1. **Поняття події та властивості як елемента класу.**

**Класом**в мові програмування називається особливий тип який має в своєму складі змінні (поля, властивості, події) та методи.

**Змінні класу:**

* **поле** — змінна одного з класичних типів мови, унікальна для кожного екземпляру;
* **об’єкт**— змінна класу об’єктного типу;
* **властивість** — змінна, доступ до якої можна здійснити тільки з використанням спеціальних методів, пов'язаних з даною властивістю.

Значення змінних процедурного типу, тобто імена процедур обробки події встановлюються автоматично, в той час, коли розробник визначає намір створити таку процедуру за допомогою інспектора об'єктів. Перелік визначених для класу подій відображається в вікні інспектора об'єктів.

Враховуючи той факт що події є властивостями, можна в програмі передбачати присвоєння для них значень, тобто в будь-який момент часу можна взяти засоби реакції на події одного об'єкту та присвоїти їх іншому. Така можливість визначається як**делегування подій***.* Делегування подій можна виконувати також при проектуванні форми за допомогою інспектора об'єктів.

Події являються окремими елементами в складі класу.

Поняття "**подія**" використовується по різному в залежності від конкретного контексту.

* По перше як реакція операційної системи на дії оператора. Наприклад, внаслідок дії оператора "натискання на ліву клавішу миші" система реагує створенням для себе спеціальної ситуації – події "виконався натиск на ліву клавішу миші". Суттю ситуації є створення спеціальної інформаційної структури — "повідомлення Windows" яка містить інформацію про параметри натиску, наприклад – про координати курсору миші під час натиску. Така ситуація термінологічно визначається як "генерація події" або "виникнення події".
* По друге подія є елементом класу в ООП, а саме властивістю процедурного типу, тобто такою змінно в складі класу значенням якої має бути ім'я деякої процедури.

1. **Поняття компонента як елемента класу.**

Класс **TComponent** представляет собой вершину в иерархии компонентов. От него порождены все компоненты, используемые в приложении, включая само приложение (класс TApplication) и форму (класс TForm). Прямыми потомками класса TComponent являются невизуальные компоненты. Класс является также базой для создания визуальных компонентов.

TObject

TPersistent

TComponent

TControl

TGraphicControl

TWinControl

Бібліотека візуальних компонентів (Visual Component Library - VCL) Delphi містить безліч визначених типів компонентів, з яких користувач може будувати свою прикладну програму. Вітрину бібліотеки — палітру компонентів, ви бачите розташовану праворуч у смузі інструментальних панелей інтегрованого середовища розробки Delphi. На цій палітрі ви можете виділити курсором миші потрібний вам компонент і перенести його на форму.



**3.Використання головних та вспливаючих меню в підпрограмі.**

В додатках надається можливість формувати меню двох видів:

* **Головне** **меню**. Таке меню належить формі та відображається під рядком заголовку вікна форми. В разі коли додаток складається з багатьох форм для зручності використання можна поєднати меню.
* **Спливаюче** (**контекстне**) **меню**. Таке меню передбачено для більшості компонент елементів керування Windows. Воно виникає спливаючи при натисненні на праву кнопку миші на клієнтській області компоненти.

З пунктом меню пов’язаний список команд меню – пункти з якими пов’язана процедура обробки подій активізації пункту меню. Команди можна поєднувати до підменю.

**Підменю** – це такі пункти, обрання яких, показує або згортає команди, що входять до нього.

Принципи створення та використання меню полягають в тому, що кожний пункт меню є об’єктом класу *TMenuItem*, який створюється автоматично в процесі побудови меню. Проектування меню можливо виконувати або за допомогою спеціального *Конструктора меню*, який входить до складу середовища або за допомогою спеціальних методів.

До складу об’єктів Delphi включено спеціальний компонент *TМenu* за схемою *TObject - TPrsistent - TComponent – TМenu*

Компонент відсутній в палітрі компонентів. Він містить методи та властивості які є загальними для двох його нащадків *TMainMenu* (опис головного меню) і *TPopupMenu* (опис спливаючого меню).

**Основні властивості:**

* *Handle* : *HMENU* — дескриптор меню
* *WindowHandle* : *HWND* — дескриптор або форми, або віконного елемента управління, з яким пов’язане з меню.
* *Items* : *TMenuItem* — містить перелік елементів меню.

**Основні методи:**

*function* *DispatchCommand*( ) — знаходить пункт меню з заданою командою та викликає метод *Click* для нього;

*function* *FindItem* — повертає посилання на об’єкт що відповідає пункту меню.

*function* *IsShortCut* — визначає комбінацію гарячих клавіш та викликає метод *Click* для нього.

**4. MDI-інтерфейс і його характеристика. Приклади застосування**

Застосування, що створюються для реалізації інформаційної системи, як правило, потребують деякої кількості форм. При проектуванні таких додатків можна використати один з **двох принципів створення інтерфейсу користувача**:

* Багатовіконний інтерфейс — MDI (Multiple Document Interface ).
* Одновіконній інтерфейс — SDI (Single Document Interface ).

**Створене за принципом MDI застосування** має батьківське вікно, в клієнтській області якого відображаються одне або більше дочірніх вікон. Ці вікна, можуть бути згорнуті, розгорнуті, змінені за розміром, але не можуть вийти за межі батьківського вікна. Їх можна по різному розташовувати в межах батьківського вікна, тобто мозаїкою або каскадом. При цьому батьківське вікно містить меню за допомогою якого здійснюється керування підпорядкованими вікнами, які не мають власного меню.

Наведемо основні вимоги щодо проектування MDI-застосування, що містить батьківське і дочірнє вікна:

1. Для батьківської форми необхідно задати значення *FormStyle := fsMDIForm*.
2. Для форм, що будуть дочірніми для батьківської необхідно задати значення *FormStyle := fsMDIСchild***.**

Якщо дочірня форма не створюється автоматично, то на її основі можна створити багато дочірніх форм шляхом виконання методу *Create* в процедурі обробки обраної події, наприклад обрання рядку меню. Для цього, наприклад, створимо головне меню форми і використаємо наступну процедуру:

*procedure TForm1.N2Click(Sender: TObject); // N2 — назва кнопки пункту меню*

*var*

*Form2 : TForm2;*

*begin*

*Form2 := TForm2.Create (Self);*

*end;*

У цій процедурі, в секції var оголошується об'єкт з ім'ям *Form2*, що належить до класу *TForm2*, тобто дочірня форма, яка була оголошена користувачем. Звичайно цей клас є нащадком класу *TForm*.

Використання властивостей типу *Read* і *Runtime Only* недоступне в інспекторі об'єктів під час розробки. Такими властивостями для форми є:

* *MDIChildCount* — ціла величина, що містить кількість відкритих дочірніх вікон MDI.
* *ActiveMDIChild* — значення TForm, що може використовуватися як дескриптор для маніпуляцій з активним підпорядкованими вікнами.
* *MDIChildren* — масив TForm з цілими індексами, що містить дескриптори дочірніх вікон в тому порядку, в якому їх відкривали.

**Методи керування підпорядкованими MDI вікнами**:

1. **Створення дочірніх вікон MDI**:

Для створення дочірніх вікон викликається метод *Create* наприклад:

*procedure TForm1.N2Click(Sender: TObject);*

*var*

*Form2: TForm2;*

*begin*

*Form2 := TForm2.Create (Self);*

*Form2.Caption := IntToStr (MDIChildCount); //у заголовок форми виведеться порядковий номер дочірньої форми*

*end;*

1. **Розташування піктограм:**

Метод *ArrangeIcons* розташовує піктограми згорнутих дочірніх вікон MDI таким чином, щоб вони рівномірно розташувались вздовж нижнього краю батьківського вікна. Метод повинен використовуватися тільки батьківським вікном (таким, у яких встановлено значення властивості *FormStyle := fsMDIForm*). Приклад звернення:

*procedure TForm1.N3Click(Sender: TObject);*

*begin*

*Form1.ArrangeIcons;*

*end;*

1. **Розташування вікон MDI каскадом**:

Метод *Cascade*. Розташовує дочірні вікна так, що вони не перекривалися. Приклад звернення:

*procedure TForm1.N4Click(Sender: TObject);*

*begin*

*Cascade;*

*end;*

1. **Розташування вікон MDI мозаїкою**:

Метод *Tile*. Розташовує дочірні вікна так, що вони не перекривалися. Область клієнта в батьківському вікні поділяється на зони, кожна з яких містить одне дочірнє вікно MDI. Приклад звернення:

*procedure TForm1.N5Click(Sender: TObject);*

*begin*

*Tile;*

*end;*

За замовчанням вікна витягнуті горизонтально. Для задання вертикального або горизонтального розміщення дочірніх вікон, властивість *TileMode* у батьківській формі встановлюється рівною *tbVertical* або *tbHorizontal*. Наприклад:

*TileMode:=tbHorizontal;*

*TileMode:=* *tbVertical;*

1. **Закриття поточного дочірнього вікна**:

Метод *Close* закриває дочірнє або батьківське вікно MDI. Метод може бути використаний для закриття активного дочірнього вікна, для цього необхідно використати властивість *ActiveMDIChild* батьківського вікна. Приклад звернення:

*procedure TForm1.N6Click(Sender: TObject);*

*begin*

*ActiveMDIChild.Close;*

*end;*

Оскільки метод *Close* не є засобом MDI, заміна на некваліфікований виклик *Close* приведе до закриття батьківського вікна. Закриття дочірнього MDI вікна мінімізує його в клієнтській частині батьківського вінка, тому ми повинні встановити у процедурі *OnClose* параметр *Action := caFree*.

1. **Обрання наступного нащадка MDI:**

Метод Next активізує наступне дочірнє вікно MDI у послідовності дочірніх вікон. Наприклад, якщо викликається метод Next батьківського вікна , і поточне підпорядковане вікно є останнім у списку (хоча б з двох вікон всього), активним стане перше дочірнє вікно з цього списку. Приклад звернення:

*procedure TForm1.N7Click(Sender: TObject);*

*begin*

*Next;*

*end;*

1. **Обрання попереднього дочірнього вікна.**

Метод *Previous* діє аналогічно методу *Next*, але вибирає дочірні вікна в зворотному порядку.

5. **SDI-інтерфейс і його характеристика. Приклади застосування**

Застосування, що створюються для реалізації інформаційної системи, як правило, потребують деякої кількості форм. При проектуванні таких додатків можна використати один з **двох принципів створення інтерфейсу користувача**:

* Багатовіконний інтерфейс — MDI (Multiple Document Interface ).
* Одновіконній інтерфейс — SDI (Single Document Interface ).

**Створене за принципом SDI застосування** може складатися з одного вікна. Повноцінне SDI-застоcування складається з головного вікна та сімейства дочірніх вікон. Коли головне вікно згортається, розгортається або закривається то спочатку відповідні дії виконуються над підпорядкованими вікнами. Кожне вікно має власне меню та розташовується на екрані незалежно від розташування інших вікон.

Керування сімействами вікон SDI схоже на керування їх MDI-аналогами. Спочатку необхідно ідентифікувати кожне вікно як головне, або як члена сімейства. Головним вікном буде вікно першої форми проекту, якщо за допомогою менеджера проектів головним не визначено інше вікно. Всі вікна, крім першого будуть дочірніми вікнами. Для всіх форм сімейства необхідно встановити значення властивості *FormStyle := fsNormal*. У головного SDI-вікна немає специфічних властивостей, таких як *ActiveMDIChild*. Члени сімейства ведуть себе як звичайні вікна, що дозволять користувачу їх відкривати, згортати, розгортати на весь екран і закривати. Але коли користувач виконує будь-яку з цих операцій на головному вікні, необхідно пересилати керуючі запити також членам сімейства, наприклад, закривати вікна членів сімейства перед закриттям основного вікна. У результаті одержується інтерфейс, схожий на інтерфейс MDI, але вони володіють більшою гнучкістю.

**Модальною** називається форма, яка повинна бути закрита перед зверненням до будь-якої іншої форми програми. Якщо користувач хоче перейти в іншу форму, не закривши поточну модальну форму, ОС блокує цю спробу і видає попереджувальний сигнал. Така заборона стосується лише поточної програми і користувач може активізувати будь-яку іншу програму. Однак програмний доступ до компонент будь-якої створеної форми є можливим, незважаючи на наявність в даний момент відкритої модальної форми.

Модальні форми часто називають інформаційними формами, діалоговими формами (або діалоговими панелями), хоча й існують немодальні діалогові та інформаційні вікна .

Для відображення форми в модальному режимі служить метод *ShowModal,* на відміну від звичайного (не модалнього) *Show*. Після виведення форми метод не завершує роботу, а чекає її закриття. Як правило, закінчення роботи з модальною формою відбувається при натисненні на розміщену на ній кнопку. Обробник події натиснення цієї кнопки повинен містити оператор, який присвоює властивості *ModalResult* ненульове значення. Існують стандартні константи, які необхідно використовувати як значення властивості *ModalResult*. Значення цієї властивості можна використовувати програмно для організації розгалуження. Однак, як правило, керування кодом результату діалогу виконується не програмно (через властивість *ModalResult* форми), а за допомогою кнопок підтвердження або відміни виконаних операцій.

Візуальне середовище Delphi дозволяє застосовувати в програмах **стандартні вікна**, імпортовані з WinAPI.

**6-23.**

**№6 Многострочные окна редактирования Memo**

Компоненты ***Memo*** и ***RichEdit*** являются окнами редактирования многострочного текста. Они так же, как и окно ***Edit***, снабжены многими функциями, свойственными большинству редакторов. В них предусмотрены типичные комбинации «горячих» клавиш: *Ctrl-C* — копирование выделенного текста в буфер обмена *Clipboard* (команда *Copy*), *Ctrl-X* — вырезание выделенного текста в буфер *Clipboard* (команда *Cut*), *Ctrl-V* — вставка текста из буфера *Clipboard* в позицию курсора (команда *Paste*), *Ctrl-Z* — отмена последней команды редактирования.

В компоненте ***Memo*** формат (шрифт, его атрибуты, выравнивание) одинаков для всего текста и определяется свойством ***Font***. Если вы сохраните в файле текст, введенный или отредактированный пользователем, то будет создан текстовый файл, содержащий только символы и не содержащий элементов форматирования. При последующем чтении этого файла в ***Memo*** формат будет определяться текущим состоянием свойства ***Font*** компонента ***Memo***, а не тем, в каком формате ранее вводился текст.

Свойства ***Alignment*** и ***WordWrap*** имеют тот же смысл, что, например, в метках, и определяют выравнивание текста и допустимость переноса длинных строк. Установка свойства ***ReadOnly*** в ***true*** задает текст только для чтения. Свойство ***MaxLength*** определяет максимальную длину вводимого текста. Если ***MaxLength = 0***, то длина текста не ограничена. Свойства ***WantReturns*** и ***WantTab*** определяют допустимость ввода пользователем в текст символов перевода строки и табуляции.

Свойство ***ScrollBars*** определяет наличие полос прокрутка текста в окне. По умолчанию ***ScrollBars*** ***=*** ***ssNone***, что означает их отсутствие. Пользователь может в этом случае перемещаться по тексту только с помощью курсора. Можно задать свойству ***ScrollBars*** значения ***ssHorizontal***, ***ssVertical*** или ***ssBoth***, что будет соответственно означать наличие горизонтальной, вертикальной или обеих полос прокрутки.

Основное свойство окон ***Memo*** и ***RichEdit*** — ***Lines***, содержащее текст окна в виде списка строк и имеющее тип ***TStrings***. Начальное значение текста можно установить в процессе проектирования, нажав кнопку с многоточием около свойства ***Lines*** в окне *Инспектора Объектов.* Перед вами откроется окно редактирования списков строк. Вы можете редактировать или вводить текст непосредственно в этом окне, или нажать кнопку *CodeEditor* и работать в обычном окне *Редактора Кода*. В этом случае, завершив работу с текстом, выберите из контекстного меню, всплывающего при щелчке правой кнопкой мыши, команду *Close Page* и ответьте утвердительно на вопрос, хотите ли вы сохранить текст в соответствующем свойстве окна редактирования.

Во время выполнения приложения вы можете заносить текст в окно редактирования с помощью методов свойства ***Lines*** типа ***TStrings***. Этот тип широко используется в свойствах многих компонентов и его описание вы можете найти в во встроенной справке Delphi. Здесь коротко укажем только на его основные свойства и методы, используемые в свойстве ***Lines***.

Весь текст, представленный одной строкой типа ***String***, внутри которой используются разделители типа символов возврата каретки и перевода строки, содержится в свойстве ***Text***.

Доступ к отдельной строке текста вы можете получить с помощью свойства ***Strings[Index: Integer]***. Индексы, как и везде в Delphi, начинаются с **0**. Так что ***Memo1.Lines.Strings[0]*** — это текст первой строки. Учтите, что если окно редактирования изменяется в размерах при работе с приложением и свойство ***WordWrap = true***, то индексы строк будут изменяться при переносах строк, так что в этих случаях индекс мало о чем говорит.

Свойство только для чтения ***Count*** указывает число строк в тексте.

Для очистки текста в окне надо выполнить процедуру ***Clear***. Этот метод относится к самому окну, а не к его свойству ***Lines***.

Для занесения новой строки в конец текста окна редактирования можно воспользоваться методами ***Add*** или ***Append*** свойства ***Lines***. Для загрузки текста из файла применяется метод ***LoadFromFile***. Сохранение текста в фале осуществляется методом ***SaveToFile***.

Пусть, например, в вашем приложении имеется окно редактирования ***Edit1***, в котором пользователь вводит имя сотрудника, и есть кнопка, при щелчке на которой в окно ***Memo1*** должна занестись шапка характеристики этого сотрудника, после чего пользователь может заполнить текст характеристики.

Обработчик щелчка на кнопке может иметь вид:

*Memo1.Clear;*

*Memo1.Lines.Add ('Х А Р А К Т Е Р И С Т И К А');*

*Memo1.Lines.Add('Сотрудник '+Edit1.Text) ;.*

*Memo1.SetFocus;*

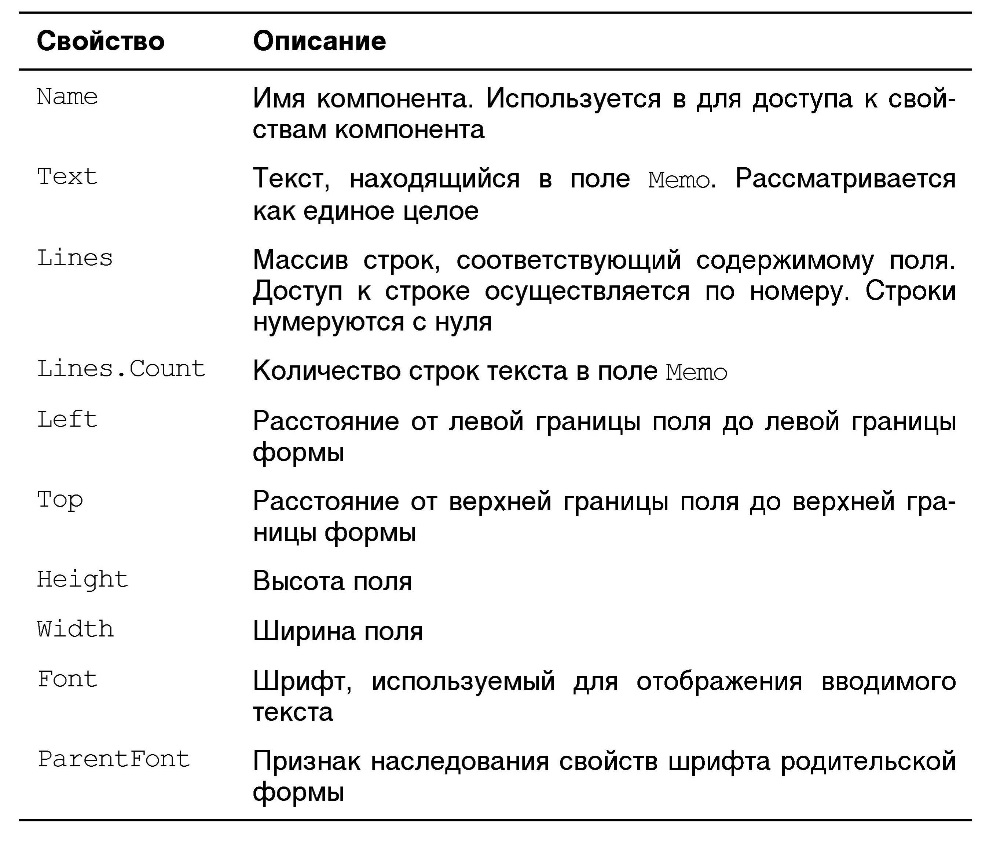
Загрузка в окно ***Memo1*** текста из файла (например, хранящейся в файле характеристики сотрудника) может осуществляться командой

*Memo1.Lines.LoadFromFile ('text.txt');*

Сохранение текста в файле может осуществляться командой

*Memo1.Lines.SaveToFile('text.txt');*

Свойство ***SelStart*** компонентов ***Memo*** и ***RichEdit*** указывает позицию курсора в тексте или начало выделенного пользователем текста. Свойство ***CaretPos*** указывает на запись, поле ***X*** которой содержит индекс символа в строке, перед которым расположен курсор, а поле ***Y*** — индекс строки, в которой находится курсор (встроенная справка Delphi утверждает другое — что свойство ***CaretPos*** содержит координаты курсора в пикселях; но, к счастью, это не так). Таким образом, учитывая, что индексы начинаются с 0, значения ***Memo1.CaretPos.Y+1*** *и****Memо1.CaretPos.X+1*** определяют соответственно номер строки и символа в ней, перед которым расположен курсор.



**№7,10 Окна редактирования Edit и MaskEdit**

В компонентах ***Edit*** и ***MaskEdit*** вводимый и выводимый текст содержится в свойстве **Text**. Это свойство можно устанавливать в процессе проектирования или задавать программно. Выравнивание текста, как это имело место в метках и панелях, невозможно. Перенос строк тоже невозможен. Текст, не помещающийся по длине в окно, просто сдвигается и пользователь может перемещаться по нему с помощью курсора. Свойство ***AutoSize*** в окнах редактирования имеет смысл, отличный от смысла аналогичного свойства меток: автоматически подстраивается под размер текста только высота, но не ширина окна.

Окна редактирования снабжены многими функциями, свойственными большинству редакторов. Например, в них предусмотрены типичные комбинации «горячих» клавиш: *Ctrl-C* — копирование выделенного текста в буфер обмена *Clipboard* (команда *Copy*), *Ctrl-X* — вырезание выделенного текста в буфер *Clipboard* (команда *Cut*), *Ctrl-V* — вставка текста из буфера *Clipboard* в позицию курсора (команда *Paste*), *Ctrl-Z* — отмена последней команды редактирования.

Свойство ***AutoSelect*** определяет, будет ли автоматически выделяться весь текст при передаче фокуса в окно редактирования. Его имеет смысл задавать равным ***true*** в случаях, когда при переключении в данное окно пользователь будет скорее заменять текущий текст, чем исправлять его. Имеются также свойства только времени выполнения ***SelLength***, ***SelStart***, ***SelText***, определяющие соответственно длину выделенного текста, позицию перед первым символом выделенного текста и сам выделенный текст. Если выделенного текста нет, то свойство **SelStart** просто определяет текущее положение курсора.

Окна редактирования можно использовать и просто как компоненты отображения текста. Для этого надо установить в **true** их свойство ***ReadOnly*** и целесообразно установить ***AutoSelect*** в ***false***. В этом случае пользователь не сможет изменять отображаемый текст и окно редактирования становится подобным меткам. Но имеются и определенные отличия. Во-первых, окна редактирования оформлены несколько иначе. А главное — окна редактирования могут вмещать текст, превышающий их длину. В этом случае пользователь может прокручивать этот текст, перемещая курсор в окне. Такими особенностями не обладает ни одна метка.

При использовании окон редактирования для вывода, ввода и редактирования чисел необходимо использовать функции взаимного преобразования строк и чисел. Для вывода это описанные при рассмотрении меток функции ***FloatToStr*** и ***IntToStr***. При вводе это функции ***StrToFloat*** — преобразование строки в значение с плавающей запятой, и ***StrToInt*** — преобразовани2е строки в целое значение. Если вводимый текст не соответствует числу (например, содержит недопустимые символы), то функции преобразования генерируют исключение ***EConvertError***. Поэтому в программе необходимо предусмотреть обработку этого исключения. Например:

***var*** *A: integer;*

***try***

*A := StrToInt(Edit1.Text);*

*...{операторы,   использующие переменную А}*

***except*** ***on*** *EConvertError****do***

*ShowMessage('Вы  ввели  ошибочное  число;* *повторите   ввод');*

Этот код обеспечивает сообщение пользователю об ошибке ввода и предотвращает ошибочные вычисления. Впрочем, это не лучший вариант предотвратить ошибочный ввод, поскольку пользователь узнает о своей ошибке только после того, как программа пытается использовать введенные данные. Лучше, если пользователь просто не сможет ввести неправильные символы. Например, если вы хотите,

чтобы пользователь мог вводить в окно редактирования ***Edit*** только цифры и символ точки, вы можете в обработчик события ***OnKeyPress*** этого компонента вставить оператор:

***if*** ***not*** *(Key****in*** *['0'..'9', ','])****then*** *Key := #0;*

Этот оператор подменит все символы, кроме цифр и запятой, нулевым символом, который не занесется в текст окна ***Edit***.

Свойство ***MaxLength*** определяет максимальную длину вводимого текста. Если ***MaxLength*** ***= 0***, то длина текста не ограничена. В противном случае значение ***MaxLength*** указывает максимальное число символов, которое может ввести пользователь.

Свойство ***Modified***, доступное только во время выполнения, показывает, проводилось ли редактирование текста в окне. Если вы хотите использовать это свойство, то в момент начала работы пользователя с текстом ***Modified*** надо установить в ***false***. Тогда при последующем обращения к этому свойству можно по его значению (***true*** или ***false***) установить, было или не было произведено редактирование.

Свойство ***PasswordChar*** позволяет превращать окно редактирования в окно ввода пароля. По умолчанию значение ***PasswordChar*** равно ***#0*** — нулевому символу. В этом случае это обычное окно редактирования. Но если в свойстве указать иной символ (например, символ звездочки «**\***»), то при вводе пользователем текста в окне будут появляться именно эти символы, а не те, которые вводит пользователь. Тем самым обеспечивается секретность ввода пароля.

Компонент ***MaskEdit*** отличается от ***Edit*** тем, что в нем можно задать строку маски в свойстве ***EditMask***. Маска состоит из трех разделов, между которыми ставится точка с запятой (;). В первом разделе — шаблоне записываются специальным образом символы, которые можно вводить в каждой позиции, и символы, добавляемые самой маской; во втором разделе записывается 1 или 0 в зависимости от того, надо или нет, чтобы символы, добавляемые маской, включались в свойство ***Text*** компонента; в третьем разделе указывается символ, используемый для обозначения позиций, в которых еще не осуществлен ввод. Прочитать результат ввода можно или в свойстве ***Text***, которое в зависимости от вида второго раздела маски включает или не включает в себя символы маски, или в свойстве ***EditText***, содержащем введенный текст вместе с символами маски.

**Таблица. Символы шаблона маски**

|  |  |
| --- | --- |
| **!** | Наличие символа «!» означает, что в ***EditText*** недостающие символы предваряются пробелами, а отсутствие символа «!» означает, что пробелы размещаются в конце. |
| **>** | Символ «>» означает, что все последующие за ним символы должны вводиться в верхнем регистре, пока не кончится маска или пока не встретится символ «<». |
| **<** | Символ «<» означает, что все последующие за ним символы должны вводиться в нижнем регистре, пока не кончится маска или пока не встретится символ «>». |
| **<>** | Символы «<>» означают, что анализ регистра не производится. |
| **\** | Символ «\» означает, что следующий за ним символ является буквенным, а не специальным, характерным для маски. Например, символ «>» после символа «\» воспримется как знак >, а не как символ, указывающий на верхний регистр. |
| **L** | Символ «L» означает, что в данной позиции должна бытъ буква. |
| **l** | Символ «l» означает, что в данной позиции может быть только буква или ничего. |
| **A** | Символ «А» означает, что в данной позиции должна быть буква или цифра. |
| **a** | Символ «а» означает, что в данной позиции может быть буква, или цифра, или ничего. |
| **C** | Символ «С» означает, что в данной позиции должен быть любой символ. |
| **c** | Символ «с» означает, что в данной позиции может быть любой символ или ничего. |
| **0** | Символ «0» означает, что в данной позиции должна быть цифра. |
| **9** | Символ «9» означает, что в данной позиции может быть цифра или ничего. |
| **#** | Символ «#» означает, что в данной позиции может быть цифра, знак «+», знак «-» или ничего. |
| **:** | Символ «:» используется для разделения часов, минут и секунд. |
| **/** | Символ «/» используется для разделения месяцев, дней и годов в датах. |
|  | Символ « » означает автоматическую вставку в текст пробела. |

Вводить маску можно непосредственно в свойство ***EditMask***. Но удобнее пользоваться специальным редактором масок, вызываемым при нажатии кнопки с многоточием в строке свойства ***EditMask*** в *Инспекторе Объектов*.

