[I. №1: Стили программирования математических вычислений: "эффективный" и "удобный", области их применения. 2](#_Toc504384659)

[II. №2: Требования к документации к модулю, к процедурам и функциям модуля. Требования к программированию процедур и функций. 3](#_Toc504384660)

[III. №3: Адресация памяти в программе. Переменные типа указатель в языке Pascal. Операции над указателями 4](#_Toc504384661)

[IV. №4: Переход от размерно-зависимого программирования с векторами и матрицами к размерно-независимому с применением указателей. 5](#_Toc504384662)

[V. №5: Динамическая память. Создание динамических переменных и их удаление. Использование динамической памяти для временного хранения промежуточной информации. 6](#_Toc504384663)

[VI. № 6: «Удобное» программирование с использованием классов математических объектов. 7](#_Toc504384664)

[VII. №7: Система обработки ошибок в модуле. 8](#_Toc504384665)

[VIII. №8. Среда Delphi, структура программы, модуля. Работа с проектом. Примеры. 9](#_Toc504384666)

[IX. №9. Компиляция программы Delphi. Работа с несколькими объектами на форме. Процедуры обработки событий. Свойства формы. Примеры. 10](#_Toc504384667)

[X. №10. Классы, инкапсуляция и наследование, полиморфизм. Примеры. 15](#_Toc504384668)

[XI. 11. Компоненты ввода, вывода, выполнения действий программы Delphi. Процедура-обработчик события создания формы. Примеры. 16](#_Toc504384669)

[XII. 12. Компоненты выбора, выполнения действия программы Delphi. Организация работы с различными типами данных. 18](#_Toc504384670)

[XIII. 13. Массивы в Delphi. Компонент StringGrid. Примеры. 20](#_Toc504384671)

[XIV. 14. Работа со строками. Компоненты ListBox, ComboBox, BitBtn. Обработка событий. Примеры. 21](#_Toc504384672)

[XV. 15. Архитектура приложения Delphi. Компонент Mainmenu. Примеры. 25](#_Toc504384673)

[XVI. 18. Текстовые файлы. Типизированные файлы. Нетипизированные файлы. 28](#_Toc504384674)

[XVII. 19. Отображение файлов в память. 29](#_Toc504384675)

[XVIII. 20. Работа с файлами как с объектами в Delphi, абстрактный класс TStream. Примеры. 31](#_Toc504384676)

[XIX. 21. Класс Exeption. Защищенные блоки. Стандартные классы исключений. Вызов исключения. Создание собственного потока. 33](#_Toc504384677)

[XX. 22. Класс TList. Свойства и методы. Тип pPointerList и pDouble. Пример. 41](#_Toc504384678)

[XXI. 23. Классы TStringList и TStrings, основные свойства и методы. Пример. 45](#_Toc504384679)

[XXII. 24. Класс TFont, основные методы и свойства. Пример. 48](#_Toc504384680)

[XXIII. 26.Класс TBrush. Методы и свойства. Пример 50](#_Toc504384681)

[XXIV. 27.Класс TCanvas. Методы и свойства. Пример 52](#_Toc504384682)

[XXV. 28.Классы TGraphic и TPicture, методы и свойства. Пример 56](#_Toc504384683)

[XXVI. 29.Редактор диаграмм, методы серий Series. Пример 59](#_Toc504384684)

[XXVII. 30.Набор обработчиков событий видимых компонентов OnMouseDown, OnMouseUp, OnMouseMove, OnClick, OnDblClick.Пример 61](#_Toc504384685)

[XXVIII. 31. События клавиатуры. Свойства компонентов, связанных с обрботкой событий. Пример 61](#_Toc504384686)

[XXIX. 32. Механизм действий. Механизм DragAndDrop. Свойства класса TControl. Пример 63](#_Toc504384687)

[XXX. 33. Механизм DragAndDock реализации причаливания в классе TControl. Пример 66](#_Toc504384688)

# №1: Стили программирования математических вычислений: "эффективный" и "удобный", области их применения.

Принципы «эффективной» парадигмы:

1. Не зависит от размерности данных, функции и процедуры работают с произвольными размерностями;
2. Никакие специальные типы данных не нужны, используются универсальные типы данных;
3. Контроль за размерностью не ведется, вся ответственность за нее возлагается на пользователя;
4. Присутствие в параметрах адресов;
5. Если в результате операции создается векторный объект, то это реализуется процедурой, а соответствующий объект программы должен быть создан, чтобы различать только адрес.

Пример: (нахождение скалярного произведения в R3)

TYPE

RealPtr = ^Real;

Function IP (pa, pb: RealPtr; n: WORD): Real;

VAR

i: word;

s: Real;

BEGIN

s:=0;

For i:=1 to n do

BEGIN

s:= s + pa^ \* pb^;

Inc (pa);

Inc (pb);

END;

IP:= s;

END;

{MAIN PROGRAM}

VAR

a, b: array [1..3] of Real;

s:=Real;

BEGIN

s:= IP (@a, @b, 3);

Writeln (s:10:6);

END;

Принципы «удобной»:

1. Размерность хранится в самом объекте и передается процедурам и функциям в составе объекта, а они, получив их в качестве параметров, обязаны проверить их размеры на соответствие параметрам. Выход за пределы размерности объекта при заполнении его координат невозможен, т.к. это контролирует процедура. Проверка на то, заполнены ли все координаты в Паскале – не возможна, но возможна, например, в С++;
2. Появляются осмысленные типы данных, например, “Vector”. В языке “Pascal” эти типы данных имеют тип «указатель»;
3. Для “удобной” парадигмы характерны наличие и многократное использование функций, создающих и уничтожающих объект;
4. Когда создается новый объект, как правило, пишется функция, которая реально создает объект, выделяя для него место в оперативной памяти. Функции, возвращающие числовые значения, выглядят так же, как и в первой парадигме. Если же она “создает” объект, то принимает новый вид;
5. В коде используется работа с указателями и полями структур. Размерности не появляются в списках параметров, т.к. они спрятаны в самих объектах.

Пример:

- создание нового вектора

Function NewVector (n: word): Vector;

Var

v: Vector;

Begin

New (v);

v^.dim:= n;

GetMem (v^.elem, n\*Sizeof (Real));

NewVector := v;

End;

- копирование вектора (из u в v)

Procedure CopyVector (u,v: Vector);

var

i: word;

pu, pv: RealPtr;

Begin

pu := u^.elem;

pv := v^.elem;

For i :=1 to u^.dim do

Begin

pv^:= pu^;

Inc (pu);

Inc (pv);

End;

End;

# №2: Требования к документации к модулю, к процедурам и функциям модуля. Требования к программированию процедур и функций.

Подпрограмма – это отдельная функционально независимая часть программы. В языке Паскаль выделяют 2 вида подпрограмм процедуры (Procedure) и функции (Function). Любая программа может содержать несколько процедур и функций. Структура любой подпрограммы аналогична структуре всей программы.

Функция отличается от процедуры тем, что результат ее работы возвращается в виде значения этой функции.

Описание процедуры начинается словом Procedure, за которым следует имя процедуры и список формальных параметров. Список параметров заключается в круглые скобки и содержит перечень параметров с указанием их типа. Затем, стоит слово var – именно таким способом компилятору указываются те параметры, в которых процедура возвращает вызвавшей ее программе результат своей работы. Слово Procedure, имя процедуры и список ее параметров образуют заголовок процедуры. За заголовком следует тело процедуры, содержащее новый раздел описаний и раздел исполнения операторов.

Проектная документация – текст, где приведены типы данных, список операций в виде заголовков процедур и функций.

Под заголовком процедуры (функции) описано на человеческом языке какую функцию она выполняет, описываются входные параметры (для функций еще и значения). Затем в документации пишется набор текстовых задач (продуманная система тестов)

**Создание процедур и функций**.

Этапы создания:

1. проектирование
2. кодирование
3. отладка и тестирование
4. документация

Проектирование - создание проектной документации. Какую информацию функция должна получить, в каком виде будут результаты возвращаться

Правило:

Процедура или функция математического характера не должна заниматься вводом или выводом информации. Все результаты должны передаваться через значение функции или через параметры по ссылке.

Имя процедуры или функции должно нести смысловую нагрузку. Внутри процедур или функций может быть список переменных (VAR) – локальные переменные процедур и функций. Создаются при начале работы функции/процедуры и уничтожаются после выполнения, поэтому могут иметь имена, совпадающие с другими.

Если документация содержит ошибки, неточности и пропуски, этот проект не будет полезен пользователю.

В процессе эксплуатации программы уважающих себя компаний происходит «слежение» за продуктами и периодическое обновление.

# №3: Адресация памяти в программе. Переменные типа указатель в языке Pascal. Операции над указателями

Указатель – переменная, возможными значениями которой могут быть адреса оперативной памяти размера 4 байта. Каждый байт оперативной памяти имеет свой адрес.

Как правило, в Паскале указатель связывается с некоторым типом данных. Такие указатели – типизированные. Для объявления типизированного указателя используется значок ^, который помещается перед соответствующим типом, например:

var

p1: ^Integer;

p2: ^Real;

type

PersonPointer = ^PersonRecord;

PersonRecord = record

Name : String;

Job : String;

Next : PersonPointer

end;

В Паскале 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 – нетипизированные. Поскольку нетипизированные указатели не связанны с конкретным типом, с их помощью удобно динамически размещать данные, структура и тип которых меняются в ходе работы программы.

POINTER –не типизированный указатель

px: ^REAL; - типизированный указатель

pn: ^WORD; - тип. указатель целочисленный

Операции над указателями

1. Присвоить значение:

pa:= @a; или pa:= @(A[i]);

pa:= Addr(a);

pa:= px; - операция копирования одного указателя в другой. (если хотя бы 1 нетипизированный или оба одного типа)

[ p:px;

pn:=p; ]

1. Инкремент указателя:

Inc(p); Inc(px); - увеличивает

A: ARRAY [1…..] OF REAL;

px:= @A; - указатель на A[1]

Inc(px); - указатель на A[2]

Операции разадресации:

px^ //(как я понял, просто пример того, что с помощью ^ можно получить значение из области памяти, на которую указывает указатель px)

TYPE

RealPtr= ^REAL;

WordPtr = ^WORD;

IntPtr= ^INTEGER;

CharPtr= ^CHAR;

StrPtr= ^STRING;

FUNCTION Norm ( pa: RealPtr; n: WORD): REAL;

VAR

S: REAL;

i: WORD;

Begin

S:= 0 ;

For i=1 to n do

BEGIN

S:= S + Sqr(pa^);

Inc(pa);

END;

Norm:= Sqrt(S);

End;

Если указатель должен повторно проходить один и тот же массив, то рекомендуется не изменять значение указателя, а использовать вспомогательный указатель, чтобы вернуть его в начало массива достаточно присвоить ему значение изначального указателя.

Если же указатель проходит массив однократно, то для работы с элементами этого массива можно использовать указатель, стоящий в списке параметров и передаваемый по значению.

# №4: Переход от размерно-зависимого программирования с векторами и матрицами к размерно-независимому с применением указателей.

«Наивная» парадигма:

1. Фиксированные размеры данных (чтобы изменить их надо перекомпилировать программный модуль, но в любом случае доступна одна размерность);
2. Массивы данной размерности объявляют именованным типом данных (например, Vector);
3. В вызывающей программе этот именованный тип данных используется для обращения к конкретным переменным.

{Напишем функцию, вычисляющую скалярное произведение двух векторов a и b размерности n}

1. Для начала нужно вспомнить работу с индексами и написать функцию подсчета скалярного произведения в векторной форме, отладить её.
2. Все размерности вносятся в раздел const А все числа, зависящие от них, заменяются на свои выражения через эти размерности. При задании новых констант необходимо тщательно следить за тем, чтобы все числа, зависящие от них, были заменены на соответствующие выражения через эти константы.
3. Размерности вносятся в список аргументов функции.
4. Удалить все описания из type. Сделать размерности переменными (убрать раздел const).
5. Все индексные выражения, входящие в список аргументов функции, заменить на указатели, указывающие на переменную того же типа, что элементы массивы.
6. Преобразование внутри тела функции.

Постепенно избавляться от индексов и полностью переходить на указатели (заменить каждое индексное выражение на указатели).

Постараться объединить удобство индексов с эффективностью указателей

( Внести в раздел TYPE тип данных "массив фиктивной длинны". Каждое индексное выражение заменить на его "фиктивный" эквивалент).

1. Оптимизация (здесь не существует конкретных указаний, так как каждая программа нуждается в собственном подходе).
2. Заметим, что проход по обоим векторам осуществляется всего один раз, поэтому от вспомогательных указателей pa и pb можно и отказаться. Вместо них будем использовать уже имеющиеся a и b.
3. Для организации цикла используем указатели. Как это осуществить уже рассказывалось в разделе **Указатели**.

Этот вариант программы не нуждается в доработке, потому последний этап для него излишен.

# №5: Динамическая память. Создание динамических переменных и их удаление. Использование динамической памяти для временного хранения промежуточной информации.

**Все переменные, объявленные в программе, размещаются в одной непрерывной области оперативной памяти, которая называется сегментом данных**. Длина сегмента данных определяется архитектурой микропроцессоров и составляет 65536 байт, что может вызвать известные затруднения при обработке больших массивов данных. С другой стороны, объем памяти ПК (обычно не менее 640 Кбайт) достаточен для успешного решения задач с большой размерностью данных. Выходом из положения служит использование динамической памяти.

Динамическая память – это оперативная память ПК, предоставляемая программе при ее работе, за вычетом сегмента данных (64 Кбайт), стека (обычно 16 Кбайт) и собственно тела программы. Ее размер составляет не менее 200-300 Кбайт. Динамическая память используется для обработки массивов большой размерности, а также для временного запоминания при работе с графическими и звуковыми средствами ПК.

При динамическом размещении заранее не известны ни тип, ни количество размещаемых данных, к ним нельзя обращаться по именам как к статическим переменным.

# № 6: «Удобное» программирование с использованием классов математических объектов.

Представим, что нам нужно создать модуль для области из аналитической геометрии. Объекты из нее — это искомые математические объекты. Поскольку мы не знаем, сколько объектов нам понадобится – мы используем указатели. Для обращения с памятью объекты создаются через конструкторы и деструкторы.

Для описания методов в ООП используются традиционные для Pascal процедуры и функции, а также особый вид процедур - конструкторы и деструкторы. Конструкторы предназначены для создания конкретного экземпляра объекта, ведь объект — это тип данных, то есть "шаблон", по которому можно создать сколько угодно рабочих экземпляров данных объектного типа (например, типа TGraphObj). Зарезервированное слово Constructor, используемое в заголовке конструктора вместо Procedure, предписывает компилятору создать особый код пролога, с помощью которого настраивается так называемая таблица виртуальных методов. Если в объекте нет виртуальных методов, в нем может не быть ни одного конструктора, наоборот, если хотя бы один метод описан как виртуальный (с последующим словом Virtual, см. метод Draw), в состав объекта должен входить хотя бы один конструктор, обращение к конструктору должно предшествовать обращению к любому виртуальному методу.

1. Конструктор и деструктор — это методы, которые используются для создания и разрушения объекта соответственно.  
   2. Объект — это экземпляр класса.  
   3. Объявив класс, мы не получим объект - его надо создать.

**Конструкторы и деструкторы.**

Методы, которые предназначены для создания и удаления [объектов](http://www.maksakov-sa.ru/TeorDelphi/KonstrDestrKlassDelphi/index.html) называются конструкторами и деструкторами соответственно. Описание данных методов отличается от обычных тем, что в их заголовках стоят ключевые слова constructor и destructor. В качестве имен конструкторов и деструкторов в базовом классе TObject и многих других классах используются имена Create и Destroy.

Прежде чем обращаться к элементам объекта, его нужно создать с помощью конструктора. Например: ObjectA:=TOwnClass.Create;

Конструктор выделяет память для нового объекта, задает нулевые значения для порядковых полей, значение nil – для указателей и полей классов, строковые поля устанавливает пустыми, а также возвращает указатель на созданный объект.

Классы объектов определяются в секции **type** глобального блока. Описание класса начинается с ключевого слова **class** и заканчивается ключевым словом **end**.

Чтобы от описания класса перейти к объекту, следует выполнить соответствующее объявление в разделе **var**:*Var math:TMatematica;*

Переменная *math* – это ссылка на экземпляр (объект в памяти), которого физически еще не существует. Чтобы сконструировать объект (выделить память для экземпляра) класса *TMatematica* и связать с ним переменную *math*, нужно в тексте программы поместить следующий оператор: *math:= TMatematica.Create;*

**Create** – это конструктор объекта; он всегда присутствует в классе и служит для создания и инициализации экземпляров. При создании объекта в памяти выделяется место только для его полей. Методы, как и обычные процедуры и функции, помещаются в область кода программы; они могут работать с любыми экземплярами своего класса и не дублируются в памяти.

После создания объект можно использовать в программе: получать и устанавливать значения его полей, вызывать его методы. Доступ к полям и методам объекта происходит с помощью имен, например: *math.x1;*

Если объект становится ненужным, он должен быть удален вызовом специального метода Destroy (освобождение памяти, занимаемой объектом), например: *math*.Destroy;

**Destroy** – это деструктор объекта; он присутствует в классе наряду с конструктором и служит для удаления объекта из динамической памяти. После вызова деструктора переменная *math* становится несвязанной и не должна использоваться для доступа к полям и методам уже несуществующего объекта. Чтобы отличать в программе связанные объектные переменные от несвязанных, последние следует инициализировать значением **nil**.

Вызов деструктора для несуществующих объектов недопустим и при выполнении программы приведет к ошибке. Чтобы избавить от лишних ошибок, в объекты ввели предопределенный метод **Free**, который следует вызывать вместо деструктора. Метод Free сам вызывает деструктор Destroy, но только в том случае, если значение объектной переменной не равно **nil**: *math*.Free;

После уничтожения объекта переменная *math* сохраняет свое значение, продолжая ссылаться на место в памяти, где объекта уже нет. Если эту переменную предполагается еще использовать, то желательно присвоить ей значение **nil**, чтобы программа могла проверить, существует объект или нет.

# №7: Система обработки ошибок в модуле.

Нужно проверить, какие возможны ошибки при данном наборе. Собрать реестр ошибок и присвоить каждой свой код. Оформить кодификатор в виде таблицы и поместить его в документацию к модулю. Предусмотреть обработку ошибок. Если при этом есть ошибка, не учтенная в кодификаторе, то он присваивает ей номер.

Кодификатор всегда начинается со строки «0». 0 – нет ошибки.

Ошибка:

- несуществующий вектор (операнда)

- несоответствие размерности

Практическая обработка ошибок

PROCEDURE OnError (place: string);

VAR ec: WORD;

Begin

IF ec <> 0 THEN

BEGIN

WRITELN;

WRITELN(‘Ошибка номер’, ec, ‘в месте’, place);

HALT;

END;

End;

OnError(‘SuccIters: NewI’) ; { Copy , F(x), C

# №8. Среда Delphi, структура программы, модуля. Работа с проектом. Примеры.

Интегрированная Среда Разработки (Integrated Development Environment — IDE) — это среда, в которой есть все необходимое для проектирования, запуска и тестирования приложений и где все нацелено на облегчение процесса создания программ. ИСР интегрирует в себе редактор кодов, отладчик, инструментальные панели, редактор изображений, инструментарий баз данных — все, с чем приходится работать.

Запускаем Delphi с помощью меню Windows Пуск | Программы. Когда мы щелкнем на пиктограмме Delphi, перед нами откроется основное окно Интегрированной Среды Разработки. Для версий Delphi, отличных от Delphi 5, окно выглядит несколько иначе, но в основных чертах окна ИСР всех версий Delphi одинаковы.

В верхней части окна ИСР мы видим *полосу главного меню*. Ее состав несколько различается от версии к версии и, кроме того, зависит от варианта Delphi, с которым мы работаем.

Ниже полосы главного меню расположены две *инструментальные панели*. Левая панель содержит два ряда *быстрых кнопок*, дублирующих некоторые наиболее часто используемые команды меню. Правая панель содержит *палитру компонентов* библиотеки визуальных компонентов (Visual Component Library — VCL). В ней содержатся и визуальные (видимые пользователю), и невизуальные компоненты (они явным образом не видны пользователю). Палитра компонентов содержит ряд страниц, закладки которых видны в ее верхней части.

