Міністерство освіти і науки України
Уманський державний педагогічний університет імені Павла Тичини

Мова програмування Lazarus

Навчальний посібник
Частина 2

Укладач Паршукова Л. М.

Умань
2021

УДК 004(075.8)

Мова програмування Lazarus: навчальний посібник / Уманський держ. пед. ун-т імені Павла Тичини ; уклад. Паршукова Л.М. – Умань: Візаві, 2021. - 140 с.

Навчальний посібник «Мова програмування Lazarus» висвітлює основні теми розділів з основ подійно- та об’єктно-орієнтованого програмування. В якості середовища програмування розглядається безкоштовне середовище програмування Lazarus на мові програмування Object Pascal (безкоштовний аналог Delphi)

УДК 004(075.8)
© Л.М. Паршукова, уклад., 2021

Зміст

1.	Тема 11. Екранна заставка	4
2.	Тема 12. Дата й час	11
3.	Тема 13. Масиви прості, багатомірні та динамічні	25
4.	Тема 14. Колекції (масиви) рядків та компоненти для них	39
5.	Тема 15. Діалоги	57
6.	Тема 16. Організація меню та панелей інструментів	70
7.	Тема 17. Модулі	83
8.	Тема 18. Блокнот - шифратор	97
9.	Тема 19. Дерева	118
10.	Тема 20. Багатовіконні додатки. SDI- і MDI-інтерфейси	130
11.	Список використаних джерел та корисні посилання	140

Тема 11. Екранна заставка

1. Постановка завдання
2. Реалізація проєкту
3. Компонент TTimer
4. Встановлення заставки в Windows

1. Постановка завдання

Час від часу для закріплення матеріалу ми будемо виконувати лабораторні роботи - реальні програмні проєкти різної складності. Цього разу ми з вами зробимо екранну заставку у вигляді електронних годин.

Насамперед, екранна заставка займає весь екран, не має системного рядка, головного меню й обрамлення головного вікна. Та й вікно в екранної заставки, як правило, одне. Ще одна відмінність - файл із екранною заставкою має розширення *.scr (від англ. screen - екран), а не *.exe, як програма. Проте, екранна заставка - та ж програма.

Подумаємо, яким чином ми можемо реалізувати таку заставку у вигляді годин. Відповідь напрошується сам собою: встановити в центрі екрану звичайну панель, на якій у властивості Caption виводити поточний час. Однак, це нудно. Нехай ця панель ще й переміщується по екрану по випадковій траєкторії! Щоб реалізувати такий проєкт, нам доведеться вивчити декілька речей: системний таймер TTimer та функцію-генератор випадкових чисел Random().

2. Реалізація проєкту

Взагалі ж, спочатку необхідно розробити алгоритм програми.

Алгоритм - набір інструкцій, що описують порядок дій виконавця для досягнення результату рішення завдання за кінцеве число дій.

Але наш проєкт досить простий, тому алгоритму, як такого, немає. Почнемо проєкт, при необхідності розробляючи алгоритм подальших дій по ходу справи. Відкрийте **Lazarus** з новим проєктом, властивість Name головної форми відразу перейменуйте в fMain. Збережіть проєкт у папку **11-01** під ім’ям **MyClock**, а модуль головної форми назвіть Main.

Тепер займемося вікном. Оскільки у вікна заставки не повинно бути системного рядка, властивість BorderStyle встановимо в bsNone. А тому що це вікно має займати весь екран, властивість WindowState переведемо в wsMaximized. Властивість Color встановимо в clBlack,

щоб форма стала чорною. І ще: властивість форми Cursor переведемо в crNone, щоб у працюючій заставці не було видно покажчика миші.

Підемо далі. Встановимо на форму мітку TLabel. Не звертайте уваги, що на чорному тлі її поки не видно на формі. Змінимо в ній деякі властивості:

- Name = lClock (так ми будемо звертатися до мітки в програмному коді; l - скорочення від label - мітка, Clock - годинники)
- Caption = 00:00:00
- Font = Times New Roman, жирний курсив, 48 пікселів. Колір шрифту виберіть жовтий.
- ParentColor = True (це змусить мітку завжди зливатися з формою, а колір шрифту буде контрастувати із загальним тлом)

Якщо ви забули, як налаштувати властивості шрифту, то нагадаю: потрібно виділити властивість Font, потім натиснути на кнопку із трьома крапками в правій частині властивості, в **Інспекторі об'єктів**:

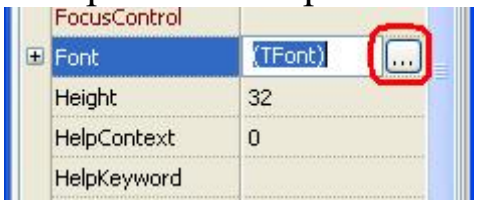


Рис. 11.1. Встановлення властивостей шрифту

Відкриється вікно, де ви зможете вибрати шрифт, накреслення, розмір і колір шрифту. Можете вибрати свої кольори для шрифту й панелі, головне, щоб вони контрастували один з одним, і добре поєднувались із чорним кольором форми, який ми встановимо пізніше.

У результаті виконаних дій у вас повинне вийти приблизно таке вікно:



Рис. 11.2. Вікно головної форми

Не хвилюйтеся із приводу того, що вікно поки маленьке, а системний рядок і обрамлення вікна є - коли програма буде працювати, вікно займе весь екран, якщо ви не наплутали з параметрами форми.

3. Компонент TTimer

Прийшла черга таймера. Він нам потрібний, щоб змусити напис мітки мінятися - показувати поточний час. Крім того, завдяки цьому компоненту ми зможемо забезпечити мітці безперервний рух.

Таймер TTimer перебуває на вкладці **System Палітри компонентів**:

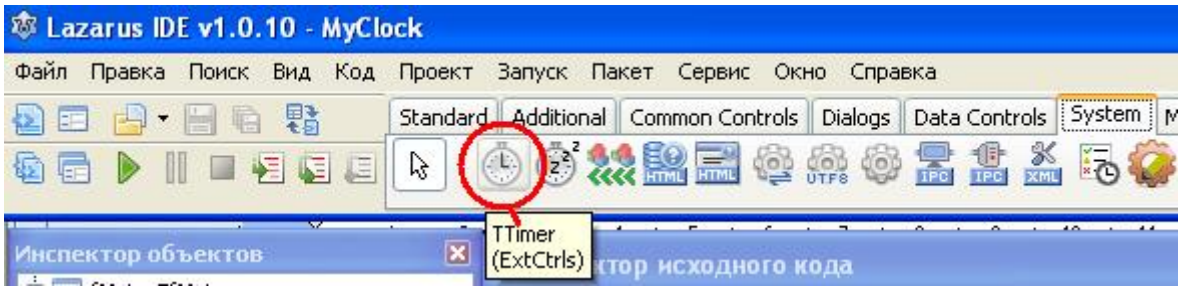


Рис. 11.3. Компонент TTimer

Компонент цей **невізуальний** - невидимий для користувача, тому його можна встановити в будь-яке місце на формі, аби тільки він вам не заважав при проєктуванні форми, не перекривав інші компоненти. Компонент буде вам видимий, незважаючи на чорні кольори форми.

У компонента є всього по одній властивості й події, які нам можуть знадобитися:

Interval - це властивість встановлює інтервал часу, коли таймер спрацює. За замовчуванням, він дорівнює 1000, що відповідає 1 секунді. Це нам і потрібно, тому залишимо значення властивості без змін. А на майбутнє запам’ятайте: якщо вам буде потрібно встановити інший час, то множте необхідну кількість секунд на 1000, і отримаєте те, що потрібно.

Далі, переходимо в **Інспекторі об’єктів** на вкладку **Події**. Нас цікавить тільки одна подія:

OnTimer - подія спрацьовує щораз, коли закінчується встановлений інтервал. Згенеруйте код для цієї події так само, як раніше генерували подію OnClick для кнопок, ми заповнимо його трохи пізніше.

От тепер нам потрібно продумати алгоритм дій - як ми будемо рухати напис. Ідея така: згенеруємо випадкове число від 0 до 3, вийде 4 варіанти. Адже в нас є 4 напрямки, куди рухатися: нагору, униз, вліво

або вправо. Залежно від отриманого випадкового числа будемо вибирати й напрямок руху. Рухати будемо, скажемо, на 50 пікселів вліво - вправо, або на 25 пікселів вниз. А як рухати? Так дуже просто! Left - це відстань у пікселях від лівого краю форми до компонента. Припустимо, у мітки Left дорівнює 100. Тобто, від краю форми до мітки 100 пікселів. Якщо ми додамо ще 50, то тим самим, зрушимо мітку вправо, а якщо навпаки, віднімемо, то зрушимо вліво. Для руху вниз використаємо властивість Top - відстань від верхньої частини форми до компонента. Рухи будуть аналогічні, але на 25 пікселів.

Якщо отримаємо напрямок 0, то рухатись будемо вправо: додавати до властивості Left мітки 50 пікселів. Якщо напрямок 1, то рухаємось вліво, віднімаючи 50 пікселів від властивості Left. Якщо напрямок 2, то рухаємось вниз: додаємо 25 пікселів до властивості Top мітки. Ну і якщо напрямок 3, то рухаємось нагору, додаючи 25 пікселів до властивості Top.

Причому може трапитися й так, що напис виявиться за межами вікна. У цьому випадку нам потрібно буде повернути його назад у ці межі. Зробимо це в такий спосіб: якщо мітка пішла вліво, то властивості Left просто привласнимо значення 0 - мітка повернеться в межі форми й виявиться притиснутою до лівого краю.

Якщо мітка пішла за правий край, то привласнимо властивості Left ширину форми мінус ширину мітки, таким чином, мітка виявиться притиснутою до правої частини форми й буде в її межах.

Точно в такий же спосіб будемо повертати мітку, якщо вона піде нагору або вниз за межі форми.

Виходить, нам буде потрібно тільки одна цілочисельна змінна, в яку ми будемо генерувати напрямок - 4 позиції. Для цього використаємо функцію Random(), що ще називають **генератором випадкових чисел**.

Random() - функція повертає випадкове число (англ. random - випадковий) від 0 до зазначеної кількісті чисел мінус один. Синтаксис дуже простий:

Random(<кількість чисел>);

Функція поверне випадкове число від 0 до <кількість чисел> - 1. Тобто, у випадку

i:= Random(4);

у змінну і потрапить випадкове число від 0 до 3. Цим ми й скористаємося. А щоб не витрачати зайву пам’ять, змінну і оголосимо з типом byte.

Незважаючи на амбіційну ідею, код події OnTimer буде досить простий:

```
procedure TfMain.Timer1Timer(Sender: TObject);
var
  i: byte; //для отримання випадкового числа
begin
  //по-перше міняємо напис мітки:
  lClock.Caption:= TimeToStr(Now);
  //отримуємо випадковий напрямок:
  i:= Random(4);
  //тепер залежно від напрямку рухаємо мітку:
  case i of
    0: lClock.Left:= lClock.Left + 50; //вправо
    1: lClock.Left:= lClock.Left - 50; //уліво
    2: lClock.Top:= lClock.Top + 50; //нагору
    3: lClock.Top:= lClock.Top - 50; //униз
  end;
  //тепер перевіряємо: чи не вийшла мітка за межі форми?
  //якщо вийшла - повертаємо її назад
  //якщо пішла вліво:
  if lClock.Left < 0 then lClock.Left:= 0;
  //якщо пішла нагору:
  if lClock.Top < 0 then lClock.Top:= 0;
  //якщо пішла вправо:
  if (lClock.Left + lClock.Width) > fMain.Width then
    lClock.Left:= fMain.Width - lClock.Width;
  //якщо пішла вниз:
  if (lClock.Top + lClock.Height) > fMain.Height then
    lClock.Top:= fMain.Height - lClock.Height;
end;
```

От і весь код! Вам може виявитися незрозумілим рядок
lClock.Caption:= TimeToStr(Now);

Таким чином, ми привласнюємо властивості Caption мітки lClock поточний час. Роботі з датою й часом у нас буде присвячена наступна лекція, так що поки на цьому не запиняємось. Весь інший код повинен вам бути зрозумілий.

Важливо, перевірити роботу заставки можна вже зараз, тільки не забудьте попередньо зберегти зміни. Щоб закрити вікно, у якого немає для цього кнопок і меню, скористайтеся стандартними кнопками Windows <**Alt + F4**>.

Але заставка на цьому ще не закінчена. Насамперед, через включену налагоджувальну інформацію програма має великий розмір, майже 15 мегабайт! Крім того, ми одержали виконуваний exe-файл, а нам потрібна заставка з розширенням *.scr. Виправляємо ці недоліки:

1. Вибираємо команду меню **<Проєкт -> Параметри проєкту>**.
2. У розділі **<Параметри компілятора>** вибираємо підрозділ **<Шляху>**.
3. У поле **<Ім'я здійсненного файлу (-o) >** замість **MyClock** вказуємо **MyClock.scr** (тобто, додаємо потрібне розширення).
4. Відключаємо прапорець **<Застосовувати згоди по іменуванню>**, щоб не було конфлікту розширень.
5. У розділі **<Параметри компілятора>** переходимо на підрозділ **<Компонування>**.
6. Відключаємо прапорець **<Генерувати налагоджувальну інформацію для GDB>**, щоб зменшити розмір отриманого файлу.
7. Зберігаємо проєкт, і знову запускаємо. У результаті в папці **11-01** одержуємо необхідний файл **MyClock.scr** розміром ледь більше 1,5 мегабайта.

Якщо вас влаштовує, що для закриття заставки доводиться натискати <**Alt + F4**>, то нічого більше робити не потрібно. Але мені б хотілося, щоб цю заставку можна було закрити кнопкою <**Esc**> (код якої 27 у таблиці символів ANSI), як будь-яку іншу нормальну заставку. А для цього потрібно саму малість доробити проєкт.

Поверніться в **Редактор форм**, і виділіть саму форму. В **Інспекторі об'єктів** перейдіть на закладку **Події** й згенеруйте подію форми OnKeyPress, що спрацьовує при натисканні на будь-яку кнопку:

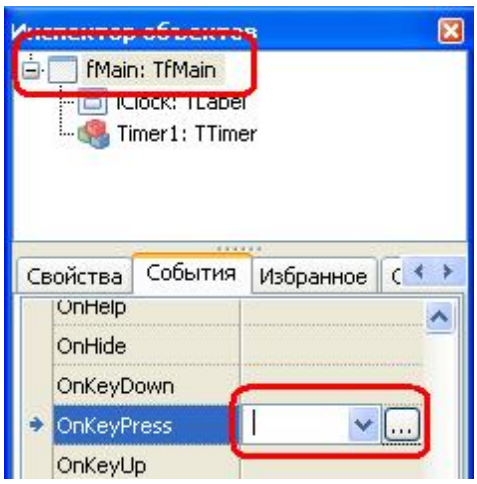


Рис. 11.4. Подія форми OnKeyPress

Її код зовсім простий:

```
procedure TfMain.FormKeyPress(Sender: TObject; var Key: char);
begin
  //якщо нажали Esc, те виходимо:
  if Key = #27 then Close;
end;
```

Збережіть проєкт і знову його запустіть. Заставка тепер повинна закриватися при натисканні клавіші **<Esc>**. **Lazarus** нам більше не потрібний, можете його закрити.

4. Встановлення заставки в Windows

Більшість із вас напевно знає, як отриману заставку встановити у Windows. Для тих, хто цього все-таки не знає, розповім докладно:

1. Знайдіть файл **MyClock.scr**, він повинен бути в папці **11-01**, де ви зберігаєте проєкти.
2. Скопіюйте цей файл у папку **C:\Windows**
3. Клацніть правою кнопкою по вільному від вікон місцю **Робочого стола** й виберіть команду «**Властивості**».
4. У вікні, що відкрилося, **«Властивості:** Екран» перейдіть на вкладку **«**Заставка».
5. Там виберіть нашу заставку й необхідний інтервал її появи, після чого натисніть **<OK>**:

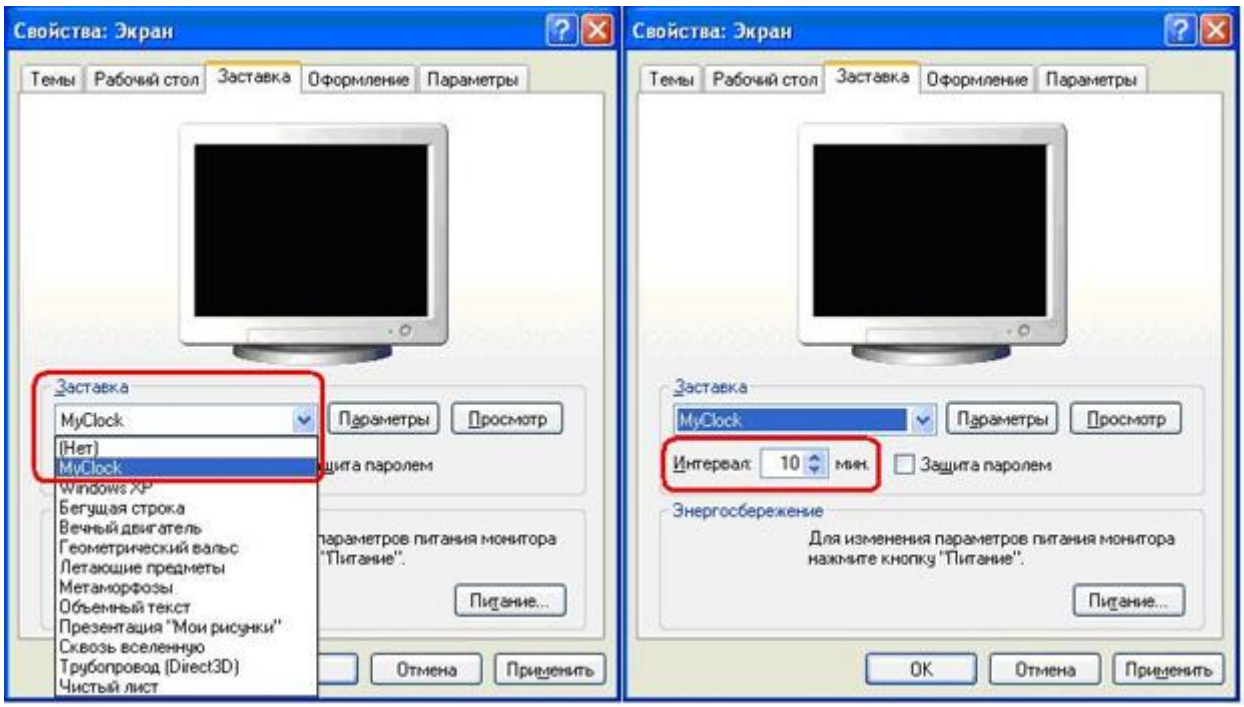


Рис. 11.5. Вибір заставки й інтервалу

Якщо користувач протягом зазначеного інтервалу не буде рухати мишею й не натисне ніякої клавіші, то з’явиться наша заставка.

Тема 12. Дата й час

1. Тип TDateTime
2. Компоненти для роботи з датою-часом
3. TCalendar
4. TDateEdit
5. Стандартні функції для роботи з датою та часом

1. Тип TDateTime

Для роботи з датою й часом в Lazarus передбачений тип TDateTime. Це універсальний тип, він дозволяє одночасно визначити й дату, і час. TDateTime займає 8 байт оперативної пам’яті й фактично, є дійсним числом (TDateTime = Double): ціла частина цього числа містить дату, а десяткова - час. Можна також для роботи з датою й часом використовувати типи відповідно, TDate і TTime, але це просто синоніми того ж типу TDateTime.

В довіднику Lazarus говориться, що мінімально можливе значення дати-часу дорівнює -693593.0, а максимальне +2958465.99999. Ціла частина такого числа позначає кількість доби, а дробова - частина доби, що пройшли з 0 годин. Додавання або віднімання одиниці із цілої

частини числа рівносильно додаванню або відніманню одного дня. Число -693593 відповідає даті 01.01.0001, а число +2958465 відповідає даті 31.12.9999, у цьому діапазоні ви й зможете працювати з датами в Lazarus.

2. Компоненти для роботи з датою-часом

На попередній лекції нам уже довелось працювати з часом, для виведення поточного часу на екран ми використовували просту мітку TLabel. Але в Lazarus для введення та виведення дати-часу є спеціальні компоненти. Знайомитись із цими компонентами будемо відразу на прикладах. Завантажте Lazarus з новим проєктом, відразу ж збережете його в папку **12-01**.