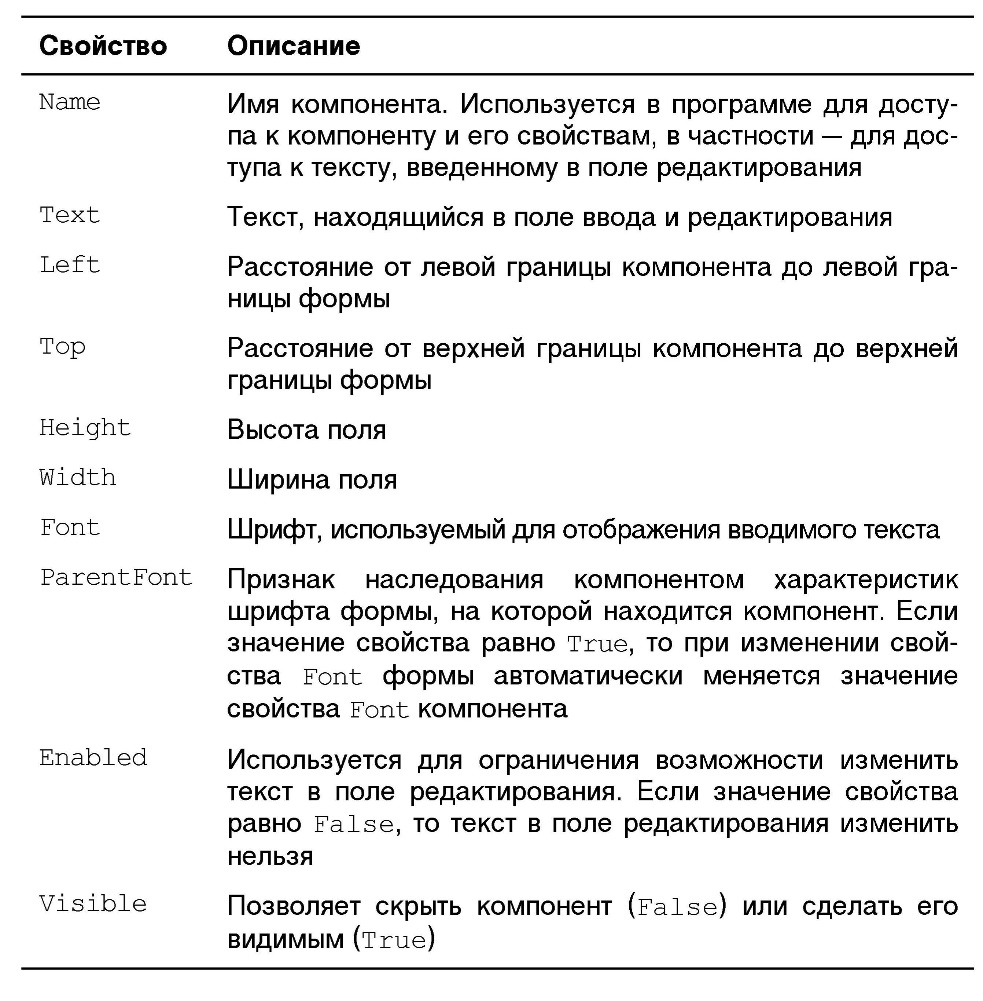
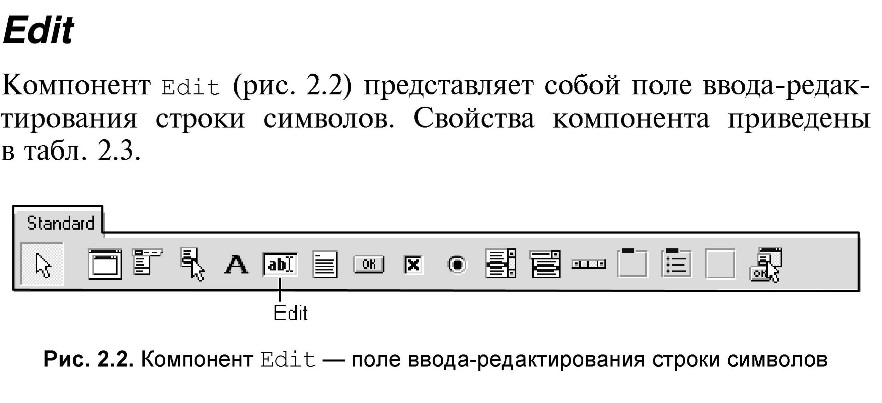
В редакторе масок окно *Sample Masks* содержит наименования стандартных масок и примеры ввода с их помощью. В окно *Input Mask* надо ввести маску. Если вы выбираете одну из стандартных масок, то окно *Input Mask* автоматически заполняется и вы можете, если хотите, отредактировать эту маску.

Окно *Character for Blanks* определяет символ, используемый для обозначения позиций, в которых еще не осуществлен ввод (третий раздел маски). Индикатор *Save Literal Characters* определяет второй раздел маски: установлен, если второй раздел равен 1, и не установлен, если второй раздел равен 0.

Кнопка *Masks* позволяет выбрать и загрузить какой-либо другой файл стандартных масок. К сожалению, среди файлов стандартных масок, поставляемых с Delphi, отсутствует маска, соответствующая российским стандартам. Но вы легко можете сами сделать себе такой файл стандартных масок. Он делается в обычном текстовом редакторе и должен сохраняться как «только текст» с расширением ***.dem***. Чтобы редактор масок Delphi видел этот файл, его надо сохранить в каталоге *Delphi\****BIN***. Каждая строка файла состоит из трех частей, разделяемых символом вертикальной черты. Первая часть состоит из пояснительного текста, появляющегося в левой панели окна *Sample Masks* редактора масок. Вторая часть — пример, который появляется в правой панели окна *Sample Masks редактора масок*. А третья часть — сама маска.

Поскольку второй раздел маски равен 0, то при чтении введенных пользователем значений свойства ***EditText*** и ***Text*** будут различаться. Свойство ***EditText*** для примера будет равно **(095) 123-45-67**, а свойство ***Text*** будет равно **0951234567**. Если второй раздел маски сделать равным 1, то значения обоих свойств будут равны **(095) 123-45-67**.

Если с помощью ***EditMask*** надо ввести, например, целое число без знака, состоящее не более, чем из двух цифр, можно задать маску **99;0;**. Если число обязательно должно быть двузначным, то маска должна иметь вид **00;0**



***№8 Button***

Простейшей и наиболее часто используемой кнопкой является кнопка ***Button***, расположенная на странице библиотеки *Standard*. Реже используется кнопка, отличающаяся, прежде всего, возможностью отобразить на ее поверхности изображение. Большинство свойств, методов и событий у этих видов кнопок одинаковы.

Основное с точки зрения внешнего вида свойство кнопки — ***Caption*** (надпись). В надписях кнопок можно предусматривать использование клавиш ускоренного доступа, выделяя для этого один из символов надписи. Перед символом, который должен соответствовать клавише ускоренного доступа, ставится символ амперсанта «&». Этот символ не появляется в надписи, а следующий за ним символ оказывается подчеркнутым. Тогда пользователь может вместо щелчка на кнопке нажать в любой момент клавишу Alt совместно с клавишей выделенного символа.

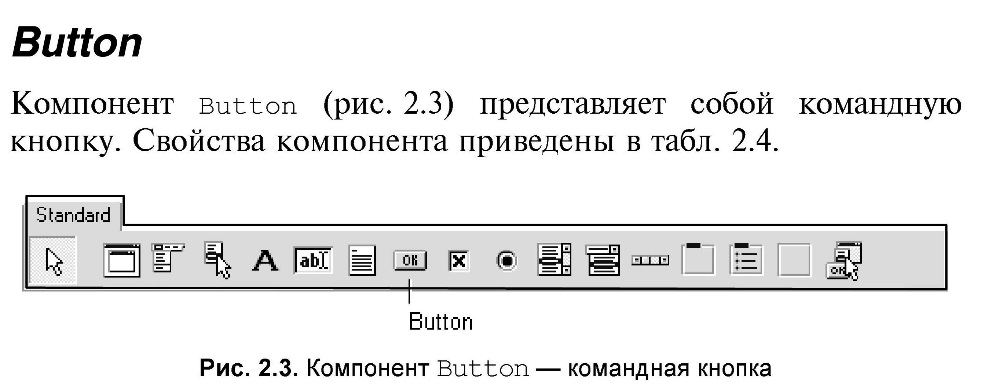
Основное событие любой кнопки — ***OnClick***, возникающее при щелчке на ней. Именно в обработчике этого события записываются операторы, которые должны выполняться при щелчке пользователя на кнопке. Помимо этого есть еще ряд событий, связанных с различными манипуляциями клавишами и кнопками мыши.

Свойство ***Cancel***, если его установить в ***true***, определяет, что нажатие пользователем клавиши Esc будет эквивалентно нажатию на данную кнопку. Это свойство целесообразно задавать равным ***true*** для кнопок «Отменить» в различных диалоговых окнах, чтобы можно было выйти из диалога, нажав на эту кнопку или нажав клавишу Esc.

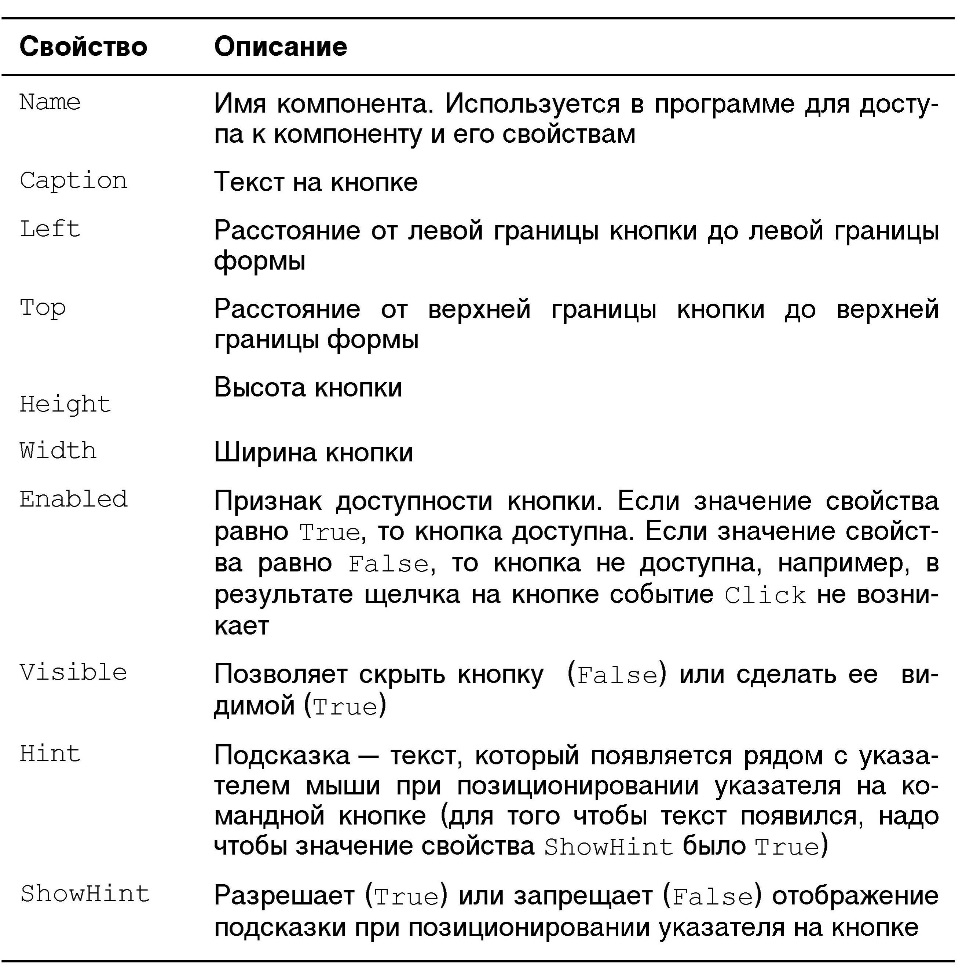
Свойство ***Default***, если его установить в ***true***, определяет, что нажатие пользователем клавиши ввода Enter будет эквивалентно нажатию на данную кнопку, даже если данная кнопка в этот момент не находится в фокусе. Правда, если в момент нажатия Enter в фокусе находится другая кнопка, то все-таки сработает именно кнопка в фокусе.

Еще одно свойство — ***ModalResult*** используется в модальных формах. В обычных приложениях значение этого свойства должно быть равно ***mrNone***.

Из методов, присущих кнопкам, имеет смысл отметить один — ***Click***. Выполнение этого метода эквивалентно щелчку на кнопке, т.е. вызывает событие кнопки ***OnClick***. Этим можно воспользоваться, чтобы продублировать какими-то другими действиями пользователя щелчок на кнопке.



№9 ***CheckBox***



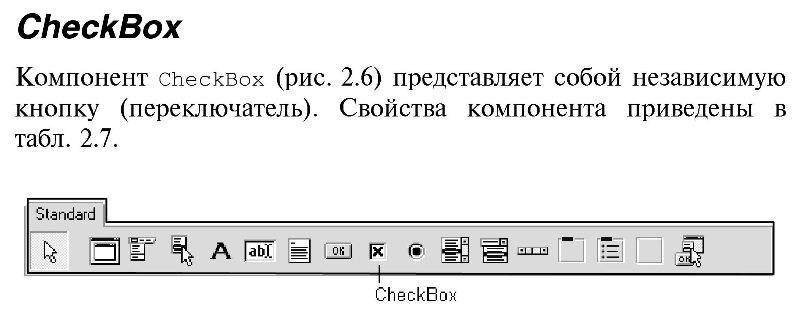
Индикаторы с флажком ***CheckBox*** используются в приложениях в основном для того, чтобы пользователь мог включать и выключать какие-то опции, или для индикации состояния. При каждом щелчке пользователя на индикаторе его состояние изменяется, проходя в общем случае последовательно через три значения: выделение (появление черной галочки), промежуточное (серое окно индикатора и серая галочка) и не выделенное (пустое окно индикатора). Этим трем состояниям соответствуют три значения свойства компонента ***State***: ***cbChecked***, ***cbGrayed***, ***cbUnchecked***. Впрочем, эти три состояния допускаются только при значении другого свойства ***AllowGrayed*** равном ***true***. Если же ***AllowGrayed =*** ***false*** (значение по умолчанию), то допускается только два состояния: выделенное и не выделенное. И ***State***, и ***AllowGrayed*** можно устанавливать во время проектирования или программно во время выполнения.

Промежуточное состояние обычно используется, если индикатор применяется для отображения какой-то характеристики объекта. Наприме

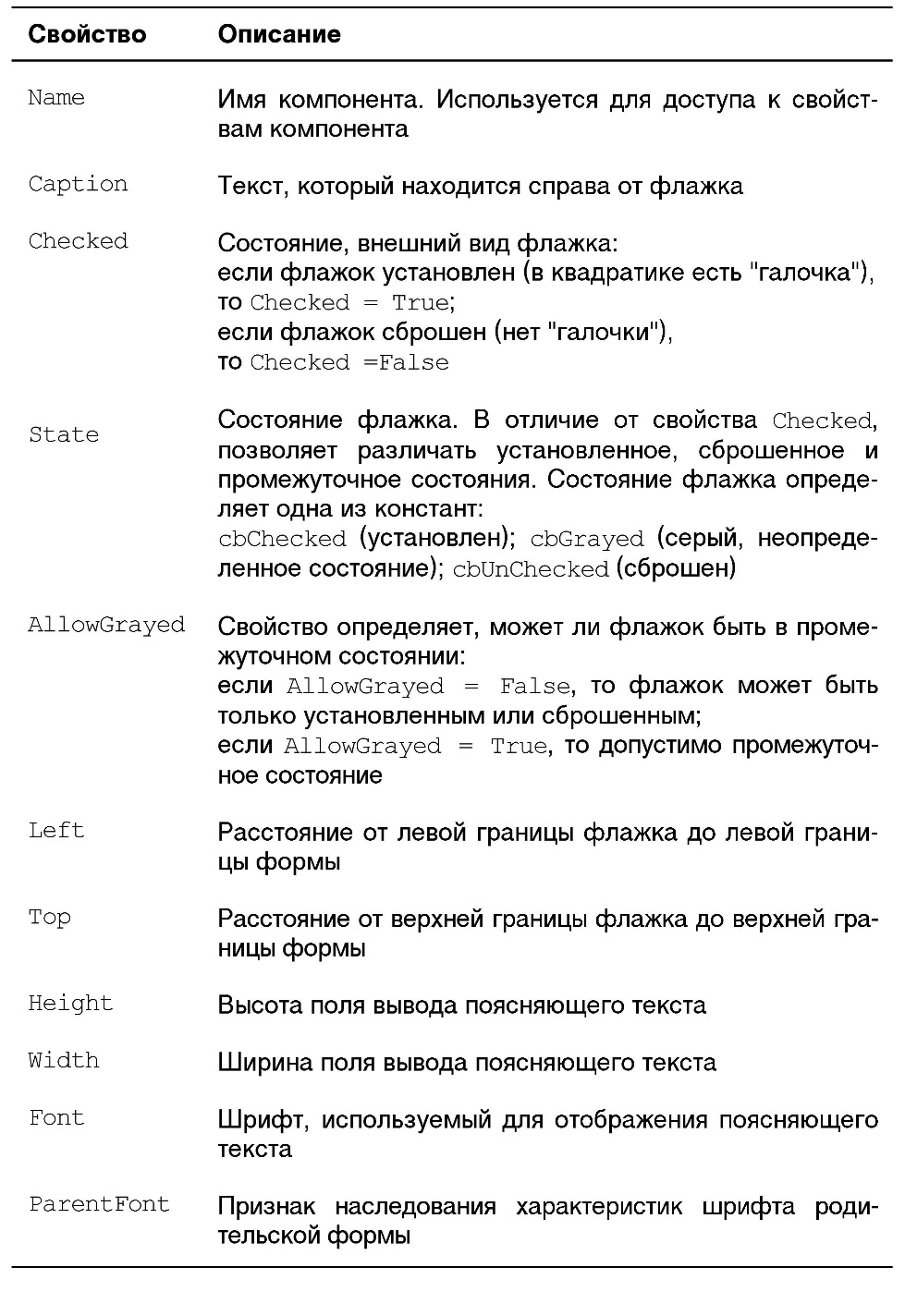
р, если индикатор призван показать, какой регистр использовался при написании какого-то фрагмента текста, то в случае, если весь текст написан в верхнем регистре индикатор может принимать выделенное состояние, если в нижнем — не выделенное, а если использовались оба регистра — промежуточное.

Проверять состояние индикатора можно не только по значению ***State***, но и по значению свойства ***Checked***. Если ***Checked*** равно ***true***, то индикатор выбран, т.е. ***State = cbChecked***. Если ***Checked*** равно ***false***, то ***State*** равно ***cbUnchecked*** или ***cbGrayed***. Установка ***Checked*** в ***true*** во время проектирования или выполнения автоматически переключает ***State*** в ***cbChecked***.

Как и в радиокнопке, в индикаторе ***CheckBox*** надпись задается свойством ***Caption***, а ее размещение по отношению к индикатору — свойством ***Alignment***.



№11 ***Label***



Для отображения различных надписей на форме используются в основном компоненты ***Label***, ***StaticText*** и ***Panel***. Первые два из этих компонентов — метки, специально предназначенные для отображения текстов.

Тексты, отображаемые в перечисленных компонентах, определяются значением их свойства ***Caption***. Его можно устанавливать в процессе проектирования или задавать и изменять программно во время выполнения приложения.

*Label1.Caption := 'Новый текст';*

Если требуется отобразить числовую информацию, можно воспользоваться функциями ***FloatToStr*** и ***IntToStr***, переводящими соответственно числа с плавающей запятой и целые в строку. Для формирования текста, состоящего из нескольких фрагментов, можно использовать операцию «*+*», которая для строк означает их склеивание (конкатенацию). Например, если в программе имеется целая переменная ***I***, отображающая число сотрудников некоторой организации, то вывести в метку ***Label1*** информацию об этом можно оператором:

*Label1.Caption := 'Число сотрудников: '+IntToStr(I);*

Во всех компонентах цвет фона определяется свойством ***Color***, а цвет надписи — подсвойством ***Color*** свойства ***Font***. Например, в большинстве меток задан цвет фона равным ***clWhite*** — белый. Если цвет специально не задавать, то цвет фона обычно сливается с цветом контейнера, содержащего метку, так что фон просто не заметен.

Таким образом, с точки зрения оформления  минимальные возможности у ***Label***.

Размещение всех рассматриваемых компонентов на форме определяется, в частности, свойствами ***Тор*** (координата верхнего края), ***Left*** (координата левого края), ***Height*** (высота), ***Width*** (ширина). Имеются также свойства, определяющие изменение размера компонента при изменении пользователем во время выполнения приложения размеров окна. Это общие свойства всех оконных компонентов: ***Align*** (выравнивание компонента по всей верхней, левой, правой, нижней частям контейнера или по всей его клиентской области), ***Anchors*** (привязка сторон компонента к сторонам контейнера), ***Constraints*** (ограничения допустимых изменений размеров). Учтите, что использование в свойстве ***Anchors*** одновременной привязки компонента к противоположным сторонам контейнера приведет к растягиванию или сжатию компонента при изменении пользователем размеров окна. При этом в метках в ряде случаев наблюдаются неприятности, связанные с перемещением надписей (если они, например, выровнены по центру). Чтобы избежать этих неприятностей, надо в обработку события формы **OnResize** вставить операторы, перерисовывающие компоненты методом ***Repaint***, например:

*StaticText1.Repaint;*

Размер меток ***Label*** определяется также свойством ***AutoSize***. Если это свойство установлено в ***true***, то вертикальный и горизонтальный размеры компонента определяются размером надписи. Если же ***AutoSize*** равно ***false***, то выравнивание текста внутри компонента определяется свойством ***Alignment***, которое позволяет выравнивать текст по левому краю, правому краю или центру клиентской области метки. В панели ***Panel*** также имеется свойство ***AutoSize***, но оно не относится к размерам надписи ***Caption***. Однако, свойство выравнивания ***Alignment*** работает и для панели.

В метке ***Label*** имеется свойство ***Wordwrap*** — допустимость переноса слов длинной надписи, превышающей длину компонента, на новую строчку. Чтобы такой перенос мог осуществляться, надо установить свойство ***WordWrap*** в ***true***, свойство ***AutoSize*** в ***false*** (чтобы размер компонента не определялся размером надписи) и сделать высоту компонента такой, чтобы в нем могло поместиться несколько строк. Если ***WordWrap*** не установлено в ***true*** при ***AutoSize*** равном ***false***, то длинный текст, не помещающийся в рамке метки, просто обрезается.

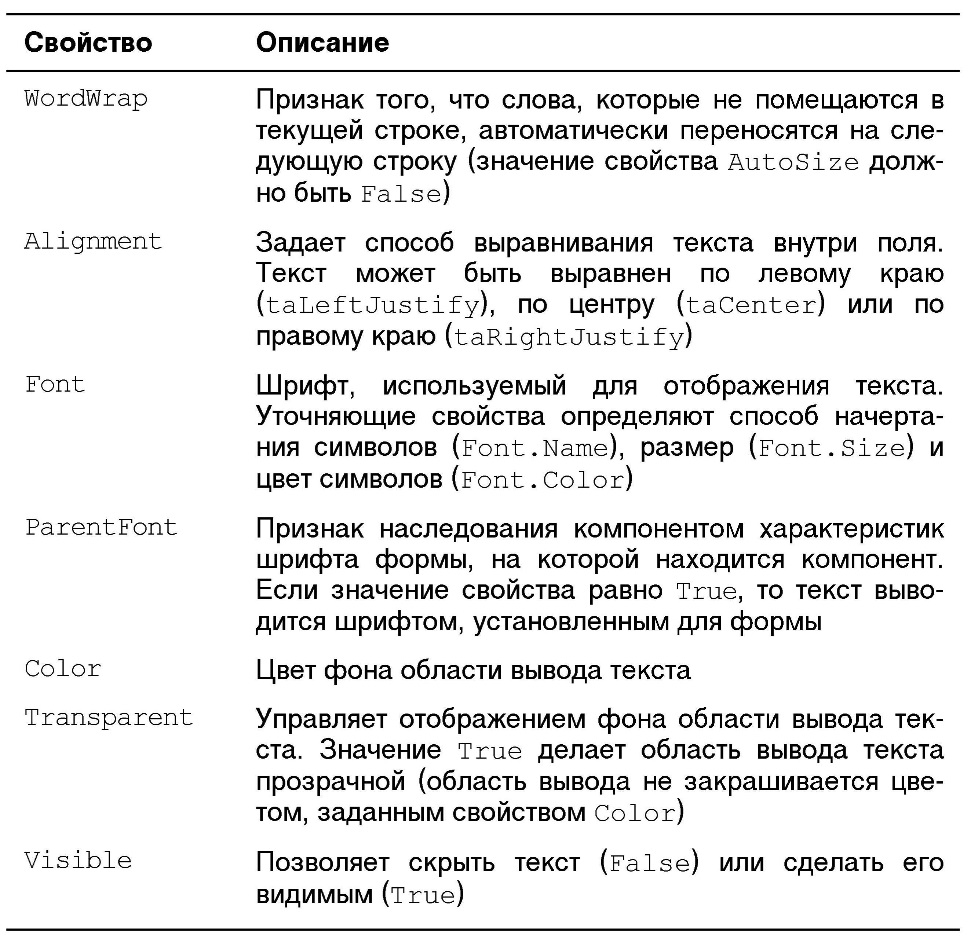
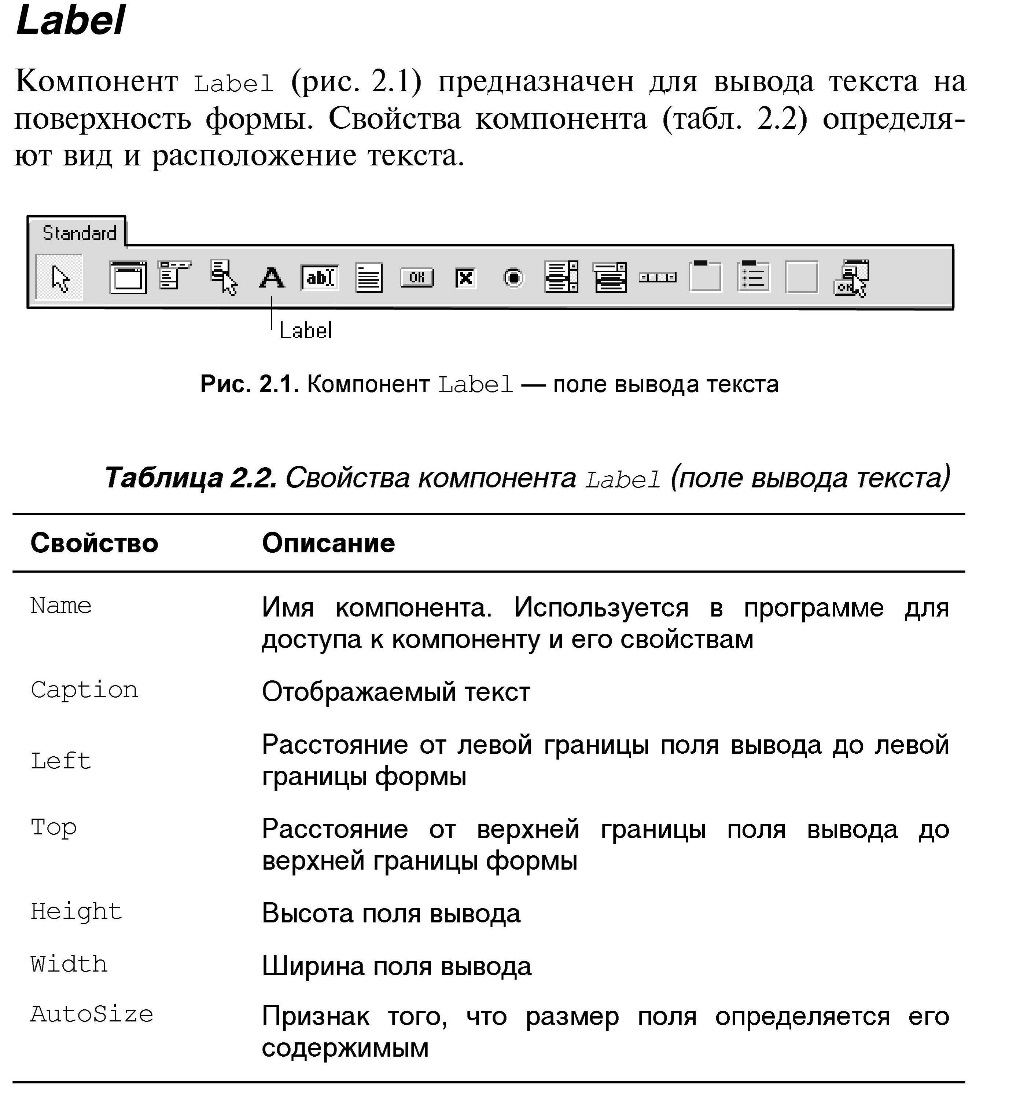
В метке ***StaticText*** перенос длинного текста осуществляется автоматически, если значение ***AutoSize*** установлено в ***false*** и размер компонента достаточен для размещения нескольких строк. Для того, чтобы в ***StaticText*** осуществлялся перенос при изменении пользователем размеров окна, надо осуществлять описанную выше перерисовку компонента методом ***Repaint*** в обработчике события формы ***OnResize***.