В основном поле окна мы видеим слева окно Инспектора Объектов (Object Inspector), с помощью которого мы в дальнейшем будем задавать свойства компонентов и обработчики событий. Правее мы можем видеть окно пустой формы, готовой для переноса на нее компонентов. Под ним расположено окно Редактора Кодов. Обычно оно при первом взгляде на экран невидимо, так как его размер равен размеру формы и окно Редактора Кодов полностью перекрывается окном формы.

Создание нового проекта приложения начинается с команды File | New Application. По этой команде открывается новый проект приложения с пустой формой. Сохранить на диске готовый проект или его заготовку можно командой File | Save Project As или File | Save All. Удобно также для сохранения использовать быстрые кнопки — третью или четвертую слева в верхнем ряду. Открыть ранее сохраненный проект можно командой File | Open или File | Open Project (вторая слева быстрая кнопка в верхнем ряду). Но если вы недавно работали с этим проектом, то удобнее воспользоваться командой File | Reopen или кнопочкой справа от быстрой кнопки Open. Эта команда дает возможность быстро выбрать проект из числа тех, с которыми вы работали последнее время.

Для компиляции и запуска на выполнение приложения надо выполнить команду Run | Run (быстрая кнопка с зеленой стрелкой, пятая в нижнем ряду).

Палитра компонентов — витрина библиотеки визуальных компонентов (Visual Component Library — VCL). Чтобы осуществить операцию переноса компонентов из палитры на форму, надо открыть соответствующую страницу библиотеки и указать курсором мыши необходимый компонент. Затем надо сделать щелчок мышью в нужном месте формы и компонент разместится там. В дальнейшем мы можем отбуксировать его в любое другое место.

Форма, на которой размещаются компоненты, является основой почти всех приложений Delphi. Ее можно понимать как типичное окно Windows. Форма является контейнером (*родителем —****Parent***) размещенных на ней компонентов.

Одной из наиболее важных частей среды Delphi является окно Редактора Кода. В действительности, если вы откроете в первый раз это окно в Delphi 5, оно может выглядеть несколько иначе и включать в себя слева еще одно встроенное окно — окно Исследователя Кода Code Explorer. В большинстве случаев вы просто можете закрыть его, щелкнув на кнопке с крестиком в его правом верхнем углу. Чтобы оно вообще больше не появлялось, выполните команду Tools | Environment Options и в открывшемся диалоговом окне на странице Explorer выключите опцию Automatically show Explorer (показывать автоматически окно Code Explorer).

Редактор Кода является полноценным программным редактором. Его можно настраивать на различный стиль работы, который нам более привычен. В редакторе применяется выделением цветом синтаксических элементов. Жирным шрифтом выделяются ключевые слова языка Object Pascal. Синим курсивом выделяются комментарии.

В заголовке окна Редактора Кода отображается имя текущего файла, того, с текстом которого вы работаете. В верхней части окна мы можем видеть также закладки или ярлычки, указывающие текущую страницу. Приложения Delphi могут использовать много исходных файлов и закладки помогают вам переходить от одного из них к другому.

В окно Редактора Кода, как и в другие окна Delphi, встроена контекстная справка. Чтобы получить справку по какому-то слову кода (ключевому слову, написанному имени функции и т.п.) достаточно установить курсор на это слово и нажать клавишу F1. Нам будет показана соответствующая тема справки.

Следующим важнейшим элементом среды разработки является Инспектор Объектов (Object Inspector). Он обеспечивает простой и удобный интерфейс для изменения свойств объектов Delphi и управления событиями, на которые реагирует объект.

Страница свойств (Properties) Инспектора Объектов (см. рис. 1.3 а), показывает свойства того объекта, который в данный момент выделен нами. Щелкнем на окне пустой формы и на странице свойств Инспектора Объектов мы сможем увидеть свойства формы. Мы можем изменять эти свойства. Например, изменить свойство **Caption** (надпись) нашей формы, написав в нем «Моя форма», и вы увидите, что эта надпись появится в полосе заголовка вашей формы.

Страница событий (Events) составляет вторую часть Инспектора Объектов. На ней указаны все события, на которые может реагировать выбранный объект. Например, если нам надо выполнить какие-то действия в момент создания формы (обычно это различные операции настройки), то мы должны выделить событие **OnCreate**. Рядом с именем этого события откроется окно с выпадающим списком. Если мы уже написали в своем приложении какие-то обработчики событий и хотите при событии **OnCreate** использовать один из них, вы можете выбрать необходимый обработчик из выпадающего списка. Если же вам надо написать новый обработчик, то сделайте двойной щелчок на пустом окне списка.

# №9. Компиляция программы Delphi. Работа с несколькими объектами на форме. Процедуры обработки событий. Свойства формы. Примеры.

В процессе компиляции проекта создается готовый к использованию файл, которым может быть приложение (ЕХЕ) или динамически загружаемая библиотека (DLL). Далее будем рассматривать только файл-приложение. Имя приложения, получаемого в результате компиляции, совпадает с име­нем файла проекта, а само приложение является автономным и не требует для своей работы дополнительных файлов **Delphi**.

Запуск процесса компиляции выполняется по команде Project | Compile <Project1> (Проект | Компилировать <проект>) или нажатием комбинации клавиш **<Ctrl>+<F9>**. В этой команде содержится имя проекта, разработка которого выполняется в настоящий момент, первоначально это Projectl. При сохранении проекта под другим именем соответственно должно изме­ниться имя проекта в команде меню.

Компиляция проекта для получения приложения может быть выполнена на любой стадии разработки проекта. Это удобно для проверки вида и пра­вильности функционирования отдельных компонентов формы, а также для проверки отдельных фрагментов создаваемого кода.

При компиляции про­екта выполняются следующие действия:

1. компилируются файлы всех модулей, содержимое которых изменилось со времени последней компиляции. В результате для каждого файла с ис­ходным текстом модуля создается файл с расширением DCU. Если ис­ходный текст модуля по каким-либо причинам недоступен компилятору, то он не перекомпилируется;
2. если в модуль были внесены изменения, то перекомпилируется не только этот модуль, но и использующие его с помощью директивы uses **МОДУЛИ**;
3. перекомпиляция модуля происходит также при изменениях объектного файла    или подключаемого файла (INC), используемых данным модулем;
4. после компиляции всех модулей проекта компилируется файл проекта и создается исполняемый файл приложения с именем файла проекта.

Кроме компиляции может быть выполнена сборка проекта. При сборке компилируются все файлы, входящие в проект, независимо от того, были в них внесены изменения или нет. Для сборки проекта используется команда меню Project | Build <Project1> (Проект | Собрать <проект1>).

Запустить проект на выполнение можно из среды **Delphi** и из среды Windows.

Выполнение проекта из среды **Delphi** осуществляется командой Run | Run (Выполнение | Выполнить) или нажатием клавиши**<F9>**. При этом созданное приложение начинает свою работу. Если в файлы проекта вносились изменения, то предварительно выполняется компиляция проекта. Запущенное приложение работает так же, как и запущенное вне среды **Delphi**, однако имеются некоторые особенности:

* нельзя запустить вторую копию приложения;
* продолжить разработку проекта можно только после завершения работы приложения;

при зацикливании (зависании) приложения его завершение необходимо выполнять средствами **Delphi**с помощью команды Run | Program Reset (Выполнение | Остановить программу) или комбинации клавиш**<Ctrl>+<F2>**.

Из среды Windows созданное приложение можно запустить как и любое другое приложение, например, с помощью Проводника.

В интерактивных системах управления процессом выполнения приложения производится с помощью событий, связанных с действиями пользователя – например, щелчком кнопкой мыши или с нажатием клавиши клавиатуры в процессе выполнения приложения.

*Событие* (Event) – это то, что происходит во время работы программы. В Delphi каждому событию присвоено имя. Например, щелчок кнопкой мыши – это событие OnClick, двойной щелчок мышью событие OnDblClick.

|  |  |
| --- | --- |
| Событие | Что происходит |
| OnClick | При щелчке кнопкой мыши |
| OnDblClick | При двойном щелчке кнопкой мыши |
| OnMouseDown | При нажатии кнопки мыши |
| OnMouseUp | При отпускании кнопки мыши |
| OnMouseMove | При перемещении мыши |
| OnKeyPress | При нажатии клавиши клавиатуры |
| OnKeyDown | При нажатии клавиши клавиатуры. События OnKeyDown и OnKeyPress – это чередующиеся, повторяющиеся события, которые происходят до тех пор, пока не будет отпущена удерживаемая клавиша (в этот момент происходит событие OnKeyUp) |
| OnKeyUp | При отпускании нажатой клавиши клавиатуры |
| OnCreate | При создании объекта (формы, элемента управления). Процедура обработки этого события обычно используется для инициализации переменных, выполнения подготовительных действий |
| OnPaint | При появлении окна на экране в начале работы программы, после появления части окна, которая, например, была закрыта другим окном, и в других случаях |
| OnEnter | При получении элементом управления фокуса |
| OnExit | При потере элементом управления фокуса |

Реакцией на событие должно быть какое-либо действие. В Delphi реакция на событие реализуется как *процедура обработки события.*Таким образом, для того чтобы программа выполняла некоторую работу в ответ на действия пользователя, программист должен написать процедуру обработки соответствующего события. Следует обратить внимание на то, что значительную часть обработки событий берет на себя компонент. Поэтому программист должен разрабатывать процедуру обработки события только в том случае, если реакция на событие отличается от стандартной или не определена. Например, если по условию задачи ограничений на символы, вводимые в поле Edit, нет, то процедуру обработки события OnKeyPress писать не надо, т. к. во время работы программы будет использована стандартная (скрытая от программиста) процедура обработки этого события.

**Action** - определяет объект *TAction*. Это объект служит для быстрой привязки действий к компонентам, в особенности - к пунктам меню и панелям инструментов. Но может быть привязан и к форме. Для управления *TAction* служат редакторы *TActionList* со страницы *Standard* и *TActionManager* со страницы *Additional*.

**ActiveControl** - определяет элемент, который имеет в данный момент фокус ввода. Если выбрать какой-либо объект во время разработки (design-time), то при запуске приложения этот объект и будет иметь фокус ввода. Также свойство может быть полезно и во время выполнения (run-time) - можно узнать, какой объект "держит" фокус в данный момент, а также можно переместить фокус на любой из объектов. Пример: разместим на форме 2 кнопки - Button1 и Button2, а также *TTimer* (страница *System*). Выбрав элемент Timer1, дважды щёлкнем в Инспекторе объектов напротив надписи OnTimer на вкладке Events, т.е. создадим обработчик события и напишем следующее: *ActiveControl:=Button2;* Теперь, запустив программу, каждую секунду фокус будет перемещаться на Button2.

**Align** - определяет выравнивание формы на экране. Свойство принимает одно из следующих значений:

* alBottom - по нижнему краю;
* alClient - вся пользовательская (клиентская) область;
* alCustom - выравнивание определяется вызовом методом объекта-родителя;
* alLeft - по левому краю;
* alNone - без выравнивания;
* alRight - по правому краю;
* alTop - по верхнему краю.

**AlphaBlend** - включает/выключает прозрачность формы.

**AlphaBlendValue** - задаёт степень непрозрачности формы: 0 - форма полностью невидима, 255 - полностью видима. Прозрачность активируется только при установке свойства AlphaBlend в True.

**Anchors** - определяет направления, по которым компоненты "привязываются" к форме. Пример: если установить у формы значения akLeft и akRight этого свойства в True, и точно также сделать у кнопки, то при изменении ширины формы размер кнопки (ширина) также будет изменяться.

**AutoScroll** - включает автоматическое появление полос прокрутки (Scroll bars) на форме, когда размеров формы недостаточно для отображения всех элементов.

**AutoSize** - включает автоматическое изменение размеров формы согласно позициям размещённых на ней элементов.

**BiDiMode** - определяет двунаправленное отображение элемента. В некоторых языках письмо осуществляется не слево-направо, а наоборот. Это свойство создано как раз для этой цели.

**BorderIcons** - определяет множество кнопок, которые отображаются в заголовке окна:

* biSystemMenu - единственный элемент, который не является кнопкой - отвечает за системное меню окна, которое вызывается комбинацией клавиш [Alt]+[Пробел].
* biMinimize - кнопка сворачивания (минимизации) окна;
* biMaximize - кнопка разворачивания окна;
* biHelp - кнопка справки.

Если хотя бы одна из кнопок сворачивания и разворачивания включена, то независимо от состояния другой, отображаются обе (но вторая естественно неактивна). Если выключены обе, они не отображаются вообще. Это не зависит от Delphi - так устроена ОС Windows.

**BorderStyle** - определяет поведение границ окна и общий тип окна:

* bsDialog - диалоговое окно (из кнопок - только "Закрыть", иконки в заголовке окна нет);
* bsNone - "чистый лист" (отсутствие у окна границ и заголовка) - применяется обычно для создания заставок во время запуска программы;
* bsSingle - обычное окно, но с запретом изменения размеров;
* bsSizeable - обычное окно (по умолчанию) - размеры формы можно изменять;
* bsSizeToolWin - упрощённое окно с уменьшенным заголовком;
* bsToolWindow - упрощённое окно с уменьшенным заголовком без возможности изменения размеров.

**BorderWidth** - ширина границы окна в пикселах. Граница является невидимой и расположена в пользовательской части формы.

**Caption** - текст заголовка формы.

**ClientHeight**, **ClientWidth** - размер клиентской (пользовательской) части формы, т.е. той, на которой располагаются компоненты.

**Color** - цвет формы.

**Constraints** - определяет минимальные и максимальные размеры высоты и ширины формы в пикселах. 0 - любое значение, т.е. без ограничений.

**Ctl3D** - свойство определяет 3D-вид формы. При выключенном - "плоское" изображение.

**Cursor** - курсор мыши в тот момент, когда он находится над формой.

**DefaultMonitor** - определяет, на каком мониторе появится форма. Имеет смысл применять это свойство только при наличии более, чем одного монитора (например, если несколько экранов).

**DockSite**, **DragKing** и **DragMode** - определяют поведение формы при осуществлении операций Drag&Drop.

**Enabled** - отвечает за общую активность формы. Если установлено в False, форма недоступна.

**Font** - шрифт, используемый на форме.

**FormStyle** - стиль формы или её поведение в MDI-приложении (многооконное приложение, где дополнительные формы располагаются "внутри" основной формы). Значения:

* fsNormal - обычная форма (значение по умолчанию);
* fsMDIChild - дочерняя (подчинённая) форма MDI-приложения;
* fsMDIForm - главная форма MDI-приложения;
* fsStayOnTop - форма находится поверх всех окон на экране.

**Height** - высота формы в пикселах. В отличие от ClientWidth является высотой с учётом заголовка и границ формы.

**HelpContext**, **HelpFile**, **HelpKeyword**, **HelpType** - свойства для связи формы с файлом справки в форма

**Hint** - текст всплывающей подсказки.

**HorzScrollBar** - свойство определяет внешний вид и поведение горизонтальной полосы прокрутки окна.

**Icon** - значок (иконка) формы. Отображается в заголовке слева от заголовка. Задаётся файлом в формате *\*.ico*.

**KeyPreview** - если свойство установлено в True, то при нажатии клавиш сначала будут вызываться обработчики формы, а только затем обработчики того компонента, который в данный момент имеет фокус ввода. События, связанные с нажатием клавиш - OnKeyDown(), OnKeyPress(), OnKeyUp().

**Left** - позиция формы на экране (левого верхнего угла) в пикселах.

**Menu** - позволяет выбрать один из компонентов-меню, который станет главным меню окна, т.е. будет отображаться вверху.

**Name** - имя формы как объекта. Может содержать только латинские буквы, цифры и знак подчёркивания, и не может начинаться с цифры. Фактически, это то имя, по которому в программе можно обратиться к форме.

**ObjectMenuItem** - используется при работе с OLE-объектами и позволяет связать пункт меню и OLE-объект: когда объект выделен, пункт меню активен и наоборот.

**OldCreateOrder** - определяет, когда происходят события OnCreate() и OnDestroy() формы. Если установлено в False, то OnCreate() произойдёт после вызова всех конструкторов, а OnDestroy() - после вызова всех деструкторов. Начальное значение - False, изменять не рекомендуется.

**ParentBiDiMode** - изменение свойства BiDiMode согласно значению объекта-предка формы.

**ParentFont** - изменение шрифта (Font) согласно значению объекта-предка.

**PixelsPerInch** - пропорции шрифта в системе (точек на дюйм).

**PopupMenu** - позволяет указать контекстное меню (объект TPopupMenu) для формы. Это меню вызывается нажатием правой кнопки мыши.

**Position** - определяет начальную позицию формы на экране, т.е. в момент её появления. Основные значения:

* poDesigned - появление в том месте, в каком форма расположена в design-time;
* poDesktopCenter - по центру рабочего стола (рекомендуемое значение);
* poScreenCenter - по центру экрана;
* poMainFormCenter - по центру главной формы приложения (для главной формы не имеет смысла).

**PrintScale** - определяет размеры формы при выводе её изображения на печать.

**Scaled** - включает масштабирование формы в соответствии с заданным значением свойства PixelsPerInch.

**ScreenSnap** - если установлено в True, то форма будет автоматически "прилипать" к краям экрана в момент перемещения.

**SnapBuffer** - определяет расстояние (в пикселах), на котором форма будет "прилипать" к краю экрана.

**ShowHint** - включает/выключает показ всплывающей подсказки (Hint).

**Tag** - специальное свойство, которое есть у всех объектов. Специального применения для этого свойства нет, поэтому оно используется для разных целей в конкретной ситуации. Свойство удобно в том случае, если нужно хранить некоторое целое число - не придётся заводить дополнительную переменную.

**Top** - позиция формы (левого верхнего угла) на экране в пикселах.

**TransparentColor** - включает/выключает прозрачность определённого цвета формы.

**TransparentColorValue** - задаёт цвет, который будет прозрачным.

**UseDockManager** - используется при реализации Drag&Drop технологии, предоставляя дополнительные возможности этого метода взаимодействия.

**VertScrollBar** - определяет внешний вид и поведение вертикальной полосы прокрутки окна.

**Visible** - определяет видимость формы на экране.

**Width** - ширина окна в пикселах, включая границы.

**WindowMenu** - свойство-аналог свойства Menu, но используемое при создании MDI-форм.

**WindowState** - одно из состояний окна:

* wsNormal - обычное состояние (занимает часть экрана);
* wsMinimized - окно свёрнуто;
* wsMaximized - окно развёрнуто на весь экран.

# №10. Классы, инкапсуляция и наследование, полиморфизм. Примеры.

В объектно-ориентированном программировании объекты объединены в классы. Класс - это проект, план строения конкретных объектов. Класс определяет, какие у данных объектов будут переменные состояния, и как они будут изменяться, какие будут методы и как они будут реализовываться. Класс нематериален, он не соответствует каким-то данным. В классе описано, какие переменные и с какими значениями могут быть у объекта. Могут быть, а не есть в данный момент. Ведь у разных объектов, относящихся к этому классу, значения этих переменных в данный момент, возможно, различны.

Объект же, наоборот, материален. Он занимает место в памяти компьютера, все его переменные имеют в каждый момент вполне определенные значения. Именно для конкретных объектов вызываются методы, хотя сами методы описаны в классе, к которому принадлежит данный объект.

Для этого служит механизм наследования - один из трех основных принципов объектно-ориентированного программирования.

Наследование дает огромные преимущества и значительно облегчает программирование. Оно позволяет не только экономить время при создании новых классов, но и дает возможность неограниченного расширения и совершенствования уже существующих классов. Можно придать новые функции классу, даже не зная деталей реализации самого объекта класса.

Некоторые свойства и методы доступны внешним объектам, а некоторые целесообразно спрятать для внутреннего пользования.

Этот принцип сокрытия - второй принцип объектно-ориентированного программирования, называемый инкапсуляцией.

Одни переменные и методы, определяемые в разделе Интерфейс, доступны для чтения, изменения и вызова, называются опубликованными, а другие, определяемые в разделе Реализация, служат исключительно нуждам самого объекта и недоступны извне, они называются приватными.

