом, вызов диалога не отличается от вызова любой другой функции. Только в операторе **uses** добавляется ссылка на модуль формы и на модуль *Forms,* без которого компилятор не поймет переменной **Application.**

УРОК №19.1 УКАЗАТЕЛЬНЫЙ (ССЫЛОЧНЫЙ) ТИП. ДИНАМИЧЕСКИЕ ПЕРЕМЕННЫЕ

Цель:

Образовательная:

Воспитательная: продолжить формирование у учащихся основ научного мировоззрения. Воспитание любви к профессии и предмету

Развивающая: развитие познавательных способностей (мышления, воображения и т.д.).

Тип занятия: урок формирования новых знаний

Форма организации учебного процесса: лекция

ХОД УРОКА

1. **Организационный момент** (требования к уроку, план работы)
2. **Изложение нового материала.**
3. **Подведение итогов урока. Обобщение знаний.**
4. **Домашнее задание.** Повторить изученный материал.

КОНСПЕКТ ПОД ЗАПИСЬ

7.4. УКАЗАТЕЛИ И ДИНАМИЧЕСКАЯ ПАМЯТЬ

7.4.1. Динамическая память

Динамическая память - это оперативная память ПК, предоставляемая программе при ее работе. Динамическое размещение данных означает использование динамической памяти непосредственно при работе программы. В отличие от этого статическое размещение осуществляется компилятором Object Pascal в процессе компиляции программы. При динамическом размещении заранее не известны ни тип, ни количество размещаемых данных.

7.4.2. Указатели

Оперативная память ПК представляет собой совокупность ячеек для хранения информации - байтов, каждый из которых имеет собственный номер. Эти номера называются *адресами,* они позволяют обращаться, к любому байту памяти. Object Pascal предоставляет в распоряжение программиста гибкое средство управления динамической памятью - так называемые указатели. *Указатель* - это переменная, которая в качестве своего значения содержит адрес байта памяти. С помощью указателей можно размещать в динамической памяти любой из известных в Object Pascal типов данных. Лишь некоторые из них (Byte, Char, ShortInt, Boolean) занимают во внутреннем представлении один байт, остальные - несколько смежных. Поэтому на самом деле указатель адресует лишь первый байт данных.

Как правило, указатель связывается с некоторым типом данных. Такие указатели будем называть *типизированными.* Для объявления типизированного указателя используется значок ^, который помещается перед соответствующим типом, например:

```
var

p1 : ^Integer;

p2 : ^Real;

type

     PerconPointer = "PerconRecord;

PerconRecord = record Name : String;

Job : String;

Next : PerconPointer ,

end;
```

Обратите внимание: при объявлении типа PerconPointer мы сослались на тип PerconRecord, который предварительно в программе объявлен не был. Как уже отмечалось, в Object Pascal последовательно проводится в жизнь принцип, в соответствии с которым перед использованием какого-либо идентификатора он должен быть описан. Исключение сделано только для указателей, которые могут ссылаться на еще не объявленный тип данных.

В Object Pascal можно объявлять указатель и не связывать его при этом с каким-либо конкретным типом данных. Для этого служит стандартный тип pointer, например:

```
     var

     p: Pointer;
```

Указатели такого рода будем называть *нетипизированньти.* Поскольку нетипизированные указатели не связаны с конкретным типом, с их помощью удобно динамически размещать данные, структура и тип которых меняются в ходе работы программы.

Как уже говорилось, значениями указателей являются адреса переменных в памяти, поэтому следовало бы ожидать, что значение одного указателя можно передавать другому. На самом деле это не совсем так. В Object Pascal можно передавать значения только между указателями, связанными с одним и тем же типом данных.

Если, например,

```
var
    pI1,pI2: ^integer;
    pR: ^Real;
    p: Pointer;
```

```
то присваивание
pI1 := pI2;
вполне допустимо, в то время как
pl1 :=pR;
```

запрещено, поскольку pI1 и pR указывают на разные типы данных. Это ограничение, однако, не распространяется на нетипизированные указатели, поэтому мы могли бы записать

```
p := pR;
pI1 := p;
```

и тем самым достичь нужного результата.