В панели размещение надписи в нескольких строках невозможно.

Можно отметить еще одно свойство меток ***Label*** и ***StaticText***, превращающее их в некоторое подобие управляющих элементов. Это свойство ***FocusControl*** — фокусируемый компонент. Если в свойстве метки ***Caption*** поместить перед одним из символов символ амперсант «&», то символ, перед которым поставлен амперсант, отображается в надписи метки подчеркнутым (сам амперсант вообще не отображается). Если после этого обратиться к свойству метки ***FocusControl***, то из выпадающего списка можно выбрать элемент, на который будет переключаться фокус, если пользователь нажмет клавиши ускоренного доступа: клавишу Alt + подчеркнутый символ. Подобные клавиши ускоренного доступа предусмотрены в управляющих элементах: разделах меню и кнопках. Благодаря свойству ***FocusControl*** метки могут обеспечить клавишами ускоренного доступа иные элементы, например, окна редактирования, в которых такие клавиши не предусмотрены. Только для того, чтобы клавиши ускоренного доступа в метках срабатывали, необходимо установить свойство ***ShowAccelChar*** этих меток в ***true***.

Для отображения текстовой информации, и даже с дополнительной возможностью прокрутки длинных текстов, можно использовать также окна редактирования ***Edit*** и ***MaskEdit*** в режиме ***Readonly***.

№12 ***RadioButton***

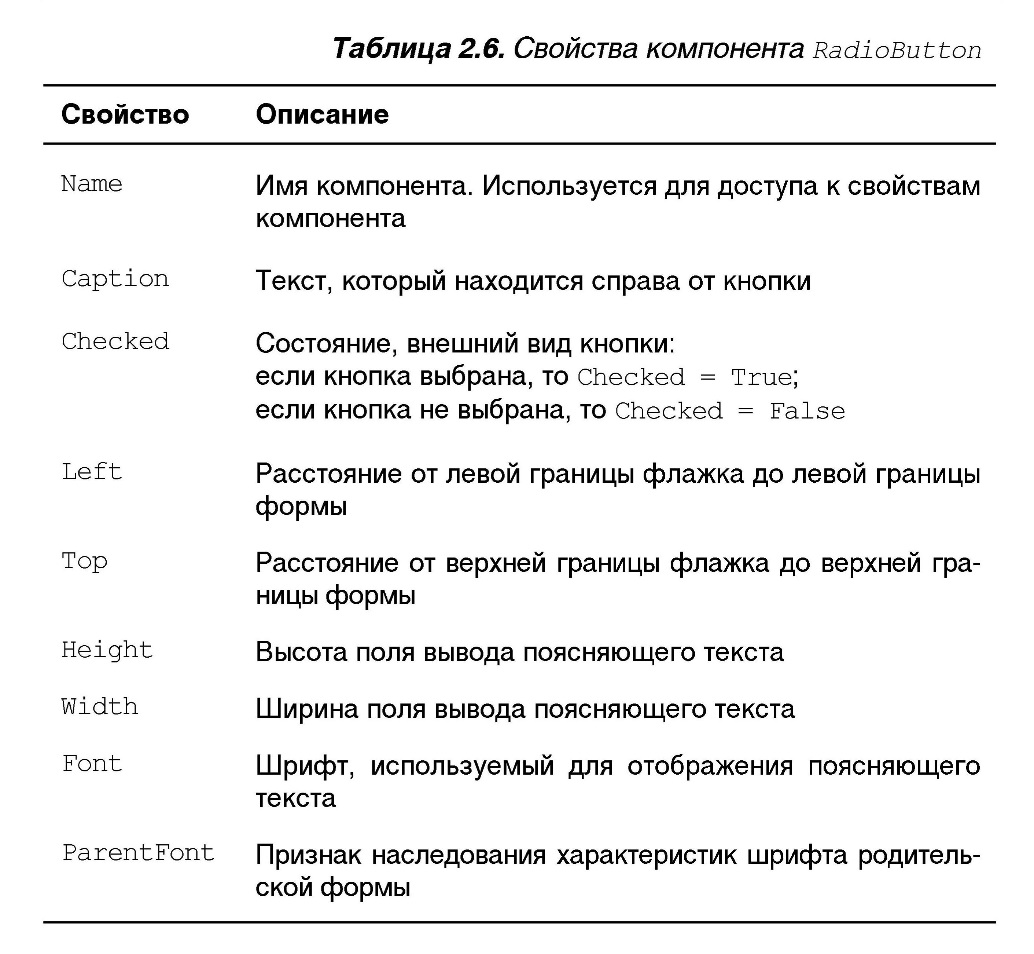
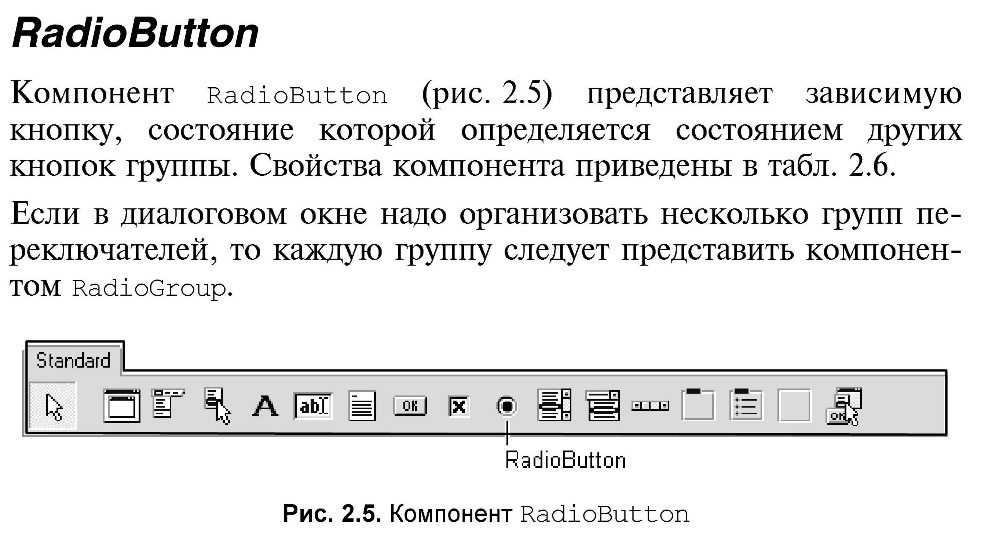


Радиокнопки образуют группы взаимосвязанных индикаторов, из которых обычно может быть выбран только один. Они используются для выбора пользователем одной из нескольких взаимоисключающих альтернатив, например, отдела, в котором работает сотрудник, или пола сотрудника. Впрочем, радиокнопки могут использоваться и для отображения аналогичных данных. В этом случае управление кнопками осуществляется программно.

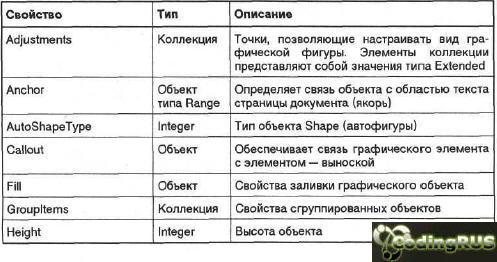
Рассмотрим свойства радиокнопки ***RadioButton***. Свойство ***Caption*** содержит надпись, появляющуюся около кнопки. Значение свойства ***Alignment*** определяет, с какой стороны от кнопки появится надпись: ***taLeftJustify*** — слева, ***taRightJustify*** — справа (это значение принято по умолчанию). Свойство ***Checked*** определяет, выбрана ли данная кнопка пользователем, или нет. Поскольку в начале выполнения приложения обычно надо, чтобы одна из кнопок группы была выбрана по умолчанию, ее свойство ***Checked*** надо установить в ***true*** в процессе проектирования. Если вы поэкспериментируете, то заметите, что и во время проектирования можно установить в ***true*** значение ***Checked*** только у одной кнопки из группы.

Размещение кнопок ***RadioButton*** в панели ***GroupBox*** дает большую свободу по сравнению с компонентом ***RadioGroup*** и позволяет разместить кнопки не регулярно.

Радиокнопки ***RadioButton*** могут размещаться не только в панели ***GroupBox***, но и в любой панели другого типа, а также непосредственно на форме. Группа взаимосвязанных кнопок в этих случаях определяется тем оконным компонентом, который содержит кнопки. В частности, для радиокнопок, размещенных непосредственно на форме, контейнером является сама форма. Таким образом, все кнопки, размещенных непосредственно на форме, работают как единая группа, т.е. только в одной из этих кнопок можно установить значение ***Checked*** в ***true***.



 №13 **Свойства объекта Shape**



**№14 Кнопка с фиксацией SpeedButton**

Кнопки ***SpeedButton*** имеют возможность отображения пиктограмм и могут использоваться как обычные управляющие кнопки или как кнопки с фиксацией нажатого состояния. Обычно они используются в качестве быстрых кнопок, дублирующих различные команды меню, и в инструментальных панелях, в которых требуется фиксация нажатого состояния.

У кнопок ***SpeedButton***, как и у других кнопок, имеется свойство ***Caption*** — надпись, но в этих кнопках оно обычно оставляется пустым, так как вместо надписи используется пиктограмма.

Изображение на кнопке задается свойством ***Glyph*** точно так же, как для кнопок ***BitBtn***. И точно так же свойство ***NumGlyphs*** определяет число используемых пиктограмм, свойства ***Layout*** и ***Margin*** определяют расположение изображения, а свойство ***Spacing*** — расстояние между изображением и надписью.

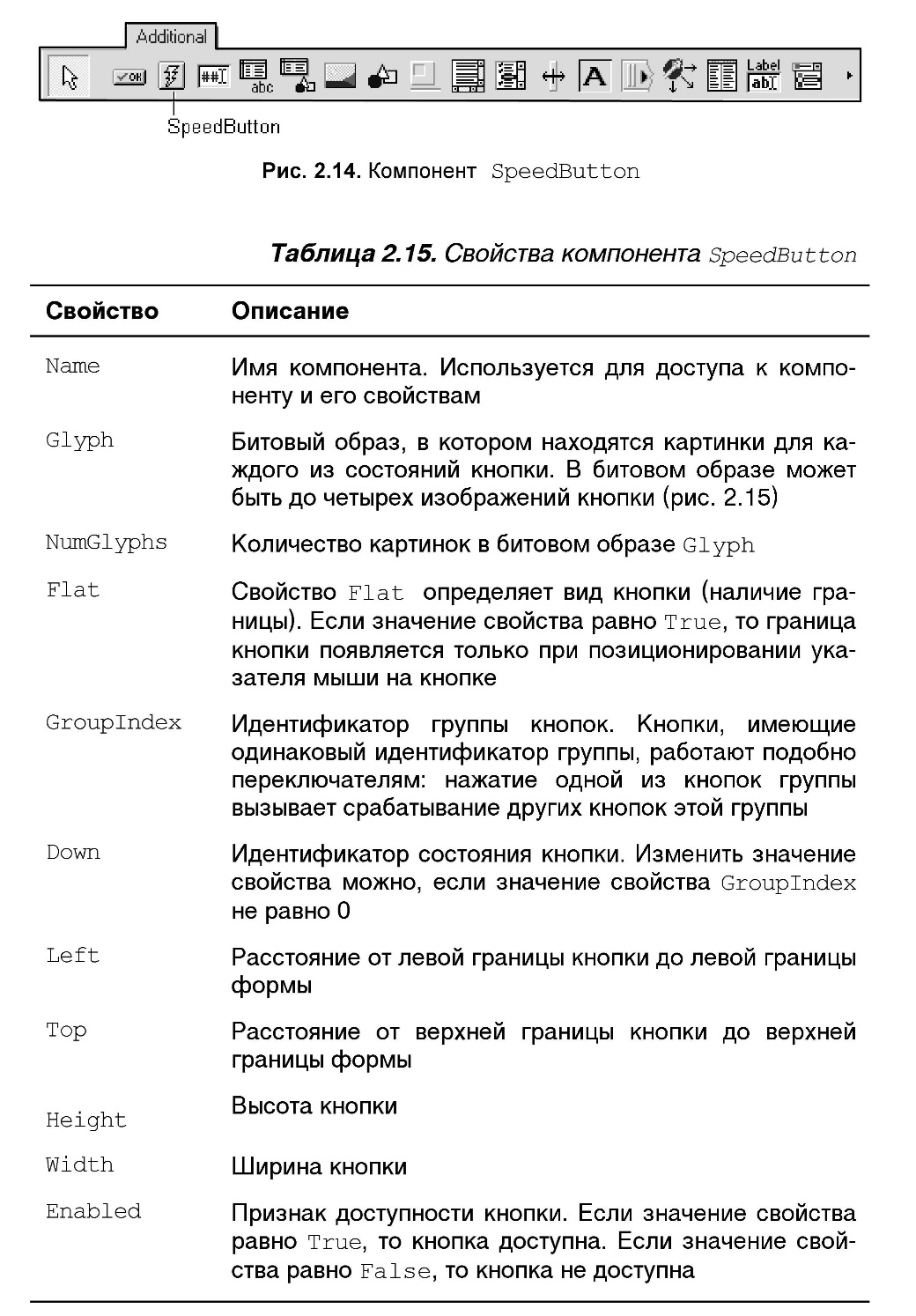
Особенностью кнопок ***SpeedButton*** являются свойства ***GroupIndex*** (индекс группы), ***AllowAllUp*** (разрешение отжатого состояния всех кнопок группы) и ***Down*** (исходное состояние — нажатое). Если ***GroupIndex = 0***, то кнопка ведет себя так же, как ***Button*** и ***BitBtn***. При нажатии пользователем кнопки она погружается, а при отпускании возвращается в нормальное состояние. В этом случае свойства ***AllowAllUp*** и ***Down*** не влияют на поведение кнопки.

Если ***Grouplndex*** ***> 0*** и ***AllowAllUp = true***, то кнопка при щелчке пользователя на ней погружается и остается в нажатом состоянии. При повторном щелчке пользователя на кнопке она освобождается и переходит в нормальное состояние (именно для того, чтобы освобождение кнопки состоялось, необходимо задать ***AllowAllUp = true***). Если свойство ***Down*** во время проектирования установлено равным ***true***, то исходное состояние кнопки — нажатое.

Если есть несколько кнопок, имеющих одинаковое ненулевое значение ***GroupIndex***, то они образуют группу взаимосвязанных кнопок из которых нажатой может быть только одна. Если одна кнопка находится в нажатом состоянии и пользователь щелкает на другой, то первая кнопка освобождается, а вторая фиксируется в нажатом состоянии. Поведение нажатой кнопки при щелчке на ней зависит от значения свойства ***AllowAllUp***. Если оно равно ***true***, то кнопка освободится, поскольку в этом случае возможно состояние, когда все кнопки группы отжаты. Если же ***AllowAllUp*** равно ***false***, то щелчок на нажатой кнопке не приведет к изменению вида кнопки. Впрочем, и в этом случае, как и при любом щелчке на кнопке, возникает событие ***OnClick***, которое может быть обработано.

Состояние кнопки во время выполнения можно определить по значению свойства ***Down***: если значение равно **true**, то кнопка нажата. Во время события ***OnClick*** значение ***Down*** уже равно тому состоянию, которое примет кнопка в результате щелчка на ней.

***№15 BitBtn***



Простейшей и наиболее часто используемой кнопкой является кнопка ***BitBtn*** расположенная на странице библиотеки *Standard*. Реже используется кнопка, отличающаяся, прежде всего, возможностью отобразить на ее поверхности изображение. Большинство свойств, методов и событий у этих видов кнопок одинаковы.

Основное с точки зрения внешнего вида свойство кнопки — ***Caption*** (надпись). В надписях кнопок можно предусматривать использование клавиш ускоренного доступа, выделяя для этого один из символов надписи. Перед символом, который должен соответствовать клавише ускоренного доступа, ставится символ амперсанта «&». Этот символ не появляется в надписи, а следующий за ним символ оказывается подчеркнутым. Тогда пользователь может вместо щелчка на кнопке нажать в любой момент клавишу Alt совместно с клавишей выделенного символа.

Основное событие любой кнопки — ***OnClick***, возникающее при щелчке на ней. Именно в обработчике этого события записываются операторы, которые должны выполняться при щелчке пользователя на кнопке. Помимо этого есть еще ряд событий, связанных с различными манипуляциями клавишами и кнопками мыши.

Свойство ***Cancel***, если его установить в ***true***, определяет, что нажатие пользователем клавиши Esc будет эквивалентно нажатию на данную кнопку. Это свойство целесообразно задавать равным ***true*** для кнопок «Отменить» в различных диалоговых окнах, чтобы можно было выйти из диалога, нажав на эту кнопку или нажав клавишу Esc.

Свойство ***Default***, если его установить в ***true***, определяет, что нажатие пользователем клавиши ввода Enter будет эквивалентно нажатию на данную кнопку, даже если данная кнопка в этот момент не находится в фокусе. Правда, если в момент нажатия Enter в фокусе находится другая кнопка, то все-таки сработает именно кнопка в фокусе.

Еще одно свойство — ***ModalResult*** используется в модальных формах. В обычных приложениях значение этого свойства должно быть равно ***mrNone***.

Из методов, присущих кнопкам, имеет смысл отметить один — ***Click***. Выполнение этого метода эквивалентно щелчку на кнопке, т.е. вызывает событие кнопки ***OnClick***. Этим можно воспользоваться, чтобы продублировать какими-то другими действиями пользователя щелчок на кнопке.

 Все сказанное выше в равной степени относится и к ***Button***, и к ***BitBtn***. Рассмотрим теперь особенности кнопки с пиктограммой ***BitBtn***. Изображение на этой кнопке задается свойством ***Glyph***. После того, как вы выбрали изображение, нажмите OK и выбранное изображение появится на вашей кнопке левее надписи.

Файл изображения для кнопки может содержать до четырех изображений пиктограмм размера 16x16. Самое левое соответствует отжатой кнопке. Второе слева соответствует недоступной кнопке, когда ее свойство ***Enabled*** равно ***false***. Третье слева изображение используется при нажатии пользователя на кнопку при ее включении. Четвертое слева изображение используется в кнопках с фиксацией ***SpeedButton*** для изображения кнопки в нажатом состоянии. Большинство изображений для кнопок использует две пиктограммы. Число пиктограмм вы можете узнать из свойства кнопки ***NumGlyphs***, которое после загрузки изображения покажет вам число пиктограмм в нем.

Расположение изображения и надписи на кнопке определяется свойствами ***Margin***, ***Layout*** и ***Spacing***. Если свойство ***Margin*** равно ***-1*** (значение по умолчанию), то изображение и надпись размещаются в центре кнопки. При этом положение изображения но отношению к надписи определяется свойством ***Layout***, которое может принимать значения: ***blGlyphLeft*** (слева, это значение принято по умолчанию), ***blGlyphRight*** (справа), ***blGlyphТор*** (вверху), ***blGlyphBottom*** (внизу). Если же ***Margin > 0***, то в зависимости от значения ***Layout*** изображение и надпись смещаются к той или иной кромке кнопки, отступая от нее на число пикселей, заданное значением ***Margin***.

Свойство ***Spacing*** задает число пикселей, разделяющих изображение и надпись на поверхности кнопки. По умолчанию ***Spacing = 4***. Если задать ***Spacing*** ***=*** ***0***, изображение и надпись будут размещены вплотную друг к другу. Если задать ***Spacing*** ***= -1***, то текст появится посередине между изображением и краем кнопки.

Еще одно свойство ***BitBtn*** — свойство ***Kind*** определяет тип кнопки. По умолчанию значение этого свойства равно ***bkCustom*** — заказная. Но можно установить и множество других предопределенных типов: ***bkOK***, ***bkCancel***, ***bkHelp***, ***bkYes***, ***bkNo***, ***bkClose***, ***bkAbort***, ***bkRetry***, ***bkIgnore***, ***bkAll***. В этих типах уже сделаны соответствующие надписи, введены пиктограммы, заданы еще некоторые свойства. Обычно все-таки лучше ими не пользоваться. Во-первых, надписи все равно надо переводить на русский язык. Во-вторых, предопределенные рисунки обычно выбиваются из общего стиля конкретного приложения. И главное — предопределение некоторых свойств, не учтенных вами, может иногда

приводить к странным результатам работы. Уж лучше использовать заказные кнопки и самому устанавливать в них все необходимые свойства.

 №16 ***Panel***

Основное назначение панели ***Panel*** другое: компоновка компонентов в окне формы.

Тексты, отображаемые в перечисленных компонентах, определяются значением их свойства ***Caption***. Его можно устанавливать в процессе проектирования или задавать и изменять программно во время выполнения приложения.

*Label1.Caption := 'Новый текст';*

Если требуется отобразить числовую информацию, можно воспользоваться функциями ***FloatToStr*** и ***IntToStr***, переводящими соответственно числа с плавающей запятой и целые в строку. Для формирования текста, состоящего из нескольких фрагментов, можно использовать операцию «*+*», которая для строк означает их склеивание (конкатенацию). Например, если в программе имеется целая переменная ***I***, отображающая число сотрудников некоторой организации, то вывести в метку ***Label1*** информацию об этом можно оператором:

*Label1.Caption := 'Число сотрудников: '+IntToStr(I);*

Во всех компонентах цвет фона определяется свойством ***Color***, а цвет надписи — подсвойством ***Color*** свойства ***Font***. Например, в большинстве меток задан цвет фона равным ***clWhite*** — белый. Если цвет специально не задавать, то цвет фона обычно сливается с цветом контейнера, содержащего метку, так что фон просто не заметен.

Компонент ***Panel*** кроме свойства ***BorderStyle*** имеет еще свойства ***BevelInner***, ***BevelOuter***, ***BevelWidth***, ***BorderWidth***, которые предоставляют возможности оформления надписи. Таким образом, с точки зрения оформления выводимого текста максимальные возможности дает ***Panel*** и минимальные — ***Label***.

Размещение всех рассматриваемых компонентов на форме определяется, в частности, свойствами ***Тор*** (координата верхнего края), ***Left*** (координата левого края), ***Height*** (высота), ***Width*** (ширина). Имеются также свойства, определяющие изменение размера компонента при изменении пользователем во время выполнения приложения размеров окна. Это общие свойства всех оконных компонентов: ***Align*** (выравнивание компонента по всей верхней, левой, правой, нижней частям контейнера или по всей его клиентской области), ***Anchors*** (привязка сторон компонента к сторонам контейнера), ***Constraints*** (ограничения допустимых изменений размеров). Учтите, что использование в свойстве ***Anchors*** одновременной привязки компонента к противоположным сторонам контейнера приведет к растягиванию или сжатию компонента при изменении пользователем размеров окна. При этом в метках в ряде случаев наблюдаются неприятности, связанные с перемещением надписей (если они, например, выровнены по центру). Чтобы избежать этих неприятностей, надо в обработку события формы **OnResize** вставить операторы, перерисовывающие компоненты методом ***Repaint***, например:

*StaticText1.Repaint;*

В панели ***Panel*** также имеется свойство ***AutoSize***, но оно не относится к размерам надписи ***Caption***. Однако, свойство выравнивания ***Alignment*** работает и для панели.

В панели размещение надписи в нескольких строках невозможно.

 №17 компоненты RadioGroup

Радиокнопки образуют группы взаимосвязанных индикаторов, из которых обычно может быть выбран только один. Они используются для выбора пользователем одной из нескольких взаимоисключающих альтернатив, например, отдела, в котором работает сотрудник, или пола сотрудника. Впрочем, радиокнопки могут использоваться и для отображения аналогичных данных. В этом случае управление кнопками осуществляется программно.

Компонент ***RadioGroup*** — панель группы радиокнопок. Это панель, которая может содержать регулярно расположенные столбцами и строками радиокнопки. Надпись в левом верхнем углу панели определяется свойством ***Caption***. А надписи кнопок и их количество определяются свойством ***Items***, имеющим тип ***TStrings***. Щелкнув на кнопке с многоточием около этого свойства в окне *Инспектора Объектов*, вы попадете в редактор списков строк. В нем вы можете занести надписи, которые хотите видеть около кнопок, по одной в строке. Сколько строчек вы запишете — столько и будет кнопок.

Кнопки, появившиеся в панели после задания значений ***Items***, можно разместить в несколько столбцов (не более 17), задав свойство ***Columns***. По умолчанию ***Columns = 1***, т.е. кнопки размещаются друг под другом.

Определить, какую из кнопок выбрал пользователь, можно по свойству ***ItemIndex***, которое показывает индекс выбранной кнопки. Индексы начинаются с 0. По умолчанию ***ItemIndex = -1***, что означает отсутствие выбранной кнопки. Если вы хотите, чтобы в момент начала выполнения приложения какаято из кнопок была выбрана (это практически всегда необходимо), то надо установить соответствующее значение ***ItemIndex*** во время проектирования. Если вы используете радиокнопки не для ввода, а для отображения данных, устанавливать значение ***ItemIndex*** можно программно во время выполнения приложения.

Компонент ***RadioGroup*** очень удобен, но не свободен от некоторых недостатков. Его хорошо использовать, если надписи кнопок имеют примерно одинаковую длину и если число кнопок в каждом столбце (при размещении их в нескольких столбцах) одинаково.

GroupBox

Компонент ***RadioGroup*** очень удобен, но не свободен от некоторых недостатков. Его хорошо использовать, если надписи кнопок имеют примерно одинаковую длину и если число кнопок в каждом столбце (при размещении их в нескольких столбцах) одинаково.

В подобных случаях желательно нерегулярное расположение кнопок. Такую возможность дают компоненты ***RadioButton***, сгруппированные панелью ***GroupBox***. Панель ***GroupBox*** выглядит на форме так же, как ***RadioGroup***, и надпись в ее верхнем левом углу также определяется свойством ***Caption***. Эта панель сама по себе пустая. Ее назначение — служить контейнером для других управляющих элементов, в частности, для радиокнопок ***RadioButton***. Отдельная радиокнопка ***RadioButton*** особого смысла не имеет, хотя и может служить индикатором, включаемым и выключаемым пользователем. Но в качестве индикаторов обычно используются другие компоненты — ***CheckBox***. А радиокнопки имеют смысл, когда они взаимодействуют друг с другом в группе. Эта группа и объединяется единым контейнером, обычно панелью

№18 ***GroupBox***.

Рассмотрим свойства радиокнопки ***RadioButton***. Свойство ***Caption*** содержит надпись, появляющуюся около кнопки. Значение свойства ***Alignment*** определяет, с какой стороны от кнопки появится надпись: ***taLeftJustify*** — слева, ***taRightJustify*** — справа (это значение принято по умолчанию). Свойство ***Checked*** определяет, выбрана ли данная кнопка пользователем, или нет. Поскольку в начале выполнения приложения обычно надо, чтобы одна из кнопок группы была выбрана по умолчанию, ее свойство ***Checked*** надо установить в ***true*** в процессе проектирования. Если вы поэкспериментируете, то заметите, что и во время проектирования можно установить в ***true*** значение ***Checked*** только у одной кнопки из группы.

Размещение кнопок ***RadioButton*** в панели ***GroupBox*** дает большую свободу по сравнению с компонентом ***RadioGroup*** и позволяет разместить кнопки не регулярно.