При наследовании все параметры и методы, описанные в родительском классе, переходят без изменений в класс-потомок. Но не всегда требуется такое буквальное копирование. Иногда просто необходимо переопределить уже существующий метод или параметр. Для того чтобы при вызове одинакового метода различные потомки вели себя по-разному, требуется замещение метода, описанного в классе-предке.

Замещать можно и переменные. В подобном разноликом поведении и заключается третий принцип объектно-ориентированного программирования - Полиморфизм.

С помощью полиморфизма потомки классов получают возможность переопределять действия, заложенные в предках.

Например, графические компоненты управления в Delphi все наследуют от своего предка метод Перерисовки, но каждый из них его замещает. В результате, когда Windows решает, что пора обновить окно и вызывает для них метод Перерисовать, каждый компонент перерисовывается по-своему.

В Delphi все классы наследуются от одного предка - базового класса TObject. Этот класс не только дает всем своим потомкам ряд нужных всем им методов, например, методы создания и уничтожения экземпляров класса, но и обеспечивает базовую совместимость любых объектов. Все компоненты в Delphi - объекты. Можно создавать новые компоненты в

Delphi.

События используются в Delphi для обеспечения определения программистом реакции на те или иные действия или запросы со стороны компонента.

Хотя описание обработчиков событий, по сути, является всего лишь полиморфным замещением методов, события в Delphi играют очень важную роль. Фактически вся программа - это описание обработчиков событий.

Программа Delphi не является сплошным последовательным выполнением кода, как в Turbo Pascal. Здесь программа реализует некоторый код только как реакцию на события -какие-то действия пользователя (нажатие кнопок, движение мыши, закрытие окон и т.п.). Когда код, описанный в обработчике, заканчивается, программа не завершается. Для завершения требуется, чтобы пользователь обычным в Windows способом закрыл главное окно приложения либо, к примеру, нажал на сделанную кнопку, в обработчике нажатия которой предусмотрен вызов процедуры Close.

# 11. Компоненты ввода, вывода, выполнения действий программы Delphi. Процедура-обработчик события создания формы. Примеры.

EDIT- однострочный редактор для ввода, вывода и модификации данных

MEMO- многострочный текстовый редактор для вывода, ввода и изменения

LABEL- метка, для размещения заголовков, и других поясняющих надписей

STATICTEXT- метка, для размещения надписей в рельефной рамке

STRINGGRID- таблица строк, используется для ввода/вывода данных в виде двумерной таблицы.

Каждый компонент получает свое имя (имя компонента и номер, например EDIT1 – первый однострочный редактор, размещенный на форме, EDIT2 - второй однострочный редактор, размещенный на форме и т.д.)

Доступ к компонентам осуществляется с помощью соответствующих свойств компонентов, причем для каждого компонента эти свойства могут быть разными.

Имя компонентаСвойство для ввода/вывода

EDITTEXT(в программе записывается EDIT1.TEXT)

MEMOTEXT(в программе записывается MEMO 1.TEXT)

LABELCAPTION (только вывод)(LABEL1.CAPTION)

STATICTEXTCAPTION (только вывод)(STATICTEXT1.CAPTION)

STRINGGRIDмассив CELLS [номер столбца, номер строки] в программе записывается STRINGGRID1.CELLS[ j,i ]

Для компонента MEMO часто используется свойство «строка» LINES в сочетании с методом «добавить» ADD. ( в программе записывается MEMO1.LINES.ADD ( ) ).

Указанные свойства этих компонентов имеют строковый тип, т.е. тип STRING, поэтому при вводе числовых данных необходимо выполнить соответствующие преобразования данных

при вводе надо перевести данные из строкового типа в числовой, а именно в целочисленный тип или вещественный тип с помощью функций STRTOINT ( ) или STRTOFLOAT ( );

при выводе данные надо перевести из целочисленного или вещественного типа в строковый с помощью функций INTTOSTR ( ) или FLOATTOSTR ( )

Для выполнения этих операций используется оператор присваивания « := »языка Паскаль.

В левой части оператора записывается имя компонента и его свойство, разделителем является точка, а в правой части – соответствующее значение.

В правой части можно записать любые данные строкового типа, т.е. записанные в апострофах или имя компонента и через точку его свойство, откуда берутся данные.

Для многострочного редактора существует несколько способов вывода данных:

способ 1Memo1.Text := ‘вывод с потерей всего выведенного ранее’;

способ 2Memo1.Text := Memo1.Text + ‘вывод с сохранением выведенного ранее’;

способ 3Memo1.Lines.Add (‘ добавить строку и вывести все, что в апострофах‘) ;

Способы 1 и 2 не вводят новой строки.

Примеры

Вывод в поле метки 1 слова МоскваLabel1.Caption := ‘ Москва ‘;

Вывод в поле метки 2 числа 25Label2.Caption := ‘ 25 ‘ ;

Вывод в поле метки 1 числа 10,23StaticText1 := ' 10.23 ‘ ;

Ввод в переменную А числа 5 из компонента Edit1А:= strtoint(Edit1.Text)

Вывод в поле Memo1 значения переменной А Memo1.Text :=inttostr (A) ;

Если выполнить вывод в поле Memo1 из компонента Edit1, то записывается команда Memo1.Text := Edit1.Text

5

- компонент Edit1

Ввод в переменную массива В[i,j] целочисленного значения из ячейки StringGrid1

B [ i,j ] :=StrToInt ( StringGrid1.Cells[ j,i ] );

Организация вывода элементов одномерного массива

Элементы одномерного массива можно вывести в разные компоненты.

1. Вывод в компонент Edit

For I := 1 to M do

Edit1.Text:=Edit1.Text + ' '+ IntToStr(A[I])

2. Вывод в компонент Label1

For I := 1 to M do

Label1.Text:=Label1.Text + ' '+ IntToStr(A[I])

3. Вывод в компонент StaticText

For I := 1 to M do

StaticText1.Caption := StaticText1.Caption + IntToStr (A[I]) + ' ';

4. Вывод в компонент Memo

4.1. В строку:

For I := 1 to M do

Memo1.Text:=Memo1.Text + ' '+ IntToStr (A[I]) { использование свойства

Text компонента Memo}

4.2. В столбик

For I := 1 to M do

Memo1.Lines.Add ( IntToStr (A[I]) ) {использование метода Add –

добавление новой строки в список}

5. Вывод одномерного массива Sum в столбец № 1 ком-та stringgrid1

for i := 1 to m do

stringgrid1.Cells[ 1, i ]:= floattostr(sum [ i ] );

В строку № 1 компонента Stringgrid :

for j := 1 to n do

stringgrid1.Cells[ j, 1 ]:= floattostr(sum [ j ] );

Организация вывода элементов двумерного массива

1. Вывод матрицы В в stringgrid2

for i := 1 to m do

for j:=1 to n do stringgrid2.cells[ j,i ]:=floattostr (b [ i,j ]);

2. Вывод матрицы В в компонент Memo с расчетом позиций чисел и в файл 'd:\rez.txt'

var ff :textfile ;

d,p,z,i,j:integer; // позиции, счетчик пробелов, длина числа

ss,aa :string; // промежуточные строки ss –строка матрицы ;

// аа-строка для числа

begin

assignfile (ff,'d:\rez.txt'); // связь с файлом на диске

rewrite (ff); // открыть файл для записи

ss:='';

d:=8; // 8 позиций на число

memo2.Lines.Add('матрица В') ; // вывод в memo2

writeln(ff,'матрица В') ; // вывод в файл

for I:=1 to m do begin

For j:=1 to n do

Begin aa:= floattostr (b[i,j]); //расчет длины числа и добавление пробелов

// до 8 символов (на 1 число)

z:= d - length (aa);

if b[i,j] <0 then z:=z+1;

for p:=1 to z do SS:=ss+' '; // 1 пробел между апострофами

ss := ss + floattostr (b[i,j]) +' '; // 2 пробела между апострофами

end;

memo2.Lines.Add(ss) ;

writeln(ff,ss) ;

ss:=''; // без пробелов между апострофами

d:=8; end;

closefile(ff);// закрыть файл

end;

end.

Вид файла D : \ rez.txt с результатами :

матрица В

10,7 2,2 3,3 4,1

10,5 6,3 1,1 -2,1

1,1 5 0,3 -1,1

Найдите событие onCreate - создание формы. Для того чтобы создать процедуру (все события описываются в процедурах) нам следует поместить курсор мыши на сроку ввода события onCreate и двойным щелчком мыши вызвать редактор кода. В редакторе кода, Delphi сам создал текст заголовка процедуры и выглядит он так:

procedure TForm1.FormCreate(Sender: TObject);

begin

end;

В разделе типов, в объявлении объекта Form1 появилось имя процедуры:

Procedure FormCreate(Sender: TObject); - поднимитесь вверх по тексту и проверьте так ли это.

Вернемся к нашей процедуре создания окна и сами напишем реакцию на создание формы. Изменим свойство - цвет окна. Между словами begin и end напишем следующую строку:

Form1.Color:= clNavy;

Запустим программу на исполнение - цвет окна изменился на тёмно- синий.

# 12. Компоненты выбора, выполнения действия программы Delphi. Организация работы с различными типами данных.

Для выполнения множественного выбора используются переключатели.

Delphi для работы с переключателями предлагает следующие компоненты:

* переключатель с независимой фиксацией (CheckBox), флажок этой компоненты можно переключать щелчком мыши;
* переключатели с зависимой фиксацией – RadioButton (кнопки выбора), RadioGroup  (группа переключателей RadioButton).

Если в группе зависимых переключателей выбран один, то в отличие от независимого переключателя, его состояние нельзя изменить повторным щелчком. Для отмены выбора зависимого переключателя нужно выбрать другой переключатель из этой группы.

Компонент **Delphi CheckBox** это флажок - независимый переключатель. Флажок **Delphi CheckBox**используется в программе для визуализации состояний включено-выключено. Каждый щелчок мышкой по компоненту **Delphi CheckBox** меняет его состояние на противоположное. Смена состояния переключателя также доступна и в программе.  
  
   **Delphi CheckBox** представляет собой сочетание небольшого окошка, которое и визуализирует наличием или отсутствием "галочки" состояние компонента, и [компонента Label](http://www.delphi-manual.ru/label.php) - заголовка, поясняющего назначение переключателя.

Основное свойство компонента CheckBox - **Checked** типа Boolean. То есть, на приведённом рисунке  
   CheckBox1.Checked = False;  
   CheckBox2.Checked = True;

Помимо свойства Checked, позволяющего управлять состоянием компонента, у Delphi CheckBox есть свойство State (состояние), которое может иметь уже три значения. Первые два значения cbChecked ("отмечено") и cbUnChecked ("не отмечено"), а третье - cbGrayed ("не определено" или "не знаю") становится доступным для выбора если установить в True свойство AllowGrayed компонента. Свойства Checked и State связаны между собой:

если свойство State равно cbChecked, свойство Checked = True;

если свойство State равно cbUnchecked или cbGrayed, свойство Checked = False.

При щелчке мышкой эти три состояния сменяют друг друга в следующей последовательности:

cbGrayed - не определено (или "не знаю");

cbChecked - отмечено;

cbUnChecked - не отмечено;

Компонент Delphi RadioGroup расположен на странице Standard Палитры Компонентов. Он представляет собой контейнер с группой компонентов RadioButton (переключателей). Такой компонент упрощает организацию работы переключателей, по сравнению другими групповыми контейнерами требующими добавлять переключатели вручную.

Компонент RadioGroup может содержать и другие компоненты, такие как редактор Edit или флажок CheckBox. Но автоматическое форматирование и распределение элементов на них не будет распространяться. В результате внешний вид приложения может пострадать тем, что элементы если и не будут перекрывать друг друга, то будут неравномерно располагаться относительно переключателей.

Свойство Columns устанавливает из скольких столбцов будет состоять группа (по умолчанию 1). От этого свойства зависит расположение переключателей, а другие компоненты, например, на компонент CheckBox, который находиться в группе RadioGroup.

Основное свойство компонента RadioGroup это Items. В нем находятся строки, которые являются заголовками переключателей. Двойным щелчком на многоточии рядом с Items в Инспекторе Объектов мы попадаем в редактор списков строк, где каждая отдельная строка будет заголовком переключателя (радиокнопки). Отсчет строк идет с нуля. Элемент Items[2] будет третьим. Для работы с элементами используются методы: Add, Delete, Clear.

Свойство Columns зададим равным двум.

Поместим компонент CheckBox в наш контейнер RadioGroup. Это мы делаем для того чтобы понять, какие визуальные неудобства он может принести при совместном использовании.

Пункты с 1 по 6 введем в редакторе строк. Щелкнув в Инспекторе Объектов на многоточие напротив Items.

Для кнопки Добавить введем такой обработчик:

RadioGroup1.Items.Add('семь');

RadioGroup1.Items.Add('восемь');

Для кнопки Очистить:

RadioGroup1.Items.Clear;

Для кнопки Удалить:

RadioGroup1.Items.Delete(RadioGroup1.ItemIndex);

Для кнопки Показать:

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

Запустим нашу программу. Нажав кнопку Добавить мы получим два новых элемента в нашей группе. Нажимая так несколько раз элементов может стать достаточно много. И компонент CheckBox будет мешать обзору, так как он не будет автоматически перемещаться как родные элементы компонента RadioGroup.

Кнопка Удалить удаляет текущий переключатель (радиокнопку).

Кнопка Очистить удаляет все родные переключатели посторонние компоненты не трогает. Компонент CheckBox останется на своем месте как и стоял.

Кнопка Показать с помощью компонента Label1 показывает нам номер номер выбранной радиокнопки. Если выбор не сделан то будет -1. Нумерация переключателей (радиокнопок) начинается с 0.

# 13. Массивы в Delphi. Компонент StringGrid. Примеры.

Массив — это структура данных, представляющая собой набор переменных одинакового типа, имеющих общее имя. Массивы удобно использовать для хранения однородной по своей природе информации, например, таблиц и списков.

Объявление массива

Массив, как и любая переменная программы, перед использованием должен быть объявлен в разделе объявления переменных. В общем виде инструкция объявления массива выгладит следующим образом:

Имя: [нижний\_индекс..верхний\_индекс] of тип

где:

имя — имя массива;

array — зарезервированное слово языка Delphi, обозначающее, что объявляемое имя является именем массива;

нижний\_индекс и верхний\_индекс — целые константы, определяющие диапазон изменения индекса элементов массива и, неявно, количество элементов (размер) массива;

тип — тип элементов массива.

Примеры объявления массивов:

temper:array[1..31] of real;

koef: array[0..2] of integer;

name: array[1..30] of string[25];

Компонент Delphi StringGrid (Таблица строк) расположен на странице Additional Палитры компонентов. Использование электронных таблиц пользователями компьютеров стало привычным и достаточно удобным средством в работе с данными. С помощью компонента StringGrid создаются таблицы, в ячейках которых содержаться произвольные текстовые данные. Он является прямым потомком компонента DrawGrid, много свойств и методов унаследовано от него.

Основное свойство компонента StringGrid - это двумерный массив Cells[ACol, ARow : integer] типа String с помощью него можно обращаться к содержимому ячеек и изменять их содержание. ACol устанавливает номер столбца, ARow - задает строку в таблице. Этим свойством можно воспользоваться только во время выполнения программы, оно доступно для записи и чтения.

Нумерация строк и столбцов идет с нуля. Свойство ColCount задает число столбцов, RowCount - число строк.

Компонент StringGrid по умолчанию слева и сверху имеет заголовочные области, они имеют серый цвет. Количество подобных строк и столбцов устанавливается в свойствах FixedCols и FixedRow. Если же для таблицы не нужны заголовочные области, то эти свойства должны быть равны нулю.

Для закрепления материала давайте создадим приложение, которое покажет нам таблицу умножения. На форму шириной (Width) равной 365 и высотой (Height) 389 поместим компонент StringGrid. Обработчик onCreate формы (возникает двойным щелчком мыши по форме) будет таким:

procedure TForm1.FormCreate(Sender: TObject);

var

x, y : integer;

begin

// Задаем число столбцов и строк таблицы

StringGrid1.ColCount := 10;

StringGrid1.RowCount := 10;

// Задаем шируну и высоту ячеек

StringGrid1.DefaultColWidth := 30;

StringGrid1.DefaultRowHeight:= 30;

// раставляем цифры от 1 до 9 в

// горизонтальной области заголовка

for x := 1 to StringGrid1.ColCount - 1 do

StringGrid1.Cells[x, 0] := IntToStr(x);

// растравляем цифры от 1 до 9 в

// вертикальной области заголовка

for y := 1 to StringGrid1.ColCount - 1 do

StringGrid1.Cells[0, y] := IntToStr(y);

// выводим результаты умножения в ячейки таблицы

for x := 1 to StringGrid1.ColCount - 1 do

for y := 1 to StringGrid1.RowCount - 1 do

StringGrid1.Cells[x, y] := IntToStr(x\*y);

end;

# 14. Работа со строками. Компоненты ListBox, ComboBox, BitBtn. Обработка событий. Примеры.

Компонент ListBox - это массив строк. ListBox может загружать данные с диска, и сохранять информацию в файл. Также ListBox может сортировать строки. Доступ к строкам предоставляет свойство Items этого компонента. В соответствии с этим, вот как происходит обращение к строке с номером i:

ListBox1.Items[i-1]; // Минус один, так как нумерация строк в компоненте начинается с нуля.

Прежде всего рассмотрим методы добавления строк в компонент Delphi ListBox:

Считывание из файла;

Добавление строки в конец списка;

Добавление строки перед строкой с номером i;

Добавление строк на этапе конструирования.

Методы добавления строк в компонент Delphi ListBox

Считывание из файла

Компонент Delphi ListBox может обращаться напрямую к текстовому файлу как для считывания информации из файла, так и для сохранения всех своих строк в текстовый файл. Каждая запись в файле будет записана в виде одной строки компонента, и при сохранении каждая строка будет сохранена в виде одной записи файла:

ListBox1.Items.LoadFromFile('Имя\_Файла'); // Процедура считывания из файла

ListBox1.Items.SaveToFile('Имя\_Файла'); // Процедура записи в файл

Добавление строки в конец списка

Программист может последовательно добавлять строки в компонент, не заботясь об их нумерации, и они будут размещаться в конец списка:

ListBox1.Items.Add('Новая\_Строка'); //Добавление строк в конец списка

Так как количество строк в компоненте равно ListBox1.Items.Count, то новая (последняя) строка имеет номер ListBox1.Items.Count-1. Это объясняется тем, что нумерация строк начинается от 0.

Добавление строки перед строкой с номером i

Программист может разместить новую строку среди существующих строк там, где ему необходимо. Для этого нужно воспользоваться методом Insert, и указать номер строки, перед которой необходимо записать новую строку:

ListBox1.Items.Insert(i, 'Новая\_Строка'); // Добавление строки перед строкой с номером i

При этом новая строка получает номер i.

Добавление строк на этапе конструирования

В Инспекторе Объектов зайдите на строчку Items и нажмите на появившуся кнопку. Появится окно редактора содержимого, где и можно внести нужную информацию. После нажатия кнопки OK содержимое редактора появится в компоненте ListBox на Форме.

Пользователь может выделить строку, щёлкнув по ней мышкой. Номер выделенной строки возвращает свойство компонента ListBox.ItemIndex. То есть, получить текст выделенной строки можно так:

S := ListBox1.Items[ListBox1.ItemIndex];

Не забывайте пользоваться оператором присоединения with. С его использованием это присваивание выглядит так:

with ListBox1 do

begin

S := Items[ItemIndex];

end;

При многократном использовании свойств компонента данная запись очень экономит силы. Кстати, оператор присоединения допускает работу с несколькими компонентами, которые нужно разделить запятой. Например, присваивание строки компонента ListBox ячейке таблицы StringGrid выглядит так:

with ListBox1, StringGrid1 do

begin

Cells[i, j] := Items[ItemIndex];

end;

Для удаления строк из компонента применяется метод Delete. Удаление строки с номером i:

ListBox1.Items.Delete(i);

Чтобы при добавлении каждая строка была отсортирована (сортировка идёт только по возрастанию), нужно установить:

ListBox1.Sorted := True;

Также возможно установить это свойство на этапе конструирования в Инспекторе Объектов.