7.4.3. Выделение и освобождение динамической памяти

Вся динамическая память в Object Pascal рассматривается как сплошной массив байтов, который называется *кучей.*

Память под любую динамически размещаемую переменную выделяется процедурой New. Параметром обращения к этой процедуре является типизированный указатель. В результате обращения указатель приобретает значение, соответствующее адресу, начиная с которого можно разместить данные, например:

```
var pI,pJ: ^Integer;
    pR: ^Real;
begin
        New (pI) ;
        New (pR) ;
    end;
```

После того как указатель приобрел некоторое значение, т. е. стал указывать на конкретный физический байт памяти, по этому адресу можно разместить любое значение соответствующего

типа. Для этого в операторе присваивания сразу за указателем без каких-либо пробелов ставится значок ^, например:

```
pJ^ := 2; // В область памяти pJ помещено значение 2
pl^ := 2*pi; // В область памяти pR помещено значение 6.28
```

Таким образом, *значение,* на которое указывает указатель, т. е. собственно данные, размещенные в куче, обозначаются значком ^, который ставится сразу за указателем. Если за указателем нет значка ^, то имеется в виду *адрес,* по которому размещены данные. Имеет смысл еще раз задуматься над только что сказанным: значением любого указателя является адрес, а чтобы указать, что речь идет не об адресе, а о тех данных, которые размещены по этому адресу, за указателем ставится ^ (иногда об этом говорят как *о разыменовании* указателя).

Динамически размещенные данные можно использовать в любом месте программы, где это допустимо для констант и переменных соответствующего типа, например:

```
pR^ := Sqr(pR") + I^ - 17;
```

Разумеется, совершенно недопустим оператор

```
pR := Sqr(pR") + I^ - 17;
```

так как *указателю* pR нельзя присвоить значение вещественного выражения. Точно так же недопустим оператор

```
pR^ := Sqr(pR) ;
```

поскольку значением указателя pR является адрес и его (в отличие от того значения, которое размещено по этому адресу) нельзя возводить в квадрат. Ошибочным будет и такое присваивание:

```
pR^' := pJ;
```

так как вещественным данным, на которые указывает pR, нельзя присвоить значение указателя (адрес).

Динамическую память можно не только забирать из кучи, но и возвращать обратно. Для этого используется процедура Dispose. Например, операторы

```
Dispose(pJ);
Dispose(pR);
```

вернут в кучу память, которая ранее была закреплена за указателями pJ и pR (см. выше).

Замечу, что процедура Dispose (pPtr) не изменяет значения указателя pPtr, а лишь возвращает в кучу память, ранее связанную с этим указателем. Однако повторное применение процедуры к свободному указателю приведет к возникновению ошибки периода исполнения. Освободившийся указатель программист может пометить зарезервированным словом nil. Помечен ли какой-либо указатель или нет, можно проверить следующим образом:

```
const
pR: ^Real = NIL;
begin
if pR = NIL then
    New (pR) ;
```

```
Dispose(pR) ;
pR := NIL;
end;
```

Никакие другие операции сравнения над указателями не разрешены.

Приведенный выше фрагмент иллюстрирует предпочтительный способ объявления указателя в виде типизированной константы с одновременным присвоением ему значения nil. Следует учесть, что начальное значение указателя (при его объявлении в разделе переменных) может быть произвольным. Использование указателей, которым не присвоено значение процедурой New или другим способом, не контролируется Delphi и вызовет исключение.

Как уже отмечалось, параметром процедуры New может быть только типизированный указатель. Для работы с нетипизированными указателями используются Процедуры GetMem И FreeMem:

```
GetMem(P, Size); // резервирование памяти;
FreeMem(P, Size); // освобождение памяти.
```

Здесь p - нетипизированный указатель; size - размер в байтах требуемой или освобождаемой части кучи.

Использование прцедур GetMem/FreeMemMet, как и вообще вся работа диамияесжой памятью, требует особой осторожности и тщателвного солюдения простого правила: освобождать нужно ровно столько пайти, сколько её было зарезервировано, и именно с того адреса, с которого она была зарезёрвирована.

7.4.4. Процедуры и функции для работы с динамической памятью

В табл. 7.14 приводится описание как уже рассмотренных процедур и функций Object Pascal, так и некоторых других, которые могут оказаться полезными при обращении к динамической памяти.