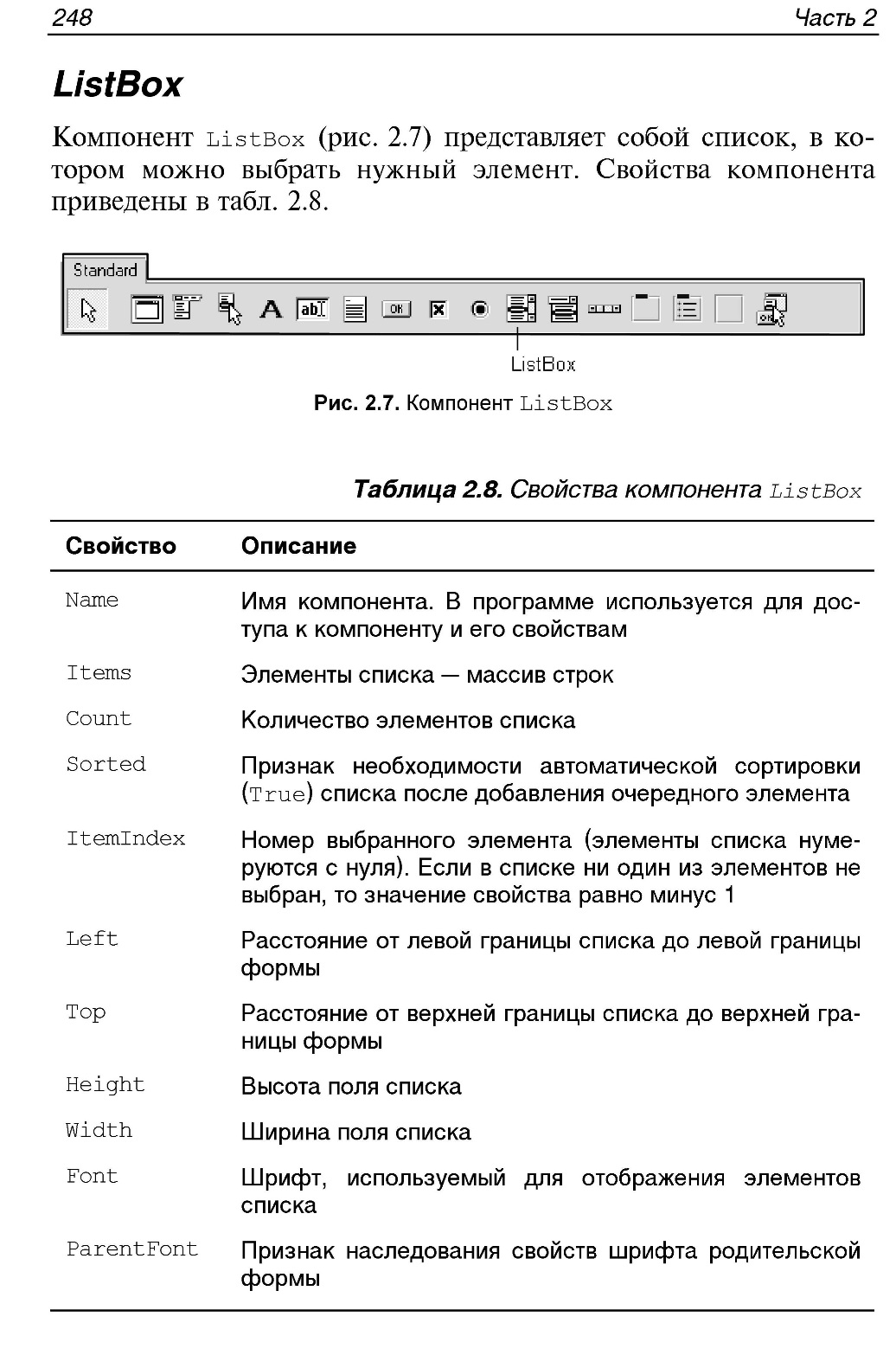
Радиокнопки ***RadioButton*** могут размещаться не только в панели ***GroupBox***, но и в любой панели другого типа, а также непосредственно на форме. Группа взаимосвязанных кнопок в этих случаях определяется тем оконным компонентом, который содержит кнопки. В частности, для радиокнопок, размещенных непосредственно на форме, контейнером является сама форма. Таким образом, все кнопки, размещенных непосредственно на форме, работают как единая группа, т.е. только в одной из этих кнопок можно установить значение ***Checked*** в ***true***.

**№19 вікно списку ListBox**

Являє собою стандартне вікно списку Windows, що дозволяє користувачеві вибирати розділи зі списку. Компонент візуальний.

ListBox – отображает список строк и позволяет пользователю выбрать из него необходимые строки.

список всегда развернут и можно допустить множественный выбор – свойство MultiSelect. Если MultiSelect = true, то значение ItemIndexсоответствует тому элементу списка, который находится в фокусе. При множественном выборе проверяется свойство Selected[Index: Integer] типBoolean. Запретить работу клавиш Ctrl и Shift при выборе можно в свойствеExtendedSelect, задав значение false.



Разница между ComboBox и от схожего по функциям компонента ListBox заключается в следующем:

* ComboBox дает возможность пользователю редактировать список, а ListBox нет;
* в ComboBox список может быть развернут или свернут, а в ListBox он всегда развернут;
* ListBox может допускать множественный выбор, а в ComboBox пользователь всегда будет выбрать один элемент.

Основные события:

* OnClick – наступает при щелчке на элементе списка;
* OnData – наступает в виртуальных списках, когда приложению надо отобразить очередную строку списка;
* OnDataObject – наступает в виртуальных списках, когда со строками виртуального списка надо связать какие-то объекты;
* OnDrawItem – наступает при необходимости перерисовать элемент списка;

OnKeyDown – наступает при нажатии пользователем любой клавиши (можно распознать нажатую клавишу в обработчике). OnKeyPress – событие наступает при нажатии пользователем клавиши символа (можно распознать вводимый символ в обработчике, а также запретить ввод). OnKeyUp –наступает, когда пользователь отпустил какую-либо клавишу (также можно распознать клавишу).

OnMouseItem – наступает при необходимости перерисовать элемент в списке с изменяемой высотой элементов.

Свойства

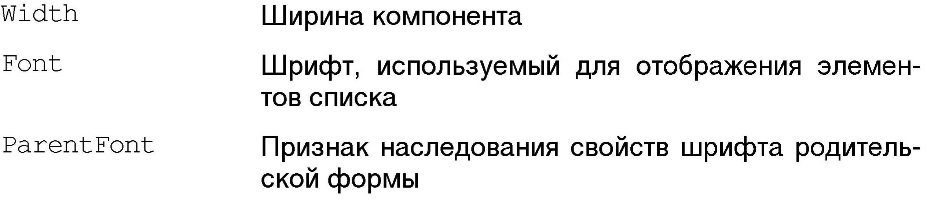
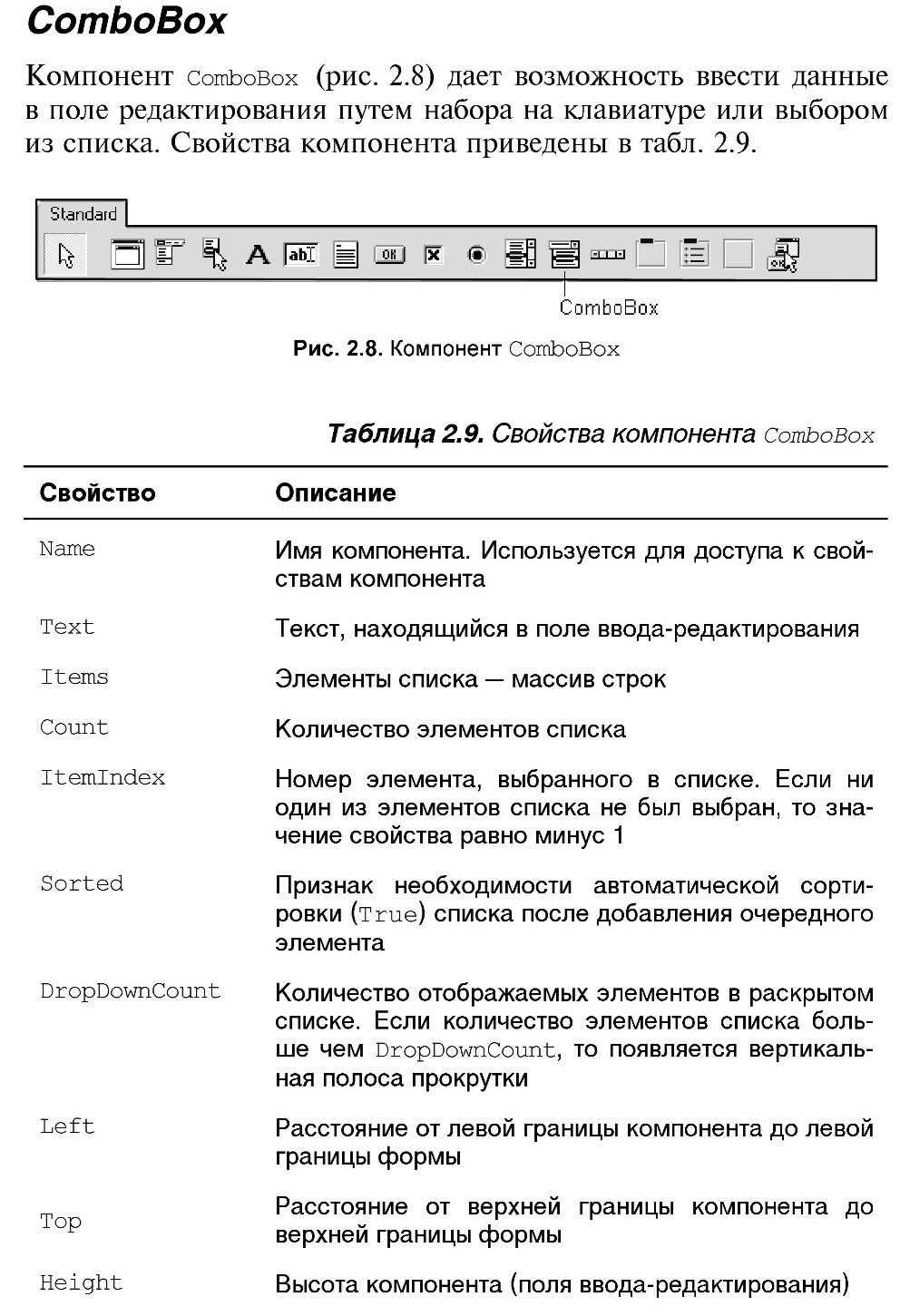
|  |  |
| --- | --- |
| Свойство | Описание |
| Action | Property Action: TBasicAction;  Определяет действие, связанное  с данным компонентом |
| Anchors | Определяет привязку данного компонента  к родительскому при изменении размеров  последнего |
| Count | Property Count: Integer;  Определяет число строк в списке |
| ItemIndex | Property ItemIndex: Integer;  Порядковый номер элемента,  выделенного в списке |
| TopIndex | Property TopIndex: Integer;  Указывает индекс элемента, видимого вверху  списка. При изменении сдвигает видимую  часть списка |

Основные методы

|  |  |
| --- | --- |
| Метод | Описание |
| Clear | Procedure Clear;  Удаляет все элементы |
| ItemAtPos | Function ItemAtPos (Pos:TPoint;  Existing: Boolean): Integer;  Возвращает индекс элемента списка,  координатами Pos |
| ItemRect | Function ItemRect(Item: Integer): TRect;  Возвращает прямоугольник, указанный  элемент Item списка |
| Show | Procedure Show;  Делает видимым невидимый индикатор |
| SetFocus | Procedure SetFocus;  Передает фокус элементу |

**редагований список ComboBox**

|  |
| --- |
| Об'єднує функції ListBox і Edit.  Компонент візуальний. |



Основные свойства Style (стили изображения списка):

* csDropDown – выпадающий список где строки и окно редактирования имеют одну и ту же высоту, позволяющий пользователям ввести или отредактировать текст;
* csSimple – развернутый список где строки и окно редактирования имеют одну и ту же высоту, позволяющий пользователям ввести или отредактировать текст;
* csDropDownList – выпадающий список, у строк одинаковая высота, не содержащий окно редактирования;
* csOwnerDrawFixed – выпадающий список типа csDropDown с графической прорисовкой элементов одинаковой высоты, задаваемой свойством ItemHeight;
* csOwnerDrawVariable — выпадающий список типа csDropDown с графической прорисовкой элементов, у которых может быть различная высота;

Основные свойства:

* Action(property Action: TBasicAction;) – задает действие, которое связанно с данным компонентом;
* Align (Talign = (alNone, alTop, alBottom, alLeft, alRight, alClient); property Align: TAlign;) – задает выравнивание компонента в контейнере;
* Anchors (type TAnchorKind = (akTop, akLeft, akRight, akBottom); type Tachors = set of TAncorKind; property Ancors: TAnchors;) – определяет привязан ли данный компонент к родительскому когда изменен размер последнего;
* DropDownCount (property DropDownCount: Integer;) – задает максимальное количество элементов, отображаемое в раскрывающемся списке без появления полосы прокрутки. Действует при всех значениях Style, кроме csSimple;
* ItemHeight (property ItemHeight: Integer;) – указывает высоту элементов, в пикселях, в выпадающем списке;
* ItemIndex (property ItemIndex: Integer;) – указывает порядковый номер элемента, который выделен в списке;
* Items (property Items: TStrings;) – массив строк списка – объект класса Tstrings. Свойства этого класса позволяют формирование и изменение списка;
* SelLength (property SelLength: Integer;) – определяет какое количество символов выделено в окне редактирования;
* SelStart (property SelStart: Integer;) – указывает позицию первого символа, который выделен в окне редактирования или, если выделения нет, то позицию курсора в нем;
* SelText (property SelText: string;) – текст выделенный в окне редактирования;
* Sorted (property Sorted: Boolean;) – указывает, будут ли строки автоматически быть отсортированы в алфавитном порядке.

Основные методы:

* Clear (procedure Clear;) – процедура удаления элементов из списка;
* SelectAll (procedure SelectAll;) – процедура выделения всего текста в окне редактирования;
* SetFocus (procedure SetFocus;) – процедура передачи фокуса элементу и активация его.

 Компонент MainMenu находится на панели инструментов Standard. Компонент MainMenu является невизуальным, позволяет конструировать и создавать на форме полосу главного меню, а также сопутствующие выпадающие меню. Компонент MainMenu отображает на форме главное меню. Конструктор меню вызывается двойным щелчком на значке MainMenu1, вам сразу показывается визуальное окно Menu Designer. В процессе проектирования свойству Menu формы присваивается ссылка на один из этих компонентов. В конструкторе меню зададим раздел Файл. Для этого в конструкторе меню по умолчанию для выделенного квадратика надо задать надпись (Caption из инспектора объектов Properties) напишите Файл и нажмите enter. Основное свойство компонента – Items (тип TMenuItem). У объекта N1 есть свойства, события и методы. Одно из свойств – Caption мы уже задали (у нас – Файл). В свойстве Name имя объекта N1.

Свойство ShortCut задает клавиши быстрого доступа к меню – горячие клавиши.

Свойство Default определяет является ли данный раздел разделом по умолчанию своего подменю.

Свойство Break используется в длинных меню, чтобы разбить список разделов на несколько столбцов.

Свойство Checked, установленное в true, указывает, что в разделе меню будет отображаться маркер флажка, показывающий, что данный раздел выбран.

RadioItem позволяет вводить маркеры, если свойство установить в true. Это свойство, установленное в true, определяет, что данный раздел должен работать в режиме радиокнопки совместно с другими разделами, имеющими то же значение свойства Grouplndex. По умолчанию значение Grouplndexравно 0. Но можно задать его большим нуля, и тогда, если имеется несколько разделов с одинаковым значением Grouplndex и с Radio-Item = true, то в них могут появляться маркеры флажков, причем только в одном из них. Если вы зададите программно в одном из этих разделов Checked = true, то в остальных разделах Checked автоматически сбросится в false.

Основные свойства

|  |  |
| --- | --- |
| Свойство | Описание |
| AutoHotKeys | Определяет, могут ли горячие клавишиэлементов меню устанавливатьсяавтоматически |
| AutoLine  Redirection | Указывает, проверяет ли Delphiпоявление в начале или конце подменюразделителей или двух разделителейподряд |
| AutoMerge | Определяет, объединяются ли главныеменю вспомогательных форм с главнымменю основной формы. |
| Images | Определяет список изображений, кото-рые могут отображаться в разделахменю слева от надписи. |
| Items | Список элементов меню типа TMenuEtem |

Основные методы

|  |  |
| --- | --- |
| Свойство | Объявление / Описание |
| Findltem | type TFindltemKind = (fkCommand, fkHandle, fkShortCut); function FindItem(Value: Word; Kind: TFindltemKind):TMenuItem;Ищет и возвращает раздел меню, идентифицируемый значением Value, которое равно: при Kind = fkCommand — идентификатору ID, используемому в сообщении Windows WM\_COM-MAND; при Kind = fkHandle — дескриптору всплывающего меню; при Kind = fkShortCut — коду горячих клавиш. |
| IsShortCut | function IsShortCut(var Message: TWMKey): Boolean;Распознает горячие клавиши, связанные с разделом меню, и выполняет этот раздел меню. |
| Merge | procedure Merge(Menu: TMainMenu);Объединяет данное меню с указанным меню Menu вспомогательной формы. |
| Unmerge | procedure Unmerge(Menu: TMainMenu);Уничтожает объединение указанного меню Menu вспомогательной формы с данным меню. |

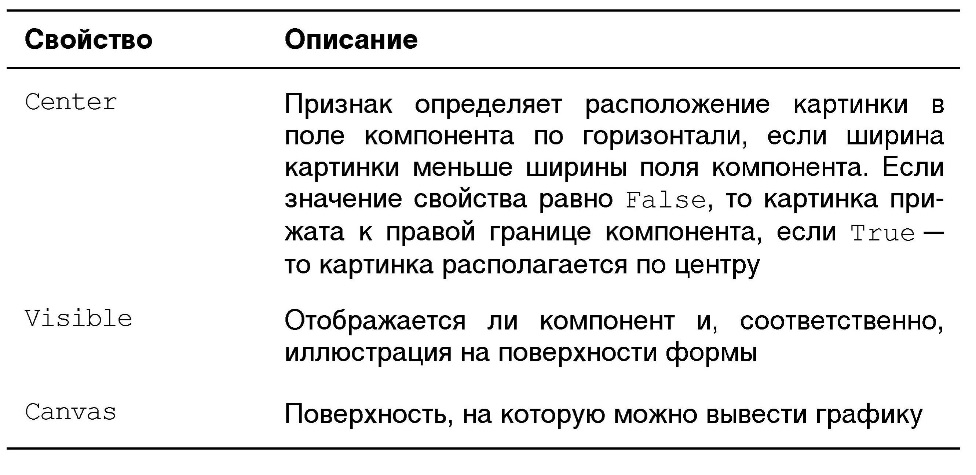
 Создавая из панели инструментов Standard компонент PopupMenu на нашей форме, вы заметите, что PopupMenu – это невизуальный компонент. ВообщеPopupMenu это всплывающее контекстное меню, которое возникает при нажатии пользователя правой кнопкой мыши.

Чтобы увидеть PopupMenu привяжите всплывающее меню к форме. Это делается на вкладке Events (события) главной формы PopupMenu. После этого при запуске проекта F9 щелкая правой кнопкой можно PopupMenu. Основные свойства можно перечесть по пальцам:Aligment (выравнивание), AutoHotKeys (горячие клавиши), AutoPopup(автоматическое появление при щелчке правой кнопки), TrackButton (какая кнопка активирует всплывающее меню, если связано с кнопкой панели инструментов).

Основные методы:

|  |  |
| --- | --- |
| Метод | Описание |
| DoPopup | Procedure DoPopup(Sender: TObject);Генерирует событие OnPopup |
| FindItem | Type TFindItemKind =(fkCommand, fkHandle, fkShortCut);Function FindItem(Value: Word, Kind:TFindItemKind): TMenuItem;  Ищет и возвращает раздел меню из  Значения Value равное Kind = fkCommand;  При Kind = fkHandle – дескриптор всплывающего  меню, при Kind = fkShortCut – код горячих клавиш |
| Popup | Procedure Popup (x,y: Integer);Показывает меню в указанной позиции экрана |

На вкладке Events (события): OnChange – событие при изменеии меню,OnPopup – событие



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |

9.3 Список изображений — компонент ImageList

Компонент **ImageList** представляет собой набор изображений одинаковых размеров, на которые можно ссылаться по индексам, начинающимся с 0. Во многих рассмотренных ранее компонентах (меню, списках и др.) встречались свойства, представляющие собой ссылки на компонент **ImageList**. Этот компонент позволяет организовать эффективное и экономное управление множеством пиктограмм и битовых матриц. Он может включать в себя монохромные битовые матрицы, содержащие маски для отображения прозрачности рисуемых изображений.

Изображения в компонент **TImageList** могут быть загружены в процессе проектирования с помощью редактора списков изображений. [Окно](http://www.fabrikaokon.ru/" \t "_blank) редактора, представленное на рис. 9.3, вызывается двойным щелчком на компоненте **TImageList** или щелчком правой кнопки мыши и выбором команды контекстного меню ImageList Editor.

|  |  |
| --- | --- |
| **Рис. 9.3** Окно редактора списков изображений |  |

В [окне](http://www.fabrikaokon.ru/" \t "_blank) редактора списков изображений вы можете добавить в списки изображения, пользуясь кнопкой Add, удалить изображение из списка кнопкой Delete, очистить весь список кнопкойClear. При добавлении изображения в список открывается обычное [окно](http://www.fabrikaokon.ru/" \t "_blank) открытия файлов изображений, в котором вы можете выбрать интересующий вас файл. Только учтите, что размер всех изображений в списке должен быть одинаковым. Как правило, это размер, используемый для пиктограмм в меню, списках, кнопках. При добавлении в список изображений для кнопок надо иметь в виду, что они часто содержат не одно, а два и более изображений (см. [раздел 5.2](http://www.beluch.ru/progr/100comp/5_2.htm)). В этих случаях при попытке добавить изображение задается вопрос: «Bitmap dimensions for... are greater then imagelist dimensions. Separate into ... separate bitmaps?» (Размерность изображения ... больше размерности списка. Разделить на ... отдельных битовых матрицы?). Если вы ответите отрицательно, то все изображения уменьшатся в горизонтальном размере и лягут как одно изображение. Использовать его в дальнейшем будет невозможно. Поэтому на заданный вопрос надо отвечать положительно. Тогда загружаемая битовая матрица автоматически разделится на отдельные изображения и потом вы можете удалить те из них, которые вам не нужны, кнопкойDelete.

Как видно из рис. 9.3, каждое загруженное в список изображение получает индекс. Именно на эти индексы впоследствии вы можете ссылаться в соответствующих свойствах разделов меню, списков, кнопок и т.д., когда вам надо загрузить в них то или иное изображение. Изменить последовательность изображений в списке вы можете просто перетащив изображение мышью на новое место.

В редакторе списков изображений вы можете, выделив то или иное изображение, установить его свойства: Transparent Color и Fill Color. Но это можно делать только в том сеансе работы с редактором списков изображений, в котором загружено данное изображение. Для изображений, загруженных в предыдущих сеансах, изменение этих свойств невозможно.

Свойство Transparent Color определяет цвет, который используется в маске для прозрачного рисования изображения. По умолчанию это цвет левого нижнего пикселя изображения. Для пиктограмм данное свойство устанавливается в **clNone**, поскольку пиктограммы уже маскированы.

Свойство Fill Color определяет цвет, используемый для заполнения пустого пространства при перемещении и центрировании изображения. Для пиктограмм данное свойство устанавливается в**clNone**.

Группа радиокнопок Options определяет способ размещения изображения битовой матрицы с размерами, не соответствующими размерам, принятым в списке:

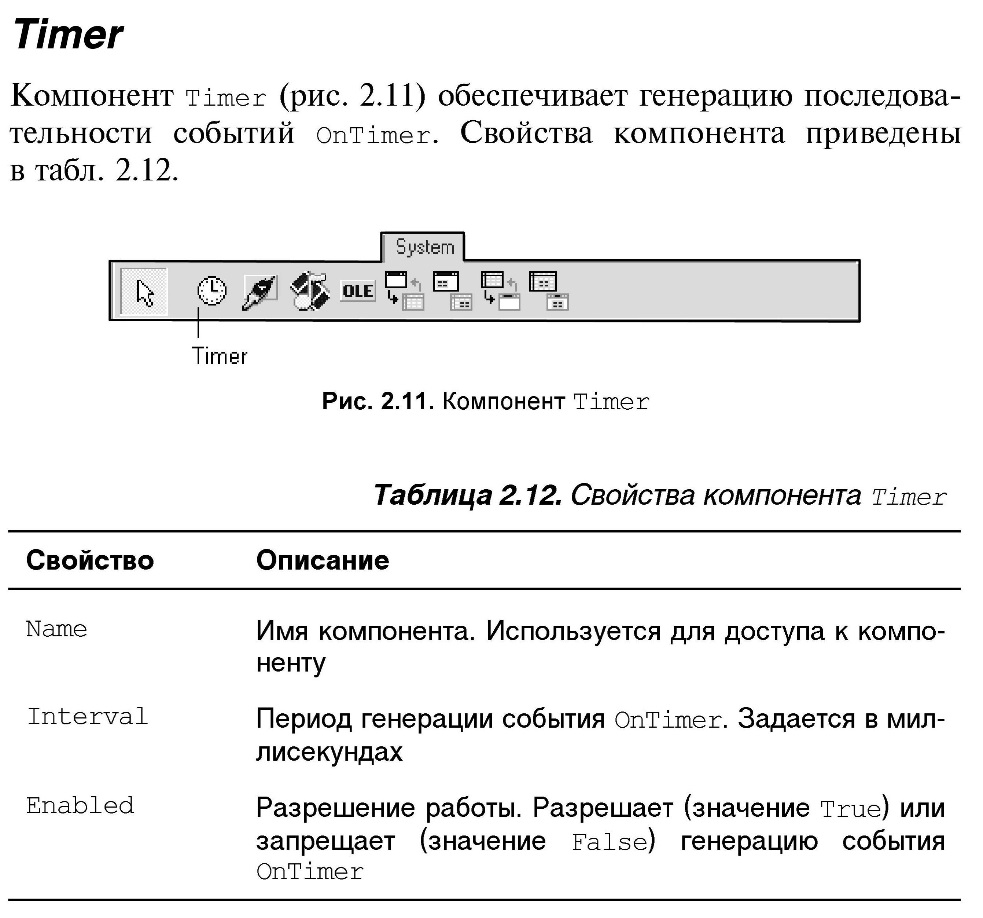
|  |  |
| --- | --- |
| **Crop** | Отображается часть изображения, помещающаяся в размер списка, начиная с левого верхнего угла |
| **Stretch** | Размеры изображения изменяются, становясь равными размерам списка. При этом возможны искажения |
| **Center** | Изображение центрируется, а если его размер больше размера списка, то не помещающиеся области отсекаются |

Теперь рассмотрим основные свойства **TImageList**:

|  |  |  |
| --- | --- | --- |
| **Свойство** | **Тип** | **Описание** |
| **Height** | **Integer** | Высота изображений в списке |
| **Width** | **Integer** | Ширина изображений в списке |
| **AllocBy** | **Integer** | Определяет количество изображений, на которое увеличивается список для добавления новых изображений |
| **Count** | **Integer** | Определяет число изображений в списке. Свойство только для чтения |

Остальные свойства определяют цвета и способы рисования изображения.

Компонент Timer в Delphi задает в приложении интервалы времени. У Таймера есть многочисленные применения: синхронизация мультипликаций, закрытие окон, с которыми пользователь некоторое время не работает, включение экранной заставки или закрытие связующих звеньев с удаленным сервером при отсутствий операций пользователя, регулярное тестирование источников информации, установки времени на отклик в обучающих программах – все эти задачи, в которые требуют задать интервалы времени, используются с помощью таймера.



У Timer в Delphi есть два управляющих свойства: Interval – значение времени в миллисекундах и Enabled – досягаемость. Через установленный интервал времени, таймер срабатывает событием OnTimer. Все заданные операции записываются в обработчике этого события.

При задании интервала (Interval) равного нулю или Enabled = false таймер становится неактивным. Для того, чтобы запустить Timer следует установить Enabled = true, при условии, что Interval – положительное число. Или же установить положительное значение Interval и значение Enabled = false.

К примеру, если нужно, чтобы через 10 секунд после запуска программы закрылась заставка, которая отображает логотип программы, то на заставке нужно разместить таймер и установить Interval = 10000, а в событии установить OnTimer = Close.

Если нужно в какой-то процедуре привести в действие таймер, который отсчитал бы установленный интервал, к примеру 10 секунд, после чего нужно выполнить некоторые операции и выключить таймер, это реализуется: при проектировании устанавливаем Enabled = true и интервал задается 0. Для запуска Timer нужно выполнить оператор Timer1.Inerval := 10000; через 10 секунд в событии OnTimer задать Nimer1.Interval :=0;

Timer в Delphi точно выдерживает установленные интервалы, даже если они велики – тысячи миллисикунд.

Основные свойства:

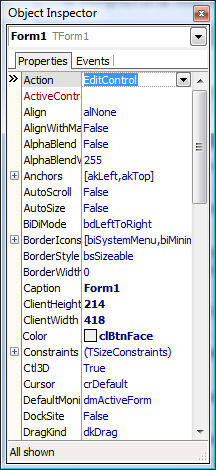
Enabled – property Enabled: Boolean;

Interval – property Interval: Cardinal;

Событие:  
OoTimer – событие происходит, когда истек очередной отрезок времени Interval.

№ 24 Прежде чем рассматривать основные компоненты Delphi, необходимо познакомиться с базовой иерархией классов этой системы.

1. Класс TObject – лежит в основе всей иерархии классов Delphi. Он обладает самыми общими методами, присущими любому объекту, описывает основные принципы поведения объектов во время работы программы (создание, уничтожение, обработка событий и др.).
2. Класс TPersistent – наследник класса TObject, реализует основные методы копирования содержимого объектов.
3. Класс TComponent – наследник класса TPersistent, основной родительский класс для всех классов, описывающих компоненты Delphi. В этот класс входит набор самых общих свойств, имеющихся у каждого компонента, и некоторые полезные методы.
4. Класс TControl – наследник класса TComponent, служит основным классом для всех визуальных элементов управления. Если элемент управления является стандартным элементом Windows, то он базируется еще на одном промежуточном классе TWinControl (наследнике класса TControl).



Свойство Name

Свойство Name содержит имя (идентификатор) компонента, по которому в программном коде производится обращение к этому компоненту. Каждому компоненту Дельфи автоматически присваивает имя исходя из типа компонента, например: Label1, Button1, Memo1. Если вы изменяете свойство Name - помните, идентификатор не должен содержать пробелов.

Свойство Owner

Свойство Owner содержит указатель на компонент, являющийся владельцем данного компонента. Форма является владельцем всех компонентов, которые в ней содержаться. Владельцем формы является приложение. Не следует путать компонент-владелец с родительским элементом управления (Parent). Родительским называется оконный элемент управления, в который непосредственно помещен дочерний компонент. Владельцем называется компонент, содержащий другие компоненты.