К сожалению, компонент не обладает возможностью добавлять только уникальные данные. Один и тот же текст запишется несколько раз. Для реализации этой возможности приходится дополнительно применять невизуальный аналог компонента ListBox - объект StringList, обладающий такой возможностью:

procedure TForm1.Button1Click(Sender: TObject);

var SL: TStringList;

S: String;

begin

SL:=TStringList.Create; // Создаём список строк типа TStringList

SL.Sorted:=True; // Устанавливаем свойство - сортировать строки

SL.Duplicates:=dupError; // Устанавливаем свойство - дубликаты вызывают ошибку добавления

try

SL.Add(S); // Пробуем добавить строку в объект StringList

ListBox1.Items.Add(S); // Добавляем строку в компонент ListBox

except

Caption:='Попытка добавить дубликат';

// Попытка добавить дубликат вызовет ошибку в первой строке секции try/except,

// прерывание и переход к секции except/end.

// Добавления в ListBox не будет.

end;

end;

Компонент ListBox автоматически добавит полосу прокрутки, если количество строк не помещается по высоте компонента. Высота каждой строки равна ItemHeight. Если нужно чтобы при добавлении новой строки полоса прокрутки точно не возникла, этот размер нужно добавить к высоте компонента. Но делать это нужно только начиная с определённого количества строк. Поэтому нужно делать проверку:

ListBox1.Items.Add('Новая\_строка');

with ListBox1 do

if (Items.Count > (Height/ItemHeight))

then Height := Height + ItemHeight;

Но если высота компонента не нацело делится на высоту строки, то прокрутка всё равно возникнет. Поэтому нужно отнять единичку:

ListBox1.Items.Add('Новая\_строка');

with ListBox1 do

if (Items.Count > (Height/ItemHeight)-1)

then Height := Height + ItemHeight;

Однако на долю секунды полоса прокрутки всё-таки появляется. Поэтому сначала сделаем проверку, а затем уже будем добавлять строку. В этом случае уже нужно отнимать не 1, а 2:

with ListBox1 do

if (Items.Count > (Height/ItemHeight)-2)

then Height := Height + ItemHeight;

ListBox1.Items.Add('Новая\_строка');

Компонент **Delphi ComboBox** представляет собой комбинацию списка строк [ListBox](http://www.delphi-manual.ru/listbox.php) со строкой ввода[Edit](http://www.delphi-manual.ru/edit.php). При этом "список строк" компонента **Delphi ComboBox** вначале скрыт, и раскрывается при щелчке мышкой по треугольничку раскрытия, который находится справа в строке ввода:

Таким образом, с помощью **Delphi ComboBox** место на Форме экономится для расмещения других элементов интерфейса программы. А при необходимости раскрытие списка строк можно вообще запретить.  
  
   Многие свойства и возможности компонента **Delphi ComboBox** по работе со строками (объектами **Items**) такие же как и у компонента [Delphi ListBox](http://www.delphi-manual.ru/listbox.php), поэтому повторно описывать их не буду. Опишу отличия и дополнительные возможности.  
  
   Итак, за возможность раскрытия компонента ComboBox отвечает стиль - свойство **Style**:

Стиль компонента ComboBox1 равен

ComboBox1.Style = csDropDown

(список может быть раскрыт, можно писать в строке ввода)

Стиль компонента ComboBox2 равен

ComboBox1.Style = csSimple

(список не может быть раскрыт, можно писать в строке ввода)

При стиле csDropDownList компонент может быть раскрыт, но свойство Text может принимать значения только одной из строк, сохранённых в компоненте, то есть пользователь лишён возможности писать в строке ввода.

Количество строк, видимых при раскрытии списка, равно DropDownCount. Если реальное количество сохранённых строк больше этого количества, то автоматически появляется полоса прокрутки. Программист может заставить список раскрыться в нужный момент. Для этого нужно свойству DroppedDown присвоить значение True. Для этой же цели пользователь может использовать клавиатурную комбинацию ALT+ВНИЗ.

Максимально допустимое количество символов в текстовой строке задаётся параметром MaxLength, причём значение 0 означает отсутствие ограничений. Свойство CharCase управляет преобразованием вводимого текста к верхнему (значение CharCase = ecUpperCase) или нижнему (значение CharCase = ecLowerCase) регистру. Значение ecNormal означает, что текст вводится без преобразования.

Основными событиями компонента Delphi ComboBox являются:

Событие Условие генерации

OnChange Изменился текст в строке ввода.

OnDropDown Список раскрывается. Это событие нужно обработать, если содержимое списка может изменяться во время работы программы. Тогда в обработчике этого события можно заново сформировать содержимое списка.

# 15. Архитектура приложения Delphi. Компонент Mainmenu. Примеры.

Delphi поддерживает создание приложений, использующих одну и\или несколько форм.

Каждое приложение имеет основную форму, появляющуюся изначально, когда программа начнет выполняться.

Формы как диалоговые окна делятся на модальные и немодальные.

**Модальные** диалоговые окна Modal должны быть закрыты прежде, чем пользователь сможет обращаться к любому уже существующему окну приложения, так как модальные окна имеют дело с критическими данными, которые должны быть определены, прежде чем продолжится работа программы.

**Немодальные** диалоговые окна Modeless не обрабатывают критических данных и поэтому дают возможность обратиться к уже существующим окнам программы.

Компонент **MainMenu** (страницы Standart Палитры компонентов) позволяет поместить главное меню в программу в виде иконки. Иконки данного типа называют "невидимыми компонентами", поскольку они невидимы во время выполнения программы.

Для создания меню следует поместить MainMenu на форму, вызвать Дизайнер меню через свойств Items в Инспекторе объектов, определить пункты меню в Дизайнере Меню. Для создания новых опций необходимо выбирать строку справа, для создания подопций - снизу. Для определения символа быстрого доступа к опции перед ним ставится символ &. Для вставки разделительной черты очередной элемент называется "-", для создания разветвленных меню - Ctrl+ **→.** Каждый элемент меню является объектом класса TmenuItem и обладает следующими особенными свойствами:

**Break: TMenuBreak** позволяет создать многоколончатый список подменю.

**Checked: Boolean** если **True,** рядом с опцией появляется галочка.

**Command: Word** используется при разработке приложений, обращающихся непосредственно к API-функциям Windows.

**Count: Integer** содержит количество опций в подчиненном меню, связанном с данным элементом.

**Default: Boolean** определяет, является ли данная опция подменю умалчиваемой, которая выделяется цветом и выбирается двойным щелчком на родительской опции.

**GroupIndex: Byte** определяет групповой индекс для зависимых опций.

**Items [Index: Integer]: TMenuItem** позволяет обратиться к любой опции подчиненного меню по ее индексу.

**MenuIndex: Integer** определяет индекс опции в списке в списке Items родительской опции.

**RadioItem: Boolean** определяет, зависит ли данная опция от выбора других опций в той же группе GroupIndex. Только одна опция группы может иметь True в свойстве Checked. Рядом с такой опцией вместо галочки изображается круг.

**Shortcut: TShortcut** задает клавиши быстрого выбора данных.

**16. Компоненты OpenDialog SaveDialog.**

Для работы с файлами предназначены стандартные процедуры и функции, имеющие те же названия и аналогичные значения, что и в языке программирования Паскаль.

Компоненты OpenDialog и SaveDialog находятся на закладке Dialogs Палитры компонентов, предназначенной для вызова стандартных диалогов Windows. Все компоненты этой страницы являются невизуальными, т.е. не видны в момент работы программы, поэтому их можно разместить в любом удобном месте формы. Внешний вид диалогов зависит от используемой версии Windows.

Диалоги Windows в порядке появления на странице Dialogs: OpenDialog - открыть файл; SaveDialog - сохранить файл; FontDialog - настроить шрифт; ColorDialog - выбор цвета; PrintDialog - печать; PrinterSetupDialog - настройка принтера; FindDialog - поиск строки; ReplaceDialog - поиск с заменой.

Компоненты OpenDialog и SaveDialog имеют идентичные свойства и отличаются только внешним видом. После вызова компонента появляется диалоговое окно, с помощью которого выбирается имя программы и путь к ней. В случае успешного завершения диалога логическая функция (метод) Execute приобретает значение True, имя выбранного файла и маршрут поиска будут содержаться в свойстве FileName.

Для фильтрации файлов, отображаемых в окне просмотра, используется свойство Filter, а для задания расширения файла, в случае, если оно не задано пользователем, - свойство DefaultExt. Для изменения заголовка диалогового окна используется свойство Title.

17. Компонент Timer. Основные свойства. Тип TDataTime. Примеры.

Компонент Delphi Timer очень простой компонент, который не виден на экране, но, тем не менее, Timer Delphi выполняет очень важные функции в программе. Delphi Timer позволяет вводить необходимые задержки между выполнением тех или иных действий.

Компонент Timer имеет всего четыре свойства и одно событие, и работать с компонентом Delphi Timer очень просто.

Delphi Timer

СВОЙСТВО НАЗНАЧЕНИЕ

Enabled "Включение-выключение" таймера

Interval Интервал срабатывания (в миллисекундах)

Name Имя компонента в программе

Tag Произвольный числовой параметр

Помещаем компонент Delphi Timer на Форму. Задаём в свойстве Interval нужный интервал времени (измеряется в миллисекундах). Переходим на вкладку Events и видим единственное событие, поддерживаемое компонентом Delphi Timer: OnTimer. Выполнив по нему двойной щелчёк, или также двойной щелчёк по самому компоненту, мы попадём в сформированный средой Delphi обработчик события, где и введём код, предусматривающий выполнение тех или иных действий.

Так как по умолчанию сойство Enabled установлено в True, то в программе через установленный в свойстве Interval промежуток времени таймер сработает, то есть выдаст событие OnTimer. Будут выполнены необходимые действия. Иногда же запланированные действия должны произойти не автоматически при старте программы, а при выполнении каких-либо других действий. В этом случае необходимо свойство Enabled в Инспекторе Объектов установить в False. Затем в необходимый момент нужно выполнить команду:

Timer1.Enabled:=True;

Учтите, что пока Timer1.Enabled равно True, компонент продолжит генерировать событие OnTimer по истечении каждого промежутка времени, равного значению свойcтва Interval. Поэтому, если нужно только единичное срабатывание, то таймер нужно остановить, причём сразу же:

procedure TForm1.Timer1Timer(Sender: TObject);

begin

Timer1.Enabled:=False;

{ Необходимые действия }

end;

Если же запрограммировать "выключение" таймера после выполнения предусмотренных в программе действий, то при достаточно малом по сравнению с продолжительностью необходимых действий значении свойства Interval таймер вновь сработает, и это может помешать ходу программы.

Компонент Delphi Timer не является очень точным и не подходит для измерения малых промежутков времени. Его точность порядка 50 миллисекунд. В качестве примера приведу простую программу, отображающую текущее время.

На форме будут только компонент Label и собственно наш компонент Delphi Timer. Свойство Timer можно оставить равным 1000, но раньше на более медленных компьютерах приходилось отображать время несколько раз в секунду, так как из-за влияния других выполняемых компьютером процессов выводимые раз в секунду показания "плавали", что было хорошо заметно на глаз. Размер шрифта возьмём побольше - 50, и подберём более гладкий, я взял Bell MT. Растянем на всю Форму: Align=alClient. Ну и, собственно, сам код:

procedure TForm1.Timer1Timer(Sender: TObject);

begin

Label1.Caption:=TimeToStr(Now);

end;

Вот и всё! Delphi Now - это системная функция, возвращающая текущую дату-время в соответствующем формате TDateTime. Если преобразовывать её в строку функцией TimeToStr, то она вернёт текущее время, если DateToStr, то текущую дату. Если мы хотим отображать в нашей программе и дату тоже, достаточно поставить ещё один компонент Label, и функцией DateToStr передавать в него дату.

Тип TDateTime содержит значение даты и времени.

Она сохраняется как переменная Double, с датой как целая часть, а время как дробная. Дата сохраняется как число дней с 30 декабря 1899. Не понятно, почему не 31 декабря. 01 января 1900 имеет значение 2.

Поскольку TDateTime фактически является Double, то вы можете выполнять над ним вычисления, как будто это было число. Это очень полезно для вычислений типа разницы между двумя датами.

Пример кода : Обнаружение различия между двумя датами

var

day1, day2 : TDateTime;

diff : Double;

begin

day1 := StrToDate('12/06/2002');

day2 := StrToDate('12/07/2002');

ShowMessage('day1 = '+DateToStr(day1));

ShowMessage('day2 = '+DateToStr(day2));

diff := day2 - day1;

ShowMessage('day2 - day1 = '+FloatToStr(diff)+' days');

end;

day1 = 12/06/2002

day2 = 12/07/2002

day2 - day1 = 30 days

# 18. Текстовые файлы. Типизированные файлы. Нетипизированные файлы.

Текстовые файлы связываются с файловыми переменными, принадлежащими к стандартному типу TextFiie. Текстовые файлы предназначены для хранения текстовой информации. Именно в такого типа файлах хранятся, например, исходные тексты программ. Компоненты (записи) текстового файла могут иметь переменную длину, что существенно влияет на характер работы с ними.

Текстовый файл трактуется в Object Pascal как совокупность строк переменной длины. Доступ к каждой строке возможен лишь последовательно, начиная с первой. При создании текстового файла в конце каждой строки ставится специальный признак eoln (End Of LiNe - конец строки), а в конце всего файла - признак eof (End Of File - конец файла). Эти признаки можно протестировать одноименными логическими функциями (см. ниже). При формировании текстовых файлов используются следующие системные соглашения:

eoln - последовательность кодов #13 (cr) и #10 (lf);

EOF -КОД #26.

Длина любого компонента типизированного файла строго постоянна, что дает возможность организовать прямой доступ к каждому из них (т. е. доступ к компоненту по его порядковому номеру).

Перед первым обращением к процедурам ввода-вывода указатель файла стоит в его начале и указывает на первый компонент с номером 0. После каждого чтения или записи указатель сдвигается к следующему компоненту файла. Переменные в списках ввода-вывода должны иметь тот же тип, что и компоненты файла. Если этих переменных в списке несколько, указатель будет смещаться после каждой операции обмена данными между переменными и дисковым файлом.

Следующая программа “зависнет”, т. к. никогда не будет прочитана вторая строка файла:

**procedure** TfmExample.bbRunClick(Sender: TObject);

**var**

F: TextFile;

S: String;

**begin**

AssignFile(F,'example.pas');

Reset(F);

**while- not** EOF(F)**do**

**begin**

Read(P,S); // *Ошибка! Бесконечный цикл!*

mmOutput.Lines.Add(S)

**end;**

CloseFile(F)

**end;**

*Нетипизированные* файлы объявляются как файловые переменные типа File и отличаются тем, что для них не указан тип компонентов. Отсутствие типа делает эти файлы, с одной стороны, совместимыми с любыми другими файлами, а с другой - позволяет организовать высокоскоростной обмен данными между диском и памятью.

При инициации нетипизированного файла процедурами Reset или Rewrite можно указать длину записи нетипизированного файла в байтах. Например, так:

**var**

F:**File;**

**begin**

AssignFile(F,'myfile.dat');

Reset(f,512);

**end.**

Длина записи нетипизированного файла указывается вторым параметром при обращении к процедурам Reset или Rewrite, в качестве которого может использоваться выражение типа *Longint.* Если длина записи не указана, она принимается равной 128 байтам.

Object Pascal не накладывает каких-либо ограничений на длину записи нетипизированного файла за исключением требования положительности и ограничения максимальной длины 2 Гбайт (для Delphi 1 длина записи ограничивается 65535). Для обеспечения максимальной скорости обмена данными рекомендуется задавать длину, которая была бы кратна длине физического сектора дискового носителя информации (512 байт). Однако операции обмена данными с дисковыми устройствами в среде Windows кэшируются, т. е. осуществляются через промежуточный буфер памяти, поэтому обычно задают Recsize = 1, что позволяет обмениваться с файлом блоками любой длины начиная с одного байта.

При работе с нетипизированными файлами могут применяться все процедуры и функции, доступные типизированным файлам, за исключением Read и write, которые заменяются соответственно высокоскоростными Процедурами BlockRead И BlockWrite:

**Procedure** BlockRead(var F:**File; var** Buf; Count: Integer [;

**var** AmtTransferred: Integer]) ;

**Procedure** BlockWrite(var F:**File; var** Buf; Count: Integer [;

**var** AmtTransferred: Integer]);

Здесь Buf - буфер: имя переменной, которая будет участвовать в обмене данными с дисками; count - количество записей, которые должны быть прочитаны или записаны за одно обращение к диску;

AmtTransferred - необязательный параметр, содержащий при выходе из процедуры количество фактически обработанных записей.

# 19. Отображение файлов в память.

Для работы с файлом динамической подкачки страниц виртуальной памяти в Windows 32 используется механизм отображения файлов в адресное пространство программы. Соответствующие функции API доступны любой программе и могут применяться к любому файлу (кстати, таким способом загружается в адресное пространство процесса исполняемый файл). В результате отображения программа может работать с файловыми данными как с данными, размещенными в динамической памяти. Такая возможность не только в большинстве случаев повышает скорость работы с данными, но и предоставляет программисту уникальные средства обработки сразу всех записей файла. Например, он может единственным оператором проверить входит ли заданный образец поиска в любую строку текстового файла. Отображение файла осуществляется в три приема.

Вначале файл создается обращением к функции

**function** FileCreate(FileName: String): Integer;

или открывается с помощью

**function** FileOpen(const FileName:

**String;** Mode: LongWord): Integer;

В обеих функциях FileName - имя файла, возможно, с маршрутом доступа. Параметр Mode определяет режим доступа к файлу и может принимать одно из следующих значений: fmOpenRead - только чтение; fmOpenWrite - только Запись; fmOpenReadWrite - чтение и запись. с помощью операции or эти константы можно комбинировать с одной из следующих регулирующих совместный доступ к файлу нескольких Программ: fmShareExclusive - совместный доступ Запрещен; fmShareDenyWrite - Другим Программам запрещается запись; fmShareDenyRead - другим программам запрещается чтение; fmSchareDenyNone - совместный доступ неограничен. Обе функции возвращают дескриптор созданного (открытого) файла или 0, если операция оказалась неуспешной.

На втором этапе создается объект отображения в память. Для этого используется такая функция:

**function** CreateFileMapping(hFile: THandle; IpFileMapping-Attributes: PSecurityAttributes; flProtect, dwMaximumSize-High, dwMaximumSizeLow: DWord; IpName: PChar): THandle;

Здесь hFile - дескриптор файла; ipFileMappingAttributes - указа-тель на структуру, в которой определяется, может ли создаваемый объект порождать дочерние объекты (обычно не может - nil); flProtect -определяет тип защиты, применяемый к окну отображения файла (см.ниже); dwMaximumSizeHigh, dwMaximumSizeLow - соответственно Старшие и младшие 32 разряда размера файла; если вы будете отображать файлы длинной до 4 ГбаЙТ, поместите В dwMaximumSizeHigh 0, а В dwMaximumSizeLow - длину файла; если оба параметра равны 0, размер окна отображения равен размеру файла; ipName - имя объекта отображения или nil.

Параметр flProtect задает тип защиты, применяемый к окну просмотра файла, и может иметь одно из следующих значений: page\_readonly - файл можно только читать (файл должен быть создан или открыт В режиме fmOpenRead); PAGE\_READWRITE - файл можно читать и записывать в него новые данные (файл открывается в режиме fmOpenReadWrite); PAGE\_WRITECOPY- файл открыт для записи и чтения, однако обновленные данные сохраняются в отдельной защищенной области памяти (отображенные файлы могут разделяться программами, в этом режиме каждая программа сохраняет изменения в отдельной области памяти или участке файла подкачки); файлоткрывается В режиме fmOpenReadWrite или fmOpenWrite; этот тип защиты нельзя использовать в Windows 95/98. С помощью операции or к параметру fiprotect можно присоединить такие атрибуты:

**sec\_commit** - выделяет для отображения физическую память или участок файла подкачки; sec\_image - информация об атрибутах отображения берется из образа файла; sec\_nocashe - отображаемые данные не кэшируются и записываются непосредственно на диск;

**sec\_reserve** - резервируются страницы раздела без выделения физической памяти. Функция возвращает дескриптор объекта отображения или 0, если обращение было неудачным.