Компоненти для роботи з датою й часом розташовуються на вкладці **Misc Палітри компонентів**. Взагалі, дуже цікаві компоненти перебувають на цій вкладці, і з деякими з них ми обов'язково ще познайомимося. Зараз же нас цікавлять усього два компоненти - календар TCalendar і рядок редагування дати TDateEdit:

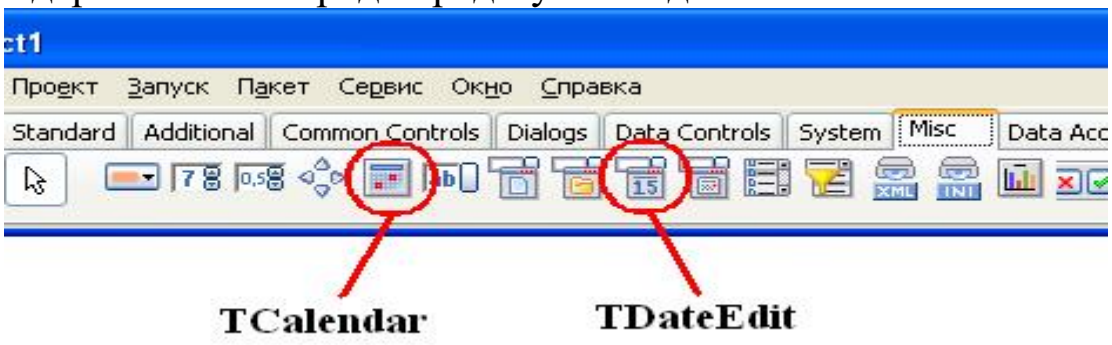


Рис. 12.1. Компоненти для роботи з датою-часом TCalendar і TDateEdit

Правда, потрібно відмітити, компоненти ці ще сирі: деякі властивості є, але не працюють, деякі працюють лише частково, деяких і зовсім немає. Так, у платному аналогу Lazarus - Delphi у компоненті TDateEdit є можливість вводити як дату, так і час. В Lazarus можна вводити тільки дату. Для введення часу, імовірно, потрібно використати звичайний TMaskEdit.

Оскільки Lazarus - проєкт новий, будемо сподіватися, що ці недоробки в майбутньому будуть виправлені.

3. TCalendar

Спочатку познайомимося з компонентом TCalendar. Встановіть його на форму, і він відразу прийме вигляд календаря, причому поточна дата буде виділена. Виділення буде як червоною окантовкою, так у календарі виділяється поточна дата, так і синім заливанням - так

у календарі виділяється обрана дата. А оскільки вибрати можна будь-яку дату, то вони не завжди збігаються. Однак з форми керувати календарем не дуже зручно. Збережіть проєкт і запустіть його. У робочому додатку календарем можна керувати: клацаючи по синіх стрілках у лівій та правій частинах заголовка календаря можна перегортати місяці, клацнувши за назвою місяця можна відкрити список, у якому вибрати інший місяць, клацнувши по році, можна вибрати або ввести вручну інший рік.

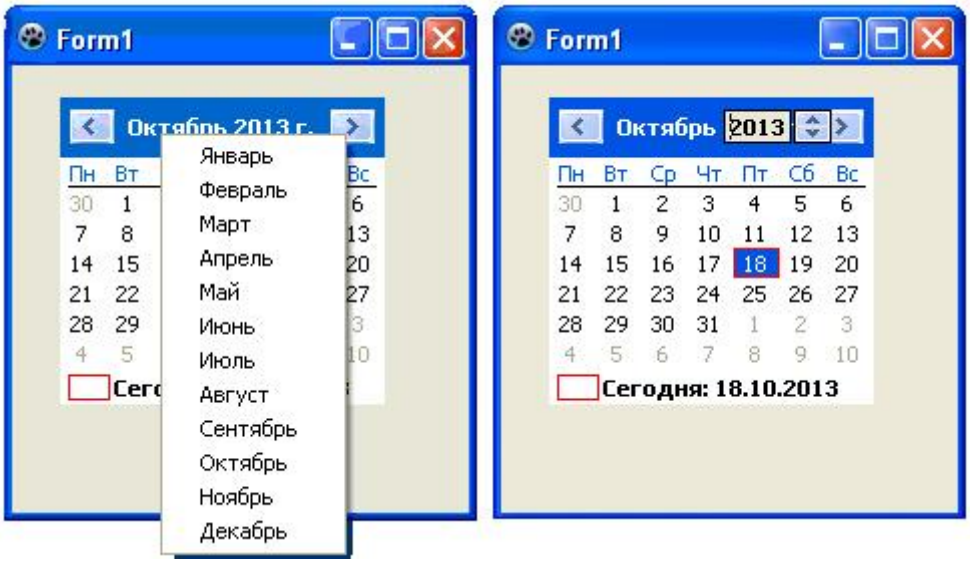


Рис. 12.2. Вибір місяця й року в календарі

Закрийте запущену програму й поверніться в проєкт. Познайомимося з деякими корисними властивостями компонента TCalendar.

Align	- вирівнювання компонента щодо форми (або іншого контейнера). Із цією властивістю ви вже знайомі, але у випадку з календарем застосовувати її не має сенсу - якщо ще можна притиснути календар до однієї зі сторін, то змінити його розміри однаково не вдасться.
BorderSpacing	- складена властивість, що розкривається. По ідеї, вона повинна набудовувати параметри обрамлення: ширину обрамлення навколо календаря або по одній з його сторін, але, на жаль, ця властивість теж не працює - як я не намагався щось змінити, вигляд календаря залишився колишнім.
Date і DateTime	- установлена в календарі дата. При зміні одного із цих властивостей, зміниться й інше, але є й відмінності: Date має тип String, а DateTime - TDateTime. Таким

	чином, якщо вам потрібно одержати дату у вигляді рядка, звертайтеся до Date. Якщо треба працювати з датою у формі TDateTime, а це часто буває потрібно, наприклад, для розрахунку дат, то звертайтеся до TDateTime. Привласнити календарю нову дату можна як через першу, так і через другу властивість, але якщо ви будете міняти дату за допомогою Date, ви повинні бути впевнені, що дата в рядку написана правильно.
DisplaySettings	- властивість повинна управляти виглядом календаря, але теж чомусь не змінює його. Має наступні підвластивості, дозволяючи (при True) або забороняючи (при False) різні дії: • dsNoMonthChange - Не дозволяти змінювати місяць • dsShowDayNames - Показувати ім'я дня тижня • dsShowHeadings - Показувати заголовок • dsShowWeekNumbers - Показувати номер тижня, у році їх 52 (цю підвластивість встановити можна). • dsStartMonday - Починати тиждень із Понеділка (в англійських і американських календарях тиждень прийнято починати з Неділі).

Перейдемо до подій компонента (перемкніться на вкладку **Події** в **Інспекторі об'єктів**). Тут нам можуть знадобитися наступні події:

- OnChange - подія наступає при будь-якій зміні в компоненті: змінили рік, місяць або день.
- OnClick - клацнули по компоненті мишею.
- OnDayChanged - змінився день.
- OnDblClick - двічі клацнули по компоненті мишею.
- OnMonthChanged - змінився місяць.
- OnYearChanged - змінився рік.

4. TDateEdit

Тепер вивчимо компонент TDateEdit. Встановіть його на форму, де-небудь під календарем. Як бачите, фактично компонент складається із двох компонентів: поля для редагування дати й кнопки, при натисканні на яку з'являється календар:



Рис. 12.3. *Компонент TDateEdi*t

Тепер перейдіть на вкладку **Властивості** в **Інспекторі об'єктів**. У компонента є наступні корисні властивості:

ButtonOnlyWhenFocused	- Показувати кнопку вибору дати тільки коли фокус введення перебуває на компоненті. За замовчуванням False, тобто, кнопку видно завжди.
ButtonWidth	- ширина кнопки. За замовчуванням дорівнює 23, але можна й змінити. Правда, знову незрозуміло навіщо.
CalendarDisplaySettings	- аналогічно властивості DisplaySettings календаря, і призначено для того ж: для зміни зовнішнього вигляду календаря, що з'являється при натисканні на кнопку.
CancelCaption	- заголовок для кнопки Скасування в діалоговому вікні **Календар**. Мова йде про прив'язаний до компонента діалог TCalendarDialog (діалоги ми будемо вивчати в іншій лекції).
DateOrder	- формат висновку дати. При виборі формату, відмінного від doNone (за замовчуванням), з'являється маска дати. Може мати наступні значення: • doDMY - послідовність дати: День, Місяць, Рік. • doMDY - послідовність дати: Місяць, День, Рік. • doNone - послідовність дати відповідає системній. • doYMd - послідовність дати: Рік, Місяць, День.
DefaultToday	- якщо True, то в працюючій програмі компонентів за замовчуванням відображає поточну дату, якщо False, то не відображає нічого.

OKCaption	- заголовок для кнопки **ОК** у діалоговому вікні **Календар**. Проблеми ті ж, що й в CancelCaption.

В **Інспекторі об'єктів** чому те не відображаються наступні необхідні властивості:

Date	- має тип TDateTime, містить обрану або введену користувачем дату.
Text	- має тип TCaption, що сполучає зі звичайним рядком. Отут обрана або введена дата зберігається у вигляді рядка.

Цими властивостями часто прийдеться користуватися, щоб довідатися, яку ж дату вибрав або ввів користувач, так що запам'ятаєте їх.

Основні події, які вам можуть знадобитися, це OnChange (при зміні), OnClick (клацнули мишею), OnKeyPress (натисли будь-яку клавішу). Події ці працюють так само, як в інших, уже відомих вам компонентах, так що затримуватися отут ми не будемо.

5. Стандартні функції для роботи з датою та часом

Ознайомимося з основними функціями, які допоможуть нам працювати з датою й часом. Для виконання прикладів раджу встановити на форму просту кнопку, і згенерувати для неї подію OnClick, у якій й будете пробувати пропоновані приклади. Отже, почнемо.

Date - Функція повертає поточну дату у форматі TDateTime. Частина числа після коми (тимчасова частина) буде дорівнює нулю. Приклад застосування:

```
var  dt: TDateTime;
begin
  dt:= Date;  //одержали в dt поточну дату
  …
```

Time - Функція повертає поточний час у форматі TDateTime. Частина числа перед комою (частина дати) буде дорівнює нулю. Приклад застосування аналогічний функції Date.

Now - Функція повертає поточну дату й час у форматі TDateTime. Обидві частини числа будуть заповнені значеннями. Приклад застосування аналогічний спільним функціям Date і Time.

DateTimeToStr() - Функція приймає як параметр дату й час у форматі TDateTime і повертає ці дані у вигляді рядка. Синтаксис функції:

```
function DateTimeToStr(DateTime: TDateTime): String
```

Приклад виведення поточної дати та часу на екран:

```
begin
  ShowMessage(DateTimeToStr(Now));
```

У результаті на екран вийде повідомлення з поточною датою й часом, наприклад:

19.04.2018 9:25:57

DateToStr() - Функція приймає як параметр дату у форматі TDateTime і виводить її у вигляді рядка. Час при цьому не виводиться. Застосування функції аналогічно DateTimeToStr.

TimeToStr() - Функція приймає як параметр час у форматі TDateTime і виводить його у вигляді рядка. Дата при цьому не виводиться. Застосування функції аналогічно DateTimeToStr.

Зверніть увагу, що формати виведення дати-часу залежать від того, яка Windows у вас встановлена.

Всі ці правила залежать від системних змінних ShortDateFormat (короткий формат дати), LongDateFormat (довгий формат дати), ShortTimeFormat (короткий формат часу) і LongTimeFormat (довгий формат часу). Повідомляти ці змінні не потрібно, вони вже присутні в системі й містять рядок з відповідними форматами (про формати ми поговоримо нижче). Так, для України у Windows короткий формат дат установлений, як dd.MM.yyyy (тобто, день із першим нулем, місяць і рік у чотирьох цифрах). Короткий формат часу буде h:nn (годину без першого нуля й хвилини із першим нулем), а довгий - h:nn:ss (година без першого нуля, хвилини й секунди із першим нулем). При бажанні, ці системні змінні можна змінити прямо в програмі, наприклад:

```
begin
  ShortDateFormat:= ‘dd.mm.yy’;
```

У результаті, рік буде виводитися у двох цифрах, а не в чотирьох, тобто замість «2018» буде «18». Однак я не рекомендую це робити. Коли вам знадобиться (а це обов'язково відбудеться) вивести дату й час у бажаному вам, або вашому замовникові, вигляді, то краще скористатися функціями, що реалізують форматний висновок, мова про які піде нижче. Тим більше що це гарантує правильне виведення дати-часу на будь-якій Windows, у будь-якій країні.

FormatDateTime() - функція виводить задану дату й час у необхідному вам форматі, що не залежить від того, яка Windows (російська або англійська) у вас встановлена. Синтаксис функції наступний:

function FormatDateTime(const Format: String; DateTime: TDateTime): String;

У функцію варто передати два параметри: рядок з необхідним форматом, і дата-час (або тільки дату, або тільки час) у форматі TDateTime. Функція поверне ці дані в зазначеному форматі. Наприклад:

```
var
  s: String;
begin
  s:= FormatDateTime('dd mmmm yyyy - hh:nn:ss', Now);
```

У результаті в рядку змінної s з'явиться рядок типу «21 Квітня 2018 - 11:41:56». Але це в теорії. На практиці замість українських літер виходять знаки «?» - позначаються проблеми сумісності Lazarus з різними кодуваннями Windows. Что-ж, рішення є й із цієї ситуації: потрібно конвертувати отриманий рядок функцією SysToUTF8(), що одержує ANSI-рядок, і гарантовано повертає формат UTF8, у якому, як ми знаємо, і працює Lazarus:

```
s:= SysToUTF8(FormatDateTime('dd mmmm yyyy - hh:nn:ss', Now));
```

Зазначений програмістом формат може містити символи, які представлені в таблиці нижче:

Таблиця 12.1. Символи формату функції FormatDateTime()	
Символи	**Опис**
c	Відображає короткий формат дати й довгий формат часу, тобто, Windows це буде у вигляді «дд. мм.гггг чч:мм:сс», наприклад: «21.04.2018 11:54:56».
d	Відображає день місяця (число) без провідного нуля (1-31)
dd	Відображає день місяця (число) із провідним нулем (01-31)
ddd	День тижня в скороченому виді, наприклад: Пн, Вт для російськомовної Windows.
dddd	День тижня в повному виді, наприклад, понеділок, вівторок.
ddddd	Короткий формат дати, наприклад: «21.04.2018»
dddddd	Довгий формат дати, наприклад: «21 Квітня 2018 р. « - для виведення дати в документ або на екран саме те.
m	Номер місяця без першого нуля (1-12).
mm	Номер місяця із першим нулем (01-12).
mmm	Коротка назва місяця, наприклад: «квіт».
mmmm	Повна назва місяця, наприклад: «Квітень».
y або yy	Рік із двох цифр, наприклад: «18»
yyyy	Рік із чотирьох цифр, наприклад: «2018»
h	Година без першого нуля (0-23)
hh	Година із першим нулем (00-23)
n	Хвилини без першого нуля (0-59)
nn	Хвилини із першим нулем (00-59)
s	Секунди без першого нуля (0-59)
ss	Секунди із першим нулем (00-59)
z	Виведення мілісекунд (0-999). Наприклад, формат «hh:nn:ss:z» виведе час приблизно так: «12:36:01:745»
t	Короткий формат часу, наприклад: «12:16»
tt	Довгий формат часу, наприклад: «12:16:53»
am/pm	Час буде виводитися так, як прийнято на заході: 12-годинне подання, після якого вказується am (до полудня) або pm (після полудня). Ці символи додаються до необхідного

	формату, наприклад, формат «t am/pm» виведе час у вигляді «12:21 pm»
a/p	Аналогічно попередньому, але замість am (до полудня) або pm (після полудня) будуть виходити a або p.

DateTimeToString() - процедура, що перетворить зазначені дату-час у рядок з використанням зазначеного формату, і результат поміщає в строкову змінну, котра вказується, як параметр за посиланням. Зазначений формат використає символи з таблицы 12.1, як і FormatDateTime().

Синтаксис:

procedure DateTimeToString(var Result: String; const Format: String; DateTime: TDateTime);

Приклад застосування:

```
Var

  s: String;
begin
  DateTimeToString(s, ‘dddddd’, Now);
  ShowMessage(SysToUTF8(s)); //отут необхідно перетворити рядок  в UTF8
```

Як бачите, для коректного виведення українського тексту нам довелося перетворити його з ANSI в UTF8 прямо всередині оператора ShowMessage(), тому що в процедурі DateTimeToString використати SysToUTF8() неможливо. Важливо, що можна було спочатку перетворити рядок, а вже потім вивести його:

```
  …
  DateTimeToString(s, ‘dddddd’, Now);
  s:= SysToUTF8(s);
  ShowMessage(s);
```

Але так довелося б робити зайвий рядок коду.

DayOfWeek() - функція приймає як параметр дату, і повертає номер дня тижня цієї дати у вигляді цілого числа. Причому робить це вона так, як прийнято на заході: першим днем тижня вважається

неділя, останнім - субота. Так що понеділок буде другим днем тижня. Синтаксис:

function DateOfWeek(Date: TDateTime): Integer;

Якщо ж вам буде потрібно вивести день тижня словом, а не числом, то можна діяти, наприклад, так:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: String;
begin
  case DayOfWeek(Now) of
    1: s:= ‘Неділя’;
    2: s:= ‘Понеділок’;
    3: s:= ‘Вівторок’;
    4: s:= ‘Середа’;
    5: s:= ‘Четвер’;
    6: s:= ‘П’ятниця’;
    7: s:= ‘Субота’;
  end; //case

   ShowMessage(‘Сьогодні ‘ + FormatDateTime(‘ddddd’, Now) + ‘, ‘ + s);
end;
```

Спочатку за допомогою case ми отримали в s необхідний день тижня. Потім ми вивели повідомлення, типу «Сьогодні 21.04.2018, Понеділок».

DecodeDate() - дуже корисна процедура. Вона розбиває зазначену дату окремо на рік, місяць і число, повертаючи ці дані у вигляді цілих чисел. Синтаксис:

procedure DecodeDate(Date: TDateTime; var Year, Month, Day: Word);

Приклад застосування:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Year, Month, Day: Word;
```

```
  s: String;
begin
  DecodeDate(Now, Year, Month, Day);
  ShowMessage('Рік - ' + IntToStr(Year) + #13 +
                'Місяць - ' + IntToStr(Month) + #13 +
                'День - ' + IntToStr(Day));
end;
```

У даному прикладі ми одержали у вигляді чисел рік, місяць і число поточної дати у відповідні змінні. Потім вивели їх на екран, перетворивши за допомогою IntToStr() у рядки, кожне дане на окремому рядку.

EncodeDate() - функція, що є протилежністю DecodeDate. Вона навпаки, збирає дату з окремих чисел, і повертає її, перетворивши у формат TDateTime. Синтаксис:

```
function EncodeDate(Year, Month, Day: Word): TDateTime;
```

Приклад застосування:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Year, Month, Day: Word; //для вказівки року, місяця й дня
  dt: TDateTime; //для одержання дати
  s: String;
begin
  //вказуємо рік, число й день:
  Year:= 1999;
  Month:= 3;
  Day:= 9;
  //одержуємо дату:
  dt:= EncodeDate(Year, Month, Day);
  //тепер виводимо її:
  ShowMessage('Указали ' + SysToUTF8(FormatDateTime('dddddd', dt)));
end;
```

Коментарі досить докладні, щоб ви розібралися з кодом. У результаті виконання прикладу буде виведене повідомлення «Вказали

9 Березень 2005 р.». Ви можете вказати й інші рік, число й день, тільки пам’ятайте, що вказувати їх потрібно правильно. Наприклад, не можна в Лютому вказувати 30 днів, а у Квітні 31. І ще: якщо ви спробуєте використати дату, меншу ніж 01.01.0001, або більшу ніж 31.12.9999, то скористатися цими функціями вам не вдасться, програма виведе помилку.

DecodeTime() - Процедура розбиває зазначений час окремо на години, хвилини, секунди та мілісекунди, повертаючи ці дані у вигляді цілих чисел. Синтаксис:

procedure DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Word);

Оскільки процедура працює аналогічно DecodeDate, наводити приклади ми не будемо.

EncodeTime() - а ця функція працює аналогічно EncodeDate(), і збирає час із окремих чисел, повертаючи їх у форматі TDateTime. Синтаксис:

function EncodeTime(Hour, Min, Sec, MSec: Word): TDateTime;

IsLeapYear() - функція, що іноді буває потрібна. Вона визначає, чи був рік високосним, чи ні. Рік вказується у вигляді цілого числа. Функція повертає True, якщо зазначений рік високосний, і False у протилежному випадку. Синтаксис:

function IsLeapYear(Year: Word): Boolean;

Приклад застосування:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Year: Word;
begin
  //вказуємо рік:
  Year:= 1917;
  //високосний чи ні?:
  if IsLeapYear(Year) then ShowMessage(‘Зазначений високосний рік’)
  else ShowMessage(‘Зазначений невисокосний рік’);
```

```
end;
```

Наостанок залишилися функції перетворення дати-часу з рядка в TDateTime, вони теж дуже часто бувають потрібні.