Свойство Tag

Свойство Tag служит для хранения произвольного целого числа или указателя, обычно используется в тех случаях, когда необходимо сохранить специфичную для данного компонента информацию.

Свойство Icon

Значение свойства Icon определяет, какая пиктограмма будет отображаться на панели задач Windows, когда приложение запущено (свернуто). Иконку приложения также можно задать в меню: Projects>Options>Application>Application Settings.

Свойство Caption

Свойство Caption содержит текст (либо название) элемента управления. Например: Label1.Caption:='Произвольный текст'; По умолчанию Delphi присваивает Caption то же значение, что и свойству Name.

Свойство Enabled

Свойство Enabled определяет, должен ли элемент управления реагировать на события мыши, клавиатуры или таймера. Если свойство имеет значение True, то элемент реагирует на события, иначе они игнорируются.

Свойство Align

Свойство Align определяет, как располагаются элементы управления внутри своего родительского элемента. Это свойство необходимо применять, если элемент управления должен оставаться на определенном месте в форме при изменении ее размеров.

Свойство Cursor

Свойство Cursor определяет изображение указателя мыши в тот момент, когда он находится на элементе управления. Например:

Button1.Cursor:=crHourGlass;

Свойство ClientHeight

Свойство ClientHeight определяет высоту рабочей области элемента управления в пикселях. Для элементов управления значение свойства ClientHeight равно значению свойства Height. Для форм свойство ClientHeight содержит высоту рабочей области формы (значение Height за вычетом высоты строки заголовка, линейки прокрутки и рамки окна).

Свойство Height

Свойство Height содержит высоту элемента управления в пикселях.

Свойство Top

Свойство Top содержит вертикальную координату (у) левого верхнего угла элемента управления (в пикселях) относительно формы или родительского элемента управления в котором данный элемент содержится.

Свойство Left

Свойство Left определяет горизонтальную координату левого края элемента управления (в пикселях) относительно формы или родительского элемента управления, в котором данный элемент содержится.

Свойство Visible

Свойство Visible определяет, будет ли данный компонент отображаться на экране. Если свойство Visible имеет значение True, то компонент виден пользователю, в противном случае - нет.

Свойство TabOrder

Свойство TabOrder определяет положение компонента в так называемой последовательности табулятора. Это последовательность, в которой компоненты становятся активными, когда пользователь нажимает клавишу [Tab].

Свойство TabStop

Свойство TabStop определяет, может ли пользователь перейти к элементу управления с помощью клавиши [Tab]. Если значение свойства TabStop - False, то данный элемент управления не входит в последовательность табулятора и пользователь не может к нему перейти с помощь клавиши [Tab]. По умолчанию значение TabStop равно True.

Свойство FormStyle

Свойство FormStyle определяет, является ли форма родительской либо дочерней или же независимой. Ниже представлены возможные значения:

fsNormal значение по умолчанию

fsMDIForm Родительская MDI-форма

fsMDIChild Дочерняя MDI-форма. (Существует лишь внутри родительской формы)

fsStayOnTop Форма, отображающаяся поверх всех окон

Свойство BorderIcons

Свойство BorderIcons определяет, какие кнопки отображаются в строке заголовка формы. Возможные значения приведены ниже:

biSystemMenu отображается значок системного меню

biMinimize отображается кнопка Свернуть

biMaximize отображается кнопка Развернуть

biHelp если значения biMinimize и biMaximize не заданы, в строке заголовка отображается значок Справка.

Свойство BorderStyle

Свойство BorderStyle определяет, отображается ли рамка форма, если да, то как именно. Ниже представлены возможные значения этого свойства:

bsDialog - Размеры формы неизменны, стандартная рамка

bsSingle - Размеры формы неизменны, простая рамка

bsNone - Размеры формы неизменны, рамки нет, также нет системных кнопок

bsSizeable - Размеры могут изменяться, стандартная рамка

bsToolWindow - Аналогично действию bsSingle, но строка заголовка меньше

bsSizeToolWin - Аналогично действию bsSizable, но строка заголовка меньше

Свойство AutoScroll

Свойство AutoScroll определяет, отображаются ли на оконном элементе управления линейки прокрутки, если этот элемент управления недостаточно велик, чтобы показать все содержащиеся в нем элементы. Если AutoScroll имеет значение True, линейки прокрутки автоматически появляются, когда они необходимы.

Свойство WindowState

Свойство WindowState определяет, как отображается форма (в обычном, свернутом или развернутом виде) - wsNormal, wsMaximized, wsMinimized.

**25.Характеристики бібліотеки компонентів.**

Бібліотека візуальних компонентів (Visual Component Library - VCL) Delphi містить безліч визначених типів компонентів, з яких користувач може будувати свою прикладну програму. Вітрину бібліотеки — палітру компонентів, ви бачите розташовану праворуч у смузі інструментальних панелей інтегрованого середовища розробки Delphi. На цій палітрі ви можете виділити курсором миші потрібний вам компонент і перенести його на форму.



Рис. 2.Палітра компонентів Delphi

Оскільки кількість сторінок у палітрі велика, і не всі закладки видно на екрані одночасно, в правій частині палітри компонентів є дві кнопки зі стрілками, спрямованими ліворуч і праворуч. Ці кнопки дозволяють переміщувати видиму на екрані частину палітри. Щоб перенести компонент на форму, треба відкрити відповідну сторінку бібліотеки та вказати курсором миші необхідний компонент. При цьому кнопка-вказівник , розміщена у лівій частині палітри компонентів, набуде вигляду не натиснутої кнопки. Це означає, що ви перебуваєте в стані, коли збираєтеся помістити компонент на форму. Помістити вибраний компонент на форму дуже просто — треба зробити клацання мишею в потрібному місці форми.



### Закладка Standard

Закладка Standard містить ряд часто використовуваних компонентів загального призначення



|  |  |  |
| --- | --- | --- |
| **Компонент** | **Тип** | **Опис** |
| фрейм | **Frame** | Панель з можливостями спадкоємства. Проектується як окреме вікно. Компонент візуальний. |
| головне меню | **MainMenu** | Дозволяє конструювати і створювати головне меню форми і випадаюче меню. Компонент невізуальний. |
| контекстне меню | **PopupMenu** | Дозволяє конструювати і створювати спливаюче контекстне меню. Компонент невізуальний. |
| мітка | **Label** | Використовується для розміщення на формах та інших контейнерах тексту, який не змінюється користувачем. Компонент візуальний. |
| вікно редагування | **Edit** | Використовується для введення користувачем однорядковий текстів. Може використовуватися для відображення тексту. Компонент візуальний. |
| багаторядкове вікно редагування | **Memo** | Використовується для введення та відображення багаторядкових текстів. Компонент візуальний. |
| командна кнопка | **Button** | Використовується для створення кнопок, якими користувач вибирає команди у додатку. Компонент візуальний. |
| контрольний  індикатор з прапорцем | **Checkbox** | Дозволяє користувачеві вмикати та вимикати різні опції. Компонент візуальний. |
| радіокнопка | **RadioButton** | Пропонують користувачеві набір альтернатив, з яких вибирається одна. Набір реалізується необхідною кількістю радіокнопок, розміщених в одному контейнері (формі, панелі і т.п.). Компонент візуальний. |
| вікно списку | **ListBox** | Являє собою стандартне вікно списку Windows, що дозволяє користувачеві вибирати розділи зі списку. Компонент візуальний. |
| редагований список | **ComboBox** | Об'єднує функції ListBox і Edit. Користувач може або ввести текст, або вибрати його зі списку. Компонент візуальний. |
| лінійка прокрутки | **Scrollbar** | Являє собою стандартну лінійку прокрутки Windows і служить для керування положенням видимої частини форм або компонентів. Компонент візуальний. |
| групове вікно | **GroupBox** | Є контейнером, що поєднує групу пов'язаних елементів управління, таких як радіокнопки RadioButton, контрольні індикатори Checkbox і т.д. Компонент візуальний. |
| група радіокнопок | **RadioGroup** | Є комбінацією групового вікна GroupBox з набором радіокнопок RadioButton; служить спеціально для створення груп радіокнопок. Можна розміщувати в компоненті кілька радіокнопок, але ніякі інші елементи управління не дозволені. Компонент візуальний. |
| панель | **Panel** | Є контейнером для групування елементів управління і менших контейнерів. Панель можна використовувати також для побудови смуг стану, інструментальних панелей, палітр інструментів. Компонент візуальний. |
| список подій | **ActionList** | Забезпечує диспетчеризацію подій компонентів. Компонент невізуальний. |

### Закладка Additional

Закладка є доповненням сторінки Standard і містить ряд часто використовуваних компонентів загального призначення.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Компонент** | | **Тип** | | **Описание** |
| кнопка с графикой | | **BitBtn** | | Используется для создания кнопок, на которых располагается битовая графика (например, кнопка OK с галочкой). Компонент визуальный. |
| кнопка с фиксацией | | **SpeedButton** | | Используется для создания инструментальных панелей и в других случаях, когда требуется кнопка с фиксацией нажатого состояния. Компонент визуальный. |
| маскированный ввод | | **MaskEdit** | | Используется для форматирования данных или для ввода символов в соответствии с шаблоном. Компонент визуальный. |
| таблица строк | | **StringGrid** | | Используется для отображения текстовой информации в таблице из строк и столбцов. Компонент визуальный. |
| таблица рисунков | | **DrawGrid** | | Используется для отображения в строках и столбцах нетекстовых данных. Компонент визуальный. |
| изображение | | **Image** | | Используется для отображения графики: пиктограмм, битовых матриц и метафайлов. Компонент визуальный. |
| формы | | **Shape** | | Используется для рисования фигур: квадратов, кругов и т.п. Компонент визуальный. |
| рамка | | **Bevel** | | Используется для рисования выступающих или утопленных  линий или прямоугольных рамок. Компонент визуальный. | | |
| окно с прокруткой | | **ScrollBox** | | Используется для создания зон отображения с прокруткой.  Компонент визуальный. | | |
| список с флажками | | **CheckListBox** | | Компонент является комбинацией свойств списка ListBox и индикаторов  CheckBox в одном компоненте. Компонент визуальный. | | |
| разделитель панелей | | **Splitter** | | Используется для создания в приложении панелей с изменяемыми  пользователем размерами. Компонент визуальный. | | |
| метка с бордюром | | **StaticText** | | Компонент подобен компоненту Label, но обеспечивает  дополнительные возможности по заданию стиля бордюра.  Компонент визуальный. | | |
| инструментальная панель | | **ControlBar** | | Используется для размещения компонентов инструментальной панели  . Компонент визуальный. | | |
| события приложения | | **ApplicationEvents** | | Перехватывает события на уровне приложения. Компонент невизуальный. | | |
| диаграммы и графики | | **Chart** | | Компонент принадлежит к семейству компонентов TChart, которые  используются для создания диаграмм и графиков. Компонент визуальный. | | |

### Закладка Win32

Сторінка Win32 містить компоненти загального призначення, які дозволяють розробляти додатки в стилі Windows 95/98 і NT 4.x. Деякі з цих компонентів аналогічні наявним на сторінці Win3.1.



|  |  |  |  |
| --- | --- | --- | --- |
| **Компонент** | **Тип** | | **Описание** |
| страница с закладкой | **TabControl** | | Позволяет организовывать страницы с закладками в стиле Windows 95, которые может выбирать пользователь. Компонент визуальный. |
| многостраничное окно | **PageControl** | | Позволяет создавать страницы в стиле Windows 95/98, управляемые закладками или иными органами управления, для экономии места на рабочем столе. Компонент визуальный. |
| список изображений | **ImageList** | | Предназначен для работы со списками изображений одинакового размера в меню, инструментальных панелях и т.п. Компонент невизуальный. |
| окно редактирования в формате RTF | **RichEdit** | | Представляет собой окно редактирования в стиле Windows 95/98, позволяющее производить выбор цвета и шрифта, поиск текста и многое другое. Компонент визуальный. |
| ползунок | **TrackBar** | | Управляющий элемент в виде ползунка в стиле Windows 95/98. Компонент визуальный. |
| отображение хода процесса | **ProgressBar** | | Используется для отображения в стиле Windows 95/98 хода процессов, занимающих заметное время. Компонент визуальный. |
| кнопка-счетчик | **UpDown** | | Кнопка-счетчик в стиле Windows 95/98 для ввода целых чисел. Компонент визуальный. |
| «горячие» клавиши | **HotKey** | | Дает возможность реализовать в приложении поддержку горячих клавиш. Компонент визуальный. |
| воспроизведение немых клипов | **Animate** | | Используется для воспроизведения немых клипов AVI, подобных используемым в Windows 95/98 изображениям копирования файлов и т.п. Компонент визуальный. |
| ввод дат и времени | **DateTimePicker** | | Ввод дат и времени с выпадающим календарем. Компонент визуальный. |
| ввод дат | **MonthCalendar** | | Ввод дат с выбором из календаря. Компонент визуальный. |
| дерево | **TreeView** | | Предоставляет возможность просмотра структуры иерархических данных в стиле Windows 95/98. Компонент визуальный. |
| списки | **ListView** | | Отображает списки в стиле Windows 95/98. Компонент визуальный. |
| заголовок | | | **HeaderControl** | Позволяет создавать составные перемещаемые заголовки в стиле  Windows 95/98. Компонент визуальный. | |
| полоса состояния | | | **StatusBar** | Полоса состояния программы, при необходимости — на нескольких  панелях. Компонент визуальный. | |
| инструментальная панель | | | **ToolBar** | Инструментальная панель для быстрого доступа к часто используемым  функциям приложения. Компонент визуальный. | |
| инструментальная перестраиваемая панель | | | **CoolBar** | Контейнер инструментальной панели, размеры которой могут  изменяться пользователем. Компонент визуальный. | |
| прокрутка страниц | | | **PageScroller** | Обеспечивает прокрутку больших окон, например, инструментальных  панелей. Компонент визуальный. | |

### Закладка System

Сторінка System містить компоненти, що дозволяють використовувати системні засоби Windows.



|  |  |  |
| --- | --- | --- |
| **Компонент** | **Тип** | **Описание** |
| таймер | **Timer** | Используется для запуска процедур, функций и событий в указанные интервалы времени. Компонент невизуальный. |
| окно для рисования | **PaintBox** | Используется для создания на форме некоторой области, в которой можно рисовать. Компонент визуальный. |
| аудио и видео плеер | **MediaPlayer** | Используется для создания панели управления воспроизведением звуковых и видео файлов, а также устройств мультимедиа. Компонент визуальный. |
| контейнер OLE | **OLEContainer** | Используется при создании области клиента для объекта OLE. Компонент визуальный. |
| диалог с сервером DDE | **DDEClientConv** | Используется клиентом DDE для организации диалога с сервером DDE. Компонент невизуальный. |
| данные, передаваемые серверу DDE | **DDEClientItem** | Используется для определения данных клиента, передаваемых в диалоге серверу DDE. Компонент невизуальный. |
| диалог с клиентом DDE | **DDEServerConv** | Компонент используется сервером DDE при проведении диалога с клиентом DDE. Компонент невизуальный. |
| данные, передаваемые клиенту DDE | **DDEServerItem** | Компонент используется для определения данных сервера, передаваемых клиенту DDE в течение диалога. Компонент невизуальный. |

**26.Основні події візуальних компонент пов`язані з клавіатурою.**

|  |  |
| --- | --- |
| **Подія** | **Опис** |
| OnKeyDown | виникає при натисненні клавіші на клавіатурі.  **Кey** - код натиснутої клавіші, **Shift** - множина натиснутих функціональних клавіш |
| OnKeyUp | виникає при відпусканні натиснутої клавіші на клавіатурі. |
| OnKeyPress | виникає при натисканні клавіші, яка друкує деякий символ.  **Key** - вже не код клавіші, а сам символ |
|  |  |
|  |  |
| OnEnter | виникає в момент, коли об’єкт отримує фокус.  Фокус вводу зазвичай відображається пунктирною рамкою на об’єкті; у полі вводу — це курсор; фокус один на все застосування, тобто працювати одночасно можна лише з одним об’єктом |

**27. Основні події візуальних компонент пов`язані з мишею.**

|  |  |
| --- | --- |
| **Подія** | **Опис** |
| onMouseDown | виникає при натисканні кнопки миші (не обов’язково лівої).  Параметр **Button** – натиснута кнопка (mbLeft - ліва кнопка, mbRight - права, mbMiddle - середня). **Shift** - множина, що вказує на натиснуті функціональні клавіші. Таким чином можна відслідкувати, наприклад, натискання при затиснутих Ctrl, Alt і т.ін., **X**, **Y** - координати курсору відносно лівого верхнього кута самого компоненту |
| onMouseUp | виникає при відпусканні натиснутої кнопки миші.  Наприклад, графічний редактор: коли кнопка миші натиснута — виконується рисування, а коли відпущена — не виконується |
| onMouseMove | виникає при наведенні курсору миші на елемент.  **X**, **Y** - координати нового положення, **Shift** – множина натиснутих функціональних клавіш |
| onClick | виникає при одинарному клацанні клавішею миші.  У кнопки подія також виникає якщо натиснути Enter або Пробіл в той момент, коли фокус вводу знаходиться на кнопці |
| onDblClick | виникає при подвійному клацанні клавішею миші. |

**28.Процедури та функції. Декларування, використання.Передача аргументів за значенням та по посиланню.**

Процедуры — это конструкции программного кода, которые позволяют создавать в программном коде некоторые подпрограммы, которые выполняют определенные операции независимо от остального программного кода. Разберем несложный пример:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | **procedure** shownumbers(n:**integer**); **var** i:**integer**; **begin** **for** i:=1 **to** n **do** showmessage(inttostr(i)); **end**;  **procedure** showsimplemessages; **begin** showmessage('This is a simple message 1!'); showmessage('This is a simple message 2!'); **end**;  **procedure** TForm1.OnCreate(Sender:**TObject**); **begin** shownumbers(5); showsimplemessages; **end**; |

Пока не будем изучать синтаксис, сначала разберемся с принципом работы процедур. Как вы уже наверное заметили, все события (например тот же OnCreate) представлены процедурами. Сначала изучим содержимое обработчика события создания формы Form1 (или точнее уже можно говорить — содержимое процедуры «procedure TForm1.OnCreate(Sender:TObject)»). Внутри этой процедуры мы видим «shownumbers(5);» и «showsimplemessages;» — это и есть вызов процедур, находящихся выше чем «procedure TForm1.OnCreate(Sender:TObject)». Как вы уже наверняка поняли, для того чтобы вызвать существующую процедуру, необходимо сначала указать ее название, а затем в скобках перечислить ее параметры. Если параметр один, то достаточно просто указать его значение. Также, процедуры могут не иметь параметров. В таком случае скобки можно опустить. Указывать сами параметры нужно в соответствующем им типе. Например строковые переменные «string» нужно указывать в кавычках  ‘ ‘.

Теперь разберем синтаксис самой процедуры. В самом начале процедуры идет ключевое слово «procedure». Затем указывается имя процедуры, а после имени указываются параметры в скобках. Указываются они перечислением через «точку с запятой». Несколько однотипных переменных подряд можно указывать через запятую. На словах это понять достаточно трудно, поэтому приведу пример:

|  |  |
| --- | --- |
| 1 | **procedure** example(a,b,c:**integer**; d:**string**; e:**real**; x1,y1:**string**); |

Затем, после объявления параметров процедуры, указывается программный код процедуры между ключевыми словами «begin» и «end» также, как это показано на первом примере. Перед «begin» можно также указать локальные переменные, создав новый раздел var. Эти переменные могут использоваться и обрабатываться только внутри самой процедурой, внутри которой они объявлены. Также внутри программного кода процедуры можно использовать параметры процедуры как обычные переменные указанного в процедуре типа. Только во время вызова процедуры, всем параметрам процедуры будет уже переданы соответствующие значения, указанные в вызове процедуры.

[warning]Очень важно при написании кода учитывать то, что сама процедура должна находиться выше в коде, чем то место где происходит ее вызов! Иначе может произойти ошибка, связанная с тем, что процедура попросту не будет найдена. [/warning]

[note]Любая процедура может вызвать любую другую процедуру, но только если вызываемая процедура стоит выше вызывающей ее процедуры.[/note]

[note]Внутри процедур можно обращаться к глобальным переменным и изменять их значения.[/note]

Надеюсь, что принцип работы процедур я достаточно вам объяснил. Теперь рассмотрим функции. Функции — это по сути те же процедуры, но только функции могут возвращать результат, т.е. какое-либо значение. Принцип устройства процедур можно очень легко понять на простом примере:

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 | **function** calc(a,b,c:**integer**; d:**string**): **integer**; **begin** result:=length(d); inc(a,5); dec(b,2); inc(result, a+b-c); **end**;  **procedure** TForm1.OnCreate(Sender:**TObject**); **var** a:**integer**; **begin** a:=calc(1,2,3,'testing'); showmessage('Результат функции равен '+inttostr(a)); **end**; |

Кстати, в этом примере мы также разберем две новые процедуры и одну новую для вас функцию, уже предусмотренные в Pascal/Delphi. Эти процедуры — «inc(a;integer;b:integer)» и «dec(a:integer;b:integer)». Первая называется инкрементом, и увеличивает целочисленное число «a» на «б» единиц. Вторая процедура называется декрементом, и делает совершенно обратную операцию инкременту, а именно уменьшает целочисленное число «a» на «b» единиц. Если переменной «b» не указать значение, то вместо числа «b» автоматически будет использоваться единица. Таким образом «inc(a);» — тоже самое что и «inc(a,1);». В качестве «b» могут выступать и отрицательные числа, что приведет к инверсии операции, т.е. «inc(a,-3);» — тоже самое что и «dec(a,3);». Процедуры инкремента и декремента использовать несколько удобнее и работают они относительно быстрее чем присвоение «a:=a+1;».

[note]Функция «length(s:string):integer» подсчитывает количество символов в строке s и возвращает результат в  виде целочисленной переменной.[/note]

В дальнейших уроках, когда я буду вам показывать новые функции или процедуры, мне будет достаточно указать ее заголовок, в котором описаны все имена, типы переменных и конечно же название самой функции или процедуры, т.к. синтаксис этих заголовков мы уже разобрали с вами в этом уроке.

Во всех функциях присутствует зарезервированная переменная под названием «result», которая хранит в себе результат функции, т.е. ее значение. Внутри программного кода функции нужно произвести присвоение переменной «result» значения, которое должна возвращать функция. Тип переменной «result» указывается после перечисления параметров (после закрывающей скобки и двоеточия), как это показано на примере. Внутри программного кода функции мы также как и в процедуре можем производить любые операции над параметрами, а также и над переменной «result». Также как и в процедуре, в функции необязательны параметры. Тогда она может выглядеть так:

|  |  |
| --- | --- |
| 1 2 3 4 | **function** example: **string**; **begin** result:='simple function'; **end**; |

Вызов функции осуществляется через присвоение, так как это показано в примере. Параметры функции указываются точно так же как и у процедуры.

Далее я приведу несколько замечаний, касающихся работы с функциями и процедурами:

[note]Функции и процедуры могут вызывать сами себя. Такой прием программирования называется рекурсией и используются чаще всего в реализации каких-либо алгоритмов.[/note]

[note]Название процедур и функций может содержать только латинские буквы, цифры и знаки подчеркивания.[/note]

[note]Функции и процедуры с любыми различиями в параметрах считаются абсолютно разными и независимыми друг от друга, даже если у них одинаковые названия. Если существует две функции или процедуры с одинаковыми названиями, но разными параметрами, то при вызове одной из этих процедур или функций будет использоваться автоматически та, которая подходит по параметрам (по их типам и количеству).[/note]

**29. Створення головного меню програми.**

Для **створення головного меню** необхідно включити до форми компонент *TMainMenu*. Цей компонент представляє головне меню та наслідує методи та властивості *TМenu*.

**Основі властивості** **у *TMainMenu:***

* *AutoMerge* – керування злиттям
* *Handle* – системний дескриптор

**Основі властивості від *TMenu:***

* *AutoHotkeys* – застосування гарячих клавіш
* *Bitmap* – малюнок для фону
* *Color* – фон
* *Images* – список малюнків для меню
* *Items* – список елементів меню
* *Palette* – палітра кольору

**Основні методи у *TMainMenu:***

* *ShortCut* – визначити гарячі клавіші
* *Merge* – злити меню
* *Unmerge* – відмінити злиття
* *Break* – розбиття «довгих» меню на декілька стовпців
* *Checked* — вказує на можливість використання в меню маркера «прапорця»

**Основні методи від *TMenu:***

* *FindItem* – отримати елемент
* *SetForm* – призначити базову форму

**Основні події:**

* *OnChange* – в разі зміни елементу

Для створення меню можна користуватися *Конструктором меню*. Для цього достатньо подвійно клацнути на піктограмі компоненту на формі. *Конструктор меню* надає можливість створювати підменю (для цього необхідно натиснути комбінацію *Ctrl+* для того рядка, для якого створюються підменю). Для забезпечення зв’язку головного меню та форми необхідно властивості *Menu* форми присвоїти значення імені головного меню. Важливою характеристикою головного меню є реалізація механізму об’єднання меню. Для того щоб реалізувати об’єднання меню необхідно виконати присвоєння властивості *AutoMerge* значення *True* для тих форм, меню яких буде додаватися до меню головної форми. Для головної форми значення цієї властивості повинно встановити властивості як *False*. Поєднання меню проходить за спеціальними правилами в основі яких лежить використання властивості *GroupIndex* відповідних об’єктів *TItemMenu*. У пунктів об’єктів одного рівня, наприклад у всіх елементів підменю верхнього рівня значення властивості збільшується. Для звичайних форм об’єднання проходить тільки на верхньому рівні в головному меню в разі їх активізації. В об’єднаному меню всі підменю будуть розташовані за збільшенням номеру групового індексу. Об’єднання виконується за правилами:

* якщо значення групового індексу в меню, що приєднується співпадає зі значенням групового індексу головного меню, то множина таких меню замінює відповідну множину пунктів головного меню;
* *AutoHotkeys* – застосування гарячих клавіш
* *Bitmap* – малюнок для фону
* *Color* – фон
* *Images* – список малюнків для меню
* *Items* – список елементів меню
* *Palette* – палітра кольору

**Основні методи у *TMainMenu:***

* *ShortCut* – визначити гарячі клавіші
* *Merge* – злити меню
* *Unmerge* – відмінити злиття
* *Break* – розбиття «довгих» меню на декілька стовпців
* *Checked* — вказує на можливість використання в меню маркера «прапорця»

**Основні методи від *TMenu:***

* *FindItem* – отримати елемент
* *SetForm* – призначити базову форму

**Основні події:**

* *OnChange* – в разі зміни елементу

Для створення меню можна користуватися *Конструктором меню*. Для цього достатньо подвійно клацнути на піктограмі компоненту на формі. *Конструктор меню* надає можливість створювати підменю (для цього необхідно натиснути комбінацію *Ctrl+* для того рядка, для якого створюються підменю). Для забезпечення зв’язку головного меню та форми необхідно властивості *Menu* форми присвоїти значення імені головного меню. Важливою характеристикою головного меню є реалізація механізму об’єднання меню. Для того щоб реалізувати об’єднання меню необхідно виконати присвоєння властивості *AutoMerge* значення *True* для тих форм, меню яких буде додаватися до меню головної форми. Для головної форми значення цієї властивості повинно встановити властивості як *False*. Поєднання меню проходить за спеціальними правилами в основі яких лежить використання властивості *GroupIndex* відповідних об’єктів *TItemMenu*. У пунктів об’єктів одного рівня, наприклад у всіх елементів підменю верхнього рівня значення властивості збільшується. Для звичайних форм об’єднання проходить тільки на верхньому рівні в головному меню в разі їх активізації. В об’єднаному меню всі підменю будуть розташовані за збільшенням номеру групового індексу. Об’єднання виконується за правилами:

* якщо значення групового індексу в меню, що приєднується співпадає зі значенням групового індексу головного меню, то множина таких меню замінює відповідну множину пунктів головного меню;
* всі пункти меню, що приєднуються, груповий індекс яких не співпадає з індексами головного меню, додаються до головного та розташовуються на відповідних значенню групового індексу місцях.

Крім автоматичного включення механізму злиття меню, керування процесом об’єднання меню можна виконати шляхом використання методів *Merge* або *UnMerge* компонента. Для форм *MDI* механізм об’єднання меню включається завжди незалежно від значення властивості *AutoMerge*.

**30. Створення спливаючого меню програми.**

Для **створення спливаючого меню** необхідно включити до форми компонент *TPopupMenu*. Цей компонент представляє спливаюче меню та наслідує властивості класу *ТMenu*. Для створення меню можна використовувати *Конструктор меню*. На відміну від головного меню спливаюче меню можна створювати не тільки для вікна форми, а також і для більшості віконних елементів-контейнерів. Для забезпечення зв’язку спливаючого меню та форми необхідно властивості *PopupMenu* компонента-контейнера (наприклад, форми) присвоїти значення імені спливаючого меню.

Спливаюче меню з’являється в точці клацання. Розташування вікна меню визначається значенням властивості Alignment:

* paLeft – лівий верхній кут
* paRight – правий верхній кут
* paCenter – середина верхньої сторони

Для виклику спливаючого меню можна використати наступний метод компоненту:

*procedure PopUp ( X,Y: integer );*

де ( X,Y ) – координати точки, що визначає місце появи меню.