Наконец, на третьем этапе создается окно просмотра, т. е. собственно отображение данных в адресное пространство программы:

**function** MapViewOfFile(hFileMappingObject: THandle; dwDesiresAccess: DWord; dwFileOffsetHigh, dwFileIffsetLow, dwNumberOfBytesToMap: DWord): Pointer;

Здесь hFileMappingObject -дескриптор объекта отображения; dwDesiresAccess - определяет способ доступа к данным и может иметь одно из следующих значений: file\_map\_write - разрешает чтение и запись, При ЭТОМ В функции CreateFileMapping должен использоваться атрибут page\_readwrite; file\_map\_read - разрешает только чтение, в функции CreateFileMapping должен использоваться атрибут

PAGE\_READONLYили PAGE\_READWRITE; FILE\_MAP\_ALL\_ACCESS - тоь же, что и

file\_map\_write; file\_map\_copy - данные доступны для записи и чтения, однако обновленные данные сохраняются в отдельной защищенной области памяти; в функции CreateFileMapping должен использоваться атрибут page\_writecopy; dwFileOffsetHigh, dwFileIffsetLow -определяют соответственно старшие и младшие разряды смещения от начала файла, начиная с которого осуществляется отображение;

dwNumberOfBytesToMap - определяет длину окна отображения (0 - длина равна длине файла). Функция возвращает указатель на первый байт отображенных данных или nil, если обращение к функции оказалось неуспешным.

После использования отображенных данных ресурсы окна отображения нужно освободить функцией function UnMapViewOfFile(IpBaseAddress: Pointer): BOOL; единственный параметр обращения к которой должен содержать адрес первого отображенного байта, т. е. адрес, возвращаемый функцией MapViewOfFile. Закрытие объекта отображения и самого файла осуществляется обращением к функции

**function** CloseHandle(hObject: THandle).

# 20. Работа с файлами как с объектами в Delphi, абстрактный класс TStream. Примеры.

В Delphi используется абстрактный класс TStream (поток данных), который является основой для работы с файлами как с объектами. В случае объектов совершенно не важно, что именно является носителем информации - дисковый ли файл, ленточный носитель или оперативная память. В специализированных потомках TStream определены стандартные методы Read, write и seek, открывающие полный доступ к файловому объекту. Сам объект создается конструктором и уничтожается деструктором.

Поток не, учитывает специфику хранящихся в файле данных, т. е. по существу работает с данными, как если бы хранились в не типизированном файле.

В следующем примере файл с именем, хранящимся в Editl.Text,

копируется В файл Edit2 . Text.

procedure TFormI.CopyFileClick(Sender:TObject) ;

var

Streami, Stream2: TStream;

begin

Streami:=TFileStream.Create(Editi.Text,

fmOpenRead or fmShareDenyWrite) ;

try

Stream2 :=TFileStream.Create(Edit2.Text,

fmOpenCreate or fmShareDenyRead);

try

Stream2.CopyFrom(Streami, Streami.Size);

finally

Stream2.Free;

finally

Stream1.Free

end;

Абстрактный класс т Stream лишь декларирует ключевые методы Read и write, которые перекрываются в его наследниках для специализации операций:

TFileStream - обмен данными с файлом;

TStringStream - обмен данными с длинной строкой (в памяти);

TMemoryStream - обмен данными с памятью;

TBLOBStream - чтение/запись полей данных типа BLOB (Binary Large Object - большой двоичный объект;

TWinsocketStream - обмен данными с сокетным подключением;

TOLEStream - обмен данными в формате OLE.

Любой наследник получает в свое распоряжение метод copyFrom, с помощью которого он может прочитать содержимое другого потока. Таким способом можно, например, файловые данные расположить в памяти или наоборот - содержимое памяти записать в дисковый файл.

Точно так же от TStream наследуются множество других полезных методов, в том числе:

Seek - пропускает нужное количество байт от текущей позиции потока;

ReadBuffer - читает из буфера в поток нужное количество байт;

Readcomponent - читает свойства компонента из потока;

WriteBuffer - записывает в поток содержимое буфера заданной длины;

WriteComponent - записывает в поток свойства компонента.

Следующий обработчик bbRunciick выведет в редактор mmOutput свойства кнопки bbRun:

procedure TfmExample.bbRunClick(Sender: TObject);

var

MemSourceStream, MemDestStream: TMemoryStream;

begin

MemSourceStream := TMemoryStream.Create;

try

MemDestStream := TMemoryStream.Create;

try

MemSourceStream.WriteComponent(bbRun);

MemSourceStream.Seek(0, soFromBeginning) ;

ObjectBinaryToText(MemSourceStream, MemDestStream);

MemDestStream.Seek(0, soFromBeginning) ;

mmOutput.Lines.LoadFromStream(MemDestStream) finally

MemDestStream.Free

end;

finally

MemSourceStream.Free

end;

end;

Комментарий к программе

Вначале свойства компонента записываются в поток MеmSourceStream. Чтобы полученным таким образом двоичным данным придать “читабельный” вид, используется процедура ObjectBinaryToText, которая преобразует данные из MemSourceStream и помещает иx в MemDestStream. Содержимое этого второго потока и выводится

# 21. Класс Exeption. Защищенные блоки. Стандартные классы исключений. Вызов исключения. Создание собственного потока.

Класс Exception является прямым потомком базового класса TObject. Вместе со своими потомками он предназначен для обработки исключительных ситуаций (исключений), возникающих при некорректных действиях программы: например, в случае деления на ноль, при попытке открыть несуществующий файл, при выходе за пределы выделенной области динамической памяти и т. п. В этом разделе рассматриваются основные свойства исключений и их использование для повышения надежности программ.

Для обработки исключений в Object Pascal предусмотрен механизм защищенного блока:

except

<обработчики исключений>

else

<операторы>

end;

finally

<операторы>

end;

Try

<операторы>

Try

<операторы>

Защищенный блок начинается зарезервированным словом try (попытаться [выполнить]) и завершается словом end. Существуют два типа защищенных блоков - except (исключить) и finally (в завершение), отличающихся способом обработки исключения. В блоке except порядок выполнения операторов таков: сначала выполняются операторы секции try... except; если операторы выполнены без возникновения исключительной ситуации, работа защищенного блока на этом прекращается, и управление получает оператор, стоящий за end; если при выполнении части try возникло исключение, управление получает соответствующий обработчик в секции except, а если таковой не найден - первый из операторов, стоящих

За словом else. В блоке finally операторы В секции finally. . .end получают управление всегда, независимо от того, возникло ли исключение в секции try.. .finally или нет. Если исключение возникло, все операторы в секции try.. .finally, стоящие за "виновником" исключения, пропускаются, и управление получает первый оператор секции finally... end. Если исключения не было, этот оператор получает управление после выполнения последнего оператора секции try.. .finally.

Обработчики исключений в блоке except имеют такой синтаксис:

on <класс исключения> do <оператор>;

Здесь on, do - зарезервированные слова; <класс исключения> -класс обработки исключения; <оператор> - любой оператор Object Pascal, кроме оператора передачи управления goto на метку вне блока except.

Обратите внимание: имя класса служит своеобразным ключом выбора, а собственно обработка осуществляется оператором, стоящим за do (этот оператор может быть составным, так что обработка исключения может занимать произвольное количество операторов Object Pascal).

Поиск нужного обработчика осуществляется с начала списка вниз до тех пор, пока не встретится класс, способный обрабатывать исключение данного типа. Если подходящего класса не обнаружено, управление передается операторам, стоящим за словом else, а если таковых нет (часть else <операторы> может опускаться), выполняется умалчиваемая обработка исключения.

Если для программиста важен лишь сам факт возникновения исключения и несущественен тип связанной с ним ошибки, он может опустить в секции except.. .end обработчики вместе со словом else, оставив в ней лишь необходимый код реакции на любую ошибку:

try

except

ShowMessage('Ощибка!') ;

end;

Защищенные блоки могут вкладываться друг в друга на неограниченную глубину, т. к. везде, где в предыдущих описаниях использовались <операторы> ИЛИ <оператор>, могут использоваться любые операторы Object Pascal, в том числе и try... except или try...finally:

try

try finally

end;

except

on EMatchError do begin

try try end;

end;

end

end;

В Delphi определены стандартные классы исключений, перечисленные в табл. 16.1. Именно эти имена, а также имена пользовательских классов (см. п. 16.1.5) могут использоваться в обработчиках исключении.

|  |  |  |
| --- | --- | --- |
| **Таблица 16.1** | | |
| **Класс** | **Родитель** | **Обрабатываемое исключение** |
| EAbort | Exception | Реализует "тихую" (без какого-либо сообщения) обработку любого исключения |
| EAbstractErrpr | Exception | Программа пытается вызвать абстрактный метод |
| EAccessViolation | Exception | Программа пыталась обратиться к не принадлежащей ей области памяти или использует недействительный указатель |
| EAppletException | Exception | Ошибка связана с созданием управляющих панелей в апплет-приложениях |
| EArrayError | Exception | Возникает из-за различного рода ошибок при работе с массивами (неверный индекс, попытка вставить элемент в массив фиксированной длины или в отсортированный массив и т. п.) |
| EAssertionFaild | Exception | Возбуждается отладочной процедурой Assert, когда тстируемое ею логическое выражение имеет значение False. |
| EBitsError | Exception | Программа пыталась обратиться к свойству Bits объекта TBits с индексом меньше нуля или больше максимально допустимого значения |
| EBrokerException | Exception | Объект-брокер не может найти сервер |
| ECacheError | Exception | Ошибка в наборе данных для компонента Tde-cisionCube |
| EClassNotFound | EFilerError | Для компонента, читаемого из потока данных, не найден соответствующий класс. Обычно возникает"в случае, когда в форму вставлен нестандартный компонент, а в библиотеке компонентов Delphi нет связанного с ним класса |
| ECommonCalendar-Error | Exception | Возникает в объектах класса TCommonCalendar и его потомках, когда вводится неверная дата |
| EComponentError | Exception | Возникает при различных манипуляциях программы с компонентом (программа не может зарегистрировать компонент, переименовать его или когда для его работы требуется интерфейс СОМ, который компонентом не поддерживается) |
| EControlC | Exception | Возникает при нажатии Ctrl-C при работе приложения в режиме консолиЛ |
| EConvertError | Exception | Ошибка преобразования в функциях StrToint или StrToFloat |
| ECorbaDispatch | Exception | Возникает в программах, использующих технологию corba, при ошибках, связанных с несовпадением интерфейсов сервера и брокера данных |
| ECorbaException | Exception | Возникает в программах, использующих технологию CORBA |
| ECorbaUser-Exception | ECorbaException | Возникает как определяемая пользователем реакция на ошибки интерфейса |
| EDatabaseError | Exception | Возникает, когда компонент обнаруживает ошибку в базе данных |
| EDateTimeError | Exception | Возбуждается компонентом TDateTimePicker при попытке ввода неверной даты или времени |
| EDBClient | EDatabaseError | Ошибка связана с неправильной работой Tcli-entDataSet |
| EDBEditError | Exception | Возникает, когда компонент пытается использовать данные, несовместимые с заданной маской |
| EDBEngineError | EDatabaseError | Связана с ошибками BDE |
| EDimensionMar-Error | Exception | Возникает, когда используемый в кубе решений набор данных не имеет агрегатных полей |
| EDimIndexError | Exception | Связана с нарушением размерности массива данных для куба решений |
| EDivByZero | EIntError | Ошибка целочисленного деления на ноль |
| EDSWriter | Exception | Ошибка при подготовке провайдером пакета данных для набора данных |
| EExternal-Exception | EStream-Error | Возникла ошибка, код которой не является предопределенным в Delphi |
| EFCreateError | EStream-Error | Ошибка при создании файла. Например, попытка создать файл на устройстве, предназначенном только для чтения, или в несуществующем каталоге |
| EFilerError | EStream-Error | Программа пытается повторно зарегистрировать в потоке один и тот же класс |
| EFOpenError | EStream-Error | Ошибка открытия потока данных. Например, попытка открыть несуществующий файл |
| EHeapException | Exception | Ошибка связана с неправильными операциями над динамической памятью |
| ElBClientError | ElBError | Ошибка связана с функционированием IBX-клиента |
| ElBError | EDatabaseError | Общая ошибка технологии IBX |
| ElBInterbase-Error | ElBError | Ошибка связана с функционированием сервера в технологии IBX |
| EInOutError | Exception | Любая ошибка в файловых операциях. Поле ErrorCode объекта этого класса содержит код ошибки |
| EInterpreterError | Exception | Возникает, когда компонент класса TDataBlockinterpeter не может интерпретировать данные блока данных |
| EIntError | Exception | Любая ошибка в целочисленных вычислениях |
| EIntfCastError | Exception | Попытка недопустимого приведения типов в OLE-объектах |
| EIntOverflow | EIntError | Ошибка целочисленного переполнения: программа пытается присвоить целочисленной переменной значение, выходящее из 32-двоичных разрядов |
| EInva1i dArgument | EMatchError | Возбуждается математическими функциями при выходе аргумента из допустимого диапазона |
| EInvalidCast | Exception | Программа пытается осуществить недопустимое преобразование типов с помощью оператора as |
| EInvalidGraphic | Exception | Программа пытается загрузить в контейнер изображение из файла, который имеет недопустимый формат (допустимыми форматами являются растр, метафайл, курсор, пиктограмма) |
| EInvalidGraphic-Operation | Exception | Программа пытается выполнить недопустимую графическую операцию |
| EInvalidGrid-Operation | Exception | Программа пытается выполнить недопустимую операцию над таблицей (например, обратиться к несуществующему столбцу или РЯДУ) |
| EInvalidImage | EFilerError | Программа пытается прочитать ресурс изображения из файла, в котором этого ресурса нет |
| EInvalidOp | EMatchError | Ошибка в операциях с плавающей точкой (недопустимая инструкция, переполнение стека сопроцессора и т. п.) |
| EInvalidOpera-tion | Exception | Не имеющий окна компонент пытается выполнить операцию, требующую дескриптора окна |
| EInvalidPointer | EHeap-Exception | Попытка использовать недействительный указатель |
| EListError | Exception | Эта ошибка связана с неверными действиями программы по отношению к разного рода спискам. Например обращение к элементу списка с индексом меньше нуля или больше максимально допустимого |
| ELowCapacity-Error | Exception | Ошибка возникает при попытке выделения памяти на устройстве, у которого нет нужной свободной памяти. |
| EMatchError | Exception | Любая ошибка при выполнении вычислений с плавающей точкой. |
| EMenuError | Exception | Ошибка при работе программы с меню. Например, при добавлении элемента с идентификатором, который уже определен в меню |
| EMCIDiviceError | Exception | Ошибка возникла в медиаплейере |
| EMethodNotFound | EFilerError | Программа прочитала из потока данных объект, но не может найти связанный с классом объекта метод |
| EMonthCalError | ECommon-Calendar-Error | Возбуждается компонентом класса TMonthcal-endar при попытке ввода неправильной даты |
| EOleCtrlError | Exception | Программа не может установить связь с элементом ActiveX |
| EOleError | Exception | Низкоуровневая ошибка в технологии OLE |
| EOleException | EOleSysError | Программа использует неверный OLE-интерфейс |
| EOleRegistration Error | EOleError | Ошибка регистрации OLE-объекта в реестре Windows |
| EOleSysError | EOleError | Возникает при неправильном выполнении команды OLE автоматизации |
| EOutlineError | Exception | Возникает при ошибке доступа к компоненту класса ToutLine |
| EOutOfMemory | EHeap-Exception | Эта ошибка возникает, когда программа запрашивает слишком большой для данной конфигурации Windows объем памяти |
| EOutOfResource | EOutOfMemory | Программа требует от Windows дескриптор окна, но Windows исчерпала лимит дескрипторов |
| EOverflow | EMatchError | Результат операций с плавающей точкой слишком велик, чтобы уместиться в регистрах сопроцессора |
| EPackageError | Exception | Возникает при ошибке доступа к пакету |
| EParserError | Exception | Ошибка преобразования текста в двоичные данные при чтении из потока |
| EPrinter | Exception | Windows сообщила программе об ошибке принтера |
| EPrivilege | Exception | Программа пытается выполнить привилегированную операцию. Привилегированные операции могут выполняться только ядром Windows |
| EPropertyConvert Error | Exception | Ошибка при чтении или записи значения свойства |
| EPropertyError | Exception | Ошибка доступа к свойству при чтении или записи |
| EPropReadOnly | Exception | Программа пытается присвоить значение свойству, из которого можно только читать (при использовании технологии OLE) |
| EPropWriteOnly | Exception | Программа пытается прочитать свойство, предназначенное только для записи |
| ERangeError | EIntError | Целочисленный результат превышает емкость целого типа данных |
| EReadError | EFilerError | Программа не может прочитать из потока данных нужного количества байт |
| EReconcileError | EData-baseError | Ошибка обновления данных в TclientDataset |
| ERegistry-Exception | Exception | Ошибка, связанная с операцией над реестром Windows |
| EResNotFound | Exception | Программа не может найти указанный ресурс в файле ресурсов |
| ESocketConnec-tionError | Exception | Ошибка связана с работой с сокетами Windows |
| ESocketError | Exception | Ошибка связана с работой с сокетами Windows |
| EStackOverflow | Exception | Исчерпан объем выделенного программе стека |
| EStreamError | Exception | Любая ошибка при работе с потоком данных |
| EStringListError | Exception | Программа ссылается на строку, индекс которой выходит из диапазона возможных значений для списка строк |
| EThread | Exception | Ситуация борьбы за общий ресурс в программе с несколькими потоками команд |
| ETreeViewError | Exception | Указан неверный индекс при обращении к TtreeView |
| EUnderflow | EMatchError | Результат операций с плавающей точкой слишком мал, чтобы уместиться в регистрах сопроцессора (исчезновение порядка) |
| EUnsupportedTypeE rror | Exception | Выбран недопустимый тип поля в качестве измерения в кубе решений |
| EUpdateError | Exception | Ошибка обновления провайдерского набора данных |
| EVariantError | Exception | Ошибка при работе с типом Variant: недопустимое приведение типов; недопустимая операция; обращение -к скалярной переменной как к варианту-массиву; индекс варианта-массива выходит из допустимых значений |
| EWin32Error | Exception | Ошибочное обращение к API-функции Windows. Свойство Message содержит номер ошибки и связанное с ней сообщение |
| EWriteError | EFilerError | Ошибка записи в поток данных |
| EZeroDivide | EMatchError | Вещественное деление на ноль |

Важно помнить, что ищется самый первый из, возможно, нескольких обработчиков, класс которого способен обрабатывать данное исключение. Если, например, в списке первым стоит EAbort, который может обработать любое исключение, ни один из стоящих за ним обработчиков никогда не получит управления. Точно так же, если указан обработчик для класса EintError, за ним бесполезно размещать ОбработчикиEDivByZero, ERangeError или EIntOverflow:

try

except

// He имеет смысла делать так:

on EintError do .....;

on ERangeError do .....;

on EDivByZero do .....;

// Надо так:

on ERangeError do .....;

on EDivByZero do .....;

on EintError do .....;

end;

При возникновении исключительной ситуации объекты классов-обработчиков создаются и уничтожаются автоматически. Если программист пожелает использовать поля или методы класса-обработчика явно, он должен поименовать автоматически создаваемый объект. Для этого перед именем класса ставится идентификатор и двоеточие:

on EObject: EClassName do .....;

Для стандартных классов такой прием фактически позволяет использовать единственное строковое свойство Message со стандартным сообщением об ошибке, которое получают все наследники класса Exception. Исключение составляет класс EInOutError, в котором для программиста может представлять интерес целочисленное свойство ErrorCode с кодом ошибки ввода/вывода.