StrToDateTime() - функція перетворить зазначені у вигляді рядків дату й час у формат TDateTime. Синтаксис:

```
function StrToDateTime(const S: string): TDateTime;
```

Приклад:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  dt: TDateTime;
begin
  dt:= StrToDateTime('21.10.2013 14:25:30');
  ShowMessage('Указали ' + FormatDateTime('c', dt));
end;
```

Зверніть увагу ось на що. У цій функції рядок потрібно встановити правильно, залежно від мови вашої Windows. Дата повинна вказуватися в короткому форматі, від часу дата відокремлюється пробілом. Сепаратором (роздільником частин дати) може бути як крапка - у російськомовній Windows, так і слеш «/» в англійській, це залежить від змінної DateSeparator. Три частини дати (y-рік, m-місяць і d-день) можуть бути у вигляді y/m/d, m/d/y, d/m/y - порядок цих частин залежить від змінної ShortDateFormat.

StrToDate() - функція перетворить зазначену у вигляді рядка дату у формат TDateTime, час при цьому не вказується. Синтаксис:

```
function StrToDate(const S: string): TDateTime;
```

StrToTime() - функція перетворить зазначений у вигляді рядка час у формат TDateTime, дата при цьому не вказується. Синтаксис:

```
function StrToDate(const S: string): TDateTime;
```

Функції діють аналогічно StrToDateTime

Тема 13. Масиви прості, багатомірні й динамічні

Мета лекції: вивчення простих, багатомірних і динамічних масивів та функцій для роботи з ними

План

1. Масив
2. Багатовимірний масив
3. Динамічний масив

1. Масив

У практиці програмування нерідко виникає необхідність обробити велику кількість однотипних даних. Наприклад, розподілити тижневий прибуток кафе по днях. Що для цього потрібно? Використати змінні типу currency, що застосовуються саме для таких випадків. І скільки змінних нам потрібно? Ну, ясне питання - по змінній на день, усього сім штук. Давайте подивимося, як таке завдання можна було б вирішити на Lazarus (приклад академічний, виконувати його не потрібно):

```
var
  day1, day2, day3, day4, day5, day6, day7: currency;
```

Досить довге оголошення вийшло, правда? А якби потрібно було порахувати не тижневий прибуток, а місячний? Або річний? Тепер припустимо, що нам потрібно порахувати суму цього прибутку. Це виглядало б приблизно так:

```
pribil:= day1 + day2 + day3 + day4 + day5 + day6 + day7;
```

А якби довелося рахувати місячний або річний прибуток? Якої довжини оператор тоді б вийшов? Але ж часто бувають і більш складні розрахунки, які потрібно провести з кожною однотипною змінною.

На щастя, у мовах програмування існує такий інструмент, як **масиви**.

Масив - це формальне об’єднання декількох однотипних змінних у єдиний набір, з одним загальним ім’ям.

Фактично, виходить ланцюжок однотипних змінних під загальним ім’ям, де в кожної такої змінної є власний номер в комірці - **індекс**. Такі змінні називаються **елементами масиву**, кожний елемент має свій номер. Оголошується масив у розділі змінних у такий спосіб:

```
var
  a: array[1..7] of currency;
```

Ключове слово array вказує, що це буде масив. Числа у квадратних дужках розділяються двома крапками й позначають діапазон індексів. Так, [1..7] означає, що індекси масиву будуть від 1 до 7, тобто, 7 штук. Після ключового слова of вказується тип масиву, такий же тип буде й у кожного елемента масиву. У результаті, компілятор створює в оперативній пам'яті сім осередків по 8 байт кожна (тип currency займає 8 байт):

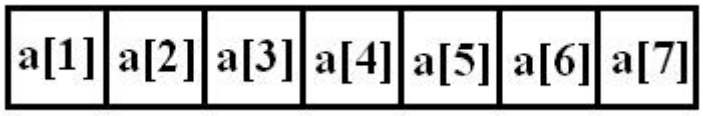


Рис. 13.1. Розміщення елементів масиву a у пам'яті

Тип даних, використовуваний у масивах, так само може бути кожним. У багатьох інших мовах програмування діапазон індексів завжди починається з 0. У мовах, заснованих на Паскалі, діапазон може починатися з будь-якої цифри: з 0, 1, або 438, якщо вам так подобається. Хоча програмісти звичайно починають діапазон або з 0, або з 1, залежно від обставин. Приклад оголошення різних масивів:

```
var
  name: array[0..30] of string;
  year: array[1..2013] of integer;
  man: array[10..25] of boolean'
```

Добре, масиви ми оголосили. Що з ними робити далі? Ми можемо звернутися до будь-якого елемента масиву, вказавши після імені у квадратних дужках індекс цього елемента. Так, ми можемо привласнити значення елементу або одержати його значення. Інакше кажучи, ми можемо працювати з елементами масиву, як з окремими змінними. Приклади:

```
…
a[1]:= 3520.45;
a[2]:= a[1] + 12.4;
ShowMessage('Прибуток за п'ятницю склала  ' + FloatToStr(a[5]));
```

…

Ну, добре, ми навчилися створювати й використовувати масиви. Але які ж переваги в масивів на відміну від простих однотипних змінних? Про першу таку перевагу ми вже говорили, що просто можна оголосити масив з елементами, розрахованими на рік:

```
var
  a: array[1..365] of currency;
```

Спробуйте оголосити 365 окремих змінних! Друга перевага набагато важливіша: обробляти елементи масиву набагато простіше, ніж окремі змінні. Наприклад, нам потрібно створити програму, яка б могла зробити переведення температур зі звичних нам Цельсіїв у незвичні американські Фаренгейти. І нехай діапазон можливих температур буде від -100 градусів до +200. Ідея така: користувач вводить потрібну температуру, а програма виводить результат на екран. Звичайно, простіше було б робити необхідний розрахунок у момент, коли користувач введе свої дані, і натисне кнопку «**розрахувати**». Але ми зробимо по-іншому: ми спочатку зробимо всі розрахунки й помістимо результати в масив, а потім будемо просто виводити потрібні дані. Масив зробимо глобальним, щоб заповнювати його в одній події, а виводити дані в іншій. Відкриваємо **Lazarus** з новим проєктом, зберігаємо його відразу в папку **13-01** там, де в нас зберігаються всі навчальні проєкти. Проєкт збережемо під ім’ям **MyTemp**, а модуль головної форми, як звичайно - **Main**. Займемося головною формою, змініть в ній наступні параметри:

- Name = fMain
- Caption = Переклад температурних шкал
- BorderStyle = bsDialog
- Position = poMainFormCenter

Далі, перейдіть на вкладку **Additional Палітри компонентів**, знайдіть і встановіть на форму компонентів TStaticText, що призначений для виведення на екран пояснювального тексту, у тому числі й многострочного. Якщо ви забули, то знайти потрібний компонент просто: потрібно просто підвести покажчик миші до компонента, і через короткий час з’явиться спливаюча підказка з ім’ям компонента.

У принципі, для цих цілей можна використати й просту мітку TLabel, але було б цікавіше познайомитися й з іншими компонентами.

В **Інспекторі об'єктів** у властивості Caption натисніть на кнопку «**…**»праворуч від властивості:

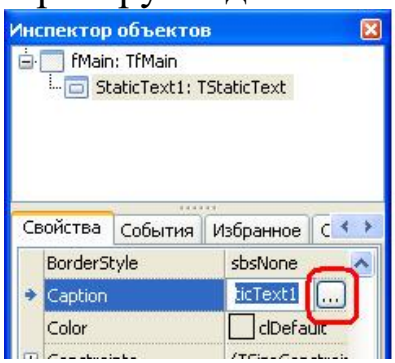


Рис. 13.2. Властивість Caption компонента TStaticText

У редакторі, що відкрився, напишіть наступний текст:

Вкажіть температуру в Цельсіях
від -100 до 200 градусів:

Саме так, у два рядки. Потім змініть розміри компонента на формі, щоб текст відображався точно в такому вигляді, як ми ввели. І ще: у властивості Alignment виберіть taRightJustify, щоб текст вирівнювався по правому краю.

Тепер перейдіть на вкладку **Misc** палітри компонентів, знайдіть там і встановіть на форму, правіше TStaticText, компонент TSpinEdit, що призначений для введення користувачем цілих чисел. Цей компонент схожий на простий TEdit, але праворуч від нього є стрілки вгору й вниз, якими користувач може збільшувати або зменшувати зазначене в рядку число. Використання цього компонента гарантує нам, що користувач не зможе ввести в нього нічого, крім цілого числа. А щоб ще гарантувати правильний діапазон цих чисел, у властивості MaxValue (максимальне можливе значення) встановіть 200, а в MinValue (мінімальне можливе значення) встановіть -100 (мінус сто). Тепер користувач не зможе ввести в це поле нічого, крім цілих чисел. Якщо ж він спробує перевищити максимальне значення, то компонент автоматично встановить число 200 - максимально можливе. І навпаки, якщо вказати число менше, ніж -100, компонент установить -100, так що нам не потрібно турбуватися про помилки. Крім того, у цьому компоненті пізніше нам знадобляться ще дві властивості: Text (встановлене значення у вигляді рядка), і Value (встановлене значення у вигляді цілого числа).

За замовчуванням, компонент недостатньо широкий, тому встановіть в його властивості Width значення 75. І ще: нам доведеться

звертатися до компонента по імені, а SpinEdit1 - це занадто довге ім’я. Змініть його властивість Name на SE1.

Ну й нарешті, встановіть на форму просту кнопку, у властивості Caption якої напишіть **Розрахувати температуру**. Напис на кнопці не вміститься, тому кнопку потрібно подовжити. Розташуєте компоненти на формі приблизно так:

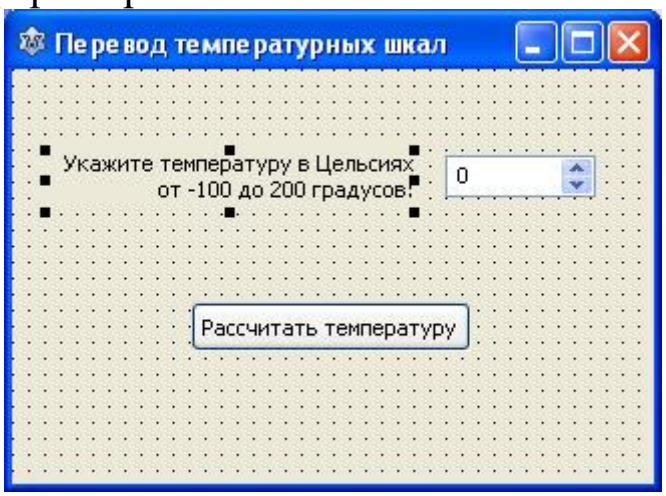


Рис. 13.3. Зовнішній вигляд форми

Тепер займемося попередніми розрахунками. Для початку нам потрібно зробити глобальний масив дійсних чисел - температура за Фаренгейтом вже не буде цілим числом. Значення по Фаренгейту обчислюються так: береться температура за Цельсієм, множиться на 9/5, потім до результату додається 32. Тому в глобальному розділі var, відразу над ключовим словом implementation повідомляємо масив atemp:

```
var
  fMain: TfMain;
  atemp: array[-100..200] of real;  //наш глобальний масив

implementation
```

Оскільки це глобальний масив, він буде створений відразу ж при завантаженні програми, і буде існувати, поки програма не завершить свою роботу. Тепер займемося заповненням цього масиву.

Виділіть форму, клацнувши по її вільному місцю, потім в **Інспекторі об’єктів** перейдіть на вкладку **Події**. Нам потрібна подія OnCreate - вона виникає тільки один раз, при створенні форми. Клацніть двічі по цій події, щоб згенерувати її. Код події наступний:

```
procedure TfMain.FormCreate(Sender: TObject);
var
  i: smallint;  //лічильник для циклу
begin
  for i:= -100 to 200 do
    atemp[i]:= i * 9/5 + 32;
end;
```

От тепер, нарешті, ми зможемо оцінити всю принадність використання масивів! Давайте розберемо цей код. У розділі var ми оголосили змінну і типу smallint. Це - лічильник для циклу for, що ми використали в коді. Тому що діапазон потрібних нам значень розташовується від -100 до +200, тип цілих чисел smallint – підходить найкраще. А тепер зверніть увагу на код:

```
  for i:= -100 to 200 do
    atemp[i]:= i * 9/5 + 32;
```

Це - сама головна «фішка» масивів - обробка всього масиву, яким би великим він не був, у простому циклі. Спочатку лічильник і приймає значення -100. Всередині циклу ми привласнюємо значення одному елементу масиву. Якщо ми замінимо змінну на її значення, як це робить компілятор у процесі роботи, то одержимо наступне:

```
    atemp[-100]:= -100 * 9/5 + 32;
```

Таким чином, ми вираховуємо переведення температури із Цельсія у Фаренгейти для конкретної температури: -100 градусів за Цельсієм, і отриманий результат привласнюємо елементу масиву atemp[-100]. Потім цикл повторюється, і збільшується на одиницю, і вже дорівнює -99. Нове значення привласнюється новому елементу масиву, і так до кінця циклу. Як ви думаєте, скільки елементів масиву буде заповнено в цьому циклі? Хто сказав 300?! А про нуль ви забули? 301 раз буде працювати цикл, і 301 різне значення буде привласнено 301 елементу масиву. А тепер уявіть, як це виглядало б для окремих змінних:

```
t_100:= -100 * 9/5 + 32;
t_99:= -99 * 9/5 + 32;
```

```
…
t200:= 200 * 9/5 + 32;
```

Замість двох рядків циклу ми одержали б 301 рядок коду з однаковими обчисленнями!

Але повернемося до нашого проєкту, адже нам ще потрібно вивести на екран результат перекладу температур. Згенеруємо для кнопки **Button1** подію OnClick. Її код буде зовсім простий:

```
procedure TfMain.Button1Click(Sender: TObject);
begin
  ShowMessage(SE1.Text + ‘ Цельсия = ‘ +
                    FloatToStr(atemp[SE1.Value]) + ‘ Фаренгейта’);
end;
```

Тут у нас усього один оператор - процедура ShowMessage(). Вона формує виведене повідомлення з 4 частин: SE1.Text - це те значення, що встановив користувач у компоненті TSpinEdit у вигляді рядка. Потім ми вставляємо в повідомлення текст ‘ Цельсія = ‘. Зверніть увагу на пробіли спочатку й наприкінці цього шматочка - без них слова будуть злипатися, а це некрасиво. Далі, ми додаємо до рядка наступний код:

```
FloatToStr(atemp[SE1.Value])
```

Тут SE1.Value - це те значення, що встановив користувач, але вже у вигляді цілого числа! Припустимо, користувач встановив значення 35, тоді цей код буде таким:

```
FloatToStr(atemp[35])
```

В елементі масиву atemp[35] у нас уже зберігається переведене у Фаренгейти значення температури 35 за Цельсієм. А функція FloatToStr() тільки переведе це значення в рядок. Таким чином, для значення 35 сформується повідомлення: «35 Цельсія = 95 Фаренгейта». Збережіть проєкт, скомпілюйте його й переконаєтеся, що програма коректно працює в зазначених межах температур, а перевищити їх користувач не зможе, якщо й захоче, як не зможе ввести нічого іншого, крім цілого числа.

2. Багатомірний масив

Масиви, які ми дотепер розбирали, були одномірними - візуально їх можна представити, як ланцюжок комірок або один рядок таблиці. Однак бувають ситуації, коли цього недостатньо. Припустимо, потрібно зберегти якусь таблицю. Одномірний масив з таким завданням не впорається. Для цього існують **багатомірні масиви**, які мають розмірність від двох і більше. Наприклад, оголосити двомірні масиви з 5 рядків і 10 колонок можна в такий спосіб:

```
var
  a: array[1..5, 1..10] of string;
  b: array[0..4, 0..9] of integer;
```

Тобто, у квадратних дужках через кому вказується діапазон спочатку рядків, а потім колонок. Візуально, такий масив можна було б представити так:

Таблиця 13.1. Подання двомірного масиву a

1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10
2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9	2,10
3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	3,10
4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8	4,9	4,10
5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9	5,10

Звернутися до окремого елемента масиву можна двома способами:

```
a[2, 9]:= ‘Рядок 1’;  //1-й спосіб
a[3][10]:= ‘Рядок 2’;  //2-й спосіб
```

У першому випадку ми звертаємося до елемента в другому рядку, дев’ятого стовпчика. У другому - до третього рядка, десятого стовпчика. Перший спосіб звертання до елемента здається компактнішим, його ми й будемо використовувати надалі.

Доробимо наш проєкт. Що, якщо ми будемо конвертувати нашу температуру не тільки у Фаренгейти, але й у Кельвіни (одиниця термодинамічної температури в СІ - Міжнародній Системі Одиниць)? Перерахування Цельсія в Кельвіни ще простіше, ніж у Фаренгейти: потрібно до температури за Цельсієм додати 273,15. Для цього нам буде потрібно замість одномірного масиву оголосити двомірний. Змініть код над службовим словом implementation:

```
atemp: array[1..2, -100..200] of real;
```

У першому рядку в нас будуть Фаренгейти, у другому - Кельвіни. А стовпчики ми відведемо під необхідний діапазон. Далі, нам буде потрібно змінити цикл у події OnCreate форми:

```
procedure TfMain.FormCreate(Sender: TObject);
var
  i: smallint;
begin
  for i:= -100 to 200 do begin
    atemp[1, i]:= i * 9/5 + 32;  //Фаренгейти
    atemp[2, i]:= i + 273.15;  //Кельвіни
  end;
end;
```

Зверніть увагу: оскільки нам у циклі потрібно виконати не один оператор, а два, нам довелося вкласти їх у програмні дужки begin...end, зробивши складовій оператор. Далі, все просто: для рядка 1 нашого масиву ми розраховуємо Фаренгейти, як минулого разу. А для рядка 2 уже Кельвіни.

Але нам буде потрібно переробити й подію кнопки OnClick:

```
procedure TfMain.Button1Click(Sender: TObject);
begin
  ShowMessage(SE1.Text + ‘ Цельсия буде дорівнює:’ + #13 +
            FloatToStr(atemp[1, SE1.Value]) + ‘ Фаренгейта’ + #13 +
            FloatToStr(atemp[2, SE1.Value]) + ‘ Кельвіна’);
end;
```

З кодом, сподіваюся, ви розберетеся самостійно? Отже, якщо користувач введе, наприклад, 36 градусів Цельсія, то натиснувши на кнопку, отримаємо наступне повідомлення:

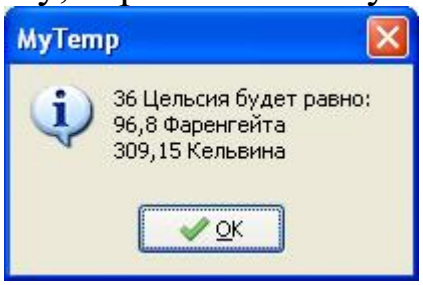


Рис. 13.4. Згенероване програмою повідомлення

Але розмірність масиву може бути й більшою. Тривимірний масив візуально можна представити в такий спосіб:

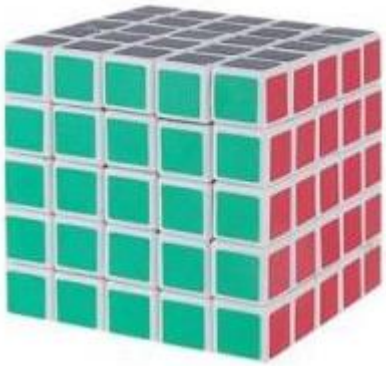


Рис. 13.5. Візуальне подання тривимірного масиву

Оголосити подібний масив можна було б так:

```
var
  a: array[1..5, 1..5, 1..5] of integer;
```

А звертатися до окремих елементів так:

```
a[2, 3, 4]:= 12;
```

Як би виглядав чотирьохмірний масив візуально взагалі неможливо представити. У вас їсти можливість працювати хоч із десятимірним масивом, однак у практиці програмування звичайно використовують одно- та двомірні масиви, і дуже рідко виникає потреба в тривимірному. Не забувайте про один з основних принципів програмування, що самі програмісти називають **KISS** (Keep It Simple, Stupid - будь простішим, дурнику). Цей принцип має на увазі, що код програми не варто ускладнювати без нестатку - складний код важче сприймається самим програмістом, більше навантажує процесор, а помилки часу виконання (run time errors - помилки, що виникають під час виконання програми, звичайно ці логічні помилки) у такому коді складніше відслідковувати.

3. Динамічний масив

У практиці програмування інший раз трапляється працювати з масивами, розмірність яких заздалегідь невідома - вона може залежати від даних, введених користувачем, від поточних параметрів, від обсягу оброблюваної інформації, так ще багато від чого. Програміст у цих

випадках не може вказати конкретні границі індексів масиву. У таких випадках використовують **динамічні масиви**.

Динамічними називаються масиви, при оголошенні яких розмір не вказується. А під час виконання програми розмір такого масиву можна змінювати.

Оголошуються динамічні масиви також у розділі var, у такий спосіб:

```
var
  da: array of integer;
```

Як бачите, ми вказуємо тільки тип масиву, але не його розмірність. Пам'ять під оголошений масив при цьому не виділяється. Надалі з динамічним масивом ми можемо робити різні операції:

- встановлювати розмірність (довжину) масиву, збільшувати або зменшувати цю розмірність (тому масив і динамічний),
- дізнаватися вищі й нижчі границі діапазону індексів масиву,
- привласнювати значення окремим елементам або навпаки,
- зчитувати ці значення.

Познайомтеся з функціями, які все це проробляють.

SetLength() - встановлює розмір масиву. Синтаксис:

SetLength(<масив>, <довжина>);

Приклад:

```
SetLength(da, 5);  //установили розмір масиву в 5 елементів
```

Отут варто обмовитися, що на відміну від звичайного масиву, початковий індекс якого може бути кожним, індексація динамічного масиву завжди починається з нуля. Тобто, індексація елементів масиву в нашому прикладі буде від 0 до 4 - всього п'ять елементів. Як тільки ми встановили розмір масиву, у пам'яті виділяється місце під нього. У нашому прикладі буде відведено по 4 байти (тип integer) для 5 елементів масиву. Звертатися до цих елементів можна звичайним чином:

```
da[0]:= 5;
da[4]:= da[0];
```

Тепер ми можемо додати елемент до масиву, встановивши для нього новий розмір:

```
SetLength(da, 6);
```

У цьому випадку, один порожній елемент буде доданий у кінець масиву. Якщо ж навпроти, ми зменшимо його розмір:

```
SetLength(da, 4);
```

то масив усікається. Якщо останній елемент містив якісь дані, вони будуть загублені. Звільнити пам’ять масиву можна, привласнивши йому нульову розмірність:

```
SetLength(da, 0);
```

Також звільнити пам’ять можна, привласнивши масиву значення nil (нічого, порожньо):

```
da:= nil;
```

Втім, цього можна й не робити: по закінченні процедури, де даний масив був оголошений, пам’ять буде звільнена автоматично.

Динамічні масиви можуть бути й багатомірними. Наприклад, двомірний масив цілих чисел можна оголосити так:

```
var
  a: array of array of integer;
```

Потім такому масиву можна привласнити розмірність, наприклад, 4 на 5:

```
SetLength(a, 4, 5);
```

Це буде, як якби ми оголосили простий масив:

```
var
  a2: array[0..3, 0..4];
```

Length() - повертає розмір динамічного масиву, тобто, кількість його елементів. Наприклад, нам потрібно подивитися розмір масиву, і якщо він порожній, то додати в нього один елемент:

```
if Length(da) = 0 then SetLength(da, 1);
```

Low() - повертає нижній індекс масиву, у динамічних масивів це завжди нуль.

High() - повертає верхній індекс масиву, але це не та ж саме, що кількість елементів. Якщо в масиву 5 елементів, то Length() поверне 5, а High() поверне 4, тому що індексація починається з нуля.

Приклад обходу масиву від першого до останнього елемента:

```
for i:= Low(da) to High(da) do
  da[i]:= i * i;
```

У прикладі ми кожному елементу привласнюємо квадрат його індексу. Давайте попрацюємо з динамічними масивами на практиці. Щоб не псувати наш конвертер температур, закрийте його й відкрийте новий проєкт. Мудрувати ми не будемо - нам потрібна тільки форма й проста кнопка на ній. Перейменовувати теж нічого не будемо, це ж просто приклад. Збережіть новий проєкт у папку **13-02**.

Згенеруйте процедуру OnClick для кнопки, її код буде таким:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;  //для запиту
  da: array of string;  //динамічний масив рядків
  i: integer;  //лічильник
begin
  //довідаємося в користувача, скільки рядків робити:
  s:= '0';
  InputQuery('Привіт!', 'Скільки рядків бажаєте створити?', s);
  //якщо нуль, то нічого не робимо, виходимо із процедури:
  if StrToInt(s) = 0 then Exit;

  //інакше встановлюємо зазначену розмірність масиву:
  SetLength(da, StrToInt(s));
  //тепер обійдемо весь масив, встановлюючи в його елементи значення:
  for i:= Low(da) to High(da) do
    da[i]:= 'Рядок №' + IntToStr(i + 1);

  //тепер знову обійдемо масив і створимо повідомлення з його рядків:
  s:= '';
  for i:= Low(da) to High(da) do
    s:= s + da[i] + #13;
```

```
  ShowMessage(s);

end;
```

Код містить досить докладні коментарі. Спочатку в рядковій змінній s ми одержуємо кількість бажаних рядків. Тому що функція InputQuery() повертає тільки рядок, нам доведеться отримувати ціле число у вигляді рядка - ніякої перевірки ми тут не робимо, це ж тільки демонстрація роботи з динамічним масивом. Тому правильність числа, що вводить, залишимо на совісті користувача.

Далі, ми перевіряємо - не чи нуль це? Якщо рядків створювати не потрібно, ми просто виходимо із процедури, пропускаючи весь інший код. Якщо ж користувач ввів якусь цифру, причому вона повинна бути більшою за нуль, то ми встановлюємо зазначену розмірність масиву:

```
SetLength(da, StrToInt(s));
```

Далі, за допомогою циклу for ми обходимо весь масив від першого до останнього елемента. У кожний елемент ми записуємо рядок: «Рядок №», додаємо номер елемента й перехід на новий рядок. Зверніть увагу, для першого рядка він буде дорівнює 0, тому ми вказуємо i + 1:

```
  for i:= Low(da) to High(da) do
    da[i]:= ‘Рядок №’ + IntToStr(i + 1);
```

Потім ми очищаємо рядкову змінну s, щоб вдруге скористатися нею, зібрати в ній повідомлення. Що й робимо в наступному циклі:

```
  for i:= Low(da) to High(da) do
    s:= s + da[i] + #13;
```

Виходить, у процедурі ми двічі обходили масив: спочатку щоб занести в нього значення, потім щоб обрахувати його. Тепер збережіть проєкт, скомпілюйте його й запустіть на виконання. Майте на увазі, якщо ви вкажете занадто велике число, то вікно повідомлення не вміститься в екрані - у цьому випадку закрити його можна буде стандартними клавішами **<Alt + F4>**.

Тема 14. Колекції (масиви) рядків і компоненти для них

1. Компонент TMemo
2. Списки вибору TListBox и TComboBox
3. Властивості та методи типу TSrings

1. Компонент TMemo

Дотепер для надання користувачеві можливості ввести текст ми використовували компоненти начебто TEdit, за допомогою яких можна ввести тільки один рядок тексту. Компонент TMemo - це, по суті, цілий текстовий редактор! Не вірите? Давайте подивимося. Відкрийте **Lazarus** з новим проєктом. Відразу ж перейменуйте форму в **fMain** (властивість Name), у властивості Caption напишіть Мій Блокнот. Збережіть проєкт як **MyNotebook** у папку **14-01**, а модулю дайте ім'я Main. Встановимо деякі інші параметри форми. Насамперед, форма буде містити редактор текстів, тому вона не повинна бути маленькою. Давайте зробимо наступні налаштування у властивостях форми:

Height = 450
Width = 750
Position = poDesktopCenter

Далі, на вкладці **Standard Палітри компонентів** знайдіть компонент TMemo:

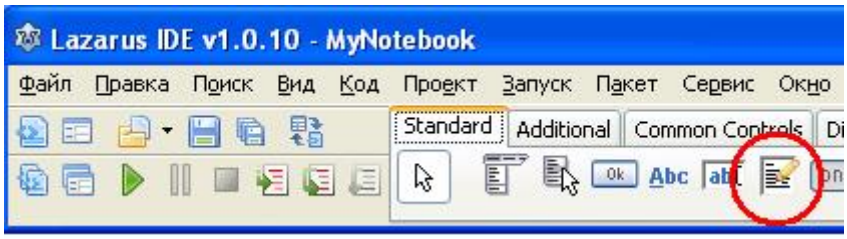


Рис. 14.1. Компонент TMemo

Встановіть його на форму. Тепер давайте розглянемо властивості компонента TMemo - уже відомі по інших компонентах властивості ми розглядати не будемо, зупинимося тільки на нових й специфічних для TMemo (компонент повинен бути виділений).

BorderStyle	- Включає/відключає обрамлення навколо компонента, і може бути: • bsNone - немає обрамлення • bsSingle - є обрамлення (за замовчуванням)

CharCase	- задає регістр виведених у компоненті символів. Може бути: • ecLowerCase - всі символи рядкові • ecNormal - символи такі, якими їх вводить користувач (за замовчуванням) • ecUppercase - всі символи прописні
Color	- кольори текстового поля. Ви можете задавати кольори як текстового поля, так і шрифту, однак майте на увазі, що серйозні програми не робляться з яскравою кольоровою гамою, тому бажано компонентам залишати їхні стандартні кольори.
Font	- шрифт тексту в компоненті. Властивість має кнопку «**…**»,за допомогою якої його можна розкрити й настроїти шрифт: назва шрифту, його розмір, накреслення, кольори й видозміна. Щодо кольорів шрифту тут справедливо те ж зауваження, що й для властивості Color. Накреслення (простий шрифт, курсив, напівжирний, напівжирний курсив) теж без потреби змінювати не варто. А от що стосується назви шрифту і його розміру - їх змінити можна. Справа в тому, що за замовчуванням у компоненті встановлений дрібний шрифт, а в деяких користувачів може бути слабкий зір. Щоб користувач даремно не напружував зір, я звичайно встановлюю шрифт Times New Roman (він виглядає презентабельніше) 12-го розміру. Шрифт виходить досить великим. Ви можете проекспериментувати із різними шрифтами й розмірами й підібрати той, що подобається особисто вам. Тільки ще одне зауваження: якщо ви думаєте, що ваша програма буде працювати й на інших комп’ютерах, то бажано вибирати стандартні шрифти. Якщо ви встановите якийсь екзотичний шрифт, якого в користувача може й не бути, то програма в нього не буде працювати нормально. Користувачеві потрібно буде додатково шукати й встановлювати ваш шрифт, а вони звичайно не терплять таких незручностей, і скоріше віддадуть перевагу якійсь іншій, аналогічній програмі.

HideSelection	- включає або виключає виділення тексту, коли компонент губить фокус. Якщо дорівнює True, то виділення тексту ховається, коли активним стає інший компонент. При False виділення залишається. Звичайно залишають значення за замовчуванням - True.
Lines	- колекція (масив) рядків. Дозволяє звертатися як до всього тексту, так і до його окремих рядків. Це складна властивість, що має свої властивості, методи й події. Методи - це те, що компонент уміє робити. Це ті ж самі процедури й функції. Lines ця властивість, але й у нього є методи, за допомогою яких можна зберігати текст у файл, зчитувати його з файлу, робити ще багато інших корисних речей. Має тип TStrings. Ми повернемося до нього трохи пізніше, фактично, вся ця лекція присвячена роботі з типом TStrings.
MaxLength	- максимальна довжина тексту, що вводить користувачем. За замовчуванням, властивість дорівнює нулю. Це означає, що обмежень на розмір тексту немає.
ReadOnly	- переводиться як «Тільки для читання». За замовчуванням, дорівнює False - користувач може як читати, так і редагувати текст. Якщо встановити в True, користувач не зможе змінювати текст. Так роблять, коли потрібно вивести для користувача багаторядковий текст, наприклад, текст ліцензії або опис програми у вікні «Про програму». Ми залишимо False, тому що зараз нам потрібно, щоб користувач міг редагувати текст.
ScrollBars	- смуги прокручування компонента. Може бути: • ssNone - немає смуг прокручування. • ssHorizontal - горизонтальна смуга прокручування, розташовується по нижньому краї компонента. Неактивна, поки текст рядка вміщається на екрані. Коли він стає більше, смуга стає активною, і користувач може прокрутити текст по горизонталі. • ssVertical - вертикальна смуга прокручування, розташовується по правому краї компонента. Неактивна, поки всі рядки вміщаються на екрані. Коли рядків більше, смуга прокручування стає

	активною, і користувач може прокручувати текст вниз. • ssBoth - присутні як вертикальна, так і горизонтальна смуги прокручування. • ssAutoHorizontal - відрізняється від ssHorizontal тим, що поки текст вміщується у вікні, смуга прокручування невидима. Принаймні, так задумували розробники. На ділі ж, ssAutoHorizontal поводиться так само, як і ssHorizontal - смуга видима, але неактивна, поки текст вміщається у вікні. Будемо сподіватися, що в майбутньому розроблювачі Lazarus виправлять цю помилку. • ssAutoVertical - те ж, що й ssAutoHorizontal, але для вертикальної смуги прокручування. • ssAutoBoth - те ж, що й ssAutoHorizontal, але для обох смуг прокручування.
WantReturns	- теж недопрацьована властивість. По ідеї, якщо її встановити в False, користувач не зможе клавішею **<Enter>** вставляти розриви рядків. Однак нові рядки як і раніше вставлюються клавішею **<Enter>** поза залежністю від налаштувань цієї властивості.
WantTabs	- от ця властивість працює. Якщо воно встановлено в False (за замовчуванням), то користувач не зможе клавішею **<Tab>** вставляти знаки табуляції - ця клавіша буде перемикати фокус із TMemo на інший компонент, якщо він є. Якщо ж властивість встановити в True, то користувач зможе вставляти знаки табуляції. Дозволяти йому вставку чи табуляції ні, залежить від призначення компонента. Якщо вам потрібно просте поле, щоб користувач міг ввести кілька рядків тексту, то табуляцію можна й не дозволяти. Якщо ж вам потрібно зробити повноцінний текстовий редактор, то табуляція необхідна.
WordWrap	- дозволяє або забороняє перенос тексту на новий рядок. Якщо встановлено True (за замовчуванням), то коли текст досягає краю, відбувається автоматичний перехід на новий рядок. Якщо встановлено False, то рядок буде тривати, поки користувач не натисне **<Enter>**. Зазвичай

	властивість залишають в True і встановлюють вертикальну смугу прокручування. Якщо вам захочеться встановити False, подбайте про те, щоб була встановлена й горизонтальна смуга прокручування.

В **Інспекторі об'єктів** відсутня ще одна важлива властивість: Text. Ця властивість має тип String і містить весь текст, всі рядки разом, включаючи символи переходу на новий рядок.

Отже, займемося налаштуванням компонента TMemo. Ім'я залишимо за замовчуванням: Memo1. Властивості Left і Top встановіть в 5. Властивість Height встановіть рівною 400, а Width - 740. Компонент зайняв майже всю форму, залишивши невеликі бордюри й місце для кнопок знизу. Тепер нам потрібно прив'язати компонент до сторін, щоб він автоматично змінював розмір при зміні користувачем розмірів вікна. Для цього розгорніть властивість Anchors (якір) і встановіть в True всі його чотири підвластивості.

Далі, нам якось потрібно забрати текст «Memo1» з компонента, що був доданий у поле автоматично. Для цього виділіть властивість Lines і клацніть по її кнопці «**…**»... Це приведе до відкриття вікна діалогу введення рядків. Зітріть там весь текст і натисніть «ОК» - текст у компоненті пропав.

Тепер встановіть вертикальну смугу прокручування (ScrollBars = ssVertical) і переконаєтеся, що в WordWrap установлено True - компонент буде автоматично переносити текст на нові рядки.

От, власне, і всі налаштування. Тепер нам потрібно домогтися від компонента трьох речей: щоб він зберігав текст у файл, щоб він зчитував текст із файлу, і щоб він очищав весь текст. Для цього нам доведеться скористатися деякими методами властивості Lines компонента. І ще нам будуть потрібні три простих кнопки, для яких внизу саме залишилося місце. Давайте першу кнопку назвемо bSave (у властивості Name), другу - bRead, і третю - bClear. Маленька буква b означає кнопку (button), хоча ви можете винайти й свої правила найменування компонентів. Нам буде потрібно змінити й властивість Caption у цих кнопок: у першій кнопці напишіть Зберегти, у другій - Читати, а в третій - Очистити. Щоб текст вміщався на кнопках просторіше, властивість Width (ширину) кнопок поставте в 90. Тепер ось ще що: за замовчуванням, усе компоненти «привязываются» до верхньої й лівої границь форми. Якщо користувач буде змінювати розміри вікна, то кнопки будуть перебувати на такій же відстані від

лівого таверхнього краю, що й при проєктуванні. А тому якщо наш Memo прив'язаний до всіх сторін, і буде збільшуватися, то закриє собою кнопки. Щоб цього уникнути, виділіть всі кнопки, відкрийте їхню властивість Anchors, akTop переведіть в False, а akBottom навпаки, в True. Тепер наші кнопки прив'язані до лівого й нижнього краю, і завжди будуть на однаковій відстані від Memo.

У вас повинна вийти така форма:

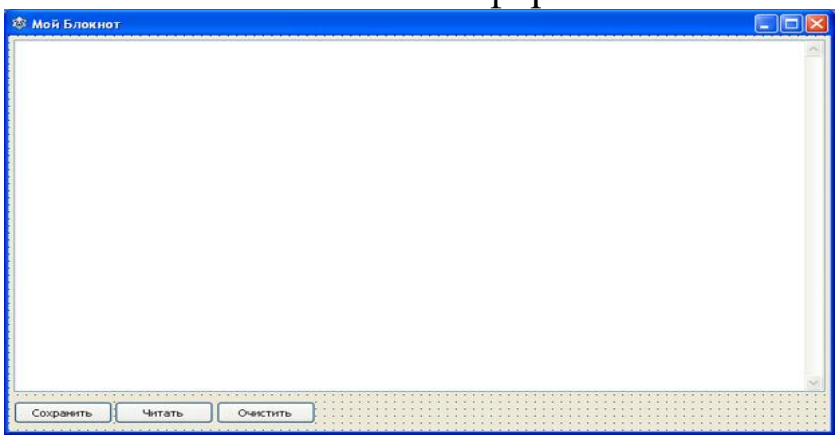


Рис. 14.2. Форма редактора текстів

Залишилося запрограмувати події OnClick для цих кнопок. Згенеруйте їх, двічі клацаючи по кнопках. От код цих подій:

```
procedure TfMain.bSaveClick(Sender: TObject);
begin
  Memo1.Lines.SaveToFile('MyText.txt');
end;

procedure TfMain.bReadClick(Sender: TObject);
begin
  if FileExists('MyText.txt') then
    Memo1.Lines.LoadFromFile('MyText.txt')
  else ShowMessage('Файл MyText.txt не існує');
end;

procedure TfMain.bClearClick(Sender: TObject);
begin
  Memo1.Lines.Clear;
end;
```

Як бачите, події дуже прості, у двох з них всього по одному рядку. Так, у рядку

```
Memo1.Lines.SaveToFile('MyText.txt');
```

ми звертаємося до компонента Memo1, його властивості Lines, і викликаємо метод цієї властивості SaveToFile().

Давайте розберемося з методами властивості Lines компонента TMemo.