Компоненти *TMainMenu* та *TPopupMenu* мають властивість *Items*, яка визначає перелік елементів меню. Якщо клацнути на її назві в вікні інспектора об’єктів, то розкриється вікно Конструктора меню. Кожний елемент меню є об’єктом класу *TMenuItem*. До складу об’єктів Delphi він включений за схемою *TObject – TPrsistent – TComponent – TМenuItem*. Компонент відсутній в палітрі компонентів. Він створюється автоматично в процесі створення меню та містить методи та властивості елемента меню.

**Основні властивості:**

* *Caption* – значення тексту рядка меню. Якщо в тексті зустрічається символ & то наступний за ним символ є акселератом тобто комбінація <Alt>+ символ, якщо текст складається з одного символу “-” то цей пункт розподілювачем;
* *ShortCut* – завдання комбінації “гаряча клавіша” для швидкого виклику пункту меню. Необхідне значення обирається з наведеного переліку значень в вікні інспектора об’єктів.
* *Enabled* – стан активності елемента меню;
* *Checked* – стан відмічання елементу меню;
* *НelpContext* – значення тексту контекстної допомоги;
* *GroupIndex* – значення групового індексу.

До складу компонента включено ряд **методів** які дозволяють на протязі часу run-time змінювати структуру дочірніх меню:

* *Parent* – посилання на батьківське меню;
* *Count* – кількість дочірніх меню;
* *Items* – перелік дочірніх меню, звернення за значенням індексу;
* *Insert* – вставити меню за значенням індексу;
* *Delete* – вилучити меню за значенням індексу;
* *IndexOf* – повернути значення індексу по імені;
* *Add* – додати елемент в кінець переліку меню;
* *Remove* – вилучити дочірнє меню за іменем.

**31.Опишіть структуру проекту Делфи.**

**Файл проекту** виглядає наступним чином:

*program Project1;*

*uses*

*Forms*

*Unit1 in "Unit 1.pas" {Form};*

*{$R \*. RES}*

*begin*

*Application.CreateForm (TForm1, Form1);*

*Application.Run;*

*end.*

Використання модуля Forms обов'язково для всіх програм, створюваних в середовищі Delphi, тому що в цьому модулі міститься опис класу TApplication

Директива *in* вказує на те, що модуль є необхідною частиною проекту, а не просто файлом, що використаються в проекті. Директива *$R* підключає до створюваного проекту необхідні ресурси. Зображення форми зберігається в вигляді Windows-ресурсу, але має розширення .dfm.

Метод *Application.CreateForm* створює форму, а метод *Application.Run* завантажує головну форму і виконує застосування.

Для кожної форми, що включається в проект, створюється окремий модуль, але можна включати в проект і модуль, не зв'язані з формами.

**Структура модуля,** що створюється для форми

*unit Unit1;*

*interfaсe*

*{Розділ інтерфейсу. Код і змінні, описані в цьому розділі, можуть використовуватися іншими модулями}*

*uses SysUtilts, WinTypes,..., Dialogs;*

*{список модулів, що повинна використати програма. Всі ці модулі є стандартними модулями і вони містять компоненти, що використовуються в програмі}*

*type*

*{Cекція опису типів описується клас, відповідний формі}*

*TForm1=class (TForm)*

*private*

*{private declarations}*

*public*

*{public declarations}*

*end;*

*var*

*Form1:TForm1;*

*implementation*

*{Розділ реалізації. Код і змінні, описані в цьому розділі, є прихованими від будь-якого коду поза даним модулем}*

*{$R. dfm}*

*end.*

**Розділ інтерфейсу** модуля складається з всього, що знаходиться між зарезервованими словами *interface* і *implementation*. Розділ інтерфейсу служить тільки для визначення. Він містить визначення констант, типів і змінних та включає в себе також визначення процедур і функцій

Якщо є модуль і необхідно використати будь-що з його вмісту, необхідно включити ім'я модуля в оператор *Uses*.

**Розділ реалізації**починається з зарезервованого слова *implementation*, та закінчується словом *end*. В розділі реалізації модуля визначаються дві основні групи: тіла функцій і процедур, що реалізують заголовки.

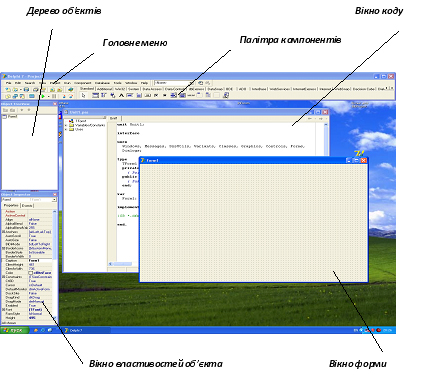
Результатом компіляції Delphi-проектів є виконавчий файл (\*.exe), або динамічна бібліотека (\*.dll).

**32.Вкажіть призначення типів файлів що за замовчуванням входять до складу проекту в Делфи.**

**Склад проекту** складається з наступних файлів:

* файл проекту (\*. dpr) — містить програму на мові Object Pascal, яка створює форми та виконує команду запуску додатку;
* файли форм (\*.dfm) — в файлах міститься опис форм проекту (опис інтерфейсу користувача)
* файли програмних модулів (\*. pas) — в файлах міститься програмний текст модулів, які визначають поведінку об'єктів включених до складу форми, тобто реалізують алгоритми користувача;
* файл ресурсів (\*. RES) — файл містить опис Windows ресурсів необхідних для проекту;
* файл опцій (\*. opt, або \*. dop ) — в файлі містяться перелік та значення параметрів проекту.

**33.Структура інтерфейсу користувача Делфи(основні інструменти середовища розробки)**



Після завантаження Delphi розглянемо вікна, що з’явились на екрані. *Головне вікно* (Main Window) використовується для управління створенням проекту і запуску створених програм.. Головне вікно має *Рядок заголовка* (Title Bar). У рядку заголовка вказується ім’я проекту.

Нижче рядка заголовка знаходиться *Головне меню* (Main Menu), яке необхідне для виконання різних функцій Delphi. Головне меню складається з таких елементів: File, Edit, Search, View, Project, Run, Component, DataBase, Tools, Help.  Рядок меню дозволяє керувати всією роботою по створенню проекту.  У рядку меню є « випадаючі» меню, за допомогою яких здійснюється управління середовищем розробки Delphi.



Під головним меню зліва знаходиться *Панель інструментів (Toolbar).*На панелі інструментів знаходяться кнопки із зображеннями, які призначені для швидкого доступу до основних команд меню.



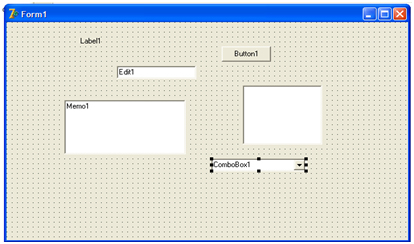
Якщо помістити курсор миші на кнопку панелі, то через деякий час з’явиться спливаюча підказка, що повідомляє про призначення даної кнопки. При роботі в середовищі кнопки панелі інструментів використовуються часто.

Права панель містить бібліотеки візуальних компонентів Visual Component Library-VCL), надалі просто *палітра компонентів*. Палітра компонентів дозволяє вибрати за допомогою кнопок візуальні та інші компоненти, із яких, як із «будівельних блоків», збирають  свій Delphi-додаток.



Палітра містить ряд сторінок, закладки яких видно в її верхній частині. Найбільш вживані із них: Standard (стандартні компоненти) і Additional (додаткові компоненти). Щоб помістити компонент у центрі вікна форми, двічі клацають на його піктограмі. Якщо потрібно розташувати компонент десь на формі, клацають один раз на піктограмі і один раз у потрібному місці форми. Вибраний компонент можна переміщати на формі, а також змінювати його розміри, перетягуючи маркери.

*Форма (Form*) є основним елементом управління при розробці проекту в Delphi. Саме на формі розміщують потрібні компоненти, задають для них потрібні властивості тощо. Увесь внутрішній простір є робочою ділянкою, яка може бути покрита крапчастою сіткою для вирівнювання компонентів. Вікно форми *Form*є початком розробки проекту і є заготовкою (макетом)  вікна проекту, який розробляється.

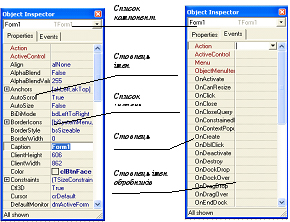


|  |  |
| --- | --- |
|  | *Розробка проекту розпочинається з форми!  Форма є контейнером, і може містити інші компоненти.* |

Форма є основою при створенні проекту в середовищі Delphi. Вікно форми можна переміщати по екрані, збільшувати, зменшувати, змінювати його властивості у вікні властивостей об’єктів, упорядковувати на ньому поставлені об’єкти і у вікні коду писати для них процедури.

|  |  |
| --- | --- |
|  | Компоненти  Delphi- це вже готові елементи інтерфейсу, які ви зможете використовувати у своїх програмах. |

*Інспектор об’єктів* (Object Inspector). За допомогою інспектора об’єктів можна задавати початкові значення властивостей об’єкта та їхню реакцію на стандартні події. Вікно інспектора об’єктів містить список компонентів поточної форми, а також дві закладки: властивостей (Properties) та подій (Events).



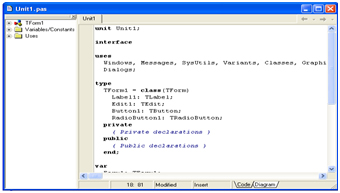
|  |
| --- |
|  |

Кожна закладка складається з двох стовпців: лівий містить назви властивостей або подій, а правий – їхні значення або назву процедури-реакції на подію. Активізувати вікно інспектора об’єктів можна за допомогою клавіші F11 або мишкою.

У вікні *дерево об’єктів* відображено всі об’єкти, які поставлені на поточну форму. Тобто у цьому вікні відображено назву форми (Form1), а також візуальні об’єкти, що встановлені на форму.



Вікно *Редактор коду* (Code Editor) призначене для написання коду програми на Object Pascal, який реалізує процедуру опрацювання певної події. Під час відкривання нового проекту в модуль Unit1.pas, який відповідає формі Form1, редактор автоматично заносить програмний код опису цієї форми. Під час додавання нових компонентів у вікно форми у програму автоматично заносяться коди з описами параметрів цих компонентів (висота, ширина, розташування, стиль тощо).



Долучення певного об’єкта чи застосування методу до нього веде до появи базового коду відповідної процедури у вікні редактора. Заготовка(шаблон) складається з заголовка процедури та ключових слів begin і end. Заготовку заповнює користувач.

**34.Компоненти для обробки тексту.**

**Отображение текста в надписях компонентов Label, StaticText и Panel**

Для отображения различных надписей на форме используются в основном компоненты ***Label***, ***StaticText*** и ***Panel***. Первые два из этих компонентов — метки, специально предназначенные для отображения текстов. Основное назначение панели ***Panel*** другое: компоновка компонентов в окне формы.

Тексты, отображаемые в перечисленных компонентах, определяются значением их свойства ***Caption***. Его можно устанавливать в процессе проектирования или задавать и изменять программно во время выполнения приложения.

*Label1.Caption := 'Новый текст';*

Если требуется отобразить числовую информацию, можно воспользоваться функциями ***FloatToStr*** и ***IntToStr***, переводящими соответственно числа с плавающей запятой и целые в строку. Для формирования текста, состоящего из нескольких фрагментов, можно использовать операцию «*+*», которая для строк означает их склеивание (конкатенацию). Например, если в программе имеется целая переменная ***I***, отображающая число сотрудников некоторой организации, то вывести в метку ***Label1***информацию об этом можно оператором:

*Label1.Caption := 'Число сотрудников: '+IntToStr(I);*

Во всех компонентах цвет фона определяется свойством ***Color***, а цвет надписи — подсвойством ***Color*** свойства ***Font***. Например, в большинстве меток задан цвет фона равным ***clWhite*** — белый. Если цвет специально не задавать, то цвет фона обычно сливается с цветом контейнера, содержащего метку, так что фон просто не заметен.

Компонент ***Panel*** кроме свойства ***BorderStyle*** имеет еще свойства ***BevelInner***, ***BevelOuter***, ***BevelWidth***, ***BorderWidth***, которые предоставляют возможности оформления надписи. Таким образом, с точки зрения оформления выводимого текста максимальные возможности дает ***Panel*** и минимальные — ***Label***.

Размещение всех рассматриваемых компонентов на форме определяется, в частности, свойствами ***Тор*** (координата верхнего края), ***Left*** (координата левого края), ***Height*** (высота), ***Width*** (ширина). Имеются также свойства, определяющие изменение размера компонента при изменении пользователем во время выполнения приложения размеров окна. Это общие свойства всех оконных компонентов: ***Align*** (выравнивание компонента по всей верхней, левой, правой, нижней частям контейнера или по всей его клиентской области), ***Anchors*** (привязка сторон компонента к сторонам контейнера), ***Constraints*** (ограничения допустимых изменений размеров). Учтите, что использование в свойстве ***Anchors*** одновременной привязки компонента к противоположным сторонам контейнера приведет к растягиванию или сжатию компонента при изменении пользователем размеров окна. При этом в метках в ряде случаев наблюдаются неприятности, связанные с перемещением надписей (если они, например, выровнены по центру). Чтобы избежать этих неприятностей, надо в обработку события формы **OnResize** вставить операторы, перерисовывающие компоненты методом ***Repaint***, например:

*StaticText1.Repaint;*

Размер меток ***Label*** и ***StaticText*** определяется также свойством ***AutoSize***. Если это свойство установлено в ***true***, то вертикальный и горизонтальный размеры компонента определяются размером надписи. Если же ***AutoSize*** равно ***false***, то выравнивание текста внутри компонента определяется свойством ***Alignment***, которое позволяет выравнивать текст по левому краю, правому краю или центру клиентской области метки. В панели ***Panel*** также имеется свойство ***AutoSize***, но оно не относится к размерам надписи ***Caption***. Однако, свойство выравнивания ***Alignment*** работает и для панели.

В метке ***Label*** имеется свойство ***Wordwrap*** — допустимость переноса слов длинной надписи, превышающей длину компонента, на новую строчку. Чтобы такой перенос мог осуществляться, надо установить свойство ***WordWrap*** в ***true***, свойство ***AutoSize*** в ***false*** (чтобы размер компонента не определялся размером надписи) и сделать высоту компонента такой, чтобы в нем могло поместиться несколько строк. Если ***WordWrap*** не установлено в ***true*** при ***AutoSize*** равном ***false***, то длинный текст, не помещающийся в рамке метки, просто обрезается.

В метке ***StaticText*** перенос длинного текста осуществляется автоматически, если значение ***AutoSize*** установлено в ***false*** и размер компонента достаточен для размещения нескольких строк. Для того, чтобы в ***StaticText*** осуществлялся перенос при изменении пользователем размеров окна, надо осуществлять описанную выше перерисовку компонента методом ***Repaint*** в обработчике события формы ***OnResize***.

В панели размещение надписи в нескольких строках невозможно.

Можно отметить еще одно свойство меток ***Label*** и ***StaticText***, превращающее их в некоторое подобие управляющих элементов. Это свойство ***FocusControl*** — фокусируемый компонент. Если в свойстве метки ***Caption*** поместить перед одним из символов символ амперсант «&», то символ, перед которым поставлен амперсант, отображается в надписи метки подчеркнутым (сам амперсант вообще не отображается). Если после этого обратиться к свойству метки ***FocusControl***, то из выпадающего списка можно выбрать элемент, на который будет переключаться фокус, если пользователь нажмет клавиши ускоренного доступа: клавишу Alt + подчеркнутый символ. Подобные клавиши ускоренного доступа предусмотрены в управляющих элементах: разделах меню и кнопках. Благодаря свойству ***FocusControl*** метки могут обеспечить клавишами ускоренного доступа иные элементы, например, окна редактирования, в которых такие клавиши не предусмотрены. Только для того, чтобы клавиши ускоренного доступа в метках срабатывали, необходимо установить свойство ***ShowAccelChar*** этих меток в ***true***.

Для отображения текстовой информации, и даже с дополнительной возможностью прокрутки длинных текстов, можно использовать также окна редактирования ***Edit*** и ***MaskEdit*** в режиме ***Readonly***.

**Окна редактирования Edit и MaskEdit**

В компонентах ***Edit*** и ***MaskEdit*** вводимый и выводимый текст содержится в свойстве **Text**. Это свойство можно устанавливать в процессе проектирования или задавать программно. Выравнивание текста, как это имело место в метках и панелях, невозможно. Перенос строк тоже невозможен. Текст, не помещающийся по длине в окно, просто сдвигается и пользователь может перемещаться по нему с помощью курсора. Свойство ***AutoSize*** в окнах редактирования имеет смысл, отличный от смысла аналогичного свойства меток: автоматически подстраивается под размер текста только высота, но не ширина окна.

Окна редактирования снабжены многими функциями, свойственными большинству редакторов. Например, в них предусмотрены типичные комбинации «горячих» клавиш: *Ctrl-C* — копирование выделенного текста в буфер обмена *Clipboard* (команда *Copy*), *Ctrl-X* — вырезание выделенного текста в буфер *Clipboard* (команда *Cut*), *Ctrl-V* — вставка текста из буфера *Clipboard* в позицию курсора (команда *Paste*), *Ctrl-Z* — отмена последней команды редактирования.

Свойство ***AutoSelect*** определяет, будет ли автоматически выделяться весь текст при передаче фокуса в окно редактирования. Его имеет смысл задавать равным ***true*** в случаях, когда при переключении в данное окно пользователь будет скорее заменять текущий текст, чем исправлять его. Имеются также свойства только времени выполнения ***SelLength***, ***SelStart***, ***SelText***, определяющие соответственно длину выделенного текста, позицию перед первым символом выделенного текста и сам выделенный текст. Если выделенного текста нет, то свойство **SelStart** просто определяет текущее положение курсора.

Окна редактирования можно использовать и просто как компоненты отображения текста. Для этого надо установить в **true** их свойство ***ReadOnly*** и целесообразно установить ***AutoSelect*** в ***false***. В этом случае пользователь не сможет изменять отображаемый текст и окно редактирования становится подобным меткам. Но имеются и определенные отличия. Во-первых, окна редактирования оформлены несколько иначе. А главное — окна редактирования могут вмещать текст, превышающий их длину. В этом случае пользователь может прокручивать этот текст, перемещая курсор в окне. Такими особенностями не обладает ни одна метка.

При использовании окон редактирования для вывода, ввода и редактирования чисел необходимо использовать функции взаимного преобразования строк и чисел. Для вывода это описанные при рассмотрении меток функции ***FloatToStr*** и ***IntToStr***. При вводе это функции ***StrToFloat*** — преобразование строки в значение с плавающей запятой, и ***StrToInt*** — преобразование строки в целое значение. Если вводимый текст не соответствует числу (например, содержит недопустимые символы), то функции преобразования генерируют исключение ***EConvertError***. Поэтому в программе необходимо предусмотреть обработку этого исключения. Например:

***var*** *A: integer;*

***try***

*A := StrToInt(Edit1.Text);*

*...{операторы, использующие переменную А}*

***except******on*** *EConvertError* ***do***

*ShowMessage('Вы ввели ошибочное число; повторите ввод');*

Этот код обеспечивает сообщение пользователю об ошибке ввода и предотвращает ошибочные вычисления. Впрочем, это не лучший вариант предотвратить ошибочный ввод, поскольку пользователь узнает о своей ошибке только после того, как программа пытается использовать введенные данные. Лучше, если пользователь просто не сможет ввести неправильные символы. Например, если вы хотите, чтобы пользователь мог вводить в окно редактирования ***Edit*** только цифры и символ точки, вы можете в обработчик события ***OnKeyPress*** этого компонента вставить оператор:

***if******not*** *(Key* ***in*** *['0'..'9', ','])* ***then*** *Key := #0;*

Этот оператор подменит все символы, кроме цифр и запятой, нулевым символом, который не занесется в текст окна ***Edit***.

Свойство ***MaxLength*** определяет максимальную длину вводимого текста. Если ***MaxLength* *= 0***, то длина текста не ограничена. В противном случае значение ***MaxLength*** указывает максимальное число символов, которое может ввести пользователь.

Свойство ***Modified***, доступное только во время выполнения, показывает, проводилось ли редактирование текста в окне. Если вы хотите использовать это свойство, то в момент начала работы пользователя с текстом ***Modified*** надо установить в ***false***. Тогда при последующем обращения к этому свойству можно по его значению (***true*** или ***false***) установить, было или не было произведено редактирование.

Свойство ***PasswordChar*** позволяет превращать окно редактирования в окно ввода пароля. По умолчанию значение ***PasswordChar***равно ***#0***— нулевому символу. В этом случае это обычное окно редактирования. Но если в свойстве указать иной символ (например, символ звездочки «**\***»), то при вводе пользователем текста в окне будут появляться именно эти символы, а не те, которые вводит пользователь. Тем самым обеспечивается секретность ввода пароля.

Компонент ***MaskEdit*** отличается от ***Edit*** тем, что в нем можно задать строку маски в свойстве ***EditMask***. Маска состоит из трех разделов, между которыми ставится точка с запятой (;). В первом разделе — шаблоне записываются специальным образом символы, которые можно вводить в каждой позиции, и символы, добавляемые самой маской; во втором разделе записывается 1 или 0 в зависимости от того, надо или нет, чтобы символы, добавляемые маской, включались в свойство ***Text*** компонента; в третьем разделе указывается символ, используемый для обозначения позиций, в которых еще не осуществлен ввод. Прочитать результат ввода можно или в свойстве ***Text***, которое в зависимости от вида второго раздела маски включает или не включает в себя символы маски, или в свойстве ***EditText***, содержащем введенный текст вместе с символами маски.

**Таблица. Символы шаблона маски**

|  |  |
| --- | --- |
| **!** | Наличие символа «!» означает, что в ***EditText*** недостающие символы предваряются пробелами, а отсутствие символа «!» означает, что пробелы размещаются в конце. |
| **>** | Символ «>» означает, что все последующие за ним символы должны вводиться в верхнем регистре, пока не кончится маска или пока не встретится символ «<». |
| **<** | Символ «<» означает, что все последующие за ним символы должны вводиться в нижнем регистре, пока не кончится маска или пока не встретится символ «>». |
| **<>** | Символы «<>» означают, что анализ регистра не производится. |
| **\** | Символ «\» означает, что следующий за ним символ является буквенным, а не специальным, характерным для маски. Например, символ «>» после символа «\» воспримется как знак >, а не как символ, указывающий на верхний регистр. |
| **L** | Символ «L» означает, что в данной позиции должна бытъ буква. |
| **l** | Символ «l» означает, что в данной позиции может быть только буква или ничего. |
| **A** | Символ «А» означает, что в данной позиции должна быть буква или цифра. |
| **a** | Символ «а» означает, что в данной позиции может быть буква, или цифра, или ничего. |
| **C** | Символ «С» означает, что в данной позиции должен быть любой символ. |
| **c** | Символ «с» означает, что в данной позиции может быть любой символ или ничего. |
| **0** | Символ «0» означает, что в данной позиции должна быть цифра. |
| **9** | Символ «9» означает, что в данной позиции может быть цифра или ничего. |
| **#** | Символ «#» означает, что в данной позиции может быть цифра, знак «+», знак «-» или ничего. |
| **:** | Символ «:» используется для разделения часов, минут и секунд. |
| **/** | Символ «/» используется для разделения месяцев, дней и годов в датах. |
|  | Символ « » означает автоматическую вставку в текст пробела. |

Вводить маску можно непосредственно в свойство ***EditMask***. Но удобнее пользоваться специальным редактором масок, вызываемым при нажатии кнопки с многоточием в строке свойства ***EditMask*** в *Инспекторе Объектов*.

В редакторе масок окно *Sample Masks* содержит наименования стандартных масок и примеры ввода с их помощью. В окно *Input Mask* надо ввести маску. Если вы выбираете одну из стандартных масок, то окно *Input Mask* автоматически заполняется и вы можете, если хотите, отредактировать эту маску.

Окно *Character for Blanks* определяет символ, используемый для обозначения позиций, в которых еще не осуществлен ввод (третий раздел маски). Индикатор *Save Literal Characters* определяет второй раздел маски: установлен, если второй раздел равен 1, и не установлен, если второй раздел равен 0.

Кнопка *Masks* позволяет выбрать и загрузить какой-либо другой файл стандартных масок. К сожалению, среди файлов стандартных масок, поставляемых с Delphi, отсутствует маска, соответствующая российским стандартам. Но вы легко можете сами сделать себе такой файл стандартных масок. Он делается в обычном текстовом редакторе и должен сохраняться как «только текст» с расширением ***.dem***. Чтобы редактор масок Delphi видел этот файл, его надо сохранить в каталоге *Delphi\****BIN***. Каждая строка файла состоит из трех частей, разделяемых символом вертикальной черты. Первая часть состоит из пояснительного текста, появляющегося в левой панели окна *Sample Masks* редактора масок. Вторая часть — пример, который появляется в правой панели окна *Sample Masks редактора масок*. А третья часть — сама маска.

Поскольку второй раздел маски равен 0, то при чтении введенных пользователем значений свойства ***EditText*** и ***Text*** будут различаться. Свойство ***EditText*** для примера будет равно **(095) 123-45-67**, а свойство ***Text*** будет равно **0951234567**. Если второй раздел маски сделать равным 1, то значения обоих свойств будут равны **(095) 123-45-67**.

Если с помощью ***EditMask*** надо ввести, например, целое число без знака, состоящее не более, чем из двух цифр, можно задать маску **99;0;** . Если число обязательно должно быть двузначным, то маска должна иметь вид **00;0;**

**Многострочные окна редактирования Memo и RichEdit**

Компоненты ***Memo*** и ***RichEdit*** являются окнами редактирования многострочного текста. Они так же, как и окно ***Edit***, снабжены многими функциями, свойственными большинству редакторов. В них предусмотрены типичные комбинации «горячих» клавиш: *Ctrl-C* — копирование выделенного текста в буфер обмена *Clipboard* (команда *Copy*), *Ctrl-X* — вырезание выделенного текста в буфер *Clipboard* (команда *Cut*), *Ctrl-V* — вставка текста из буфера *Clipboard* в позицию курсора (команда *Paste*), *Ctrl-Z* — отмена последней команды редактирования.

В компоненте ***Memo*** формат (шрифт, его атрибуты, выравнивание) одинаков для всего текста и определяется свойством ***Font***. Если вы сохраните в файле текст, введенный или отредактированный пользователем, то будет создан текстовый файл, содержащий только символы и не содержащий элементов форматирования. При последующем чтении этого файла в ***Memo*** формат будет определяться текущим состоянием свойства ***Font*** компонента ***Memo***, а не тем, в каком формате ранее вводился текст.

Компонент ***RichEdit*** работает с текстом в обогащенном формате *RTF.* При желании изменить атрибуты вновь вводимого фрагмента текста вы можете задать свойство ***SelAttributes***. Это свойство типа ***TTextAttributes***, которое в свою очередь имеет подсвойства: ***Color*** (цвет), ***Name*** (имя шрифта), ***Size*** (размер), ***Style*** (стиль) и ряд других. Например, введите на форму компонент ***RichEdit***, диалог выбора шрифта ***FontDialog*** и кнопку ***Button***, которая позволит пользователю менять атрибуты текста. В обработчик щелчка кнопки можно было бы ввести текст:

***if*** *FontDialog1.Execute* ***then***

***with*** *RichEdit1.SelAttributes* ***do***

***begin***

*Color:=FontDialog1.Font.Color;*

*Name:=FontDialog1.Font.Name;*

*Size:=FontDialog1.Font.Size;*

*Style:=FontDialog1.Font.Style;*

***end****;*

*RichEdit1.SetFocus;*

В приведенном коде присваивается поочередно значение каждого свойства. Но этот текст можно кардинально сократить, воспользовавшись тем, что объекты ***SelAttributes*** и ***Font*** совместимы по типу. Поэтому можно присвоить сразу все свойства одного объекта другому:

***if*** FontDialog1.Execute ***then***

RichEdit1.SelAttributes.Assign(FontDialog1.Font);

RichEdit1.SetFocus;

Запустите приложение и увидите, что вы можете менять атрибуты текста, выполняя отдельные фрагменты различными шрифтами, размерами, цветами, стилями. Устанавливаемые атрибуты влияют на выделенный текст или, если ничего не выделено, то на атрибуты нового текста, вводимого начиная с текущей позиции курсора (позиция курсора определяется свойством ***SelStart***).

В компоненте имеется также свойство ***DefAttributes***, содержащее атрибуты по умолчанию. Эти атрибуты действуют до того момента, когда изменяются атрибуты в свойстве ***SelAttributes***.

Но значения атрибутов в ***DefAttributes***сохраняются и в любой момент эти значения могут быть методом ***Assign*** присвоены атрибутам свойства ***SelAttributes***, чтобы вернуться к прежнему стилю.

Свойство ***DefAttributes***доступно только во время выполнения. Поэтому его атрибуты при необходимости можно задавать, например, в обработчике события ***OnCreate***.