Например:

Reset(F) ;

while not EOF(F) do

begin

end;

CloseFile(F) ;

except

on E: EInOutError do

ShowMessage('При выполнении файловой операции возникла'+ ' ошибка ?'+ IntToStr(E.ErrorCode));

end;

В некоторых ситуациях программисту бывает необходимо инициировать собственное исключение. Для этого он использует зарезервированное слово raise (возбудить). Если это слово встретилось в секции try.. .exception или try.. .finally, немедленно начинают свою работу секции соответственно except... end и finally... end. Если оно встретилось в except.. .end или finally.. .end, считается, что данный защищенный блок на текущем уровне вложенности (блоки могут быть вложенными) завершил свою работу и управление передается вышестоящему уровню.

Слово raise возбуждает исключение самого общего класса Exception. Если программист желает возбудить исключение конкретного типа (не важно - стандартного или собственного), он должен явно указать класс создаваемого в этот момент объекта путем вызова его конструктора. Например, следующий оператор возбудит ошибку ввода/вывода:

raise EInOutError.Create('Ощибка!') ;

Такой прием - единственная возможность возбудить нестандартное исключение, обрабатываемое пользовательским классом.

Программист может создать собственный класс обработки исключений, объявив его потомком Exception или любого другого стандартного класса (этим другим чаще всего бывает класс EAbort). Объявление нестандартного класса имеет смысл только тогда, когда вам необходимо научить программу распознавать некорректные наборы данных и соответствующим образом на них реагировать.

Пусть, например, в программе используется цикл ввода целочисленных значений из текстового файла, их проверки и преобразования. Проверка заключается в простом контроле неотрицательности очередного числа после ввода и его положительности после преобразования. Перед проверкой необходимо получить строку из файла (здесь может возникнуть ошибка EinOutError) и преобразовать ее в целую величину (здесь возможна ошибка EConvertError); после проверки осуществляется обработка величины, в процессе которой может возникнуть ошибка EIntError.

# 22. Класс TList. Свойства и методы. Тип pPointerList и pDouble. Пример.

Класс TList очень полезный универсальный контейнер списков. Он отличается от массивов, в которых он обеспечивает более богатые функциональные возможности.

В частности объекты TList могут быть отсортированы. Эта сортировка может быть с использованием любых выбранных критериев. Например, список может содержать набор объектов, которые имеют строку и численные поля. Вы можете отсортировать список по строке, по числу, по обоим, с возрастанием или убыванием, как Вы желаете. И пересортировать позже по другим критериям.

Показанный пример кода показывает такую сортировку.

Ключевые свойства и методы упомянуты ниже.

Свойство Capacity

Используется для установления размера (число указателей на объекты) списка. Предварительно установив в разумное значение, можно избежать множественных перераспределений памяти.

Свойство Count

Число элементов (указателей) в списке. Может быть прочитано или записано. Если размер уменьшен в результате изменения значения Count, то удаляются элементы в конце списка.

Свойство Items

Позволяет обращаться к элементам в списке. Например, myList.Items[2] ; возвращает 3-ий элемент в списке. Это свойство, заданное по умолчанию, вышеупомянутое может быть упрощено до myList[2];.

Свойство List

Возвращает элементы в массиве.

Метод Add

Добавляет элемент в конец списока.

Метод Assign

Заменяет список содержанием другого списка.

Метод Clear

Удаляет все элементы списка, устанавливая Count в 0.

Метод Delete

Удаляет элемент из списка по его позиции в списке.

Метод Remove

Удаляет элемент из списка по его объектному указателю.

Метод Exchange

Меняет позиции двух элементов

Метод Move

Перемещает элемент в новую позицию списка.

Метод Insert

Вставляет новый элемент в список в данную позицию.

Метод First

Получает первый элемент в списке.

Метод Last

Получает последний элемент в списке.

Метод Sort

Сортирует список в соответствии с вашими указанными критериями. Сортировка списка проводится внутри TList, но каждая пара элемента сравнивается, вызывая функцию, которую вы указали для этого метода.

Метод IndexOf

Выдает позицию указанного объекта в списке.

Тип pPointerList определен следующим образом:

type

pPointerList = PPointerList;

TPointerList = array [0..MaxListSize] of Pointer;

Константа MaxListSize для Delphi 1 ограничена значением 16379 элементов. Для старших версий Delphi она ограничивается доступной памятью.

**unit Unit1;**  
  
**interface**  
  
**uses**  
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
  Dialogs, StdCtrls;  
  
**type**  
  **// Определение класса клиента**  
  TCustomer = class  
    private  
      **// Поля данных этого нового класса**  
      CustomerName   : String;  
      CustomerNumber : Integer;  
  
    public  
      **// Свойства для чтения значений этих данных**  
      property Name : String  
          read CustomerName;  
      property Number : Integer  
          read CustomerNumber;  
  
      **// Коструктрор**  
      constructor Create(const CustomerName   : String;  
                         const CustomerNumber : Integer);  
  end;  
  
  **// Определение класса формы**  
  TForm1 = class(TForm)  
    procedure FormCreate(Sender: TObject);  
  
  private  
    **// TList объект мы использует в этом коде**  
    myList : TList;  
  
    **// Метод для показа содержимого нашего объекта списка**  
    procedure ShowListContents;  
  
  public  
  
  end;  
  
**var**  
  Form1: TForm1;  
  
**implementation**  
  
**{$R \*.dfm}**  
  
  
**// Конструктор Customer**  
**// --------------------------------------------------------------------------**  
**constructor TCustomer.Create(const CustomerName   : String;**  
                             const CustomerNumber : Integer);  
**begin**  
  **// Сохранение переданных параметров**  
  self.CustomerName   := CustomerName;  
  self.CustomerNumber := CustomerNumber;  
**end;**  
  
  
**// Программа сортировки TList : сравните клиентов по имени**  
**// --------------------------------------------------------------------------**  
**// Возвращенное целое число имеет следующее значение :**  
**//**  
**//   > 0 : (положительное) Item1 является меньше чем Item2**  
**//     0 : Item1 равно Item2**  
**//   <0 : (negative) больше чем item2**  
**function compareByName(Item1 : Pointer; Item2 : Pointer) : Integer;**  
**var**  
  customer1, customer2 : TCustomer;  
**begin**  
  customer1 := TCustomer(Item1);  
  customer2 := TCustomer(Item2);  
  
  **// Теперь сравнение строк**  
  if      customer1.Name > customer2.Name  
  then Result := 1  
  else if customer1.Name = customer2.Name  
  then Result := 0  
  else Result := -1;  
**end;**  
  
  
**// Подпрограмма для показа содержимого нашего списка**  
**// --------------------------------------------------------------------------**  
**procedure TForm1.ShowListContents;**  
**var**  
  i : Integer;  
**begin**  
  **// И повторный показ списка**  
  for i := 0 to myList.Count-1 do  
  begin  
    ShowMessage(TCustomer(myList[i]).Name+' is customer number '+  
                IntToStr(TCustomer(myList[i]).Number));  
  end;  
**end;**  
  
**// Конструктор формы**  
**// --------------------------------------------------------------------------**  
**procedure TForm1.FormCreate(Sender: TObject);**  
**var**  
  customer : TCustomer;  
  
**begin**  
  **// Создание объекта TList для хранения набора объектов клиент**  
  myList := TList.Create;  
  
  **// Создание нескольких объектов клиентов и добавление их в наш объект список**  
  customer := TCustomer.Create('Neil Moffatt', 123);  
  myList.Add(customer);  
  customer := TCustomer.Create('Bill Gates', 64);  
  myList.Add(customer);  
  
  **// Мы можем добавить объект, не присваивая в промежуточную переменную**  
  myList.Add(TCustomer.Create('Henry Cooper', 999));  
  myList.Add(TCustomer.Create('Alan Sugar', 2));  
  
  **// Теперь показываем список**  
  ShowListContents;  
  
  **// Теперь мы сортируем список в последовательность имён и повторяем показ**  
  myList.Sort(compareByName);  
  ShowListContents;  
  
  **// Теперь удаляем и вставляем некоторые объекты**  
  **// Обратите внимание, что индексы начинаются с 0**  
  myList.Insert(2, TCustomer.Create('Added as item 3', 33));  
  myList.Delete(4);  
  
  **// И повторно показываем список**  
  ShowListContents;  
**end;**  
  
**end.**

# 23. Классы TStringList и TStrings, основные свойства и методы. Пример.

TStringList - полезный тип класса. Он чрезвычайно полезен для многих видов обработок списков. Элементы в строковом списке могут быть вставлены, перемещены и отсортированы.

Список может быть сформирован строка за строкой, или загружен из большой строки разделенной запятой или даже из текстового файла. TStringList происходит от TStrings. Вы можете использовать и TStrings, но это не рекомендуется, так как он не полный - некоторые из его методов абстрактны. TStringList осуществляет эти абстрактные методы (Clear, Delete и Insert). Мы рассмотрим основные свойства и методы TStringList, включая полученные из TStrings.

Свойство Count

Возвращает число строк в списке.

Свойство Capacity

Устанавливает или получает текущую вместимость строкового списка. Вы можете управлять этой вместимостью по необходимости.

Свойство Strings

Получает или корректирует строку по данному индексу в списке (первый элемент списка имеет индекс 0).

Обратите внимание, что свойство Strings является свойством, заданным по умолчанию. Это означает, что вы можете использовать его без его указания:

myName := names.Strings[4];

является эквивалентным:

myName := names[4];

Свойство Text

Устанавливает или получает список в виде большой строки. Эта строка будет содержать каждую строку закончивающуюся комбинацией символов перевода каретки и перевода строки (CRLF). Полезно для загрузки из визуального объекта, который может содержать многочисленные строки текста.

Свойство CommaText

Получает или устанавливает список в виде большой строки. Эта строка будет иметь список строк разделенных запятыми. Это полезно для загрузки из экспорта текстовой электронной таблицы. Если при получении строка содержит вложенные пространства, то она будет заключена в двойные кавычки.

Cвойство DelimitedText

Получает или устанавливает список через большую строку. Эта строка содержит список строк разделенных значением Delimiter (по умолчанию - запятая). Строки, содержащие вложенные пробелы должны быть заключены в QuoteChar (по умолчанию - ").

Свойство QuoteChar

Используется для замыкания строк, которые имеют вложенные пробелы при использовании DelimitedText.

Свойство Delimiter

Используется для разделения строк при использовании DelimitedText

. Свойство Names

Строки в строковом списке могут быть обработаны, как пары название/значение, как во втором примере кода. Каждая строка не должна иметь никаких внедренных пробелов, и содержать знак =. Это очень полезное понятие. См. свойства Value и ValueFromIndex, и метод IndexOfName.

Свойство Values

Возвращает значение для данного названия, когда используются строки пары название/значение (см. выше).

Свойство ValueFromIndex

Возвращает значение по индексу строки (начинается с 0), когда используются пары название/значение.

Свойство CaseSensitive

Когда true, Delphi обрабатывает строки чувствительно к регистру при выполнении некоторых операций, таких как Sort.

Свойство Duplicates

Это свойство может иметь одно из следующих перечислимых TDuplicates значений:

dupIgnore Игнорирует (отбрасывает) дубликаты

dupAccept Позволяют дубликаты

dupError Выбрасывает исключение, если имеются дубликаты

Свойство Sorted

Когда true, все строки будут добавляться в свою позицию отсортированной последовательности. Когда false, они будут добавляться в конец. См. также метод Sort.

Свойство Objects

Возвращает объект, связанный со строкой по данному индексу, если он существует.

Метод Add

Добавляет данную строку в список, возвращая ее позицию в списке (начинается с 0).

Метод Append

Так же как и Add, но без возвращения индексного значения.

Метод Insert

Вставляет строку в заданную индексом позицию. Позиция 0 вызовет вставку в начало.

Метод Delete

Удаляет строку по данному индексу.

Метод Clear

Удаляет все строки из списка.

Метод Move

Перемещает строку из одной позиции в другую, сдвигая другие строки соответственно.

Метод Exchange

Перестанавливает две строки в списке, идентифицированные по их индексным позициям.

Метод IndexOf

Получает индекс позиции строки в списке соответствующей данной строке. Если строка не найдена, то возвращается -1.

Метод IndexOfName

Получает индекс позиция первой пары название/значение строки, где название соответствует данной строке. Если не найдена - возвращается -1.

Метод Find

То же самое, что и IndexOf, но с использованием списков сортированных строк.

Метод Sort

Если Sorted является ложным, то это вызовет сортировку списка.

Метод AddStrings

Добавляет строки из другого списка.

Метод Assign

Заменяет текущий список содержанием другого списка.

Метод LoadFromFile

Очень полезный метод, загружает строковый список из текстового файла. Каждая текстовая строка (законченая CRLF - см. DelimitedText) становится строкой списка.

Метод SaveToFile

Сохраняет строковый список в текстовый файл.

**var**  
  animals : **TStringList**;            **// Определение нашей переменной списка строк**  
  i       : Integer;  
**begin**  
  **// Определение объекта списка строк, и указание нашей переменной на него**  
  animals := TStringList.Create;  
  
  **// Теперь добавляем несколько названия в наш список**  
  animals.Add('Cat');  
  animals.Add('Mouse');  
  animals.Add('Giraffe');  
  
  **// Теперь показываем этих животных**  
  for i := 0 to animals.Count-1 do  
    ShowMessage(animals[i]);  **// animals[i] эквивалентно animals.Strings[i]**  
**end;**

# 24. Класс TFont, основные методы и свойства. Пример.

С помощью класса TFont создается объект-шрифт для любого графического устройства (экрана, принтера, плоттера и т. п.).

**Свойства класса:**

|  |  |
| --- | --- |
| property Color: TColor; | Цвет шрифта |
| property Charset: TFontCharSet; | Набор символов. Для русскоязычных программ это свойство обычно имеет значение DEFAULTCHARSET или russian charset. Используйте значение OEMCHARSET для отображения текста MS-DOS (альтернативная кодировка) |
| property FontAdapter: IChangeNotifier; | Поставляет информацию о шрифте в компоненты ActveX |
| property Handle:hFont; | Дескриптор шрифта. Используется при непосредственном обращении к API-функциям Windows |
| property Height: Integer; | Высота шрифта в пикселях экрана |
| property Name: TFontName; | Имя шрифта. По умолчанию имеет значение MS  Sans Serif |
| property Pitch: TFontPitch; | Определяет способ расположения букв в тексте: значение fpFixed задает моноширинный текст, при котором каждая буква имеет одинаковую ширину; значение fpVariabel определяет ропорциональный текст, при котором ширина буквы зависит от ее начертания; fpDefault определяет ширину, принятую для текущего шрифта |
| property PixelPerInch: Integer; | Определяет количество пикселей экрана на один дюйм реальной длины. Это свойство не следует изменять, т. к. оно используется системой для обеспечения соответствия экранного шрифта шрифту принтера |
| property Size: Integer; | Высота шрифта в пунктах *(1/72* дюйма). Изменение этого свойства автоматически изменяет свойство Height и наоборот. |
| property Style: TFontStyles; | Стиль шрифта. Может принимать значение как комбинация следующих признаков: fsBold (жирный), fsltalic (курсив), fsUnderline (подчеркнутый), fsStrikeOut (перечеркнутый) |

Для некоторых случаев может оказаться полезным метод

**procedure** Assign(Source: TPersistent);

с помощью которого значения свойств шрифтового объекта source присваиваются свойствам текущего шрифта. Метод не изменяет свойство pixeiperinch, поэтому его можно использовать для создания шрифта принтера по экранному шрифту и наоборот.

Класс TPen, методы и свойства. Пример

С помощью класса треп создается объект-перо, служащий для вычерчивания линий. Свойства класса:

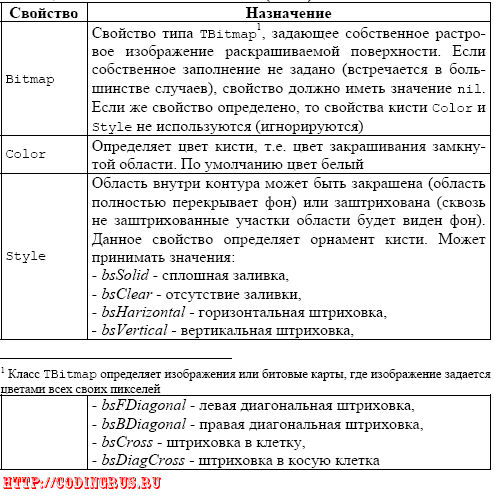
|  |  |  |  |
| --- | --- | --- | --- |
|  | property Color: TColor; | | Цвет вычерчиваемых пером линий |
|  | property Handle: Inte  ger; | | Дескриптор пера. Используется при непосредственном обращении к API-функциям Windows |
|  | property Mode: TPenMode; | | Определяет способ взаимодействия линий с фоном (см.ниже) |
|  | property Style: TPenStyle; | | Определяет стиль линий (см. рис. 16.2). Учитывается только для толщины линий 1 пиксель. Для толстых линий стиль всегда ps Solid (сплошная) |
|  | property Width: Integer; | | Толщина линий в пикселях экрана |
|  | Свойство Mode может принимать одно из следующих значений: | | |
|  | pmBlack | | Линии всегда черные. Свойства Color и Style игнорируются |
|  | pmWhite | | Линии всегда белые. Свойства Color и Style игнорируются |
|  | pmNop | | Цвет фона не меняется (линии не видны) |
|  | pmNot | | Инверсия цвета фона. Свойства Color и Style игнорируются |
|  | pmCopy | | Цвет линий определяется свойством Color пера |
|  | pmNotCopy | | Инверсия цвета пера. Свойство Style игнорируется |
|  | | pmMergePenNot | Комбинация цвета пера и инверсионного цвета фона |
|  | | pmMaskPenNot | Комбинация общих цветов для пера и инверсионного цвета фона. Свойство *Style*игнорируется |
|  | | pmMergeNotPen | Комбинация инверсионного цвета пера и фона |
|  | | pmMaskNotPen | Комбинация общих цветов для инверсионного цвета пера и фона. Свойство *Style*игнорируется |
|  | | pmMerge | Комбинация цветов пера и фона |
|  | | pmNotMerge | Инверсия цветов пера и фона. Свойство Style игнорируется |
|  | | pmMask | Общие цвета пера и фона |
|  | | pmNotMask | Инверсия общих цветов пера и фона |
|  | | pmXor | Объединение цветов пера и фона операцией XOR |
|  | | pinNotXor | Инверсия объединения цветов пера и фона операцией XOR |

# 26.Класс TBrush. Методы и свойства. Пример

Объекты класса TBrush (кисти) служат для заполнения внутреннего пространства замкнутых фигур.

***Свойства класса:***

|  |  |
| --- | --- |
| property Bitmap: TBitmap; | Содержит растровое изображение, которое будет использоваться кистью для заполнения. Если это свойство определено, свойства Color и Style игнорируются |
| property Color: TColor; | Цвет кисти |
| property Handle: Integer; | Дескриптор кисти. Используется при не посредственном обращении к API- функциям Windows |
| property Style: TBrushStyle; | Стиль кисти (см. рис. 16.3) |

 Класс **TBrush** задает характеристики кисти, которой заливаются  
внутренние области изображения. Основные свойства кисти представ-  
лены в табл. 1.2.  
**Таблица 1.2. Свойства объекта Brush (кисть)**  
  
Программно определить характеристики кисти можно, используя  
следующее присваивание:

with Image1.Canvas do  
begin  
Brush.Color:=clRed;  
Brush.Style:= bsCross;  
Rectangle(10,10,100,100)  
end

В рассмотренном фрагменте программного кода рисуется прямо-  
угольник, заполненный красной штриховкой в клетку.  
Рассмотрим пример назначения битового стиля кисти:

procedure TForm1.FormCreate(Sender: TObject);  
var  
Bitmap: TBitmap; //переменная для хранения образа кисти  
begin  
Bitmap:=TBitmap.Create;//создание объекта – битового образа1  
try  
//Загрузка битового изображения из файла Picture.bmp  
Bitmap.LoadFromFile('Picture.bmp');  
//Назначение битового изображения в качестве стиля кисти  
Image1.Canvas.Brush.Bitmap:=Bitmap;  
//Рисуем закрашенный эллипс  
Image1.Canvas.Ellipse(20,20,180,180);  
finally  
Image1.Canvas.Brush.Bitmap:=nil;  
//Освобождение памяти, выделенной под изображение  
Bitmap.Free  
end  
end;

# 27.Класс TCanvas. Методы и свойства. Пример

Этот класс создает “канву”, на которой можно рисовать чертежными инструментами - пером, кистью и шрифтом. Объекты класса ЕСаnvas автоматически создаются для всех видимых компонентов, которые должны уметь нарисовать себя. Они инкапсулируют объекты Font, pen, Brush, а также многочисленные методы, использующие эти объекты.