Таблица 7.14. Средства Object Pascal для работы с памятью

Function Addr(X): Pointer;	Возвращает адрес аргумента X. Аналогичный результат возвращает операция @
Procedure Dispose (var P: Pointer) ;	Возвращает в кучу фрагмент динамической памяти, который ранее был зарезервирован за типизированным указателем Р
Procedure Free-Mem(var P: Pointer; Size: Integer) ;	Возвращает в кучу фрагмент динамической памяти, который ранее был зарезервирован за нетипизированным указателем Р
Procedure Get-Mem(var P: Pointer; Size: Integer) ;	Резервирует за нетипизированным указателем Р фрагментдинамической памяти требуемого размера Size

Procedure New(var P: Pointer) ;	Резервирует фрагмент кучи для размещения переменной и помещает в типизированный указатель P адрес первого байта
Function SizeOf(X): Integer;	Возвращает длину в байтах внутреннего представления указанного объекта *X*

Windows имеет собственные средства работы с памятью. В табл. 7.15 перечислены соответствующие API-функции и даны краткие пояснения. За более полной информацией обращайтесь к справочной службе в файлах WIN32. hlp или WIN32S. hlp.

Таблица 7.15. Средства Windows для работы с памятью

CopyMemory	Копирует содержимое одного блока памяти в другой блок. Блоки не должны перекрываться хотя бы частично
FillMemory	Заполняет блок памяти указанным значением
GetProcessHeap	Возвращает дескриптор кучи для текущей программы
GetProcessHeaps	Возвращает дескрипторы куч для всех работающих программ
GlobalAlloc	Резервирует в куче блок памяти требуемого размера
GlobalDiscard	Выгружает блок памяти
GlobalFlags	Возвращает информацию об указанном блоке памяти
GlobalFree	Освобождает блок памяти и возвращает его в общий пул памяти
GlobalHandle	Возвращает дескриптор блока памяти, связанного с заданным указателем
GlobalLock	Фиксирует блок памяти и возвращает указатель на его первый байт
GlobalMemoryStatus	Возвращает информацию о доступной памяти (как физической, так и виртуальной)
GlobalReAlloc	Изменяет размер и атрибуты ранее зарезервированного блока памяти

GlobalSize	Возвращает размер в байтах блока памяти
GlobalUnlock	Снимает фиксацию блока памяти и делает его перемещаемым
HeapAlloc	Резервирует в куче неперемещаемый блок памяти
HeapCompact	Удаляет фрагментацию кучи
HeapCreate	Создает для программы новую кучу
HeapDestroy	Возвращает кучу в общий пул памяти
HeapFree	Освобождает блок памяти, зарезервированный функциями HeapAlloc или HeapReAlloc
HeapLock	Делает указанную кучу доступной только для текущего потока
HeapReAlloc	Изменяет размер и/или свойства кучи
HeapSize	Возвращает размер кучи в байтах
HeapUnlock	Делает указанную кучу доступной для любых потоков текущего процесса
HeapValidate	Проверяет состояние кучи или размещенного в ней блока памяти
IsBadCodePtr	Сообщает, может ли вызывающая программа читать данные из указанного адреса памяти (но не из блока памяти)
IsBadHugeReadPtr	Сообщает, может ли вызывающая программа читать данные из указанного блока памяти
IsBadHugeWritePtr	Сообщает, может ли вызывающая программа изменять содержимое указанного блока памяти
IsBadReadPtr	Сообщает, может ли вызывающая программа читать данные из указанного блока памяти
IsBadStringPtr	Сообщает, может ли программа читать содержимое строки, распределенной в куче
IsBadWritePtr	Сообщает, может ли вызывающая программа изменять содержимое указанного блока памяти
LocalAlloc	Аналогична GlobalAlloc

:: LocalDiscard	Аналогична GloalDiscard
'LocalFlags	Аналогична GlobalFlags
LocalFree	Аналогична Global Free
LocalHandle	Аналогична GlobalHandle
LocalLock	Аналогична GlobalLock
LocalReAlloc	Аналогична GlobalReAlloc
LocalSize	Аналогична GlobalSize
LocalUnlock	Аналогична GlobalUnlock
MoveMemory	Копирует один блок памяти в другой. Блоки могут перекрываться
VirtualAlloc	Резервирует блок виртуальной памяти
VirtualFree	Освобождает блок виртуальной памяти
VirtualLock	Фиксирует блок виртуальной памяти
VirtualProtect	Изменяет права доступа текущей программы к виртуальному блоку памяти
VirtualProtectEx	Изменяет права доступа указанной программы к виртуальному блоку памяти
VirtualQuery	Возвращает свойства виртуального блока памяти по отношению к вызывающей программе
VirtualQueryEx	Возвращает свойства виртуального блока памяти по отношению к указанной программе
VirtualUnloc'k	Снимает фиксацию блока виртуальной памяти
ZeroMemory	Заполняет блок памяти нулями