За выравнивание, отступы и т.д. в пределах текущего абзаца отвечает свойство ***Paragraph*** типа ***TParaAttributes***. Этот тип в свою очередь имеет ряд свойств:

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

Значения подсвойств свойства ***Paragraph*** можно задавать только в процессе выполнения приложения, например, в событии создания формы или при нажатии какой-нибудь кнопки. Значения подсвойств свойства ***Paragraph*** относятся к тому абзацу, в котором находится курсор. Например, каждый из следующих операторов осуществит соответственное выравнивание текущего абзаца:

*RichEdit1.Paragraph.Alignment:=taLeftJustify; // Влево*

*RichEdit1.Paragraph.Alignment:=taCenter; // По центру*

*RichEdit1.Paragraph.Alignment:=taRightJustify;// Вправо*

Следующий оператор приведет к тому, что текущий абзац будет отображаться как список, т.е. с маркерами:

*RichEdit1.Paragraph.Numbering:=nsBullet*;

Уничтожение списка в текущем абзаце осуществляется оператором

*RichEdit1.Paragraph.Numbering:=nsNone;*

В целом, если с помощью компонента ***ActionList*** определено некоторое действие ввода и уничтожения списка, названное ***ABullet***, то операторы обработки соответствующего действия могут иметь вид:

***if*** *(ABullet.Checked)* ***then***

*RichEdit1.Paragraph.Numbering:=nsNone*

***else*** *RichEdit1.Paragraph.Numbering :=nsBullet;*

*ABullet.Checked:=****not*** *ABullet.Checked;*

Они обеспечивают переключение соответствующей быстрой кнопки и раздела меню из нажатого состояния (отмеченного) в ненажатое с соответствующим изменением вида текущего абзаца.

Свойства ***TabCount*** и ***Tab*** имеют смысл при вводе текста только при значении свойства компонента ***WantTabs = true***. Это свойство разрешает пользователю вводить в текст символ табуляции. Если ***WantTabs* *=* *false***, то нажатие пользователем клавиши табуляции просто переключит фокус на очередной компонент и символ табуляции в текст не введется.

Мы рассмотрели основные отличия ***Memo*** и ***RichEdit***. Теперь остановимся на общих свойствах этих окон редактирования.

Свойства ***Alignment*** и ***WordWrap*** имеют тот же смысл, что, например, в метках, и определяют выравнивание текста и допустимость переноса длинных строк. Установка свойства ***ReadOnly*** в ***true*** задает текст только для чтения. Свойство ***MaxLength*** определяет максимальную длину вводимого текста. Если ***MaxLength = 0***, то длина текста не ограничена. Свойства ***WantReturns*** и ***WantTab*** определяют допустимость ввода пользователем в текст символов перевода строки и табуляции.

Свойство ***ScrollBars*** определяет наличие полос прокрутка текста в окне. По умолчанию ***ScrollBars* *=* *ssNone***, что означает их отсутствие. Пользователь может в этом случае перемещаться по тексту только с помощью курсора. Можно задать свойству ***ScrollBars*** значения ***ssHorizontal***, ***ssVertical*** или ***ssBoth***, что будет соответственно означать наличие горизонтальной, вертикальной или обеих полос прокрутки.

Основное свойство окон ***Memo*** и ***RichEdit*** — ***Lines***, содержащее текст окна в виде списка строк и имеющее тип ***TStrings***. Начальное значение текста можно установить в процессе проектирования, нажав кнопку с многоточием около свойства ***Lines*** в окне *Инспектора Объектов.* Перед вами откроется окно редактирования списков строк. Вы можете редактировать или вводить текст непосредственно в этом окне, или нажать кнопку *CodeEditor* и работать в обычном окне *Редактора Кода*. В этом случае, завершив работу с текстом, выберите из контекстного меню, всплывающего при щелчке правой кнопкой мыши, команду *Close Page* и ответьте утвердительно на вопрос, хотите ли вы сохранить текст в соответствующем свойстве окна редактирования.

Во время выполнения приложения вы можете заносить текст в окно редактирования с помощью методов свойства ***Lines*** типа ***TStrings***. Этот тип широко используется в свойствах многих компонентов и его описание вы можете найти в во встроенной справке Delphi. Здесь коротко укажем только на его основные свойства и методы, используемые в свойстве ***Lines***.

Весь текст, представленный одной строкой типа ***String***, внутри которой используются разделители типа символов возврата каретки и перевода строки, содержится в свойстве ***Text***.

Доступ к отдельной строке текста вы можете получить с помощью свойства ***Strings[Index: Integer]***. Индексы, как и везде в Delphi, начинаются с **0**. Так что ***Memo1.Lines.Strings[0]*** — это текст первой строки. Учтите, что если окно редактирования изменяется в размерах при работе с приложением и свойство ***WordWrap = true***, то индексы строк будут изменяться при переносах строк, так что в этих случаях индекс мало о чем говорит.

Свойство только для чтения ***Count***указывает число строк в тексте.

Для очистки текста в окне надо выполнить процедуру ***Clear***. Этот метод относится к самому окну, а не к его свойству ***Lines***.

Для занесения новой строки в конец текста окна редактирования можно воспользоваться методами ***Add***или ***Append***свойства ***Lines***. Для загрузки текста из файла применяется метод ***LoadFromFile***. Сохранение текста в фале осуществляется методом ***SaveToFile***.

Пусть, например, в вашем приложении имеется окно редактирования ***Edit1***, в котором пользователь вводит имя сотрудника, и есть кнопка, при щелчке на которой в окно ***Memo1*** должна занестись шапка характеристики этого сотрудника, после чего пользователь может заполнить текст характеристики.

Обработчик щелчка на кнопке может иметь вид:

*Memo1.Clear;*

*Memo1.Lines.Add ('Х А Р А К Т Е Р И С Т И К А');*

*Memo1.Lines.Add('Сотрудник '+Edit1.Text) ;.*

*Memo1.SetFocus;*

Загрузка в окно ***Memo1*** текста из файла (например, хранящейся в файле характеристики сотрудника) может осуществляться командой

*Memo1.Lines.LoadFromFile ('text.txt');*

Сохранение текста в файле может осуществляться командой

*Memo1.Lines.SaveToFile('text.txt');*

Свойство ***SelStart*** компонентов ***Memo*** и ***RichEdit*** указывает позицию курсора в тексте или начало выделенного пользователем текста. Свойство ***CaretPos*** указывает на запись, поле ***X*** которой содержит индекс символа в строке, перед которым расположен курсор, а поле ***Y*** — индекс строки, в которой находится курсор (встроенная справка Delphi утверждает другое — что свойство ***CaretPos*** содержит координаты курсора в пикселях; но, к счастью, это не так). Таким образом, учитывая, что индексы начинаются с 0, значения ***Memo1.CaretPos.Y+1*** *и* ***Memо1.CaretPos.X+1*** определяют соответственно номер строки и символа в ней, перед которым расположен курсор.

**Ввод и отображение целых чисел — компонент SpinEdit**

В Delphi имеются специализированные компоненты, обеспечивающие ввод целых чисел, один из них — ***SpinEdit***. Компонент ***SpinEdit*** представляет собой сочетание ***Edit*** и ***UpDown***, оформленное как отдельный тип компонента.

Свойство **Thousands** определяет наличие или отсутствие разделительного пробела между каждыми тремя цифрами разрядов вводимого числа.

Свойства ***MinValue*** и ***MaxValue*** компонента задают соответственно минимальное и максимальное значения чисел, свойство ***Increment*** задает приращение числа при каждом нажатии на кнопку. Свойство ***Value*** определяет текущее значение числа.

**35.Загальні події елементів управління та їх характеристики.**

К основным элементам управления относят следующие компоненты:

* кнопки;
* флажки;
* переключатели;

|  |  |
| --- | --- |
| **Подія** | **Опис** |
| onMouseDown | виникає при натисканні кнопки миші (не обов’язково лівої).  Параметр **Button** – натиснута кнопка (mbLeft - ліва кнопка, mbRight - права, mbMiddle - середня). **Shift** - множина, що вказує на натиснуті функціональні клавіші. Таким чином можна відслідкувати, наприклад, натискання при затиснутих Ctrl, Alt і т.ін., **X**, **Y** - координати курсору відносно лівого верхнього кута самого компоненту |
| onMouseUp | виникає при відпусканні натиснутої кнопки миші.  Наприклад, графічний редактор: коли кнопка миші натиснута — виконується рисування, а коли відпущена — не виконується |
| onMouseMove | виникає при наведенні курсору миші на елемент.  **X**, **Y** - координати нового положення, **Shift** – множина натиснутих функціональних клавіш |
| onClick | виникає при одинарному клацанні клавішею миші.  У кнопки подія також виникає якщо натиснути Enter або Пробіл в той момент, коли фокус вводу знаходиться на кнопці |
| onDblClick | виникає при подвійному клацанні клавішею миші. |
| OnKeyDown | виникає при натисненні клавіші на клавіатурі.  **Кey** - код натиснутої клавіші, **Shift** - множина натиснутих функціональних клавіш |
| OnKeyUp | виникає при відпусканні натиснутої клавіші на клавіатурі. |
| OnKeyPress | виникає при натисканні клавіші, яка друкує деякий символ.  **Key** - вже не код клавіші, а сам символ |
| OnResize | виникає при зміні розмірів об’єкту. |
| OnPaint | виникає при прорисовуванні об’єкту на екран. |
| OnEnter | виникає в момент, коли об’єкт отримує фокус.  Фокус вводу зазвичай відображається пунктирною рамкою на об’єкті; у полі вводу — це курсор; фокус один на все застосування, тобто працювати одночасно можна лише з одним об’єктом |
| OnExit | виникає в момент, коли об’єкт втрачає фокус вводу. |
| OnClose | виникає при закритті форми. |
| OnCloseQuery | виникає при спробі закрити форму.**CanClose** — логічне значення, тип даних — Boolean. Якщо після виконання обробника значення змінної CanClose виявиться False, то закриття форми виконано не буде. |
| OnDestroy | виникає при знищенні форми.  Після закриття форма знищується і звільняється зайнята нею пам'ять |
| OnShow | виникає при показі форми на екран. |
| OnHide | виникає при приховуванні форми з екрану |
| OnActivate | виникає при в момент, коли форма становиться активною, тобто колі їй передається фокус вводу.  Наприклад, коли користувач клацнув по форму |
| OnDeactivate | виникає при в момент, коли форма втрачає фокус |
| OnChange | виникає при в момент зміні чогось  Наприклад, у поля введення TEdit ця подія спрацьовує коли змінюється текст у цьому полі |

**36. Загальні властивості елементів управління та їх характеристики.**

К основным элементам управления относят следующие компоненты:

* кнопки;
* флажки;
* переключатели;

**Кнопки**

Кнопки являются представителями класса TButton. Они являются одними из наиболее распространенных элементов управления Windows.

Основные свойства класса TButton отличающие его от предков:

* **Caption**: TCaption – строка текста на кнопке;
* **Cancel**: Boolean – управляет синхронизацией экранной кнопки, с физической кнопкой Esc на клавиатуре. На форме может быть только одна кнопка со свойством Cancel, равным true.
* **Default**: Boolean – аналог предыдущего свойства, но синхронизирует с клавишей Enter.
* **ModalResult**: TModalResult – при щелчке на кнопке значение свойства ModalResult родительской формы устанавливается равным свойству ModalResult кнопки. Используется при создании окон диалога
* **Font**: TFont        –  параметры шрифта надписи кнопки

**Флажки**

Флажки являются представителями класса TCheckBox и служат для управления состоянием чего-либо. Флажки могут иметь два состояния: включено или выключено, но допускается и третье (неопределенное) — флажок на сером фоне.  Третье состояние флажков можно наблюдать, например, в программе установки Windows при выборе устанавливаемых компонентов.

Основные свойства флажков:

* **Caption**: TCaption – поясняющая надпись;
* **Checked**: Boolean – показывает, установлен флажок (true), нет или третье состояние (false);
* **State**: TCheckBoxState= (cbUnchecked. cbChecked. cbGrayed) – определяет состояние флажка:

ü cbChecked – установлен;

ü cbUnchecked – неустановлен;

ü cbGrayed – установлен и закрашен серым.

* **AllowGrayed**: Boolean – определяет, может (true) или нет (false) флажок иметь третье состояние.

Рассмотрим в качестве примера использования флажка управления реакцией на щелчок по кнопке.

**Переключатели**

Можно выделить два класса переключателей TRadioButton и TRadioGroup. Класс TRadioButton представляет отдельные переключатели, которые группируются по признаку принадлежности к тому или иному контейнеру. Все переключатели, помещаемые в один контейнер (форма, фрейм и т. п.), считаются входящими в одну группу, в которой может быть установлен только один переключатель. Класс TRadioGroup представляет собой готовую группу переключателей.

**Класс TRadioButton** во многом аналогичен флажку. Основные свойства класса:

* **Caption**: TCaption – поясняющая надпись;
* **Checked**: Boolean – определяет состояние переключателя.

Событие **OnClick** для компонента TRadioButton имеет одну особенность: оно вызывается только при установке переключателя. Если переключатель уже установлен, то щелчок на нем левой кнопки мыши не вызовет события OnClick.

Рассмотрим пример, подобный предыдущему, где для управления реакцией на щелчок на кнопке будем использовать переключатели.

**37.Призначення і характеристика компонентів вкладки Standard**

Закладка Standard містить ряд часто використовуваних компонентів загального призначення



|  |  |  |
| --- | --- | --- |
| **Компонент** | **Тип** | **Опис** |
| фрейм | **Frame** | Панель з можливостями спадкоємства. Проектується як окреме вікно. Компонент візуальний. |
| головне меню | **MainMenu** | Дозволяє конструювати і створювати головне меню форми і випадаюче меню. Компонент невізуальний. |
| контекстне меню | **PopupMenu** | Дозволяє конструювати і створювати спливаюче контекстне меню. Компонент невізуальний. |
| мітка | **Label** | Використовується для розміщення на формах та інших контейнерах тексту, який не змінюється користувачем. Компонент візуальний. |
| вікно редагування | **Edit** | Використовується для введення користувачем однорядковий текстів. Може використовуватися для відображення тексту. Компонент візуальний. |
| багаторядкове вікно редагування | **Memo** | Використовується для введення та відображення багаторядкових текстів. Компонент візуальний. |
| командна кнопка | **Button** | Використовується для створення кнопок, якими користувач вибирає команди у додатку. Компонент візуальний. |
| контрольний  індикатор з прапорцем | **Checkbox** | Дозволяє користувачеві вмикати та вимикати різні опції. Компонент візуальний. |
| радіокнопка | **RadioButton** | Пропонують користувачеві набір альтернатив, з яких вибирається одна. Набір реалізується необхідною кількістю радіокнопок, розміщених в одному контейнері (формі, панелі і т.п.). Компонент візуальний. |
| вікно списку | **ListBox** | Являє собою стандартне вікно списку Windows, що дозволяє користувачеві вибирати розділи зі списку. Компонент візуальний. |
| редагований список | **ComboBox** | Об'єднує функції ListBox і Edit. Користувач може або ввести текст, або вибрати його зі списку. Компонент візуальний. |
| лінійка прокрутки | **Scrollbar** | Являє собою стандартну лінійку прокрутки Windows і служить для керування положенням видимої частини форм або компонентів. Компонент візуальний. |
| групове вікно | **GroupBox** | Є контейнером, що поєднує групу пов'язаних елементів управління, таких як радіокнопки RadioButton, контрольні індикатори Checkbox і т.д. Компонент візуальний. |
| група радіокнопок | **RadioGroup** | Є комбінацією групового вікна GroupBox з набором радіокнопок RadioButton; служить спеціально для створення груп радіокнопок. Можна розміщувати в компоненті кілька радіокнопок, але ніякі інші елементи управління не дозволені. Компонент візуальний. |
| панель | **Panel** | Є контейнером для групування елементів управління і менших контейнерів. Панель можна використовувати також для побудови смуг стану, інструментальних панелей, палітр інструментів. Компонент візуальний. |
| список подій | **ActionList** | Забезпечує диспетчеризацію подій компонентів. Компонент невізуальний. |

**38. Призначення і характеристика компонентів вкладки Additional**

Закладка є доповненням сторінки Standard і містить ряд часто використовуваних компонентів загального призначення.



|  |  |  |
| --- | --- | --- |
| **Компонент** | **Тип** | **Описание** |
| кнопка с графикой | **BitBtn** | Используется для создания кнопок, на которых располагается битовая графика (например, кнопка OK с галочкой). Компонент визуальный. |
| кнопка с фиксацией | **SpeedButton** | Используется для создания инструментальных панелей и в других случаях, когда требуется кнопка с фиксацией нажатого состояния. Компонент визуальный. |
| маскированный ввод | **MaskEdit** | Используется для форматирования данных или для ввода символов в соответствии с шаблоном. Компонент визуальный. |
| таблица строк | **StringGrid** | Используется для отображения текстовой информации в таблице из строк и столбцов. Компонент визуальный. |
| таблица рисунков | **DrawGrid** | Используется для отображения в строках и столбцах нетекстовых данных. Компонент визуальный. |
| изображение | **Image** | Используется для отображения графики: пиктограмм, битовых матриц и метафайлов. Компонент визуальный. |
| формы | **Shape** | Используется для рисования фигур: квадратов, кругов и т.п. Компонент визуальный. |
| рамка | **Bevel** | Используется для рисования выступающих или утопленных линий или прямоугольных рамок. Компонент визуальный. |
| окно с прокруткой | **ScrollBox** | Используется для создания зон отображения с прокруткой. Компонент визуальный. |
| список с флажками | **CheckListBox** | Компонент является комбинацией свойств списка ListBox и индикаторов CheckBox в одном компоненте. Компонент визуальный. |
| разделитель панелей | **Splitter** | Используется для создания в приложении панелей с изменяемыми пользователем размерами. Компонент визуальный. |
| метка с бордюром | **StaticText** | Компонент подобен компоненту Label, но обеспечивает дополнительные возможности по заданию стиля бордюра. Компонент визуальный. |
| инструментальная панель | **ControlBar** | Используется для размещения компонентов инструментальной панели. Компонент визуальный. |
| события приложения | **ApplicationEvents** | Перехватывает события на уровне приложения. Компонент невизуальный. |
| диаграммы и графики | **Chart** | Компонент принадлежит к семейству компонентов TChart, которые используются для создания диаграмм и графиков. Компонент визуальный. |

**39. Призначення і характеристика компонентів вкладки Win32**

Сторінка Win32 містить компоненти загального призначення, які дозволяють розробляти додатки в стилі Windows 95/98 і NT 4.x. Деякі з цих компонентів аналогічні наявним на сторінці Win3.1.



|  |  |  |
| --- | --- | --- |
| **Компонент** | **Тип** | **Описание** |
| страница с закладкой | **TabControl** | Позволяет организовывать страницы с закладками в стиле Windows 95, которые может выбирать пользователь. Компонент визуальный. |
| многостраничное окно | **PageControl** | Позволяет создавать страницы в стиле Windows 95/98, управляемые закладками или иными органами управления, для экономии места на рабочем столе. Компонент визуальный. |
| список изображений | **ImageList** | Предназначен для работы со списками изображений одинакового размера в меню, инструментальных панелях и т.п. Компонент невизуальный. |
| окно редактирования в формате RTF | **RichEdit** | Представляет собой окно редактирования в стиле Windows 95/98, позволяющее производить выбор цвета и шрифта, поиск текста и многое другое. Компонент визуальный. |
| ползунок | **TrackBar** | Управляющий элемент в виде ползунка в стиле Windows 95/98. Компонент визуальный. |
| отображение хода процесса | **ProgressBar** | Используется для отображения в стиле Windows 95/98 хода процессов, занимающих заметное время. Компонент визуальный. |
| кнопка-счетчик | **UpDown** | Кнопка-счетчик в стиле Windows 95/98 для ввода целых чисел. Компонент визуальный. |
| «горячие» клавиши | **HotKey** | Дает возможность реализовать в приложении поддержку горячих клавиш. Компонент визуальный. |
| воспроизведение немых клипов | **Animate** | Используется для воспроизведения немых клипов AVI, подобных используемым в Windows 95/98 изображениям копирования файлов и т.п. Компонент визуальный. |
| ввод дат и времени | **DateTimePicker** | Ввод дат и времени с выпадающим календарем. Компонент визуальный. |
| ввод дат | **MonthCalendar** | Ввод дат с выбором из календаря. Компонент визуальный. |
| дерево | **TreeView** | Предоставляет возможность просмотра структуры иерархических данных в стиле Windows 95/98. Компонент визуальный. |
| списки | **ListView** | Отображает списки в стиле Windows 95/98. Компонент визуальный. |
| заголовок | **HeaderControl** | Позволяет создавать составные перемещаемые заголовки в стиле Windows 95/98. Компонент визуальный. |
| полоса состояния | **StatusBar** | Полоса состояния программы, при необходимости — на нескольких панелях. Компонент визуальный. |
| инструментальная панель | **ToolBar** | Инструментальная панель для быстрого доступа к часто используемым функциям приложения. Компонент визуальный. |
| инструментальная перестраиваемая панель | **CoolBar** | Контейнер инструментальной панели, размеры которой могут изменяться пользователем. Компонент визуальный. |
| прокрутка страниц | **PageScroller** | Обеспечивает прокрутку больших окон, например, инструментальных панелей. Компонент визуальный. |

**40. Призначення і характеристика компонентів вкладки System**

Сторінка System містить компоненти, що дозволяють використовувати системні засоби Windows.



|  |  |  |
| --- | --- | --- |
| **Компонент** | **Тип** | **Описание** |
| таймер | **Timer** | Используется для запуска процедур, функций и событий в указанные интервалы времени. Компонент невизуальный. |
| окно для рисования | **PaintBox** | Используется для создания на форме некоторой области, в которой можно рисовать. Компонент визуальный. |
| аудио и видео плеер | **MediaPlayer** | Используется для создания панели управления воспроизведением звуковых и видео файлов, а также устройств мультимедиа. Компонент визуальный. |
| контейнер OLE | **OLEContainer** | Используется при создании области клиента для объекта OLE. Компонент визуальный. |
| диалог с сервером DDE | **DDEClientConv** | Используется клиентом DDE для организации диалога с сервером DDE. Компонент невизуальный. |
| данные, передаваемые серверу DDE | **DDEClientItem** | Используется для определения данных клиента, передаваемых в диалоге серверу DDE. Компонент невизуальный. |
| диалог с клиентом DDE | **DDEServerConv** | Компонент используется сервером DDE при проведении диалога с клиентом DDE. Компонент невизуальный. |
| данные, передаваемые клиенту DDE | **DDEServerItem** | Компонент используется для определения данных сервера, передаваемых клиенту DDE в течение диалога. Компонент невизуальный. |

**41. Критерії якості графічного інтерфейсу користувача**

* Скорость работы пользователя
* Количество человеческих ошибок
* Скорость обучения
* Соответствие интерфейса задачам пользователя (субъективное удовлетворение)

Скорость работы пользователя

Длительность выполнения работы пользователем состоит из длительности восприятия исходной информации, длительности интеллектуальной работы (в смысле – пользователь думает, что он должен сделать), длительности физических действий пользователя и длительности реакции системы.

Необходимо максимально облегчать возвращение пользователей к работе и проектировать интерфейс так, чтобы пользователи возможно меньше о нем думали.

Открывайте новые диалоговые окна не в центре экрана, а в центре текущего действия пользователя (если они не будут перекрывать важную информацию на экране, разумеется)

Убирайте с экрана все диалоги с вопросами, на которые в течение пяти минут не был дан ответ

Количество человеческих ошибок

Наибольшее количество человеческих ошибок при пользовании ПО раскладывается на четыре типа:

* Ошибки, вызванные недостаточным знанием предметной области.
* Опечатки.
* Несчитывание показаний системы.
* Моторные ошибки.

Нужно направлять усилия на

\* плавное обучение пользователей *в процессе* работы.

\* снижение требований к бдительности.

\* повышение разборчивости и заметности индикаторов.

\*снижение чувствительности системы к ошибкам.

Это можна сделать 3 способами:

\* блокировка потенциально опасных действий пользователя до получения подтверждения правильности действия.

\* проверка системой всех действий пользователя перед их принятием.

\* самостоятельный выбор системой необходимых команд или параметров, при котором от пользователя требуется только проверка.

Не делайте опасные для пользователя кнопки кнопками по умолчанию

Всегда показывайте границы диапазона во всплывающей подсказке

Наличие человеческих ошибок, которых нельзя обнаружить и исправить до окончательного совершения действия, всегда свидетельствует о недостаточно хорошем дизайне

Скорость обучения

Обучение есть действие: если обучаться легко, пользователям будет достаточно слабого стимула, если тяжело, стимул придется увеличивать.

В случаях компьютерных систем стимул есть вещь почти исключительно добровольная. Это значит, что пользователь обучится пользоваться программой или сайтом только в том случае, если он будет уверен, что это сделает его жизнь легче и приятней.

Пользователь будет учиться какой-либо функции, только если он знает о еѐ существовании, поскольку, не обладая этим знанием, он не способен узнать, что за еѐ использование жизнь даст ему награду. т.е. одного стимула недостаточно, если пользователь не знает, за что этот стимул дается.

Поэтому разумно привести более совершенный список средств обучения:

\* общая «понятность» системы

\* обучающие материалы.

Рассчитывайте на средних пользователей, а не новичков или на профессионалов: средних пользователей, как-никак, абсолютное большинство

Избегайте создавать элементы управления, функциональность которых зависит от контекста

Последовательность в реализации интерфейса есть первое условие качества результата Система должна индицировать все свои состояния

Соответствие интерфейса задачам пользователя (субъективное удовлетворение)

Исследования показали, что пользователи воспринимают одинаково положительно как убогие, но приятные интерфейсы, так и простые, эффективные, но сухие и скучные.

Но значительно легче и приятнее пользоваться эстетически привлекательными объектами.

\*Избегайте развязности в визуале. Лучше быть поскромнее.

\*Избегайте ярких цветов.

\*Избегайте острых углов в визуале.

\*Старайтесь сделать визуал максимально более легким и воздушным.

\*Старайтесь добиваться контраста не сменой насыщенности элементов, но расположением пустот.

\*Старайтесь сделать интерфейс максимально насыщенным визуальными закономерностями. Под закономерностью я понимаю любое методически выдерживаемое соответствие свойств у разных объектов, например, высота кнопок может быть равна удвоенному значению полей диалогового окна.

\*Всемерно старайтесь использовать модульные сетки, т.е. привязывайте все объекты к линиям воображаемой сетки, которую выдерживайте во всем интерфейсе.

\*Старайтесь привязывать все размеры и координаты (как минимум пропорции диалоговых окон) к золотому сечению (0.618 х 0.382).

Большинство сообщений об ошибках в действительности не являются собственно сообщениями об ошибках. На самом деле они показывают пользователю, что система, которой он пользуется:

\* недостаточно гибка, чтобы приспособиться к его действиям.

\* недостаточно умна, чтобы показать ему его возможные границы его действия.

\* самоуверенна и считает, что пользователь дурак, которым можно и нужно помыкать.

# **42.**  Интегрированная среда разработки приложений Delphi, ее возможности. Основные элементы среды: палитра компонентов, инспектор объектов, форма с редактором кода. Структура программы: файлы проекта, модулей, ресурсов, настройки. Создание Windows-приложений. Пример

Создание прикладных программ, или приложений, в Delphi выполняется в интегрированной среде разработки IDE (Integrated Development Environment). IDE служит для организации взаимодействия с программистом и включает в себя ряд окон, содержащих различные элементы управления. С помощью средств интегрированной среды разработчик может проектировать интерфейсную часть приложения, а также писать программный код, и связывать его с управляющими элементами. При этом вся работа по созданию приложения, включая отладку, происходит в IDE.   
Главные составные части среды программирования  
Интегрированная среда программирования Delphi состоит из следующих основных частей:

1. Главное окно
2. Дизайнер форм (Form Designer)
3. Редактор исходного текста (Editor Window)
4. Палитра компонентов (Component Palette)
5. Инспектор объектов (Object Inspector)

В главном [окне](http://www.fabrikaokon.ru/" \t "http://daxnow.narod.ru/index/_blank) отображается Главное меню, Панель инструментов и Палитра компонентов (см. рисунок). Главное меню содержит обширный набор команд для доступа к функциям Delphi. Панель инструментов находится под Главным меню в левой части Главного [окна](http://www.fabrikaokon.ru/" \t "http://daxnow.narod.ru/index/_blank) и содержит 15 кнопок быстрого доступа для вызова часто используемых команд Главного меню. Палитра компонентов находится справа от Панели инструментов и содержит множество компонентов, размещаемых в создаваемых приложениях. Все компоненты делятся на группы, каждая из которых в Палитре компонентов располагается на отдельной странице (Standard, Additional, Win 32 и др.). 

Рисунок - Приложение Delphi.

Дизайнер форм в Delphi служит для создания визуального интерфейса программы (рис. 9). Дизайнер Форм первоначально состоит из одного пустого [окна](http://www.fabrikaokon.ru/" \t "http://daxnow.narod.ru/index/_blank), которое заполняется компонентами, выбранными в Палитре компонентов.   
Редактор исходного текста (Редактор кода) представляет собой текстовый редактор, с помощью которого можно создавать и редактировать программный код (рис. 9).   
Инспектор объектов состоит из двух страниц, каждую из которых можно использовать для определения поведения данного компонента (рис. 9). Первая страница – это список свойств, вторая – список событий. Если нужно изменить что-нибудь, связанное с определенным компонентом, то обычно делается это в Инспекторе объектов. Страница событий связана с Редактором кода. Если дважды щелкнуть мышкой на правую сторону какого-либо пункта, то соответствующий данному событию код автоматически запишется в Редактор. Сам Редактор немедленно получит фокус, и появится возможность добавить код обработчика данного события.