***Свойства класса:***

|  |  |
| --- | --- |
| property Brush: TBrush; property ClipRect: TRect; | Объект-кисть Определяет текущие размеры области, нуждающейся в прорисовке |
| property CopyMode: TCopyMode; | Устанавливает способ взаимодействия растрового изображения с цветом фона |
| property Font: TFont; | Объект-шрифт |
| property Handle: Integer; | Дескриптор канвы. Используется при непосредственном обращении к API-функциям Windows |
| property LockCount: In  tegers; | Счетчик блокировок канвы. Увеличивается на единицу при каждом обращении к методу Lock и уменьшается на единицу при обращении к Unlock |
| property Pen: TPen; | Объект-перо |
| property PenPos:TPoint; | Определяет текущее положение пера в пикселях относительно левого верхнего угла канвы |
| property Pixels[X,Y:Integer]: TColor; | Массив пикселей канвы |

Свойство copyMode используется при копировании части одной канвы (источника) в другую (приемник) методом copyRect и может иметь одно из следующих значений:

|  |  |
| --- | --- |
| cmBlackness | Заполняет область рисования черным цветом |
| cmDestInvert | Заполняет область рисования инверсным цветом фона |
| cmMergeCopy | Объединяет изображение на канве и копируемое изображение операцией and |
| cmMergePaint | Объединяет изображение на канве и копируемое изображение операцией OR |
| cmNotSrcCopy | Копирует на канву инверсное изображение источника |
| cmNotSrcErase | Объединяет изображение на канве и копируемое изображение операцией OR и инвертирует полученное |
| cmPatCopy | Копирует образец источника |
| cmPatInvert | Комбинирует образец источника с изображением на канве с  помощью операции XOR |
| cmPatPaint | Комбинирует изображение источника с его образцом с помощью операции OR, затем полученное объединяется с изображением на канве также с помощью OR |
| cmSrcAnd | Объединяет изображение источника и канвы с помощью операции AND |
| cmSrcCopy | Копирует изображение источника на канву |
| cmSrcErase | Инвертирует изображение на канве и объединяет результат с изображением источника операцией AND |
| cmSrcInvert | Объединяет изображение на канве и источник операцией XOR |
| cmSrcPaint | Объединяет изображение на канве и источник операцией OR |
| cmWhitness | Заполняет область рисования белым цветом |

 С помощью свойства pixels все пиксели канвы представляются в виде двухмерного массива точек. Изменяя цвет пикселей, можно прорисовывать изображение по отдельным точкам.

Методы класса:

|  |  |
| --- | --- |
| procedure Arc(X1, Y1, X2, Y2, ХЗ, Y3, Х4, Y4: Integer); | Чертит дугу эллипса в охватывающем прямоугольнике (Xl,Yl)-(X2,Y2). Начало дуги лежит на пересечении эллипса и луча, проведенного из его центра в точку (ХЗ, Y3), а конец - на пересечении с лучом из центра в точку (Х4, Y4). Дуга чертится против часовой стрелки (см. рис. 16.4, а) |
| procedure BrushCopy  (const Dest: TRect;  Bitmap: TBitmap;  const Source: TRect;  Color: TColor); | Копирует часть изображения Source на участок канвы Dest. Color указывает цвет в Dest, который должен заменяться на цвет кисти канвы. Метод введен для совместимости с ранними версиями Delphi. Вместо него следует пользоваться классом Т Image List |
| procedure Chord(XI,  Yl, X2, Y2, X3, Y3, | Чертит сегмент эллипса в охватывающем прямоуголь нике (XI, У1) - (X2, Y2). Начало дуги сегмента лежит на пересечении эллипса и луча, проведенного из его центра в точку (X3, Y3), а конец - на пересечении с лучом из центра в точку (X4, Y4). Дуга сегмента чертится против часовой стрелки, а начальная и конечная точки дуги соединяются прямой (см. рис. 16.4, б) |
| procedure CopyRect (Dest: TRect; Canvas: TCanvas;Source: TRect) ; | Копирует изображение Source канвы Canvas в участок Dest текущей канвы. При этом разнообразные специальные эффекты достигаются с помощью свойства CopyMode |
| procedure Draw(X, Y:Integer; Graphic:TGraphic) ; | Осуществляет прорисовку графического объекта Graphic так, чтобы левый верхний угол объекта расположился в точке (X, Y) |
| procedure DrawFocusRect (const Rect:TRect) ; | Прорисовывает прямоугольник с помощью операции XOR, поэтому повторная прорисовка уничтожает ранее вычерченный прямоугольник. Используется в основном для прорисовки нестандартных интерфейсных элементов при получении ими фокуса ввода и при потере его |
| procedure Ellipse(XI, Yl, X2,  Y2: Integers; | Чертит эллипс в охватывающем прямоугольнике (XI, Yl) - (X2, Y2). Заполняет внутреннее пространство эллипса текущей кистью |
| procedure FillRect(const Rect: TRect); | Заполняет текущей кистью прямоугольную область Rect, включая ее левую и верхнюю границы, но не затрагивая правую и нижнюю границы |
| procedure Flood-  Fill(X, Y: Integer;  Color: TColor; Fill Style: TFillStyle); | Производит заливку канвы текущей кистью. Залив  ка начинается с точки (X, Y) и распространяется вовсе стороны от нее. Если FillStyle=fsSurface, заливка распространяется на все соседние точки с цветом Color. Если FillStyle=fsBorder, наоборот, заливка прекращается на точках с этим цветом |
| procedure FrameRect(const Rect: TRect); | Очерчивает границы прямоугольника Rect текущей кистью толщиной в 1 пиксель без заполнения внутренней части прямоугольника |
| procedure LineTo(X,  Y: Integer); | Чертит линию от текущего положения пера до точки  (X,Y) |
| procedure Lock; | Блокирует канву в многопоточных приложениях для предотвращения использования канвы в других по токах команд |
| procedure MoveTo(X,Y: Integer); | Перемещает перо в положение (X, Y) без вычерчивания линий |
| procedure Pie(XI,  Yl, X2, Y2, X3, Y3,  X4, Y4: Longint); | Рисует сектор эллипса в охватывающем прямоугольнике (XI, Yl) - (X2, Y2). Начало дуги лежит на пересечении эллипса и луча, проведенного из его центра в точку (X3, Y3), а конец - на пересечении случом из центра в точку (X4, Y4). Дуга чертится против часовой стрелки. Начало и конец дуги соединяются прямыми с ее центром (см. рис. 16.4, в) |
| procedure Polygon.  (Points: array of TPoint) ; | Вычерчивает пером многоугольник по точкам, заданным в массиве Points. Конечная точка соединяется с начальной и многоугольник заполняется кистью. Без заполнения используйте - Polyline |
| procedure Polyline  (Points: array of  TPoint); | Вычерчивает пером ломаную прямую по точкам, заданным в массиве Points |
| procedure Rectan  gle(XI, Yl, X2, Y2:  Integer); | Вычерчивает и заполняет прямоугольник (XI, Yl) -S(X2, Y2). Для вычерчивания без заполнения используйте FrameRect или Polyline |
| procedure Refresh; | Устанавливает в канве умалчиваемые шрифт и кисть |
| procedure RoundRect(Xl, Yl, X2,Y2, X3, Y3: Inte ger) ; | Вычерчивает и заполняет прямоугольник (XI,Yl)- (X2, Y2) со скругленными углами. Прямоугольник (XI, Yl) - (X3, Y3) определяет дугу эллипса для округления углов (см. рис. 16.4, г) |
| procedure Stretch Draw (const Rect:TRect; Graphic:  TGraphic ); | Вычерчивает и при необходимости масштабирует графический объект Graphic так, чтобы он полностью занял прямоугольник Rect |
| function TextExtent  (const Text:String): TSize; | Возвращает ширину и высоту прямоугольника, охватывающего текстовую строку Text |
| function TextHeight  (const Text: String): Integer; | Возвращает высоту прямоугольника, охватывающего текстовую строку Text |
| procedure TextOut(X,  Y: Integer; const Text: String) ; | Выводит текстовую строку Text так, чтобы левый  верхний угол прямоугольника, охватывающего текст, располагался в точке (X, Y) |
| procedure TextRect(Rect: TRect; X, Y:  Integer; const Text:  String) ; | Выводит текстовую строку Text так, чтобы левый верхний угол прямоугольника, охватывающего текст, располагался в точке (х, Y). Если при этом какая-либо часть надписи выходит из границ прямоугольника Rect, она отсекается и не будет видна |
| function TextWidth  (const Text:String): Integer; | Возвращает ширину прямоугольника, охватывающего текстовую строку Text |
| function TryLock:  Boolean; | Пытается заблокировать канву. Если она не заблокирована другим потоком команд, - True, в противном случае ничего не делает и возвращает False |
| procedure Unlock; | Уменьшает на 1 счетчик блокировок канвы |

# 28.Классы TGraphic и TPicture, методы и свойства. Пример

Важное место в графическом инструментарии Delphi занимают классыTGraphic и TPicture.

TGraphic - это абстрактный класс, инкапсулирующий общие свойства и методы трех своих потомков: пиктограммы (TIcon), метафайла (TMetafile) и растрового изображения (TBitmap). Общей особенностью потомков TGraphic является то, что обычно они сохраняются в файлах определенного формата. Пиктограммы представляют собой небольшие растровые изображения, снабженные специальными средствами, регулирующими их прозрачность. Для файлов пиктограмм обычно используется расширение ico. Метафайл - это изображение, построенное на графическом устройстве с помощью специальных команд, которые сохраняются в файле с расширением wmf или emf. Растровые изображения - это произвольные графические изображения в файлах со стандартным расширением bmp.

***Свойства класса TGraphic:***

|  |  |
| --- | --- |
| property Empty: Boolean; | Содержит True, если с объектом не связано графическое изображение |
| property Height: Integer; | Содержит высоту изображения в пикселях |
| property Modified: Boolean; | Содержит True, если графический объект изменялся |
| property Palette: HPALETTE; | Содержит цветовую палитру графического объекта |
| property PaletteModified: Boolean; | Содержит True, если менялась цветовая палитра графического объекта |
| property Transparent: Boolean; | Содержит True, если объект прозрачен для фона, на котором он изображен |
| property Width: Integer; | Содержит ширину изображения в пикселях |
| ***Методы Класса****TGraphic:* | |
| procedure LoadFromrile(const FileName:String) ; | Загружает изображение из файла FileName |
| procedure LoadFromStream(Stream: TStream); | Загружает изображение из потока данных Stream |
| procedure SaveToClipboardFormat (var AFormat: Word;var AData: THandle; varAPalette: HPALETTE); | Помещает графическое изображение Adata и его цветовую палитру APalette в буфер межпрограммного обмена в формате Af ormat |
| procedure SaveToFile(constFileName: Strings; | Сохраняет изображение в файле FileName |
| procedure SavePoStream(Stream: TStream); | Сохраняет изображение в потоке Stream |

Полнофункциональный класс TPicture инкапсулирует в себе все необходимое для работы с готовыми графическими изображениями - пиктограммой, растром или метафайлом. Его свойство Graphic может содержать объект любого из этих типов, обеспечивая нужный полиморфизм методов класса.

Свойства класса TPicture:

|  |  |
| --- | --- |
| property Bitmap: TBitmap; | Интерпретирует графический объект как растровое изображение |
| property Graphic: TGraphic; | Содержит графический объект |
| property Height: Integer; | Содержит высоту изображения в пикселях |
| property Icon: TIcon; | Интерпретирует графический объект как пиктограмму |
| property Metafile: TMetafile; | Интерпретирует графический объект как метафайл |
| property width: Integer; | Содержит ширину изображения в пикселях |

|  |
| --- |
| ***Методы класса*** ***TPicture:*** |

|  |  |
| --- | --- |
| procedure Assign(Source: TPersistent) ; | Связывает собственный графический объект Graphic с объектом Source |
| procedure LoadFromClipboardFormat  (AFormat: Word;AData: THandle;  APalette:HPALETTE) ; | Ищет в буфере межпрограммного обмена Clipboard зарегистрированный формат AFormat и, если формат найден, загружает из буфера изображение AData и его палитру APalette |
| procedure LoadFromFile(const FileName:String); | Загружает изображение из файла FileName |
| class procedure Register; ClipboardFormat(AFormat:  Word; AGraphicClass: TGraphicClass) ; | Используется для регистрации в Clipboard  нового формата изображения |
| class procedure Register; FileFormat (const AExtension, ADescription: String; AGraphicClass: Tgraphic Class) ; | Используется для регистрации нового файлового формата |
| class procedure RegisterFileFormatRes(const AExtension: String; ADescriptionResID: Integer; Agraphic Class: TGraphicClass); | Используется для регистрации нового формата ресурсного файла |
| procedure SaveToClipboardFormat (var AFormat: Word; var AData: THandle; varAPalette: HPALETTE); | Помещает графическое изображение AData и его цветовую палитру APalette в буфер межпрограммного обмена в формате AFormat |
| procedure SaveToFile(const  FileName: Strings; | Сохраняет изображение в файле FileName |
| class function SupportsClipboardFormat(AFormat: Word): Boolean; | Возвращает True, если формат Aformat зарегистрирован в буфере межпрограммного обмена Clipboard |
| class procedure UnregisterGraphicClass(AClass:TGraphicClass); | Делает недоступными любые графические объекты класса Aclass |

# 29.Редактор диаграмм, методы серий Series. Пример

Теперь рассмотрим компонент **Chart**. Этот компонент позволяет строить различные диаграммы и графики, которые выглядят очень эффектно (рис. 4.8). Компонент **Chart** имеет множество свойств, методов, событий, так что если все их рассматривать, то этому пришлось бы посвятить целую главу. Поэтому ограничимся рассмотрением только основных характеристик **Chart**. А с остальными вы можете ознакомиться во встроенной справке Delphi или просто опробовать их, экспериментируя с диаграммами.

Компонент **Chart** является контейнером объектов **Series** типа **TChartSeries** — серий данных, характеризующихся различными стилями отображения. Каждый компонент может включать несколько серий. Если вы хотите отображать график, то каждая серия будет соответствовать одной кривой на графике. Если вы хотите отображать 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 вы можете 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.

Разместите один или два (если захотите воспроизвести рис. 4.8) компонента **Chart** на форме и посмотрите открывшиеся в Инспекторе Объектов свойства. Приведем пояснения некоторых из них.

|  |  |
| --- | --- |
| **AllowPanning** | Определяет возможность пользователя прокручивать наблюдаемую часть графика во время выполнения, нажимая правую кнопку мыши. Возможные значения:**pmNone** — прокрутка запрещена, **pmHorizontal**, **pmVertical** или **pmBoth** — разрешена соответственно прокрутка только в горизонтальном направлении, только в вертикальном или в обоих направлениях. |
| **AllowZoom** | Позволяет пользователю изменять во время выполнения масштаб изображения, вырезая фрагменты диаграммы или графика курсором мыши (на рис. 4.8 б внизу показан момент просмотра фрагмента графика, целиком представленного на рис. 4.8 а). |
| **Title** | Определяет заголовок диаграммы. |
| **Foot** | Определяет подпись под диаграммой. По умолчанию отсутствует. Текст подписи определяется подсвойством **Text**. |
| **Frame** | Определяет рамку вокруг диаграммы. |
| **Legend** | Легенда диаграммы — список обозначений. |
| **MarginLeft , MarginRight , MarginTop, MarginBottom** | Значения левого, правого, верхнего и нижнего полей. |
| **BottomAxis, LeftAxis, RightAxis** | Эти свойства определяют характеристики соответственно нижней, левой и правой осей. Задание этих свойств имеет смысл для графиков и некоторых типов диаграмм. |
| **LeftWall, BottomWall, BackWall** | Эти свойства определяют характеристики соответственно левой, нижней и задней граней области трехмерного отображения графика (см. рис. 4.8 а, нижний график). |
| **SeriesList** | Список серий данных, отображаемых в компоненте. |
| **View3d** | Разрешает или запрещает трехмерное отображение диаграммы. |
| **View3DOptions** | Характеристики трехмерного отображения. |
| **Chart3DPercent** | Масштаб трехмерности (для рис. 4.8 это толщина диаграммы и ширина лент графика). |

Рядом со многими из перечисленных свойств в Инспекторе Объектов расположены кнопки с многоточием, которые позволяют вызвать ту или иную страницу Редактора Диаграмм — многостраничного окна, позволяющего установить все свойства диаграмм. Вызов Редактора Диаграмм возможен также двойным щелчком на компоненте **Chart** или щелчком на нем правой кнопкой мыши и выбором команды Edit Chart во всплывшем меню.

Если вы хотите попробовать воспроизвести приложение, показанное на рис. 4.8, сделайте двойной щелчок на верхнем компоненте **Chart**. Вы попадете в окно Редактора Диаграмм (рис. 4.9) на страницу Chart, которая имеет несколько закладок. Прежде всего вас будет интересовать на ней закладка Series. Щелкните на кнопке Add — добавить серию. Вы попадете в окно (рис. 4.10), в котором вы можете выбрать тип диаграммы или графика. В данном случае выберите Pie — круговую диаграмму. Воспользовавшись закладкой Titles вы можете задать заголовок диаграммы, закладка Legend позволяет задать параметры отображения легенды диаграммы (списка обозначений) или вообще убрать ее с экрана, закладка Panel определяет вид панели, на которой отображается диаграмма, закладка 3D дает вам возможность изменить внешний вид вашей диаграммы: наклон, сдвиг, толщину и т.д.

Когда вы работаете с Редактором Диаграмм и выбрали тип диаграммы, в компонентах **Chart** на вашей форме отображается ее вид с занесенными в нее условными данными (см. рис. 4.11).

Поэтому вы сразу можете наблюдать результат применения различных опций к вашему приложению, что очень удобно.

Страница Series, также имеющая ряд закладок, дает вам возможность выбрать дополнительные характеристики отображения серии. В частности, для круговой диаграммы на закладкеFormat полезно включить опцию Circled Pie, которая обеспечит при любом размере компонента **Chart** отображение диаграммы в виде круга. На закладке Marks кнопки группы Styleопределяют, что будет написано на ярлычках, относящихся к отдельным сегментам диаграммы: Value — значение, Percent — проценты, Label — названия данных и т.д. В примере рис. 4.8 включена кнопка Percent, a на закладке General установлен шаблон процентов, обеспечивающий отображение только целых значений.

Вы можете, если хотите, добавить на этот компонент **Chart** еще одну тождественную серию, нажав на закладке Series страницы Chart кнопку Clone, а затем для этой новой серии нажать кнопку Change (изменить) и выбрать другой тип диаграммы, например, Bar. Конечно, два разных типа диаграммы на одном рисунке будут выглядеть плохо. Но вы можете выключить индикатор этой новой серии на закладке Series, а потом предоставить пользователю выбрать тот или иной вид отображения диаграммы (ниже будет показано, как это делается).

Выйдите из Редактора Диаграмм, выделите в вашем приложении нижний компонент **Chart** и повторите для него задание свойств с помощью Редактора Диаграмм. В данном случае вам надо будет задать две серии, если хотите отображать на графике две кривые, и выбрать тип диаграммы Line. Поскольку речь идет о графиках, вы можете воспользоваться закладками Axis иWalls для задания координатных характеристик осей и трехмерных граней графика.

На этом проектирование внешнего вида приложения завершается.

# 30.Набор обработчиков событий видимых компонентов OnMouseDown, OnMouseUp, OnMouseMove, OnClick, OnDblClick.Пример

При выборе элемента управления возникает событие **OnClick** типа TNotifyEvent, которое также называют**событием нажатия**. Обычно оно возникает при щелчке мышью на компоненте. При разработке приложений событие OnClick является одним из наиболее часто используемых.

Приведем в качестве примера процедуру обработки события выбора элемента Edit1:

Для некоторых компонентов событие OonClick может возникать и при других способах нажатия элемента управления, находящегося в фокусе ввода, например, для компонента **Button** — с помощью клавиши <Пробел> или <Enter>, а для компонента CheckBox — с помощью клавиши <Пробел>.

При щелчке любой кнопкой мыши генерируются еще два события: **onMouseDown** типа TMouseEvent, возникающее при нажатии кнопки мыши, и **OnMouseUp** типа TMouseEvent — при отпускании кнопки.

При двойном щелчке левой кнопкой мыши в области компонента, кроме того, генерируется событие**onDblClick** типа TNotifyEvent.

***События возникают в следующем порядке:***

**OnMouseDown ? OnClick ? OnMouseUp ? OnDblClick ? OnMouseDown ? OnMouseUp.**

# 31. События клавиатуры. Свойства компонентов, связанных с обрботкой событий. Пример

Еще одним важным средством, с помощью которого пользователь может управлять работой приложения, является клавиатура. При всей кажущейся сложности этого технического устройства существуют всего три события, связанных с клавиатурой:

- OnKeyDown – нажатие любой клавиши;

- OnKeyPress – нажатие клавиш ASCII-символов;

- OnKeyUp – отпускание любой клавиши.

Только активный компонент, который имеет фокус ввода, может обрабатывать события клавиатуры (из этого следует, что это могут быть только потомки класса TWinControl). Если активных элементов нет, то события от клавиатуры будет получать форма. Кроме того, можно настроить работу приложения так, чтобы события клавиатуры всегда перехватывала форма, а не активный элемент управления. Для этого достаточно установить свойство KeyPreview формы в true.

При возникновении событий от клавиатуры для активного компонента вызываются соответствующие процедуры их обработки. Событие OnKeyPress обычно используется в том случае, когда необходима реакция на нажатие одной нефункциональной (символьной) клавиши на клавиатуре. Например, оно не возникает при нажатии таких клавиш, как Shift или F4. События OnKeyDown и OnKeyUp чаще всего используются для определения нажатия комбинации клавиш, а также нажатия различных функциональных клавиш.

Рассмотрим обработку событий клавиатуры на примере.

Допустим, необходимо создать приложение с формой, в заголовке которой можно было бы видеть, какая символьная клавиша была нажата, а на метке Label, размещенной на форме, отображался бы код отпущенной клавиши. Выход из приложения должен осуществляться по нажатию клавиши F10. Для решения поставленной задачи, прежде всего, нужно установить свойство формы KeyPreview = true. Таким образом, все события от клавиатуры бу-дет получать форма. А чтобы форма обрабатывала события, как указано выше, требуются следующие обработчики событий:

procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);

begin

Form1.Caption:= 'Нажата клавиша: ' + Key;

end;

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

begin

if Key = VK\_F10 then Form1.Close;

end;

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

begin

Label1.Caption:= 'Код нажатой клавиши: ' + IntToStr(Key);

end;

В представленных обработчиках событий параметр Key позволяет узнать, какая клавиша была нажата, а значение параметра Shift дает возможность определить, были ли нажаты клавиши Alt, Ctrl, Shift, а также кнопки мыши.

В обработчике события OnKeyDown для проверки нажатия клавиши F10 используется стандартная константа VK\_F10, которая содержит код данной клавиши. В Delphi существует полный набор таких констант для каждой клавиши. Их полный перечень можно найти в справочной литературе.

# 32. Механизм действий. Механизм DragAndDrop. Свойства класса TControl. Пример

Интерфейс переноса Drag-and-Drop

Интерфейс переноса и приема компонентов появился достаточно давно. Он обеспечивает взаимодействие двух элементов управления во время выполнения приложения. При этом могут выполняться любые необходимые операции. Несмотря на простоту реализации и давность разработки, многие программисты (особенно новички) считают этот механизм малопонятным и экзотическим. Тем не менее использование Drag-and-Drop может оказаться очень полезным и простым в реализации. Сейчас мы в этом убедимся.

Для того чтобы механизм заработал, требуется настроить соответствующим образом два элемента управления. Один должен быть источником (Source), второй — приемником (Target). При этом источник никуда не перемещается, а только регистрируется в качестве такового в механизме.

**Примечание**

Один элемент управления может быть одновременно источником и приемником.

Пользователь помещает указатель мыши на нужный элемент управления, нажимает левую кнопку мыши и, не отпуская ее, начинает перемещать курсор ко второму элементу. При достижении этого элемента пользователь отпускает кнопку мыши. В этот момент выполняются предусмотренные разработчиком действия. При этом первый элемент управления является источником, а второй — приемником.

После выполнения настройки механизм включается и реагирует на перетаскивание мышью компонента-источника в приемник. Группа методов-обработчиков обеспечивает контроль всего процесса и служит для хранения исходного кода, который разработчик сочтет нужным связать с перетаскиванием. Это может быть передача текста, значений свойств (из одного редактора в другой можно передать настройки интерфейса, шрифта и сам текст); перенос файлов и изображений; простое перемещение элемента управления с места на место и т. д. Пример реализации Drag-and-Drop в Windows — возможность переноса файлов и папок между дисками и папками.

Как видите, можно придумать множество областей применения механизма Drag-and-Drop. Его универсальность объясняется тем, что это всего лишь средство связывания двух компонентов при помощи указателя мыши. А конкретное наполнение зависит только от фантазии программиста и поставленных задач.

Весь механизм Drag-and-Drop реализован в базовом классе TControl, который является предком всех элементов управления. Рассмотрим суть механизма.

Любой элемент управления из Палитры компонентов Delphi является источником в механизме Drag-and-Drop. Его поведение на начальном этапе переноса зависит от значения свойства

type TDragMode = (dmManual, dmAutomatic);

property DragMode: TDragMode;

Значение dmAutomatic обеспечивает автоматическую реакцию компонента на нажатие левой кнопки мыши и начало перетаскивания — при этом механизм включается самостоятельно.

Значение dmManual (установлено по умолчанию) требует от разработчика обеспечить включение механизма вручную. Этот режим используется в том случае, если компонент должен реагировать на нажатие левой кнопки мыши как-то иначе. Для инициализации переноса используется метод

procedure BeginDrag(Immediate: Boolean;

Threshold: Integer = -1);

Параметр immediate = True обеспечивает немедленный старт механизма. При значении False механизм включается только при перемещении курсора на расстояние, определенное параметром Threshold.

О включении механизма сигнализирует указатель мыши — он изменяется на курсор, определенный в свойстве

property DragCursor: TCursor;

Еще раз напомним, что источник при перемещении курсора не изменяет собственного положения, и только в случае успешного завершения переноса сможет взаимодействовать с приемником.

Приемником может стать любой компонент, в котором создан метод-обработчик

procedure DragOver(Source: TObject; X, Y: Integer; State: TDragState;

var Accept: Boolean);

Он вызывается при перемещении курсора в режиме Drag-and-Drop над этим компонентом. В методе-обработчике можно предусмотреть селекцию источников переноса по нужным атрибутам.

Если параметр Accept получает значение True, то данный компонент становится приемником. Источник переноса определяется параметром source. Через этот параметр разработчик получает доступ к свойствам и методам источника. Текущее положение курсора задают параметры X и Y. Параметр state возвращает информацию о характере движения мыши:

type TDragState = (dsDragEnter, dsDragLeave, dsDragMove);

dsDragEnter — указатель появился над компонентом; dsDragLeave — указатель покинул компонент; dsDragMove — указатель перемещается по компоненту.

Приемник должен предусматривать выполнение некоторых действий в случае, если источник завершит перенос именно на нем. Для этого используется метод-обработчик

type TDragDropEvent = procedure(Sender, Source: TObject; X, Y: Integer)

of object;

property OnDragDrop: TDragDropEvent;

который вызывается при отпускании левой кнопки мыши на компоненте-приемнике. Доступ к источнику и приемнику обеспечивают параметры Source и Sender соответственно. Координаты мыши возвращают параметры X и Y.

При завершении переноса элемент управления — источник — получает соответствующее сообщение, которое обрабатывается методом

type TEndDragEvent = procedure(Sender, Target: TObject; X, Y: Integer)

of object;

property OnEndDrag: TEndDragEvent;

Источник и приемник определяются параметрами Sender и Target соответственно. Координаты мыши определяются параметрами X и Y.

Для программной остановки переноса можно использовать метод EndDrag источника (при обычном завершении операции пользователем он не используется):

procedure EndDrag(Drop: Boolean);

Параметр Drop = True завершает перенос. Значение False прерывает перенос.

Теперь настало время закрепить полученные знания на практике. Рассмотрим небольшой пример. В проекте DemoDragDrop на основе механизма Drag-and-Drop реализована передача текста между текстовыми редакторами и перемещение панелей по форме (рис. 27.1).

Для однострочного редактора Edit1 определены методы-обработчики источника. В методе EditiMouseDown обрабатывается нажатие левой кнопки мыши

и включается механизм переноса. Так как свойство DragMode для Edit1 имеет значение dmManual, то компонент без проблем обеспечивает получение фокуса и редактирование текста.

Метод EditiEndDrag обеспечивает отображение информации о выполнении переноса в источнике.

Для компонента Edit2 определены методы-обработчики приемника. Метод Edit2DragOver проверяет класс источника и разрешает или запрещает прием.

Метод Edit2DragDrop осуществляет перенос текста из источника в приемник.

**Примечание**

Обратите внимание, что оба компонента TEdit одновременно являются источниками и приемниками. Для этого каждый из них использует методы-обработчики другого. А исходный код методов настроен на обработку владельца как экземпляра класса TEdit.

Форма, как приемник Drag-and-Drop, обеспечивает перемещение панели Panel2, которая выступает в роли источника. Метод FormDragOver запрещает прием любых компонентов, кроме панелей. Метод FormDragDrop осуществляет перемещение компонента.

Панель не имеет своих методов-обработчиков, т. к. работает в режиме dmAutomatic и не нуждается в дополнительной обработке завершения переноса.  
  
Copied from: <http://articles.org.ru/delphi7/Glava27/index2.php>

# 33. Механизм DragAndDock реализации причаливания в классе TControl. Пример

Интерфейс присоединения Drag-and-Dock

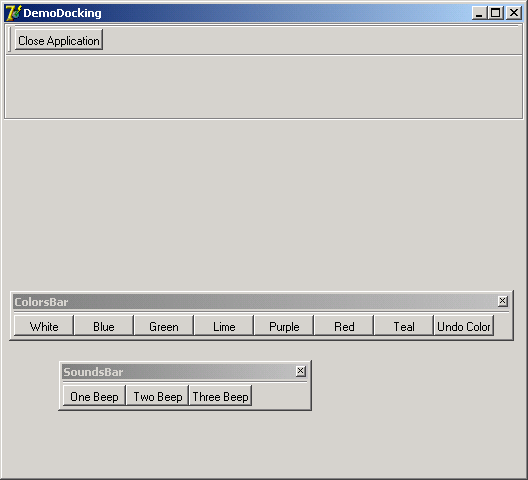
Эта возможность появилась в Delphi 4. Она "подсмотрена" опять-таки у разработчиков из Microsoft, внедривших плавающие панели инструментов в MS Office, Internet Explorer и другие продукты (рис. 27.2).

Речь идет о том, что ряд элементов управления (а конкретно — потомки класса xwinControl) могут служить носителями (доками) для других элементов управления с возможностью их динамического перемещения из одного дока в другой при помощи мыши. Перетаскивать можно практически все — от статического текста до форм включительно. Пример использования техники Drag-and-Dock дает сама среда разработки Delphi — с ее помощью можно объединять на экране различные инструменты, такие как Инспектор объектов и Менеджер проекта.

Как и в случае с технологией перетаскивания Drag-and-Drop, возможны два варианта реализации техники Drag-and-Dock: автоматический и ручной. В первом случае дело сводится к установке нужных значений для нескольких свойств, а остальную часть работы берет на себя код VCL; во втором, как следует из названия, вся работа возлагается на программиста.

Итак, что же нужно сделать для внедрения Drag-and-Dock? В Инспекторе объектов необходимо изменить значение свойства DragKind на dkDock, a свойства DragMode — наdmAutomatic. Теперь этот элемент управления можно перетаскивать с одного носителя-дока на другой.

Носителем других компонентов (доком) может служить потомок TwinControl. У него есть свойство Docksite, установка которого в True разрешает перенос на него других компонентов. Если при этом еще и установить свойство AutoSize в True, док будет автоматически масштабироваться в зависимости от того, что на нем находится. В принципе, этими тремя операциями исчерпывается минимальный обязательный набор.



***Рис. 27.2.****Плавающие панели инструментов*

Естественно, для программиста предусмотрены возможности контроля за этим процессом. Каждый переносимый элемент управления имеет два события, возникающие в моменты начала и конца переноса:

type TStartDockEvent = procedure(Sender: TObject;

var DragObject: TDragDockObject) of object;

TEndDragEvent = procedure(Sender, Target: TObject; X, Y: Integer) of object;

В первом из методов sender — это переносимый объект, a DragObject — специальный объект, создаваемый на время процесса переноса и содержащий его свойства. Во втором sender — это также переносимый объект, a Target — объект-док.

Док тоже извещается о событиях во время переноса:

type TGetSitelnfoEvent = procedure(Sender: TObject; DockClient: TControl;

var InfluenceRect: TRect; MousePos: TPoint;

var CanDock: Boolean)

of object;

TDockOverEvent = procedure(Sender: TObject; Source: TDragDockObject;

X, Y: Integer; State: TDragState; var Accept: Boolean) of object;

TDockDropEvent = procedure(Sender: TObject;

 Source: TDragDockObject;

X, Y: Integer) of object;

TUnDockEvent = procedure(Sender: TObject; Client: TControl; NewTarget:

TWinControl; var Allow: Boolean) of object;

Как только пользователь нажал кнопку мыши над переносимым компонентом и начал сдвигать его с места, всем потенциальным докам (компонентам, свойство которыхDocksite установлено в True) рассылается событие onGetsiteinfo. С ним передаются параметры: кто хочет "приземлиться" (параметр Dockclient) и где (MousePos). В ответ док должен сообщить решение, принимает он компонент (параметр CanDock) и предоставляемый прямоугольник (infiuenceRect) или нет. При помощи этого события можно принимать только определенные элементы управления, как показано в примере:

procedure TForml.PanellGetSitelnfо(Sender: TObject; DockClient: TControl; var InfiuenceRect:

TRect; MousePos: TPoint; var CanDock: Boolean);

begin

if DockClient is TBitBtn then CanDock := False;

end;

Два последующих события в точности соответствуют своим аналогам из механизма переноса Drag-and-Drop). Событие onDockOver происходит при перемещении перетаскиваемого компонента над доком, OnDockDrop — в момент его отпускания. Наконец, onUnDock сигнализирует об уходе компонента с дока и происходит в момент его "приземления" в другом месте.

Между доком и содержащимися на нем элементами управления есть двусторонняя связь. Все "припаркованные" элементы управления содержатся в векторном свойствеDockclients, а их количество можно узнать из свойства

DockClientCount:

s : = ' ' ;

for i := 0 to Panell.DockClientCount-1

do AppendStr(s,Panell.DockClients[i].Name+#$D#$A); ShowMessage(s) ;

С другой стороны, если элемент управления находится на доке, то ссылка на док располагается в свойстве HostDocksite. С ее помощью можно установить, где находится элемент, и даже поменять свойства дока:

procedure TMyForm.ButtonlEndDock(Sender, Target: TObject; X, Y: Integer); begin

(Sender as TControl).HostDockSite.SetTextBuf(pChar((Sender as TControl).Name));

end;

Компоненты можно не только переносить с одного дока на другой, но и отпускать в любом месте. Хотя сам по себе компонент TControl и его потомки не являются окнами Windows, но специально для этого случая создается окно-носитель. Свойство FloatingDockSiteClass как раз и определяет класс создаваемого окна. По умолчанию для большинства компонентов значение этого свойства равно TCustomDockForm. Это — форма, которая обладает свойствами дока и создается в момент отпускания элемента управления вне других доков. Внешне она ничем не отличается от обычной стандартной формы. Если вы хотите, чтобы ваша плавающая панель инструментов выглядела по- особенному, нужно породить потомка от класса TCustomDockForm и связать свойство FloatingDockSiteCiass с этим порожденным классом:

TMyCustomFloatingForm = class(TCustomDockForm)

 public

constructor Create(AOwner: TComponent);

 override;

 end;

constructor TMyCustomFloatingForm.Create(AOwner: TComponent};

 begin

inherited Create(AOwner);

BorderStyle := bsNone;

 end;

procedure TForml.FormCreate(Sender: TObject);

 begin

ToolBarl.FioatingDockSiteCiass := TMyCustomFloatingForm; end;

В этом примере решена типовая задача — сделать так, чтобы несущее окно плавающей панели инструментов не содержало заголовка. Внешний вид таких панелей приведен на рис. 27.3.

Переносить компоненты можно не только с помощью мыши, но и программно. Для этого есть пара методов ManualDock и ManualFioat. В приводимом ниже примере нажатие кнопки с именем BitBtnl переносит форму custForm на док MainForm.Paneil и размещает ее по всей доступной площади (параметр выравнивания alclient). Нажатие кнопки BitBtn2 снимает эту форму с дока и выравнивает ее по центру экрана. В свойствах UndockHeight и undockwidth хранятся высота и ширина элемента управления на момент, предшествующий помещению на док:

procedure TMainForm.BitBtnlClick(Sender: TObject);

 begin

GustForm.ManualDock

(MainForm.Pane11,nil,alClient);

 end;

procedure TMainForm.BitBtn2Click(Sender: TObject);

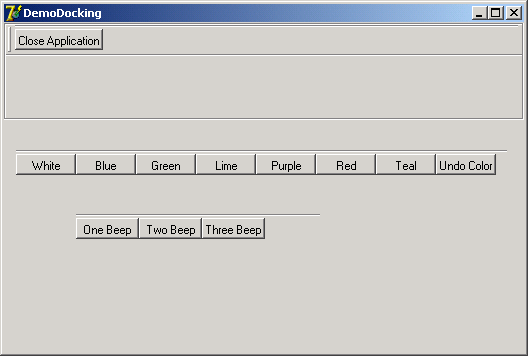
 begin

with CustForm do

begin ManualFloat(Rect((Screen.Width-UndockWidth) div 2,

(Screen.Height-UndockHeight) div 2, (Screen.Width+UndockWidth) div 2, (Screen.Height+UndockHeight) div 2) );

 end;



***Рис. 27.3.****Плавающие панели инструментов без заголовка окна*

Полное рассмотрение внутреннего устройства механизмов Drag-and-Dock потребовало бы расширения объема этой главы. Тем, кто хочет использовать их на все 100%, рекомендуем обратиться к свойствам useDockManager и DockManager. Последнее представляет собой СОМ-интерфейс, позволяющий расширить возможности дока, вплоть до записи его состояния в поток (классTStream).

Усовершенствованное масштабирование

В класс TControl добавлены свойства, позволяющие упростить масштабирование форм и находящихся на них компонентов.

Свойство Anchors:

TAnchorKind = (akLeft, akTop, akRight, akBottom);

TAnchors = set of TAnchorKind; property Anchors: TAnchors;

отвечает за привязку компонентов к определенным краям формы при масштабировании. По умолчанию любой компонент привязан к верхней и левой сторонам ([akLeft, akTop]), т. е. не двигается при стандартном масштабировании. Но, изменив значение этого свойства, можно сделать так, чтобы компонент находился, к примеру, все время в нижнем правом углу.

С другой стороны, если прикрепить все четыре стороны, то получится интересный и нужный во многих случаях эффект. Такой компонент увеличивается и уменьшается вместе с формой; но в то же время сохраняется расстояние до всех четырех ее краев.

Свойство constraints представляет собой набор ограничений на изменение размеров компонента. Оно содержит четыре свойства: MaxHeight, Maxwidth, MinHeight и Minwidth. Как легко догадаться из названий, размеры компонента могут меняться только в пределах значений этих четырех свойств.

Наконец, большинство элементов управления получили свойство Autosize, позволяющее им автоматически масштабироваться при изменении содержимого (скажем, надписи на кнопке).