**Организация проекта, основные файлы проекта**   
Проект в Delphi состоит из форм, модулей, установок параметров проекта, ресурсов и т.д. Вся эта информация размещена в файлах: файл проекта (.dpr); файл модуля (.pas); описание формы (.dfm); описание ресурсов (.res); параметры проекта (.dof); параметры среды (.cfg); исполняемый файл (.exe).  
Типичная головная программа приложения имеет вид:  
program Project1;  
uses  
Forms,  
Unit1 in 'Unit1.pas' {Form1};  
{$R \*.res}  
begin  
Application.Initialize;  
Application.CreateForm(TForm1, Form1);  
Application.Run;  
end.  
Текст модуля имеет вид.  
unit Unit1;  
interface   
{Список подключаемых модулей}  
uses  
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,  
Dialogs;  
{Объявление класса формы}  
type  
TForm1 = class(TForm)  
private  
{ Private declarations }  
public  
{ Public declarations }  
end;  
var   
Form1: TForm1;  
{Сюда могут включаться объявления типов, констант, переменных, функций и процедур, к которым будет доступ из других модулей}  
implementation   
{$R \*.dfm}  
{Сюда могут помещаться предложения **uses**, объявления типов, констант, переменных, к которым не будет доступа из других модулей. Тут же должны быть реализации всех объявленных в разделе **interface** функций и процедур, а также могут быть реализации любых дополнительных, не объявленных ранее функций и процедур.}  
end.

## **43.** Элементы управления ActiveX

|  |
| --- |
| Теперь [займемся](http://moneyman.ru/" \t "http://www.helloworld.ru/texts/comp/lang/delphi/delphi4/_blank) увлекательным делом — создадим элементы управления ActiveX. Рассмотрим, что собой представляв ActiveX и чем элементы управления ActiveX лучше или хуже стандартных компонентов Delphi на базе библиотек VCL.      Технология AcliveX компании Microsoft представляет собой технологию ОСХ, переделанную для обеспечения пepедачи элементов управления через службы Internet, в основном через World Wide Web. Я уверен, что читатель спросит "Это все хорошо, но как технология ActiveX может помочь мне?". Но неужели вам не хочется создать элементы управления, которые можно использовать не только в Delphi, но и в таких средах программирования, как Borland C++ Builde Visual C++ и Visual Basic? Технология ActiveX предоставляет вам такую возможность.      Чтобы использовать эту возможность, Delphi предлагает с помощью среды DAX заключить потомка класса TWin Control в элемент управления ActiveX. Ниже перечислены платформы, на которых сертифицированы для выполнены  элементы управления ActiveX, поставляемые с Delphi.   * Borland Delphi версии 2 и 3. * Borland C++ Builder. * Borland Paradox 8. * Borland IntraBuilder. * Microsoft Visual C++. * Microsoft Visual Basic версии 4 и 5. * Microsoft Internet Explorer 3.01. * Microsoft ActiveX Control Pad. * Microsoft Frontpage.       Создание элемента управления ActiveX предусматривает выполнение следующих этапов.   1. Элемент управления VCL должен быть вставлен в палитру компонентов Delphi и должен быть потомком класса TWinControl. Чтобы создать элемент управления ActiveX, являющийся потомком TGraphicControl. в большинстве случаев достаточно изменить базовый класс на TCustomControl. Этот способ использован в примере, который будет приведен немного позже. 2. Запустите мастера ActiveX Control, выбрав команду File/New и дважды щелкнув на пиктограмме ActiveX Control во вкладке ActiveX. В [окне](http://www.fabrikaokon.ru/" \t "http://www.helloworld.ru/texts/comp/lang/delphi/delphi4/_blank) мастера выберите компонент VCL, который нужно преобразовать в элемент ActiveX, укажите имя элемента ActiveX, включите или отключите информацию о лицензировании, версии и т.п. После щелчка на кнопке OK Delphi создаст библиотеку типов со всеми свойствами, объявленными как public и published, методами и событиями. Дополнительно создаются два файла исходного кода: в одном из них содержатся определения из библиотеки типов (интерфейсы, диспинтерфейсы и т.д.), а во втором — реализации методов интерфейса, определенных в первом файле. 3. В процессе создания библиотеки типов Delphi может преобразовать не все конструкции языка то ли потому, что нет смысла преобразовывать отдельные конструкции, то ли потому, что она не знает, как преобразовать некоторые конструкции языка Object Pascal. Независимо от причины, по которой преобразование не выполнилось, в определение элемента ActiveX можно легко добавить любые свойства, методы и события. Для этого откройте библиотеку типов (с помощью команды View/Type Library), внесите необходимые изменения и щелкните на кнопке Refresh панели инструментов редактора библиотек типов, что позволит Delphi обновить код. 4. Создайте код всех методов Get и Set, в которых не реализована необходимая функциональность или которые оставлены пустыми, как добавления к библиотеке типов. Это относится и к обработчикам событий. 5. Теперь вы, возможно, пожелаете создать одну или несколько страниц свойств. В средах, не поддерживающих инспекторов свойств, это даст возможность пользователю, щелкнув правой кнопкой мыши, вызвать некоторое подобие инспектора свойств для изменения значений свойств элемента управления. Для этого, как минимум, нужно создать основную страницу свойств. Delphi обладает несколькими стандартными страницами свойств для управления цветом, шрифтами, изображением и списком строк Delphi. 6. Откомпилируйте и зарегистрируйте элемент управления. 7. ОСХ-файл можно импортировать в любую среду разработки, поддерживающую элементы управления ActiveX, которые созданы в Delphi. |

**44.** Delphi позволяет многократно использовать одни и те же объекты в качестве шаблонов при дальнейшей разработке приложений. Для хранения таких объектов используется специальное хранилище Repository. При разработке приложения можно добавить в него объект из репозитория через [окно](http://www.fabrikaokon.ru/" \t "http://www.delphifaq.ru/razrabotka-prilozheniy-v-sisteme-delphi/_blank) New Items, которое вызывается командой File|New|Other. Большое количество объектов, находящихся в хранилище, распределены по нескольким страницам: New (базовые элементы), Forms (формы), Projects (проекты), Dialogs (диалоги), Data Modules (модули данных) и др. Кроме того, пользователь может помещать в репозиторий свои заготовки.

Для добавления объекта к проекту необходимо выбрать страницу и указать объект. Объекты можно добавлять к проекту различными способами, в зависимости от состояния переключателя в нижней части окна:

* Copy – копирование объекта. Изменения, внесённые в проекте в копию объекта, не влияют на оригинал;
* Inherit – от объекта в хранилище порождается новый объект, который добавляется к проекту;
* Use – использование объекта из хранилища. Изменение этого объекта в проекте приводит к изменению объекта в хранилище и во всех проектах, использующих этот объект аналогичным образом (как Use).

## Repository – репозитарий объектов

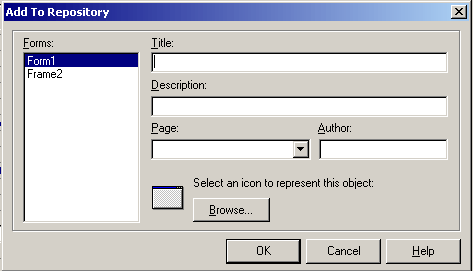
Delphi позволяет многократно использовать одни и те же объекты в качестве шаблонов при дальнейшей разработке приложений. Для хранения таких объектов используется специальное хранилище Repository. При разработке приложения можно добавить в него объект из репозитория через [окно](http://www.fabrikaokon.ru/" \t "http://www.delphifaq.ru/razrabotka-prilozheniy-v-sisteme-delphi/_blank) New Items, которое вызывается командой File|New|Other. Большое количество объектов, находящихся в хранилище, распределены по нескольким страницам: New (базовые элементы), Forms (формы), Projects (проекты), Dialogs (диалоги), Data Modules (модули данных) и др. Кроме того, пользователь может помещать в репозиторий свои заготовки.

Для добавления объекта к проекту необходимо выбрать страницу и указать объект. Объекты можно добавлять к проекту различными способами, в зависимости от состояния переключателя в нижней части окна:

* Copy – копирование объекта. Изменения, внесённые в проекте в копию объекта, не влияют на оригинал;
* Inherit – от объекта в хранилище порождается новый объект, который добавляется к проекту;
* Use – использование объекта из хранилища. Изменение этого объекта в проекте приводит к изменению объекта в хранилище и во всех проектах, использующих этот объект аналогичным образом (как Use).

Способ многократно использовать формы (а так же и фреймы)– использование репозитария объектов Delphi. Репозитарий объектов – это централизованное хранилище форм, фреймов и заготовок проектов, предназначенных для повторного использования в среде Delphi. Форма, которую мы видим при выборе пункта меню File/New/Other, отображает содержимое репозитария. Добавить форму или фрейм в репозитарий не составляет никакого труда. Просто щелкните на них правой кнопкой мыши и выберите пункт меню “Add to repository”. На экран будет выведен диалог добавления.

Для помещения формы в репозиторий, следует сохранить её в папке ObjRepos. Затем выполнить команду Add to Repository через контекстное меню. Если использовать команду Project|Add to Repository, то в репозиторий будет добавлен проект, а не форма. Проекты из репозитория копируются целиком, с родительскими классами, что замедляет работу и увеличивает объём памяти. В диалоговом окне Add to Repository надо записать сведения о помещаемой в архив форме: название, пояснения, имя автора разработки, название страницы хранилища.



Элементы этого диалога достаточно понятны:

Список Forms – список форм и фреймов, доступных для добавления.

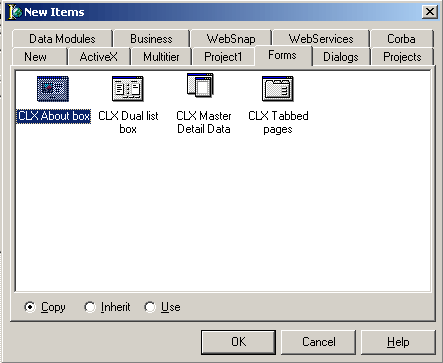
Title – наименование под которым элемент будет виден в репозитарии

Description – описание элемента

Page – страница, на которой элемент будет отображаться.

Author – сведения об авторе данного элемента.

Поместить форму из репозитария объектов в проект несложно, более того, вы это фактически уже неоднократно делали – File/New/Other. Обратим внимание только на группу переключателей внизу формы: их три Copy, Inherit, Use. Copy – копирует форму из репозитария в новый модуль. Inherit – наследует класс объекта, описанного в репозитарии. Use – просто включает модуль из репозитария в текущий проект.



Диалог настройки страниц репозитария и добавленных объектов вызывается пунктом меню IDE Tools/Repository. Если вы любитель покопаться во внутренностях Delphi, то скажу, что описание репозитария хранится в файле delphi32.dro. Файл описания имеет формат ini файла Windows.

## **45.** Фреймы

В Delphi 5 введен новый компонент, который помогает поддерживать стилистическое единство приложения. Это **Frame** — фрейм. Он представляет собой нечто среднее между панелью и формой. С формой его роднит то, что он:

* проектируется отдельно, как самостоятельное окно
* имеет свой модуль — файл **.pas**
* имеет возможности наследования, причем даже более широкие, чем у формы, так как может наследоваться даже внутри одного приложения
* может включаться в Депозитарий и использоваться так же, как и форма, включая наследование

С панелью фрейм роднит то, что он:

* не является самостоятельным окном Windows и может отображаться только на форме или другом контейнере
* имеет свойства, методы, события, подобные панели, а не форме

Таким образом, фрейм — это панель, т.е. некий фрагмент [окна](http://www.fabrikaokon.ru/" \t "http://beluch.ru/progr/100comp/_blank) приложения, но способный переноситься на разные формы, в разные приложения и допускающий использование преимуществ наследования.

Начать проектирование нового фрейма можно командой File | New Frame или командой File | New и выбором пиктограммы Frame на странице New [окна](http://www.fabrikaokon.ru/" \t "http://beluch.ru/progr/100comp/_blank) Депозитария. В обоих случаях перед вами откроется окно фрейма, подобное окну формы, а в Редакторе Кода вы увидите текст заготовки модуля фрейма:

**unit** Unit2;Interface // Открытый интерфейс фрейма {Список подключаемых модулей}**uses** Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs; {Объявление класса фрейма}**type** TFrame2 = **class**(TFrame) {Сюда Delphi помещает объявления компонентов, размещаемых на фрейме. Не добавляйте сюда ничего вручную} **private** // Закрытый раздел класса {Private declarations} {Сюда могут помещаться объявления переменных, функций и процедур, включаемых в класс фрейма, но не доступных для других модулей} **public** // Открытый раздел класса {Public declarations} {Сюда могут помещаться объявления переменных, функций и процедур, включаемых в класс фрейма и доступных для других модулей} **end**; {Сюда могут помещаться объявления типов, констант, переменных, функций и процедур, к которым будет доступ из других модулей, но которые не включаются в класс фрейма. Они будут едины для всех объектов фреймов}**implementation** // Реализация модуля{$R \*.DFM} {Сюда могут помещаться предложения **uses**, объявления типов, констант, переменных, к которым не будет доступа из других модулей. Они будут едины для всех объектов фреймов. Тут же должны быть реализации всех объявленных в разделе **interface** функций и процедур, а также могут быть реализации любых дополнительных, не объявленных ранее функций и процедур}**end**.

Комментарии в приведенном тексте поясняют, куда и что можно помещать в модуле. Те переменные, объявления которых вы поместите в объявление класса, будут индивидуальны для каждого объекта фрейма. Объявления имеют обычный для класса вид. Например:

A: integer;

Переменные, объявления которых вы поместите вне объявления класса, будут едины для всех объектов фрейма. Они объявляются как обычные переменные. Например:

**var** A: integer;

На фрейм вы можете так же, как на форму, переносить и размещать любые компоненты, устанавливать их свойства, писать обработчики их событий и т.п.

Давайте создадим чисто тестовый фрейм, чтобы на его примере продемонстрировать проектирование фрейма, его использование, доступ к различным его элементам и наследование свойств.

Начните новое приложение и выполните команду File | New Frame. Перенесите на фрейм групповую панель **GroupBox** (см. [раздел 7.2](http://beluch.ru/progr/100comp/7_2.htm)). Перенесите на панель метку **Label** и три кнопки**Button**. Разместите все эти компоненты примерно так, как показано на рис. 7.9, изменив соответственно их надписи (**Caption**) и назвав кнопки соответственно **BSetup**, **BInc**, **BShow**.

|  |  |
| --- | --- |
| **Рис. 7.9** Пример фрейма |  |

Давайте введем в наш модуль в разных местах объявления целых переменных, а в обработчики событий кнопок введем операторы, манипулирующие ими и отображающие результат в метке. Модуль фрейма может приобрести следующий вид:

**unit** UFrame;**interfaceuses** Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;**type** TFrame2 = **class**(TFrame) GroupBox1: TGroupBox; BSetup: TButton; Label1: TLabel; BInc: TButton; BShow: TButton; **procedure** BSetupClick(Sender: TObject); **procedure** BIncClick(Sender: TObject); **procedure** BShowClick(Sender: TObject); **private** { Private declarations } {Переменная А видна только в данном модуле} A: integer; **public** { Public declarations } {Переменная В видна в других модулях через объект фрейма} B:integer; **end**;{Переменная С видна в других модулях}**var** C:integer;**implementation**{$R \*.DFM}{Переменная D видна только в данном модуле}**var** D:integer;**procedure** TFrame2.BSetupClick(Sender: TObject);**begin** A:=1; B:=1; C:=1; D:=1; Label1.Caption := 'A=' + IntToStr(A) + ' B=' + IntToStr(B) + ' C=' + IntToStr(C) + ' D=' + IntToStr(D);**end**;**procedure** TFrame2.BIncClick(Sender: TObject);**begin** Inc(A); Inc(B); Inc(C); Inc(D); Label1.Caption := 'A=' + IntToStr(A) + ' B=' + IntToStr(B) + ' C=' + IntToStr(C) + ' D=' + IntToStr(D);**end**;**procedure** TFrame2.BShowCiick(Sender: TObject);**begin** Label1.Caption := 'A='+IntToStr(A) + ' B=' + IntToStr(B) + ' C=' + IntToStr(C) + ' D=' + IntToStr(D);**end**;**end**.

В модуле введены переменные:

* **А** — введена в закрытый раздел класса; видна только в процедурах данного класса в этом модуле; независимые друг от друга переменные **А** будут содержаться в каждом объекте фрейма.
* **В** — введена в открытый раздел класса; в других модулях можно получить доступ к **В** через имя объекта фрейма; независимые друг от друга переменные **В** будут содержаться в каждом объекте фрейма.
* **С** — введена в открытый интерфейс вне описания класса; доступна в других модулях; имеется единственный экземпляр **С**, независимо от числа объектов фреймов.
* **D** — введена в реализацию класса; доступна только в данном модуле; имеется единственный экземпляр **D**, независимо от числа объектов фреймов.

Введенные в модуль обработчики щелков на кнопках обеспечивают сброс всех переменных на 1 (процедура **TFrame2.BSetupClick**), увеличение всех переменных на 1 (процедура**TFrame2.BIncClick**), отображение текущего состояния переменных (процедура **TFrame2.BShowClick**).

Теперь давайте разместим несколько экземпляров фрейма на форме. Перейдите в основную форму приложения и выберите в палитре компонентов **Frame** (первая кнопка на страницеStandard). Появится диалоговое [окно](http://www.fabrikaokon.ru/" \t "http://beluch.ru/progr/100comp/_blank), в котором будет спрашиваться, какой фрейм вы хотите разместить на форме. Выберите ваш фрейм **Frame2** и он появится на форме. Можете отбуксировать его, как обычный компонент, в нужное место. Повторите эту операцию еще раз и разместите на форме второй фрейм (рис. 7.10). Добавьте кнопку и метку, задав ее свойство **Align**равным **alTop** и свойство **Alignment** равным **taCenter**.

Вы получили форму, содержащую два объекта — фрейма. Можете изменить какие-то свойства объектов. Например, изменить надписи (**Caption**) групповых панелей **GroupBox** (см. рис. 7.10 а).

|  |  |  |
| --- | --- | --- |
| **Рис. 7.10** Пример использования фреймов; форма (а) и приложение в работе (б) | а) |  |
|  | б) |  |

После того, как вы изменили эти свойства, они перестают наследоваться из класса фрейма. А остальные свойства продолжают наследоваться. В этом легко убедиться. Перейдите в модуль фрейма (рис. 7.9) и измените у фрейма стиль шрифта (**Font.Style**) на жирный. Вы увидите, что в обоих объектах главной формы шрифт тоже станет жирным. Верните во фрейме шрифт на обычный и он синхронно изменится в объектах. А теперь установите в одном из фреймов на форме шрифт жирным. Повторив после этого эксперимент с изменением шрифта в исходном фрейме, вы увидите, что теперь шрифт меняется только в том объекте формы, в котором вы его не изменяли вручную. Таким образом объекты наследуют только те свойства, которые не были в них установлены вручную.

Теперь давайте напишем обработчик щелчка на кнопке главной формы. Прежде всего взгляните на текст модуля этой формы. Вы увидите, что в нем в описании класса формы появились две строки:

Frame21: TFrame2; Frame22: TFrame2;

Это объявления объектов фреймов. Все компоненты, размещенные на фреймах, напрямую из модуля формы не видны. Доступ к ним можно получить только через объекты **Frame21** и **Frame22**. Имена компонентов, размещенных во фреймах, локальные. Несмотря на то, что во фреймах имеются кнопки с именами **BShow**, вы можете назвать тем же именем кнопку на форме.

Поместите в обработчик щелчка на этой кнопке оператор

Label1.Caption := 'В(Frame21)='+IntToStr(Frame21.B) + ' B(Frame22)='+IntToStr(Frame22.B) + ' C=' + IntToStr(C);

Он отображает в метке **Label1** значения переменных **В** объектов фреймов и значение переменной **С** класса фрейма. Значения переменных **А** и **D** отобразить невозможно, поскольку эти переменные недоступны из внешних модулей. Если вы попытаетесъ отобразить их значения, компилятор выдаст сообщение об ошибке.

Сохраните ваше приложение, оттранслируйте его и выполните. Манипулируя кнопками вы легко сможете убедиться (см. рис. 7.10 б), что переменные **А** и **В** независимы для каждого фрейма, а переменные **С** и **D** одинаковы. Точнее оба фрейма оперируют с одними и теми же переменными **С** и **D**.

Рассмотренный фрейм не имел никакого практического значения. Давайте построим более полезный пример. Во многих диалогах при установке различных опций фигурирует фрагмент, фрейм которого показан на рис. 7.11. Фрагмент включает в себя панель **GroupBox**, [окно](http://www.fabrikaokon.ru/" \t "http://beluch.ru/progr/100comp/_blank) редактирования, в котором пользователь может написать имя файла, и кнопку Обзор, которая позволяет выбрать файл в стандартном диалоге Windows открытия файла. Если путь к файлу длинный, то полное имя файла с путем может не помещаться в [окне](http://www.fabrikaokon.ru/" \t "http://beluch.ru/progr/100comp/_blank) редактирования. Поэтому полезно для него предусмотреть всплывающее окно, которое отображало бы полное имя файла вместе с путем и всплывало бы, если пользователь задержал над ним курсор мыши.

Давайте построим подобный фрейм и опробуем его в работе. Начните новое приложение и выполните команду File | New Frame. Перенесите на фрейм групповую панель **GroupBox**. Перенесите в эту панель [окно](http://www.fabrikaokon.ru/" \t "http://beluch.ru/progr/100comp/_blank) редактирования **Edit**, кнопку **Button**, диалог **OpenDialog** (см. [раздел 8.2](http://beluch.ru/progr/100comp/8_2.htm)) и компонент **ApplicationEvents** — перехватчик событий приложения (см. [раздел 9.4](http://beluch.ru/progr/100comp/9_4.htm)). Расположите компоненты примерно так, как показано на рис. 7.11.

|  |  |
| --- | --- |
| **Рис. 7.11** Фрейм выбора файла |  |

Задайте в свойстве **Filter** диалога **OpenDialog** какой-то фильтр файлов, например, «все файлы|\*.\*». Свойство **ShowHint** (показать ярлычок подсказки) в компонентах **Edit** и **Button** установите в**true**. В кнопке **Button** кроме того можете написать текст подсказки **Hint**, например, «Выбор файла|Выбор файла из каталога».

В обработчик события **OnShowHint** компонента **ApplicationEvents** занесите оператор:

**if** HintInfo.HintControl = Edit1 **then** **begin** HintStr := Edit1.Text; ApplicationEvents1.CancelDispatch; **end**;

Этот оператор в момент, когда должен отображаться ярлычок, проверяет, не является ли источником этого события (**HintInfo.HintControl**) [окно](http://www.fabrikaokon.ru/" \t "http://beluch.ru/progr/100comp/_blank) редактирования **Edit1**. Если да, то текст ярлычка (**HintStr**) подменяется текстом, содержащимся в окне редактирования и принимаются меры (метод **CancelDispatch**), чтобы это событие не обрабатывалось другими компонентами**ApplicationEvents**, которые могут присутствовать в приложении. Пояснение всех этих операций см. в [разделе 9.4](http://beluch.ru/progr/100comp/9_4.htm).

Теперь введите в модуль фрейма глобальную переменную **FileName** типа **string**, в которой будет отображаться выбранный файл. В обработчик щелчка на кнопке введите оператор

**if** OpenDialog1.Execute **then** **begin** Edit1.Text := OpenDialog1.FileName; FileName := OpenDialog1.FileName; **end**;

который вызывает диалог открытия файла и помещает в [окно](http://www.fabrikaokon.ru/" \t "http://beluch.ru/progr/100comp/_blank) редактирования **Edit1** и в переменную **FileName** имя файла, выбранного пользователем, вместе с путем к нему.

В обработчик события **OnExit** компонента **Edit1** поместите оператор

FileName := Edit1.Text;

заносящий в переменную **FileName** имя файла, если пользователь не пользовался диалогом, а просто написал в [окне](http://www.fabrikaokon.ru/" \t "http://beluch.ru/progr/100comp/_blank) имя файла.

Программирование фрейма закончено. Теперь создайте тестовую программу, использующую этот фрейм. Предположим, что вам нужно разместить на форме два фрагмента, описанных вами во фрейме. Перейдите в основной модуль вашего приложения и разместите на форме так, как вы уже делали, два объекта вашего фрейма (рис. 7.12 а).

|  |  |  |
| --- | --- | --- |
| **Рис. 7.12** Приложения с двумя фреймами выбора файла: его форма (а) и приложение во время выполнения (б) | а) |  |
|  | б) |  |

Теперь вы можете поменять что-то в размещенных на форме объектах фреймах, изменить надписи групповых панелей, шрифты и т.п. Сохраните ваше приложение вместе с модулем фрейма, оттранслируйте его и проверьте в работе (рис. 7.12 б).

Вы разработали достаточно полезный фрейм и хотели бы его сохранить для использования в будущих приложениях. Это легко сделать, внеся его в Депозитарий. Щелкните на своем фрейме правой кнопкой мыши и выберите из всплывшего меню раздел Add To Repository. Перед вами откроется [окно](http://www.fabrikaokon.ru/" \t "http://beluch.ru/progr/100comp/_blank), представленное на рис. 7.13. В верхнем его [окне](http://www.fabrikaokon.ru/" \t "http://beluch.ru/progr/100comp/_blank) редактирования Title вы должны написать название вашего фрейма — подпись под его пиктограммой при входе в Депозитарий. В следующем окне — Description можете написать более развернутое пояснение. Его может увидеть пользователь, войдя в Депозитарий, щелкнув правой кнопкой мыши и выбрав во всплывшем меню форму отображения View Details. В выпадающем списке Page вы можете выбрать страницу Депозитария, на которой хотите разместить пиктограмму своего фрейма. Впрочем, вы можете указать и новую страницу с новым заголовком (Мои формы на рис. 7.13). В результате она появится в Депозитарии.

В [окне](http://www.fabrikaokon.ru/" \t "http://beluch.ru/progr/100comp/_blank) Author вы можете указать сведения о себе как об авторе. Наконец, если стандартная пиктограмма вас не устраивает, вы можете выбрать другую, щелкнув на кнопке Browse.... После выполнения всех этих процедур щелкните на кнопке OK и ваш фрейм окажется включенным в Депозитарий.

Теперь вы можете использовать его в последующих ваших приложениях. Для этого вам надо будет выполнить команду File | New и в открывшемся диалоговом окне New Items отыскать ваш фрейм (рис. 7.14).

|  |  |
| --- | --- |
| **Рис. 7.13** Окно добавления фрейма в Депозитарий |  |

|  |  |
| --- | --- |
| **Рис. 7.14** [Окно](http://www.fabrikaokon.ru/" \t "http://beluch.ru/progr/100comp/_blank) New Items с включенным новым фреймом |  |

В нижней части [окна](http://www.fabrikaokon.ru/" \t "http://beluch.ru/progr/100comp/_blank) расположены три радиокнопки, которые определяют, как именно вы хотите заимствовать фрейм из Депозитария: Сору — копировать, Inherit — наследовать, Use — использовать. Если включена кнопка Сору, то файлы фрейма просто будут скопированы в ваше приложение. При этом никакой дальнейшей связи между исходным фреймом и копией не будет. Вы можете спокойно изменять свойства вашей копии и это никак не отразится на фрейме, хранящемся в Депозитарии. А если вы в дальнейшем что-то измените во фрейме, хранящемся в Депозитарии, то эти изменения никак не затронут вашего приложения, куда вы до этого скопировали фрейм.

При включенной кнопке Inherit вы получите в своем проекте фрейм, наследующий размещенному в Депозитарии. Это значит, что если вы что-то измените во фрейме, хранящемся в Депозитарии, то это отразится при перекомпиляции во всех проектах, которые наследуют этот фрейм. Однако, изменения в наследуемых фреймах никак не скажутся на свойствах фрейма, хранящегося в Депозитарии.

При включенной кнопке Use вы получите режим использования. В этом случае в ваш проект включится сам фрейм, хранящийся в Депозитарии. Значит любое изменение свойств фрейма, сделанное в вашем проекте, отразится и на хранящемся в Депозитарии фрейме, и на всех проектах, наследующих или использующих этот фрейм (при их перекомпиляции).

Таким образом, режим Inherit целесообразно использовать для всех модулей вашего проекта, а режим Use — для изменения базового фрейма. Тогда усовершенствование вами базового фрейма будет синхронно сказываться на всех модулях проекта при их перекомпиляции.

Введенный в Delphi 5 компонент фрейм благодаря использованию наследования позволяет обеспечить единство стилистических решений не только внутри приложения, но и в рамках серии разрабатываемых вами приложений. Вам достаточно один раз разработать какие-то часто применяемые фреймы, включить их в Депозитарий, а затем вы можете использовать их многократно во всех своих проектах.