SaveToFile() - зберегти текст у зазначений файл, у нашому випадку, це файл **MyText.txt**. Розширення *.txt традиційно використовується для текстових файлів, так що його краще вказувати, хоч це й не є обов'язковим. У прикладі ми вказали просто ім'я файлу, без адреси. Файл буде створений у поточній папці, там же, де й програма. Якщо ви бажаєте вказати якесь конкретне місце, то вказуйте ім'я файлу разом з адресою, наприклад:

```
Memo1.Lines.SaveToFile('C:\MyText.txt');
```

Якщо такого файлу не було, він буде створений. Якщо файл був, він перезапишеться. Це буде звичайний текстовий файл, який можна відкрити будь-яким текстовим редактором.

LoadFromFile() - цей метод навпаки, зчитує текст із зазначеного файлу в компонент. Однак файл повинен існувати насправді, інакше програма викличе помилку. Саме тому спочатку ми робимо перевірку на існування файлу:

```
if FileExists('MyText.txt') then
  Memo1.Lines.LoadFromFile('MyText.txt')
else ShowMessage('Файл MyText.txt не існує');
```

Якщо зазначений файл існує, функція FileExists() поверне True. Якщо ж файлу з якоїсь причини немає, то ніякого читання не буде. Замість цього вийде повідомлення, що файл не існує, і процедура завершить роботу.

Clear - очищає текст в Memo. У прикладі ми звернулися до властивості Lines, але можна звернутися й до самого компонента, тому що в нього теж є метод Clear. Замість

```
Memo1.Lines.Clear;
```

можна написати просто

Memo1.Clear;

Результат буде однаковий. Збережіть проєкт, запустіть, і випробуйте його в роботі. Не забудьте поекспериментувати зі зміною розміру вікна, щоб подивитися, як будуть виглядати «прив'язані» до сторін форми компоненти.

Крім перерахованих вище інструментів, TMemo має й деякі інші корисні методи. Зараз ми говоримо про методи саме компонента TMemo, а не про його властивості Lines, майте це на увазі

CopyToClipboard	- копіює виділений у компоненті текст у **Буфер обміну**. Приклад: Memo1.CopyToClipboard;
CutToClipboard	- вирізує виділений текст і поміщає його в **Буфер обміну**. Приклад: Memo1.CutToClipboard;
PasteFromClipboard	- вставляє в компонент текст із **Буфера обміну** в позицію, де перебуває курсор. Приклад: Memo1.PasteFromClipboard;
SelectAll	- виділити весь текст. Приклад: Memo1.SelectAll;
Undo	- скасувати останні зміни в тексті. Приклад: Memo1.Undo;

Окремо варто сказати й про події компонента TMemo. Виділіть його, і в Інспекторі об'єктів перейдіть на вкладку Події. Нас цікавлять тільки основні події, які можуть нам згодитися в роботі. Приклади я приводити не буду, події генеруються так само, як OnClick, яким ми неодноразово користувалися.

OnChange	- подія відбувається, коли текст у компоненті змінений. Наприклад, користувач ввів або навпаки, видалив символ, завантажив текст із файлу або ще якимось чином змінив вміст Memo.

OnClick	- подія відбувається, коли користувач клацає по компоненті мишею.
OnDblClick	- подія відбувається, коли користувач двічі клацає по компоненті мишею.
OnEnter	- подія відбувається, коли компонент стає активним. Або, як говорять, одержує фокус введення.
OnExit	- подія відбувається, коли компонент губить фокус введення (активним стає інший компонент).
OnKeyDown	- подія виникає, коли користувач натискає, але ще не відпускає клавішу на клавіатурі. Оброблювач звичайно використовують для розпізнавання натискання клавіші
OnKeyPress	- подія виникає, коли користувач натиснув і відпустив клавішу на клавіатурі. Оброблювач звичайно використовують для розпізнавання натиснутої клавіші. Якщо користувач ввів неприпустимий символ, його можна заборонити або замінити іншим символом.
OnKeyUp	- подія виникає, коли користувач відпускає клавішу на клавіатурі. Оброблювач звичайно використовують для розпізнавання натиснутої клавіші.
OnUTF8KeyPress	- те ж, що й OnKeyPress, але не для ANSI-, а для UTF8-символів. При роботі з українськими символами потрібно використовувати цю подію.

Збережіть проєкт, скомпілюйте й запустіть його. Випробуйте редагування, збереження тексту у файл, читання його з файлу.

2. Списки вибору TListBox і TComboBox

В Lazarus є ще два компоненти, які використовують тип TStrings, або говорячи інакше, масиви рядків. Це списки вибору TListBox і TComboBox. Такі списки звичайно використовують у програмах не для редагування тексту, а для вибору користувачем одного або декількох рядків. TListBox - це прямокутний список, схожий на TMemo, а TComboBox – випадаючий список.

Щоб познайомитися з ними, закриємо поточний проєкт і створимо новий. Відразу ж форму перейменуємо в **fMain**, збережемо проєкт у папку **14-02** під ім'ям **MyStrings**, а модулю головної форми, як звичайно, дамо ім'я Main. У властивості Caption форми напишіть Вибір міста, властивість BorderStyle зробіть bsDialog, щоб користувач не міг змінювати розміри вікна. Властивість Position зробіть poDesktopCenter, щоб вікно з'являлося по центрі - такі дріб'язки потрібно відразу ж встановлювати, щоб програма добре виглядала. Розміри форми зробимо такими: Height = 210, Width = 340.

Властивості Name у компонентів у цьому проєкті ми міняти не будемо. Так буде краще виглядати при звертанні до цих компонентів всередині коду. Хоча в реальних проєктах рекомендується давати компонентам відповідному їхньому призначенню назви. Ну, от вам приклад. Якщо на формі всього одна кнопка, то її ім'я можна залишити за замовчуванням - Button1. Отже зрозуміло, про яку кнопку мова йде. А от якщо їх три, як у минулому проєкті, тоді накраще дати їм зрозумілі імена. bSave, bRead, і bClear - ясно, що мова йде про кнопки, раз перша b (button), і що це кнопки «**Зберегти**», «**Читати**» і «**Очистити**». Якщо у вас слабкі знання англійського - не лихо, можете давати українські назви, але писати латиницею: kSohranit, kChitat і kOchistit (k - knopka). Оскільки в нашому прикладі всі компоненти будуть у єдиному екземплярі, міняти їхні властивості Name ми не будемо.

Повернемося до проєкту. На вкладці **Standard Палітри компонентів** знайдіть компоненти TListBox і TComboBox:

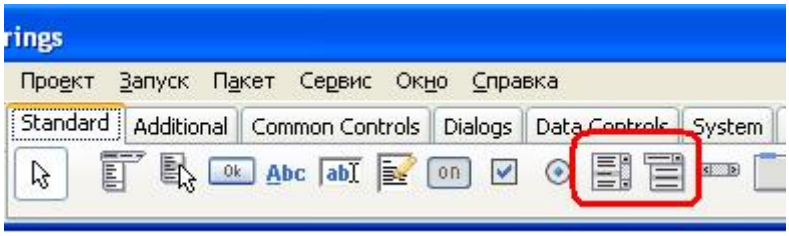


Рис. 14.3. Компоненти TListBox і TComboBox

У лівому верхньому куті форми встановітье TListBox. Зверніть увагу на його властивості в **Інспекторі об'єктів**. Ми торкнемося тільки деяких, специфічні для TListBox.

Columns	- кількість стовпців у списку. Якщо властивість дорівнює нулю або одиниці, то список виходить в один

	стовпець. Якщо більше - то стільки ж буде й стовпців. Якщо всі рядки не вміщаються в списку, смуга прокручування буде з'являтися автоматично. Звичайно цю властивість не міняють - рядки в один стовпець використаються найчастіше.
ExtendedSelect	- розширене виділення. Має на увазі, що користувач може виділити відразу цілий діапазон рядків, якщо, утримуючи **<Shift>**, клацніть спершу по першому, а потім по останньому рядку діапазону. За замовчуванням, дорівнюватиме True - таке виділення дозволене. У кожному разі, користувач може виділяти вибіркові рядки, клацаючи по них по черзі й утримуючи при цьому **<Ctrl>**. Все вищесказане працює, тільки якщо властивість MultiSelect = True. Не будемо змінювати цю властивість.
Items	- головна властивість компонента, масив (колекція) рядків. Те ж саме, що й властивість Lines у компонента TMemo, має той же тип TStrings, і відповідно, має ті ж властивості й методи. Редактор рядків також відкривається при натисканні на кнопку «**…**»праворуч від властивості.
MultiSelect	- можливість виділення декількох рядків. За замовчуванням, дорівнює False - багаторядкове виділення заборонене. Якщо ви побажаєте дозволити користувачеві багаторядкове виділення, то вам у коді необхідно обходити весь список, щоб виявити виділені рядки.
Sorted	- сортування списку. Якщо дорівнює True, список сортується за алфавітом.

Все інше приблизно відповідає компоненту TMemo. Отже, встановимо в нашого ListBox1 наступні властивості:

- Left, Top = 10
- Height, Width = 150
- MultiSelect = True

Тепер у властивості Items кнопкою «**…**»відкрийте редактор рядків і внесіть наступні рядки (кожне місто на окремому рядку):

Москва

Санкт-Петербург

Київ

Вільнюс

Будапешт

Відень

Париж

Берлін

Лондон

Мехіко

Оттава

Лос-Анжелес

Сан-Франциско

Нью-Йорк

Якщо є бажання відсортувати список міст за алфавітом, встановите Sorted = True.

Тепер займемося компонентом TComboBox. Встановите його правіше ListBox1. У нього теж не так багато специфічних властивостей:

ArrowKeysTraverseList	- при True (за замовчуванням) дозволяє переміщатися за списком кнопками зі стрілками. Хоча в мене й при False так можна було переміщатися, може бути, у наступних версіях Lazarus буде інакше.
AutoComplete	- при значенні True дозволяє вибирати елементи зі списку, фільтруючи їх. Тобто, якщо в рядку списку ви ввели «А», то вийдуть тільки ті елементи, які починаються на «А». Якщо ви ввели наступну «б», залишаться тільки ті, що починаються на «Аб». І т.д.
AutoCompleteText	- перемикачі автозавершення введення, що розкриває властивість із п’ятьома перемикачами. При True перемикач включений, при False - виключений. Є наступні перемикачі: • cbactEnabled - включення опції AutoComplete.

	• cbactEndOfLineComplete - виконує AutoComplete, тільки якщо курсор перебуває наприкінці рядка. • cbactRetainPrefixCase - зберігає ті ж символи, які ввів користувач. • cbactSearchAscending - при True пошук підходящих рядків ведеться у висхідному порядку, при False - у спадному. • cbactSearchCaseSensitive - при True пошук чутливий до регістра символів (розрізняє великі та малі), при False - немає. Однак, на жаль, і ця властивість поки не працює. Я включав всі перемикачі, але спроба фільтрації рядків однаково приводить до помилок.
ItemIndex	- це дуже важлива властивість - індекс виділеного рядка. За замовчуванням дорівнює -1, тобто, ніякий рядок не виділений. Індексація рядків починається з нуля, але оскільки, ми не будемо вручну заповнювати ComboBox1, то залишимо -1.
Items	- такий же масив рядків, як і в TListBox. Його також можна заповнити в процесі проєктування, натиснувши на «**…**»і викликавши **Редактор рядків**. Однак ми не будемо зараз це робити. Заповнювати список і змінювати ItemIndex ми будемо програмно, у коді, адже потрібно освоїти й цю можливість!
Sorted	- як і в TListBox, ця властивість сортує список за алфавітом.
Text	- текст, що виводиться в рядку компонента. При зміні ItemIndex міняється й властивість Text. За замовчуванням, там написане ім’я компонента. Але нам це не потрібно, тому просто очистіть цю властивість.

Отже, установимо наступні параметри компонента ComboBox1:

Left = 177
Top = 10
Width = 150

Рядка у властивості Items ми не заповнювали, інші властивості не змінювали.

Далі, нам знадобиться проста кнопка TButton. Її параметри:

Left = 10
Top = 170
Width = 150
Caption = Копіювати в ComboBox

І наостанок, нам знадобиться TEdit, щоб виводити в нього обрану в ComboBox1 рядок. Його параметри:

Left = 177
Top = 172
Width = 150
ReadOnly = True (користувач не може редагувати цей рядок)
Text = ‘‘ (тобто, очистити цю властивість)

У результаті у вас повинна вийти така форма:

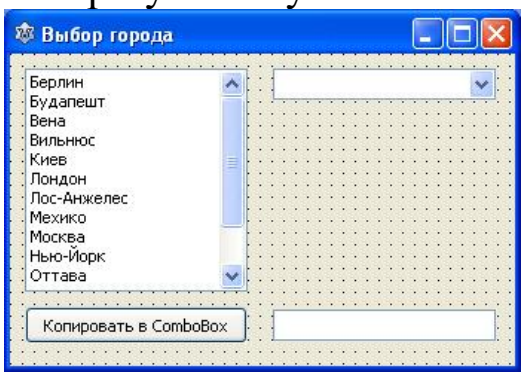


Рис. 14.4. Форма проєкту

Суть програми в наступному: користувач вибирає в **ListBox1** один або кілька міст, натискає кнопку «**Копіювати в ComboBox**». Ми програмно обійдемо весь список **ListBox1**, від першого до останнього рядка, і кожен обраний користувачем рядок додамо в список ComboBox1. Якщо ми щось у нього додали, нам доведеться встановити ItemIndex = 0, тобто, виділити перший рядок списку (інакше відразу не буде видно, додалося туди щось чи ні). Далі, якщо в рядку ComboBox з’явилося якесь місто, нам потрібно буде продублювати його в рядок Edit1.

Приклад, загалом, простий, але він демонструє майже всі можливості списків. Для реалізації задуманого нам буде потрібно

всього дві події: натискання OnClick на кнопку **Button1**, і зміна OnChange у компоненті ComboBox1. Згенеруйте їх, от код цих подій:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  i: integer; //лічильник
begin
  //очистимо ComboBox:
  ComboBox1.Clear;
  //далі в циклі обходимо весь ListBox, і якщо рядок
  //виділений - копіюємо його в ComboBox:
  for i:= 0 to ListBox1.Count -1 do
    if ListBox1.Selected[i] then
      ComboBox1.Items.Add(ListBox1.Items.Strings[i]);
  //якщо ComboBox не порожній, виділимо перший елемент, інакше
  //не виділяємо нічого:
  if ComboBox1.Items.Count > 0 then begin
    ComboBox1.ItemIndex:= 0;
    //копіюємо в Edit1 виділений рядок з ComboBox:
    Edit1.Text:= ComboBox1.Items.Strings[ComboBox1.ItemIndex];
  end
  else ComboBox1.ItemIndex:= -1;
end;

procedure TfMain.ComboBox1Change(Sender: TObject);
begin
  //копіюємо в Edit1 виділений рядок з ComboBox:
  Edit1.Text:= ComboBox1.Items.Strings[ComboBox1.ItemIndex];
end;
```

Коментарі досить докладні, але все-таки прояснимо деякі моменти. Властивість Items в TListBox і TComboBox, а також властивість Lines в TMemo мають тип TStrings, тому поводяться схожим чином - що працює в одного компонента, буде працювати й в іншого. Пам’ятаєте, як ми очищали Memo1? Те ж саме можна зробити й з TListBox, і з TComboBox:

```
  ComboBox1.Clear;
```

А от, як ми реалізували обхід всіх рядків в ListBox1:

```
//виділений - копіюємо його в ComboBox:
for i:= 0 to ListBox1.Count -1 do
  if ListBox1.Selected[i] then
    ComboBox1.Items.Add(ListBox1.Items.Strings[i]);
```

Властивість Count компонента ListBox1 повертає кількість рядків у компоненті. В **Інспекторі об'єктів** його не видно, однак всі типи TStrings мають цю властивість, а виходить, ми зможемо в такий же спосіб довідатися кількість рядків Memo, і в ComboBox. Оскільки індексація рядків починається не з одиниці, а з нуля, то в останнього рядка списку номер індексу буде Count - 1.

Далі, властивість Selected повертає True, якщо зазначений в індексі рядок виділений. У принципі, ми можемо чи подивитися виділена якийсь конкретний рядок, наприклад, дев'ятий:

```
if ListBox1.Selected[8] then
```

але набагато зручніше, звичайно, обійти весь список у циклі for, як це було зроблено в проєкті.

Далі ми дивимося - якщо поточний рядок виділений, то ми копіюємо його вміст у список ComboBox1:

```
ComboBox1.Items.Add(ListBox1.Items.Strings[i]);
```

Вміст кожного конкретного рядка можна одержати через його властивість Items, що містить безпосередньо масив рядків Strings. Вказавши індекс потрібного нам рядка, ми одержимо його вміст. Наприклад, одержати вміст третього рядка в ComboBox і Memo можна було б так:

```
s:= ComboBox1.Items.Strings[2];
s:= Memo1.Lines.Strings[2];
```

Метод Add() типу TStrings додає зазначений рядок у кінець списку. Точно так ми могли б додати рядок у кінець списку Memo або ListBox:

```
Memo1.Lines.Add('Новий рядок');
ListBox1.Items.Add('Новий рядок');
```

Ось таким способом ми заповнюємо список ComboBox1 тими рядками, які були виділені в ListBox1. Однак, оскільки користувач міг

і не виділяти ніяких рядків, і все-таки натиснути кнопку, ми робимо перевірку:

```
    //якщо ComboBox не порожній, виділимо перший елемент, інакше
    //не виділяємо нічого:
    if ComboBox1.Items.Count > 0 then begin
      ComboBox1.ItemIndex:= 0;
      //копіюємо в Edit1 виділений рядок з ComboBox:
      Edit1.Text:= ComboBox1.Items.Strings[ComboBox1.ItemIndex];
    end
    else ComboBox1.ItemIndex:= -1;
  end;
```

Отут все досить прозоро: якщо список ComboBox1 порожній, то його властивість Count буде дорівнює нулю. У цьому випадку, нам потрібно привласнити властивості ItemIndex значення -1, тобто, ніякий рядок не виділений. Якщо спробувати встановити значення 0, тобто, виділити перший рядок, відбудеться помилка, адже рядка немає! Якщо все-таки рядок є, то ми, по-перше, виділяємо в списку перший рядок:

```
ComboBox1.ItemIndex:= 0;
```

і, по-друге, ми копіюємо його в Edit1:

```
Edit1.Text:= ComboBox1.Items.Strings[ComboBox1.ItemIndex];
```

Отут відбувається ось що: ComboBox1.ItemIndex поверне поточний індекс, тобто, індекс виділеного рядка. А ComboBox1.Items.Strings[x] поверне вміст рядка під індексом x, як це відбувалося в ListBox1 у циклі.

І наостанок, у події OnChange компонента ComboBox1 ми дублюємо код копіювання виділеного тексту в Edit1 - це подія буде спрацьовувати, коли користувач вибере в списку якесь інше місто.

3. Властивості й методи типу TSrings

На завершення розглянемо основні властивості й методи типу TStrings - базового класу масиву рядків, широко використовуваного в деяких компонентах. Це може бути як властивість Lines компонента TMemo, так і властивості Items компонентів TListBox і TComboBox. Контейнер TRadioGroup, що ми з вами вивчали в лекции №7, також має властивість Items типу TStrings. Є різні властивості цього типу й в

інших компонентів, як стандартних, так і створених сторонніми розробниками. На жаль, створювати змінні такого типу не можна, для цього служить інший клас - TStringList, що є спадкоємцем TStrings, і має більш розширені можливості. Але про нього ми будемо говорити вже в іншій лекції. А зараз розглянемо основні можливості типу TStrings.

Таблиця 14.1. Властивості й методи типу (класу) TStrings	
Найменування	**Опис**
Властивості	
Count	Кількість рядків у списку
Strings	Індексований масив рядків. До окремого рядка звертаються за його індексом, наприклад Strings[0] - перший рядок масиву.
Text	Містить текст всіх рядків, включаючи символи переходу на новий рядок, у вигляді єдиного рядка.
Методи	
Add	Додає рядок у кінець списку й повертає індекс доданого рядка. Втім, що повертає значення можна ігнорувати, як це робили ми в прикладі.
Append	Додає рядок у кінець списку, але на відміну від Add, не повертає ніякого значення.
AddStrings	Додає в кінець списку інший список рядків, що також має тип TStrings.
Clear	Очищає список.
Delete	Видаляє зі списку рядок за його індексом. Наприклад, видалити другий рядок з Memo можна так: Memo1.Lines.Delete(1);
Insert	Вставляє в список рядок за зазначеним індексом. Весь інший текст зрушується вниз. Якщо тексту немає, рядок буде вставлена за індексом 0.

LoadFromFile	Зчитує рядок із зазначеного файлу й заповнює ним компонент.
SaveToFile	Навпаки, зчитує рядок з компонента, і зберігає його у зазначений файл. Якщо файлу немає, він буде створений, якщо є - перезаписаний.

Тема 15. Діалоги

1. Діалоги
2. TOpenDialog и TSaveDialog
3. TColorDialog
4. TFontDialog
5. TCalendarDialog
6. TCalculatorDialog
7. Компонент TImage та діалоги TOpenPictureDialog, TSavePictureDialog

1. Діалоги

У будь-якій великій програмі є діалоги.

Діалоги - це стандартні системні вікна, призначені для одержання інформації від користувача. В Lazarus діалоги - невізуальні (невидимі користувачеві) компоненти, робота яких приводить до виведення на екран стандартних діалогових вікон операційної системи.

Інформація, яку потрібно отримати від користувача, може бути самою різною: який файл відкрити, який файл створити, куди зберегти, який вибрати шрифт, кольори, картинку…Можна, звичайно, створювати всі ці діалоги вручну, але набагато зручніше користуватися стандартними діалогами - це простіше, і ці діалоги використовують мову встановленої операційної системи. Тим більше що від кожного діалогу нам з вами буде потрібно всього декілька властивостей, і один метод - легше нікуди! Всі діалоги перебувають на вкладці **Dialogs Палітри компонентів**.

2. TOpenDialog і TSaveDialog

Діалоги TOpenDialog і TSaveDialog призначені, як неважко догадатися з їхньої назви, для відкриття й збереження файлів:

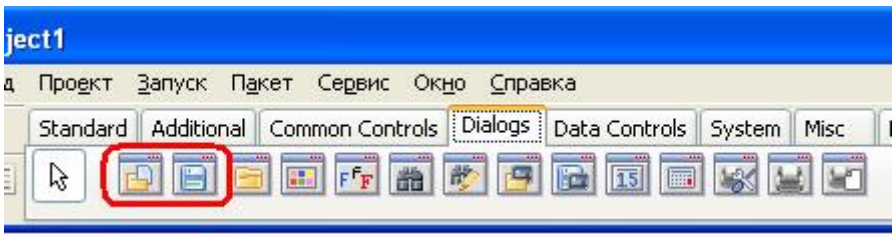


Рис. 15.1. Компоненти TOpenDialog і TSaveDialog

Вивчати роботу діалогів ми з вами будемо відразу на конкретних прикладах. Щоб не робити зайвої роботи, скористаємося проєктом **MyNotebook** з минулої лекції, що ви повинні були зберегти в папку **14-01**. Якщо ви пам'ятаєте, у тому проєкті для зберігання тексту ми використовували файл **MyText.txt**, що не дуже зручно - а раптом користувачеві захочеться працювати з багатьма файлами, а не тільки з одним? От саме для цього нам і знадобляться два цих діалоги.

Отже, відкрийте **Lazarus**. Якщо у вас відкрився новий проєкт (або попередній), закрийте його командою **Файл -> Закрити**. Далі, командою **Файл -> Відкрити** знайдемо й відкриємо проєкт **MyNotebook** (для цього потрібно вказати файл **MyNotebook.lpi** або **MyNotebook.lpr**). Зверніть увагу: тільки що ми скористалися стандартним діалогом відкриття файлу!

Коли проєкт із редактором тексту відкриється, встановітьe в будь-яке місце форми по одному компоненті TOpenDialog і TSaveDialog. Ці компоненти будуть невидимі для користувача, тому їх можна встановити в будь-яке місце форми, наприклад, прямо поверх Memo1. Виділіть компонент TOpenDialog, розберемося з його властивостями. Як бачите, їх не дуже багато, та й потрібні нам будуть не всі.

DefaultExt	- розширення імені файлу за замовчуванням. Залежно від того, з яким типом файлів нам доведеться працювати, таке в них буде й розширення. Якщо ми працюємо з текстом, то краще іменам файлів давати розширення txt. Це необов'язково, але так системі простіше буде зрозуміти, з яким типом файлів їй працювати. Так, якщо ви в Провіднику клацніть двічі по нашому колишньому файлу MyText.txt, то відкриється стандартний Блокнот Windows, у якому буде завантажений текст із цього файлу.
FileName	- ім'я файлу. Можна відразу ж вказати ім'я файлу, з адресою й розширенням, але звичайно цю властивість залишають порожньою. Після того, як діалог спрацює, і

	користувач вибере файл для відкриття, то в цій властивості буде й адреса, і ім’я цього файлу. А вони нам будуть потрібні в методі Memo1.LoadFromFile()
Filter	- фільтр типів файлів. Тут можна задати фільтрацію файлів по їхніх типах, використовуючи маску файлів. У масці знак «*» означає будь-яка кількість будь-яких символів. Наприклад, у фільтрі можна вказати маски *.txt, що означає «будь-який файл із розширенням txt», і(або) *.*, що означає «будь-який файл із будь-яким розширенням або без розширення».
InitialDir	- папка (директорія, каталог), використовувана за замовчуванням. Тут вказується адреса папки, з якої діалог почне свою роботу. Заповнювати цю властивість має сенс лише тоді, коли потрібні файли у вас будуть зберігатися в якомусь одному, конкретному місці. У нашому випадку ми не знаємо, де користувачеві захочеться зберігати свої файли, тому заповнювати цю властивість не потрібно.
Name	- ім’я компонента. Із цією властивістю ви вже знайомі, але отут потрібно зробити одне зауваження. Пам’ятаєте, у минулій лекції говорилось, що якщо компонент один, то його можна не перейменовувати? Із кнопками ця порада гарна - до кнопок у коді нам не доводиться звертатися за іменем. А от до діалогів потрібно звертатися неодноразово. Судіть самі, що простіше буде набрати на клавіатурі: OpenDialog1.FileName або OD.FileName? Тому порада: перейменовуйте діалоги, залишаючи лише перші літери імені. Якщо вам доведется в одній формі використати два однакових діалоги, то можна їх назвати OD1 і OD2.
Title	- заголовок вікна діалогу. За замовчуванням, він уже містить потрібний текст: «Відкрити існуючий файл». Якщо ж вам захочеться встановити свою, нестандартну назву, наприклад «Відкрити мій текстовий файл», можете вписати його тут.

Із властивостями розібралися, залишився метод. Метод цей - Execute.

Execute викликає стандартне діалогове вікно, і дає користувачеві зробити вибір. Якщо користувач зробив свій вибір, то Execute поверне True. Якщо користувач відмовився робити вибір і закрив вікно, не вказавши файлу, метод поверне False.

Настроїмо наш TOpenDialog:

- DefaultExt = .txt
- Name = OD

Тепер займемося фільтром. Виділіть властивість Filter і клацніть по кнопці «**…**»праворуч від нього. Відкриється редактор фільтрів, що ми заповнимо в такий спосіб:

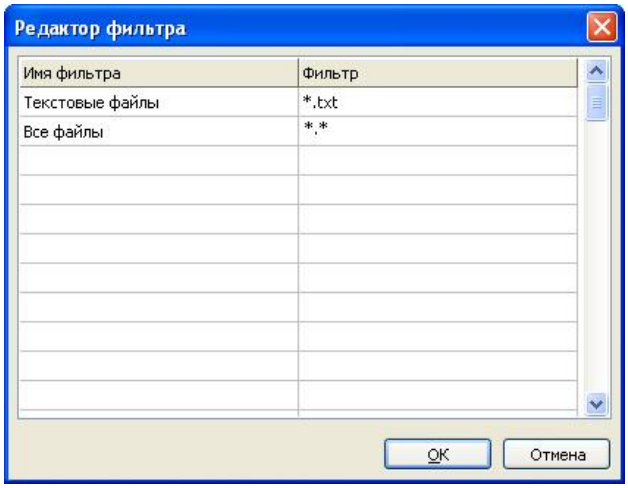


Рис. 15.2. Фільтри для TOpenDialog

Коли ви натиснете «**ОК**», Lazarus у властивості Filter згенерує наступний фільтр:

Текстові файли|*.txt|всі файли|*.*

Цей рядок можна було б написати й вручну, але скористатися **Редактором фільтрів** все-таки зручніше.

Тепер, коли ми настроїли наш діалог відкриття файлів OD, займемося кодом. Якщо ви пам’ятаєте, то завантаження тексту з файлу в нас відбувалася при натисканні кнопки «**Читати**», тобто, bRead. Клацніть двічі по цій кнопці, і ви потрапите на код її події OnClick. Наведемо текст цієї події:

```
procedure TfMain.bReadClick(Sender: TObject);
begin
  if FileExists(‘MyText.txt’) then
    Memo1.Lines.LoadFromFile(‘MyText.txt’)
```

```
  else ShowMessage('Файл MyText.txt не існує');
end;
```

Нагадаймо, ми спочатку перевіряли, чи існує цей файл, і потім, залежно від результату, використовували або одну, або іншу дію. Отож, з діалогами цього робити не потрібно! Якщо користувач вибрав файл, то нема сенсу перевіряти - чи існує він. Звичайно, існує, інакше користувач не зміг би його вибрати. Відчуваєте різницю? Новий код події буде такий:

```
procedure TfMain.bReadClick(Sender: TObject);
begin
  if OD.Execute then Memo1.Lines.LoadFromFile(OD.FileName);
end;
```

А тепер різницю видно? Ми перевіряємо - якщо діалог спрацював, то ми викликаємо метод LoadFromFile, передаючи йому як параметр обраний користувачем файл. Не забувайте, що у властивості FileName діалогу втримується ім'я обраного файлу разом з його адресою й розширенням. Якщо ж діалог не спрацював, то й робити нічого не потрібно.

Тепер займемося діалогом збереження файлу TSaveDialog. А отут нам все вже відомо - всі властивості й метод Execute у нього в точності такі, як і в TOpenDialog, хіба що текст у властивості Title інший. Саме тому й подано ці діалоги парою.

Змініть ім'я компонента на SD. Властивості DefaultExt і Filter налаштуйте так само, як в OD. Все інше залишається без змін. Тепер нам потрібно вписати код збереження файлу. Робиться це в події OnClick кнопки «**Зберегти**». Новий код події такий:

```
procedure TfMain.bSaveClick(Sender: TObject);
begin
  if SD.Execute then Memo1.Lines.SaveToFile(SD.FileName);
end;
```

Код простій, і повинен уже бути вам зрозумілий без усяких коментарів. Збережіть проєкт і запустіть його на виконання. Подивіться, як працює збереження й завантаження файлів. Тут є один недолік, якщо ви помітили: при збереженні вже відкритого файлу

програма однаково запитує, куди зберігати текст. Цей недолік ми виправимо в найближчій лабораторній роботі.

3. TColorDialog

TColorDialog - діалог вибору кольорів. Він дуже простий. Усе, що нам від нього потрібно, ця властивість Color і метод Execute. Якщо діалог відбувся, то складна властивість Color буде містити обрані користувачем кольори. Ці кольори можна привласнити властивості Color якогось іншого компонента. Наприклад, властивість Color компонента TMemo відображає кольори тла даного компонента. Не варто використовувати в проєкті яскраві кольори, але наша програма навчальна, до того ж ми будемо встановлювати тільки ті розцвічення, які вибере користувач. А тому, встановіть на форму ще одну кнопку TButton, властивість Name якої перейменуйте в bColor, а у властивості Caption напишіть Кольори. Не забудьте зняти у властивості Anchors закріплення до верхнього краю, і встановити закріплення до нижнього, як ми робили це з іншими кнопками.

Далі, нам буде потрібно встановити на форму компонентів TColorDialog:

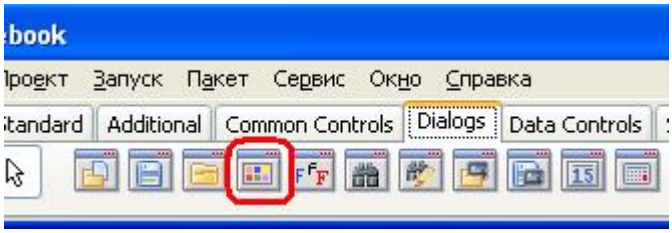


Рис. 15.3. Компонент TColorDialog

Цей компонент, як і всі інші компоненти, невізуальний, тому встановлюйте його, куди бажаєте. Властивість Name діалогу перейменуйте в CD. Оброблювач OnClick для кнопки bColor дуже простій:

```
procedure TfMain.bColorClick(Sender: TObject);
begin
  if CD.Execute then Memo1.Color:= CD.Color;
end;
```

Якщо діалог відбувся, тлу редактора привласнюємо ті кольори, що вибрав користувач. Збережіть проєкт, запустіть його й спробуйте змінити кольори.

4. TFontDialog

TFontDialog - діалог вибору шрифту. У цьому діалозі користувач може вибрати як сам шрифт, так і його розміри, накреслення, ефекти й

кольори. Не плутайте кольори тла й кольори шрифту! Білий або жовтий текст, наприклад, прекрасно читається на синьому або чорному тлі. Нам буде потрібно встановити на форму один такий компонент:

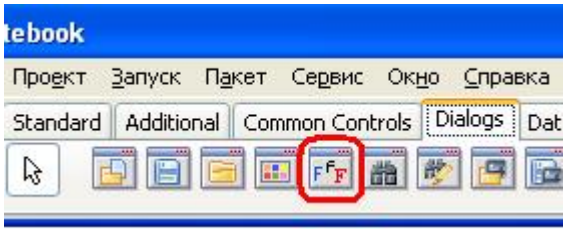


Рис. 15.4. Компонент TFontDialog

Властивість Name компонента перейменуйте в FD. У коді нам буде потрібно його складна властивість Font, що буде містити той шрифт, що вибрав користувач, і який ми зможемо привласнити властивість Font іншого компонента.

Також нам буде потрібна кнопка з ім'ям bFont і текстом Шрифт у властивості Caption. Не забувайте про прив'язки кнопок до нижньої границі форми. Оброблювач OnClick для кнопки такий:

```
procedure TfMain.bFontClick(Sender: TObject);
begin
  if FD.Execute then Memo1.Font:= FD.Font;
end;
```

5. TCalendarDialog

TCalendarDialog - діалог вибору дати. У діалозі користувач може вибрати дату, що потім можна буде вставити прямо в текст.

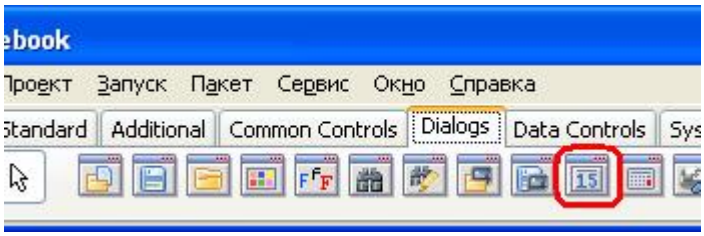


Рис. 15.5. Компонент TCalendarDialog

Як і звичайний календар, цей діалог має властивість Date, що містить обрану користувачем дату у форматі TDateTime. Властивість Name діалогу пропоную перейменувати в DD (Date Dialog), оскільки CD на формі вже є, це діалог вибору кольору. Нам буде потрібно кнопка bDate з написом Дата в Caption. Код оброблювача кнопки наступний:

```
procedure TfMain.bDateClick(Sender: TObject);
begin
  if DD.Execute then Memo1.Lines.Add(DateToStr(DD.Date));
end;
```

Не потрібно пояснювати, що отут ми вставляємо в кінець тексту дату, попередньо перетворивши її з TDateTime в String?

6. TCalculatorDialog

TCalculatorDialog - діалог-калькулятор. Дуже корисний, особливо в бухгалтерських програмах. Дозволяє користувачеві вивести калькулятор, зробити необхідні розрахунки, і результат вставити в текст:

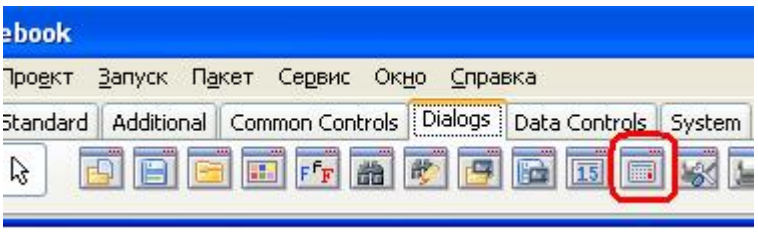


Рис. 15.6. Компонент TCalculatorDialog

У компонента є властивість Value, що має тип Double, тобто, виводить результат у вигляді речовинного числа. Перейменуйте діалог в Calc (тому що CD уже є). Для реалізації діалогу нам також буде потрібно кнопка bCalc з написом Caption. Код кнопки:

```
procedure TfMain.bCalcClick(Sender: TObject);
begin
  if Calc.Execute then Memo1.Lines.Add(FloatToStr(Calc.Value));
end;
```

Зверніть увагу, діалог вважається виконаним, якщо користувач по завершенні розрахунків натиснув на кнопку з «галочкою»:

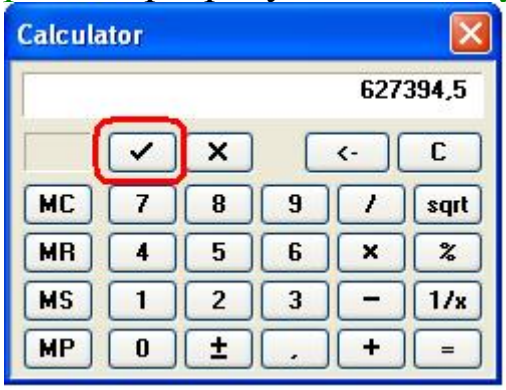


Рис. 15.7. Кнопка завершення діалогу TCalculatorDialog

Тільки в цьому випадку подія Execute поверне True, і код виконається, вставивши результат обчислень у текст. Якщо ж користувач закриє діалог кнопкою із хрестиком, клавішами **<Alt+F4>** або якось інакше, то діалог не буде виконаний, і результат втратиться.

У результаті всіх наших дій, наш редактор повинен виглядати й працювати приблизно так:

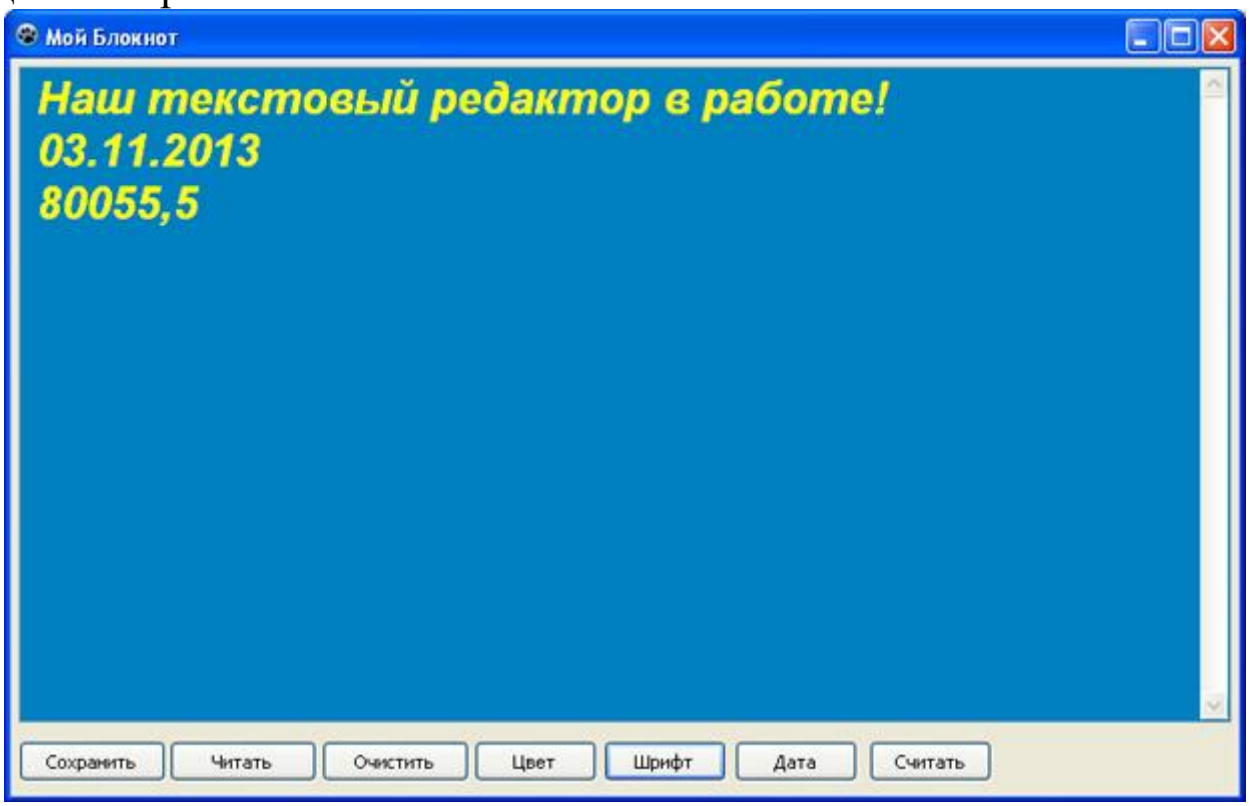


Рис. 15.8. Остаточний вигляд редактора

На цьому даний проєкт можете закрити, він нам більше не знадобиться.

7. Компонент TImage і діалоги TOpenPictureDialog, TSavePictureDialog

Останній розділ теми присвятимо роботі із графікою, в Lazarus є й така можливість. Щоб не псувати наші попередні досягнення, відкриємо **Lazarus** з новим проєктом. Як звичайно, головну форму назвемо **fMain**, збережемо проєкт у папку **15-01** під ім'ям **MyImageBrowser**, а модуль головної форми назвемо Main. Зробимо наступні настроювання властивостей у головної форми:

Caption = Перегляд зображень
Height = 450
Position = poDesktopCenter
Width = 700

На вкладці **Additional Палітри компонентів** знайдіть компонент TImage:

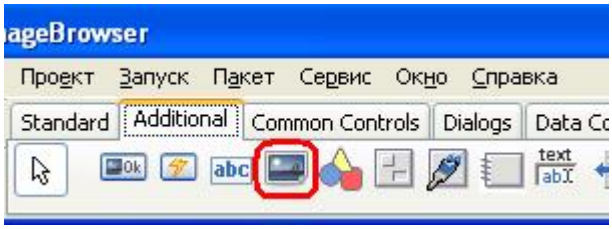


Рис. 15.9. Компонент TImage

Встановіть його на форму. Компонент TImage є контейнером для показу зображень - графічних файлів. Сам компонент за замовчуванням має такі ж кольори, як форма, а його границі позначені пунктирною лінією. TImage дозволяє завантажувати графічні файли як на етапі проєктування, так і в процесі роботи програми. У першому випадку зображення зчитується, і надалі стає частиною проєкту, його ресурсом. У другому випадку зображення не стає частиною проєкту, воно займає пам'ять із моменту його відкриття користувачем, і до закриття програми, або поки користувач не завантажить інше зображення. Ми випробуємо обидва варіанти, але для початку настроїмо положення й розміри компонента. Ім'я залишимо за замовчуванням - Image1. Оборотний увага на властивості компонента в **Інспекторі об'єктів**.

AutoSize	- автоматичний розмір. Якщо дорівнює True, компонент TImage буде підганяти свій розмір під реальний розмір зображення.
Center	- при значенні True виводить зображення по центрі компонента.
Picture	- основна властивість, має спеціальний тип TPicture і містить саме зображення. За допомогою цієї властивості можна завантажити зображення з файлу в компонент і під час проєктування, і під час виконання програми. Можна також зберегти зображення у файл, хоча це має сенс тільки для графічних редакторів, що дозволяють створювати й змінювати зображення.
Proportional	- при значенні True зображення буде змінювати розміри, зберігаючи пропорції зображення - відношення висоти до ширини. Якщо значення False, то при зміні розмірів

	зображення може бути перекручено: надмірно витягнуто в довжину або завширшки.
Stretch	- при значенні True розмір зображення буде підбудовуватися під розмір TImage. Звичайно або картинку підганяють під розмір контейнера, або контейнер під розмір картинки. У першому випадку True буде у властивості AutoSize, у другому - в Stretch.
Transparent	- прозорість. Діє тільки на бітові матриці, на файли з форматом bmp. Застосовується, коли потрібно сховати фонові кольори малюнка. Допустимо, у редакторі **Paint** ви на білому тлі намалювали синє коло. Зберегли в bmp-файл, що потім завантажили в TImage. Отож, якщо Transparent = False, те ви одержите картинку, як є - синє коло в білому прямокутному тлі. Якщо ж Transparent = True, то фонові кольори буде замінятися на кольори компонента під TImage, тобто, стане прозорим. Фоновим вважається колір самого нижнього лівого пікселю - якщо він білий, то ці кольори відображатися не будуть.

Потрібними нам методами володіє складна властивість Picture, що саме є об'єктом. Подібно масивам рядків, властивість Picture має такі методи, як LoadFromFile і SaveToFile, хоча останній метод застосовується в основному, у графічних редакторах.

Установимо наступні властивості компонента Image1:

Left, Top = 5
Height = 385
Width = 690
AutoSize = False
Center = True
Proportional = True
Stretch = True

Інші властивості змінювати не будемо. Спробуємо завантажити картинку. Виділіть властивість Picture і клацніть по кнопці «**…**»праворуч від нього. Відкриється діалог завантаження зображень. Нажавши кнопку «**Завантажити**», ви відкриєте стандартний діалог «**Відкрити файл зображень**». Тут ви можете вибрати бажаний файл, переглядаючи в правій частині діалогу мініатюрне зображення файлу. Вибравши потрібний файл, натисніть кнопку «**Відкрити**». Обраний

файл відобразиться в **Діалозі завантаження зображень**, після чого натисніть кнопку «**ОК**». Обране зображення потрапить в TImage, причому буде відображено по центрі контейнера, збереже пропорції й піджене свій розмір під розміри контейнера.

Однак нам не потрібне тло для форми, нам потрібна програма для перегляду різних зображень, а така програма завантажує картинки на вимогу користувача. Тому видалимо зображення. Для цього знову виділіть Picture і відкрийте **Діалог завантаження зображень**. Там натисніть на кнопку «**Очистити**», після чого натисніть на «**ОК**». Діалог закриється, контейнер Image1 знову буде порожній.

Тепер нам потрібний діалог відкриття графічних файлів - TOpenPictureDialog, що перебуває на вкладці **Dialogs Палітри компонентів**:

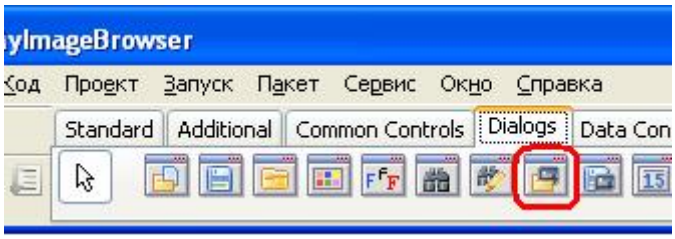


Рис. 15.10. Компонент TOpenPictureDialog

Подібно іншим діалогам, TOpenPictureDialog є невізуальним, його можна встановити на будь-яке місце, наприклад, прямо посередині Image1. Властивостей у нього також небагато, причому всі потрібні властивості уже заповнені. Відкрийте, наприклад, **Редактор фільтрів** у властивості Filter, і подивіться на ту кількість графічних форматів, з якими ви можете працювати! Однак ім'я діалогу занадто довге, так що перейменуємо його властивість Name на OPD.

Також унизу нам знадобиться кнопка **TButton**, що ми дамо ім'я bOpen, а у властивості Caption напишемо **Завантажити зображення**. Щоб напис помістився на кнопці, встановіть її властивість Width рівним 160, або просто розтягніть кнопку за допомогою миші або кнопкою зі стрілкою вправо, утримуючи натиснутої кнопку **<Shift>**.

Давайте скористаємося прикладом попереднього додатка, і подкорегуємо властивість Anchors у контейнера й кнопки: Image1 «прив'яжіть» до всіх сторін форми, щоб він міг змінювати розмір разом з формою, а кнопку «відв'яжіть» від верхньої границі й «прив'яжіть» до нижньої.

Згенеруйте подію OnClick для кнопки, його код:

```
procedure TfMain.bOpenClick(Sender: TObject);
begin
  if OPD.Execute then Image1.Picture.LoadFromFile(OPD.FileName);
end;
```

Як бачите, нічого складного, робота діалогу абсолютно схожа на роботу інших діалогів. Збережіть проєкт, запустіть його, спробуйте завантажувати різні графічні файли й змінювати розміри форми - розміри картинки також повинні мінятися, зберігаючи пропорції:

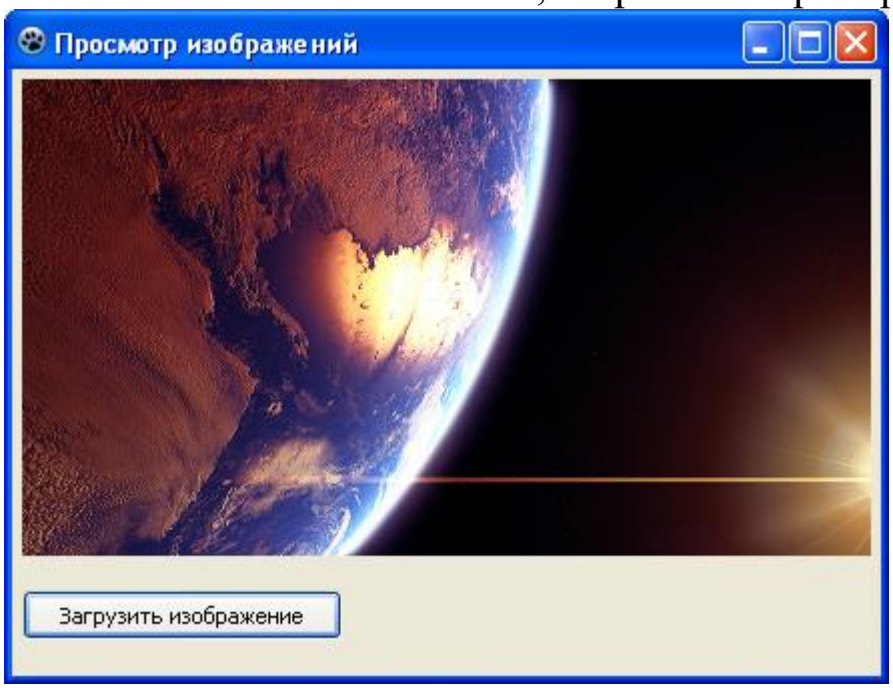


Рис. 15.11. Програма MyImageBrowser із завантаженим зображенням і зменшеним розміром

Діалог TSavePictureDialog має ті ж властивості, що й TOpenPictureDialog і призначено для збереження зображення у файл. Наша програма тільки переглядає файли, а не редагує їх, тому тут потреби в TSavePictureDialog немає. Однак в наступній темі ми все-таки використаємо його, щоб зберігати файл із зображенням під іншим ім'ям.

Крім пройдених діалогів вкладка **Dialogs** містить і інші, більш специфічні діалоги. Наприклад, діалоги пошуку й заміни тексту, настроювання принтера й друку. Однак всі діалоги працюють схожим образом, так що при необхідності, ви зможете самостійно розібратися з роботою кожного із цих діалогів.

Тема 16. Організація меню й панелей інструментів

1. Головне меню
2. Спливаюче меню
3. Компонент TImageList
4. Гарячі клавіші
5. Панель інструментів

1.Головне меню

Інтерфейс програми може містити два типи меню: головне й спливаюче. Головне меню являє собою рядок у верхній частині екрана. Першою командою меню, як правило, є «**Файл**», останньою – «**Довідка**». Реалізується головне меню компонентом TMainMenu, що перебуває на вкладці **Standard Палітри компонентів**:

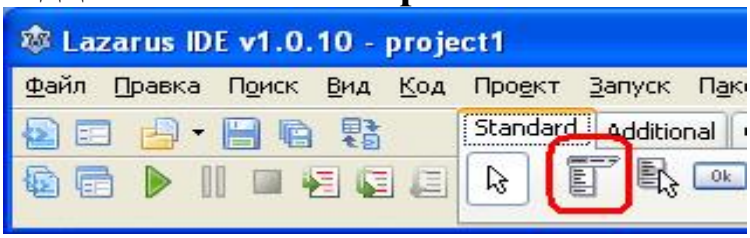


Рис. 16.1. Компонент TMainMenu

Але давайте знайомитися із цим компонентом відразу на практиці. Отже, відкрийте **Lazarus** з новим проєктом. Відразу ж перейменуйте головну форму в **fMain**, проєкт збережіть в папку **16-01** під ім'ям **ImageViewer**, а модулю головної форми дайте ім'я Main. В Caption форми напишіть Перегляд зображень. А саму форму зробіть більшою, хоча б 450*700 пікселів.

Тепер встановіть на форму компонентів TMainMenu - компонент невізуальний, його можна встановити в будь-якому місці, наприклад, прямо посередині форми. Нам не потрібно звертатися до компонента по імені, так що ім'я можна не змінювати. Щоб створити меню, потрібно двічі клацнути по компоненті, і відкриється **Редактор меню**. За замовчуванням, в **Інспекторі об'єктів** виявиться виділеним перший (і єдиний) пункт меню **MenuItem1**:

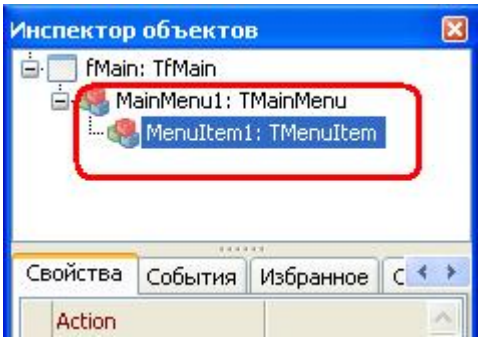


Рис. 16.2. Виділений пункт меню

Тепер нам доведеться змінити деякі налаштування, тому що ім'я MenuItem1 нам ні про що не говорить. Властивість Name перейменуйте в FileMenu, у властивості Caption напишіть Файл. Відразу ж ви побачите, як змінилася назва на кнопці в **Редакторі меню**, і слово «Файл» з'явилося у верхній частині вікна форми. Клацати по ньому не потрібно, інакше буде створений оброблювач натискання на пункт меню, а нам це не потрібно - для пункту «Файл» оброблювач не передбачений.

Далі, у **Редакторі меню** клацніть по кнопці «**Файл**» **правою** кнопкою миші, і в що відкрилися спливаючому меню виберіть команду «**Вставити новий пункт (після)**». У **Редакторі меню** з'явилася друга кнопочка з написом «New Item2». Її необхідно перейменувати, як і першу - переконаєтеся, що саме вона виділена в **Інспекторі об'єктів**, перейменуйте Name в WindowMenu, а в Caption напишіть Вікно. Тепер наше меню містить два пункти: «**Файли**» і «**Вікно**»:

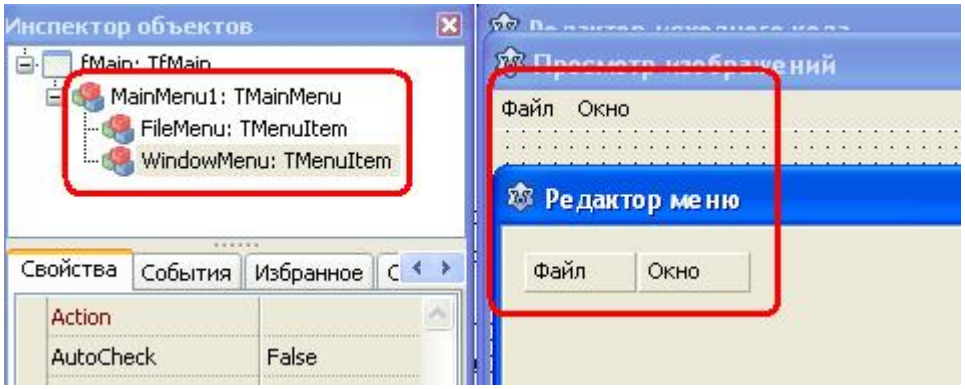


Рис. 16.3. Установка пунктів меню

Навчимося створювати підпункти меню. Клацніть правою кнопкою миші по кнопці «**Файл**» у **Редакторі меню** й виберіть команду «**Створити підменю**». Відразу ж на кнопці «**Файл**» з'явилася стрілка вниз, що вказує, що пункт меню містить підпункти. І сам підпункт - кнопочку з написом «**New Item3**», розташовану нижче «**Файл**». Переконаємося, що в **Інспекторі об'єктів** виділений цей пункт, Name змінимо на FileOpen, Caption - на Відкрити.

Отут ідея така: головні пункти меню називати, як **FileMenu**, **WindowMenu**, **OptionsMenu**, **HelpMenu** і т.п. Останнє - **Menu** буде вказувати на те, що це головний пункт. Підменю йменувати інакше - спочатку йде ім'я головного пункту, потім команда. Наприклад, для головного пункту **FileMenu** ми могли б створити такі команди, як

FileOpen, FileSaveAs, FileClose і т.п. Це лише рекомендація, правила ви можете придумати й свої, головешці, щоб ви не заплуталися в назвах головного пункту і його підпунктів.

Клацніть правою кнопкою миші по кнопці «**Відкрити**», виберіть команду «**Вставити новий пункт (після)**». Отриманий пункт перейменуйте в **FileSaveAs**, а в Caption напишіть Зберегти як.

Нижче аналогічним образом вставте новий підпункт. У цього підпункту ім'я змінювати не будемо, а у властивості Caption ми впишемо всього один символ «-». У такий спосіб у меню вставляються роздільники - горизонтальні риски між командами.

Далі вставимо ще один підпункт. Назвіть його **FileExit**, а в Caption напишіть Вихід. У результаті у вас має вийти щось подібне:

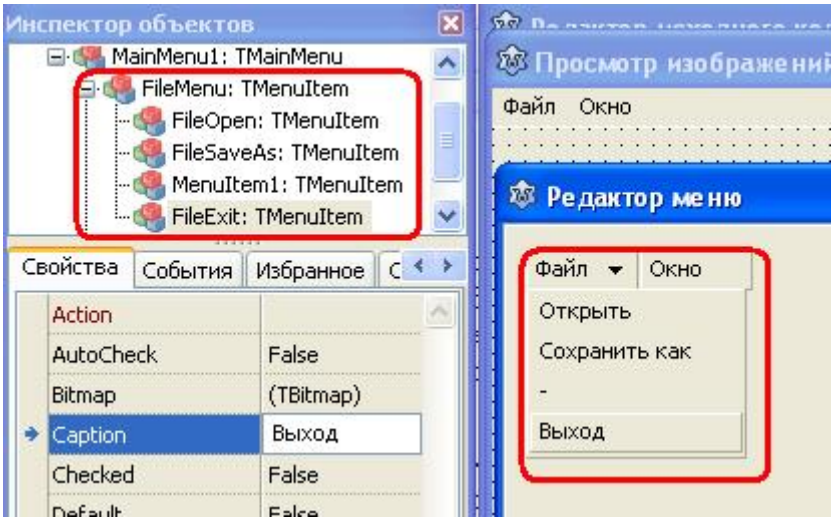


Рис. 16.4. Меню «Файл»

У нас ще залишився неопрацьованим пункт «**Вікно**». Виділіть його та створіть наступні підпункти з парою властивостей (Name - Caption): WindowNormal - Нормальне, WindowMinimize - Згорнути, WindowMaximize - Розгорнути. Як тільки ви це зробите, можете закривати **Редактор меню** - він нам більше не потрібний.

У центральну частину вікна помістіть компонент TImage, у який ми й будемо завантажувати обрані користувачем зображення. Налаштуйте компонент у такий спосіб:

- Align = alClient
- Center = True
- Proportional = True
- Stretch = True

Крім того, на форми нам знадобляться два діалоги: TOpenPictureDialog, що ми перейменуємо в OPD, і TSavePictureDialog,

що назвемо SPD. Як і TMainMenu, діалоги будуть невидимі користувачеві, тому їх можна встановити на бу-яку місце форми.

Тепер займімося програмуванням пунктів меню. Виберіть на форми пункт «**Файл -> Вихід**», і автоматично буде створений оброблювач OnClick для цього пункту. В налаштуванні впишіть усього одну команду:

```
procedure TfMain.FileExitClick(Sender: TObject);
begin
  Close;
end;
```

Як уже говорилося, оператор Close у дочірньому (неголовному) вікні закриває це вікно, а в головному вікні завершує робочі програми. Створіть у такий же спосіб оброблювач для «**Файл -> Відкрити**», з його кодом ви повинні бути знайомі по попередньому проєкті:

```
procedure TfMain.FileOpenClick(Sender: TObject);
begin
  if OPD.Execute then Image1.Picture.LoadFromFile(OPD.FileName);
end;
```

Для «**Файл -> Зберегти як**» код буде схожий:

```
procedure TfMain.FileSaveAsClick(Sender: TObject);
begin
  if SPD.Execute then Image1.Picture.SaveToFile(SPD.FileName);
end;
```

З пунктом «**Файл**» розібралися, залишився пункт «**Вікно**» і три його підпункти. Тут ми будемо міняти стан вікна в fMain.WindowState. Код для цих подій наступний:

```
procedure TfMain.WindowMaximizeClick(Sender: TObject);
begin
  fMain.WindowState:= wsMaximized;
end;

procedure TfMain.WindowMinimizeClick(Sender: TObject);
begin
  fMain.WindowState:= wsMinimized;
end;
```

```
procedure TfMain.WindowNormalClick(Sender: TObject);
begin
  fMain.WindowState:= wsNormal;
end;
```

Не переплутайте назви подій. Збережіть проєкт, запустіть його й спробуйте в роботі. Програма повинна нормально завантажувати файли, змінювати стан вікна, завершувати роботу. При збереженні файлу вказуйте не тільки бажане ім’я, але й розширення, наприклад newfile.jpg - компонент TImage не вміє вгадувати формат і самостійно підставляти потрібне розширення.

2.Спливаюче меню

Ми з вами раз у раз користуємося спливаючим меню. Це меню, що з’являється, коли користувач натискає праву кнопку миші. Реалізується воно компонентом TPopupMenu, що перебуває поруч із TMainMenu на **Палітрі компонентів**. Оскільки воно також невізуально, встановлюйте його на будь-яке місце на формі. Редактор спливаючого меню також викликається подвійним натисканням на компонент (або натисканням на «…»у властивості Items). Редагується воно за зразком. Виділіть перший (і поки єдиний) пункт цього меню, у властивості Name напишіть pFileMenu, у властивості Caption напишіть Файл. Далі, клацніть правою кнопкою миші по кнопці цього пункту в **Редакторі меню**, і виберіть «**Створити підменю**». Створіть тут два підрозділи: pFileOpen і pFileSaveAs із властивостями Caption, відповідно, **Відкрити** й **Зберегти як**.

Аналогічним образом, створіть пункт pWindowMenu (Caption = Вікно) з підпунктами pWindowNormal, pWindowMinimize і pWindowMaximize (Caption = Нормальне, Згорнути, Розгорнути).

Далі, створіть головний пункт із Caption = «-» (ім’я не змінюємо, це просто роздільник), і ще один пункт Name = CloseMenu, Caption = Вихід. У вас має вийти таке меню:

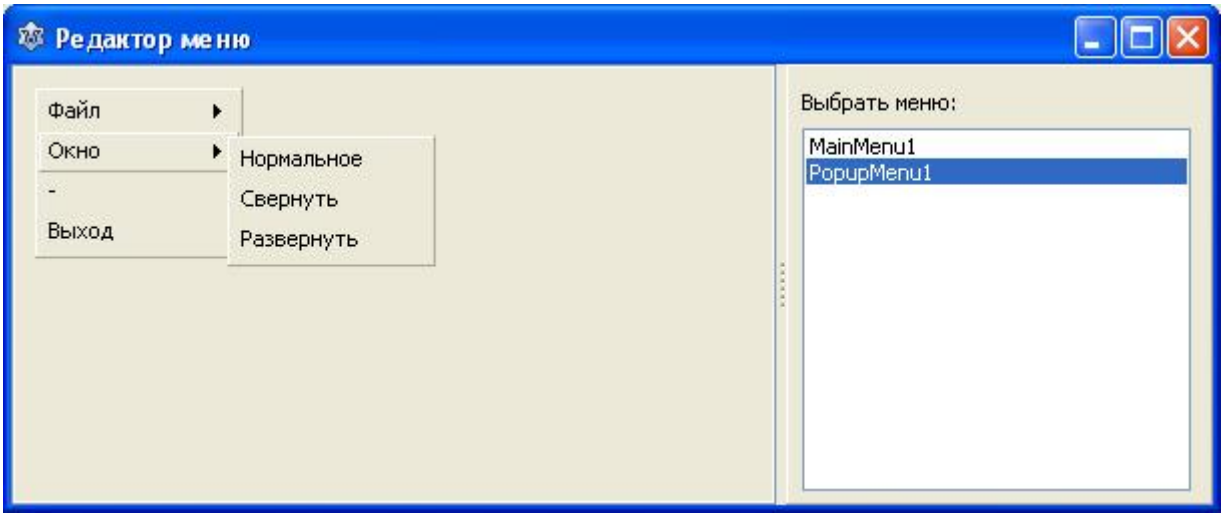


Рис. 16.5. *Спливаюче меню*

Меню ми зробили, але не поспішаєте закривати **Редактор меню**. Якщо ви думаєте, що для спливаючого меню ми знову будемо писати той же самий код, то ви помиляєтеся, все набагато простіше. Виділіть підпункт «**Файл -> Відкрити**» - клацніть один раз, а не двічі, інакше буде створений оброблювач OnClick, а нам це не потрібно. Якщо це все-таки трапилося, вам доведеться видалити порожній каркас процедури цього оброблювача, а також оголошення цієї процедури в розділі type у верхній частині модуля форми. Тільки не переплутайте назви процедур - всі пункти спливаючого меню в нас починаються з букви p.

Отже, ми виділили підпункт спливаючого меню «**Файл -> Відкрити**». В **Інспекторі об'єктів** перейдіть на вкладку **Події**, відкрийте список у подію OnClick і виберіть із цього списку подія FileOpenClick - це подія відкриття файлу в головному меню, що ми раніше запрограмували:

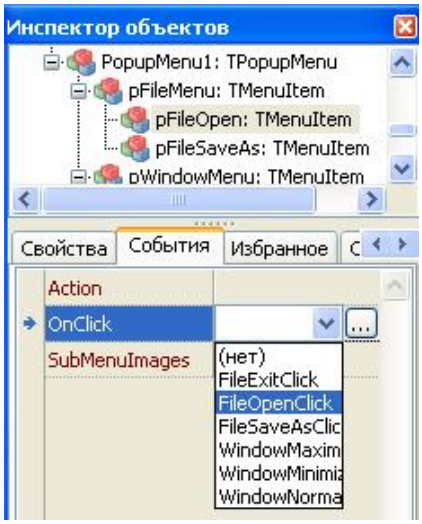


Рис. 16.6. Вибір готової події в OnClick

Таким чином, коли користувач вибере цю команду в спливаючому меню, буде викликана відповідна подія головного меню, так що нам не потрібно двічі писати той самий код. Виберіть відповідні події головного меню для команд, що залишилися, спливаючого меню **«Файл -> Зберегти** як», **«Вікно ->** Нормальне», **«Вікно ->** Згорнути», **«Вікно ->** Розгорнути» і «Вихід». Після цього можна закрити **Редактор меню**.

Але й цього недостатньо, щоб користувач міг викликати спливаюче меню - його ще потрібно прив'язати до якого-небудь компонента. Звичайно його прив'язують або до форми, або до панелі. Якщо на формі дві панелі, то для кожної можна зробити своє спливаюче меню, зі своїми командами. У нас всю частину форми займає компонент Image1, до нього й прив'яжемо спливаюче меню. Виділіть Image1, перейдіть на вкладку **Властивості Інспектори об'єктів**, і у властивості PopupMenu виберіть наш PopupMenu1. Тепер, якщо користувач натисне над Image1 праву кнопку миші, вийде спливаюче меню. Якщо ви все зробили правильно, то команди спливаючого меню будуть викликати відповідні команди головного меню, і все буде працювати нормально:

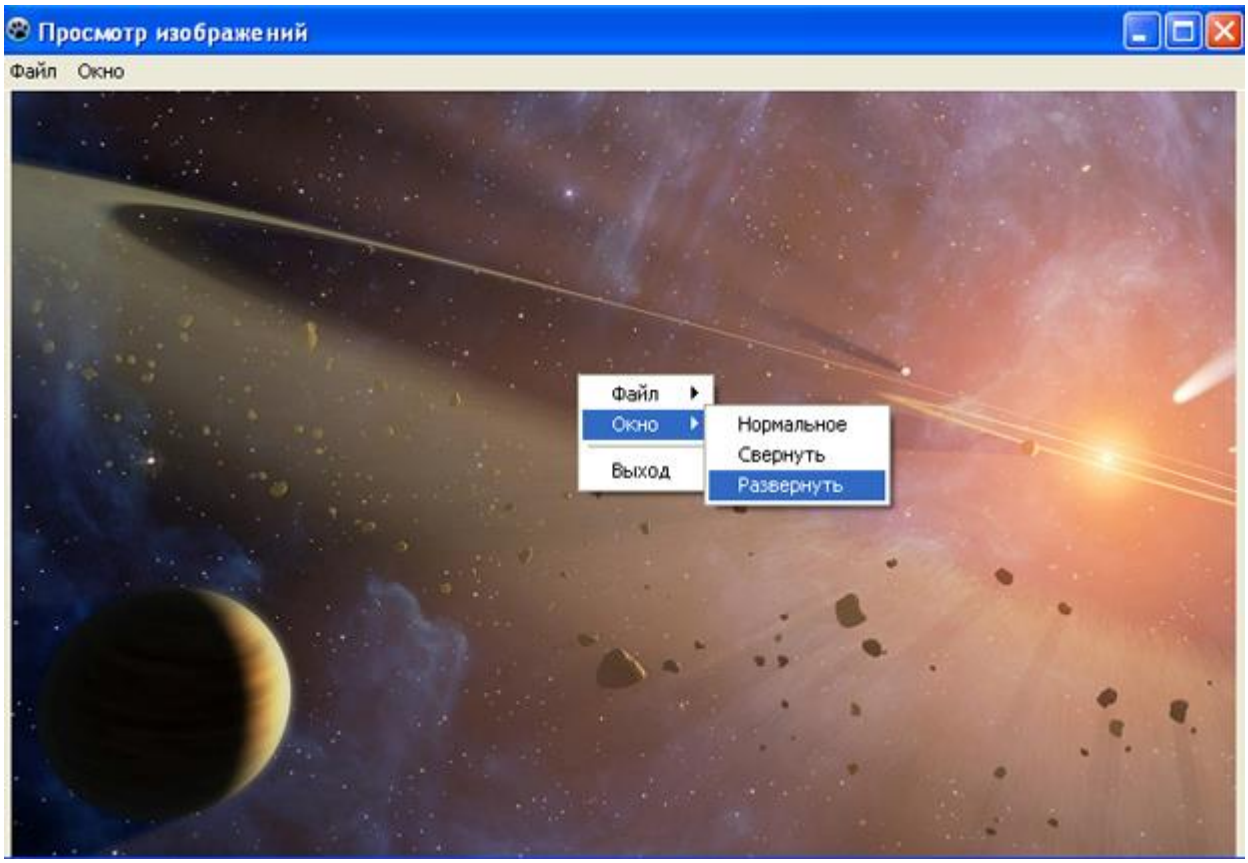


Рис. 16.7. *Робота спливаючого меню в завантаженому додатку*

3. Компонент TImageList

TImageList - це контейнер для зберігання списку зображень. Як правило, в TImageList зберігають зображення для меню й кнопок панелей інструментів. Компонент TImageList перебуває на вкладці **Common Controls Палітри компонентів**:

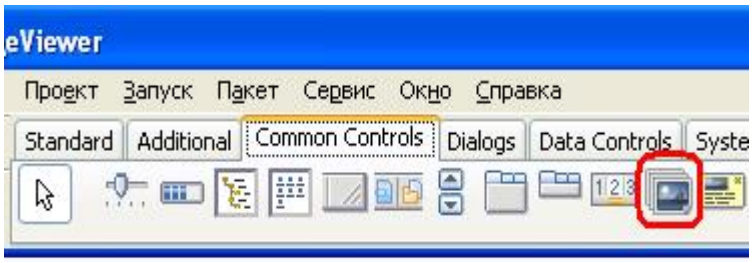


Рис. 16.8. *Компонент TImageList*

Компонент невізуальний, так що його також можна встановити на будь-яке вільне місце. Клацніть двічі по ньому, щоб відкрити **Редактор ImageList**. У **Редакторі** натисніть кнопку «**Додати**», відкриється діалогове вікно «**Додати зображення**». Тепер от що: для пунктів меню нам потрібні маленькі зображення, 16*16 пікселів. За замовчуванням, Lazarus установлюється в папку **C:\Lazarus**. Якщо ви не змінювали цю папку, то картинки для пунктів меню у вас будуть за адресою

C:\Lazarus\images\menu

Зображень там не так, щоб дуже багато, набагато більше цікавих зображень, та й цілих безкоштовних колекцій зображень можна знайти на просторах Інтернету. Однак на якийсь час обійдемося тим, що вже є. Відкрийте зазначену адресу в діалозі вибору зображень. Нам потрібна картинка для пункту меню «**Файл -> Відкрити**». Отут краще підійде файл **menu_project_open.png**. Виберіть цей файл і натисніть кнопку «**Відкрити**» - файл потрапить у список під індексом 0. Як і в інших списках, індексація в TImageList починається з 0, індекс -1 означає, що зображення не обрано. Знову натисніть кнопку «**Додати**» у редакторі. Для «**Файл -> Зберегти як**» підійде файл **menu_project_saveas.png**. Зображення встане під індексом 1.

Втім, зображення можна переміщувати за списком, міняти їх місцями за допомогою кнопок «**Перемістити вгору**» і «**Перемістити вниз**». Під індексом 2 встановите файл **menu_exit.png** - це буде зображення для «**Файл -> Вихід**». Для пунктів меню «**Вікно**»

підходящих зображень немає, тому ми залишимо ці команди без відповідних картинок.

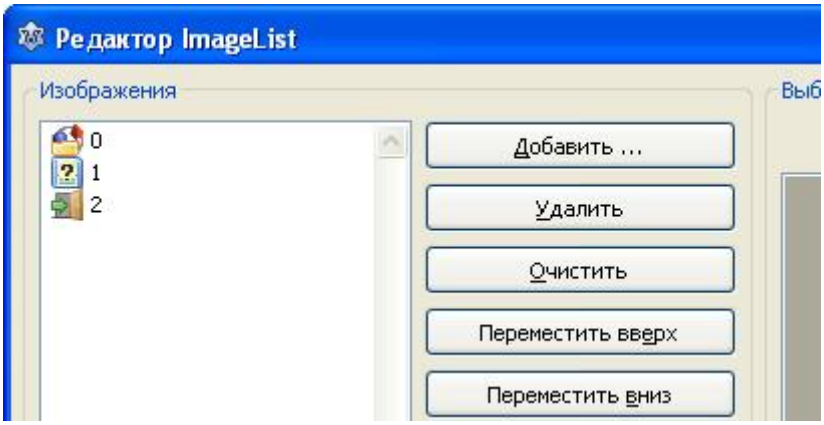


Рис. 16.9. *Три встановлених зображення в редакторі ImageList*

Натисніть кнопку «**OK**», щоб закрити редактор. Тепер нам потрібно, щоб головне меню побачило ці зображення. Виділіть компонент MainMenu1. У його властивості Images виберіть наш ImageList1. Таким чином, ми зв'язали головне меню зі списком зображень.

Далі, двічі клацніть по MainMenu1, щоб відкрити редактор меню. Виділіть підпункт «**Відкрити**» пункту «**Файл**». Ви бачите, що його властивість ImageIndex дорівнює -1, тобто, зображення не обрано. Відкрийте список зображень і виберіть перше зображення, з індексом 0. Для підпункту «**Зберегти як**» встановіть індекс 1. Для «**Вихід**» - індекс 2. У результаті пункти меню «**Файл**» стануть відображатися разом із зображеннями:

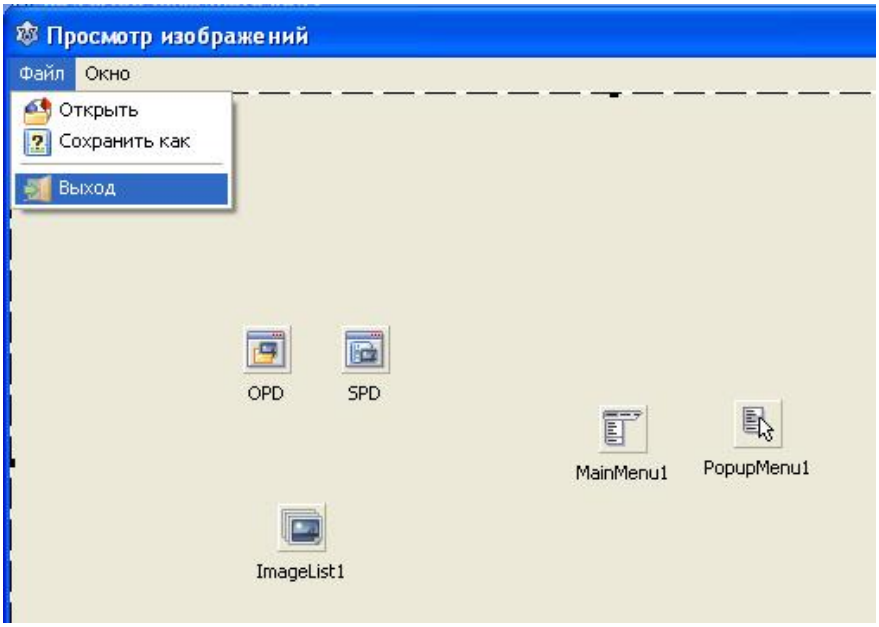


Рис. 16.10. *Меню із зображеннями*

Великої користі від цього немає, але проєкт прикрашає. Те ж саме зробіть для відповідних пунктів спливаючого меню: спочатку в його властивості Images виберіть наш список ImageList1. Потім відкрийте редактор меню, виділіть по черзі пункти «**Відкрити**», «**Зберегти як**» і «**Вихід**», виберіть для них відповідні ImageIndex. Відразу ж у редакторі меню ви не побачите змін, однак закрийте його, збережіть проєкт і запустіть програму на виконання. Клацніть правою кнопкою миші по вікну та отримаєте меню із зображеннями:

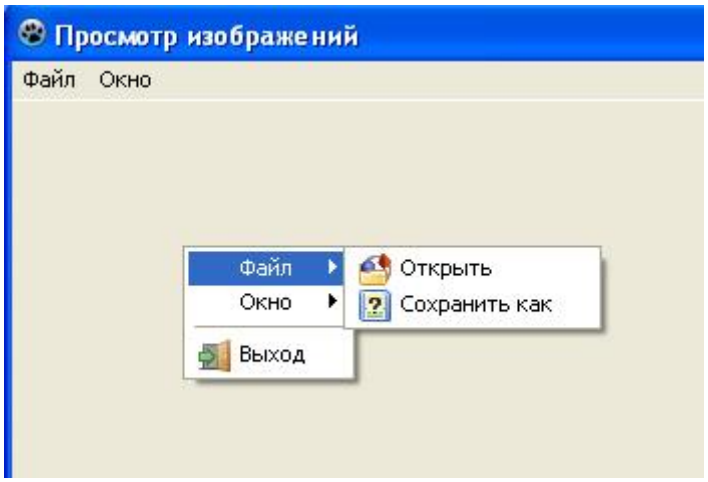


Рис. 16.11. *Спливаюче меню із зображеннями*

4. Гарячі клавіші

Отут самий час згадати про одну важливу властивість підпунктів меню, як головного, так і спливаючого - ShortCut.

ShortCut - властивість, що дозволяє вибрати для команди меню сполучення клавіш, так називані «гарячі клавіші». Якщо якесь сполучення клавіш привласнене одній з команд, то користувачеві необов'язково буде лізти в меню, щоб виконати цю команду, досить буде натиснути відповідне сполучення клавіш. Так, для команди меню **Файл->Відкрити** традиційно застосовують сполучення **<Ctrl+0>**; для команди **Файл->Зберегти** - **<Ctrl+S>**; для команди **Файл->Вихід** можна використати **<Ctrl+Q>**. Відкрийте редактор головного меню, виділіть по черзі в меню **Файл** підпункти «**Відкрити**», «**Зберегти як**» і «**Вихід**». Виберіть для цих команд відповідні сполучення у властивості ShortCut. У результаті меню буде мати такий вигляд:

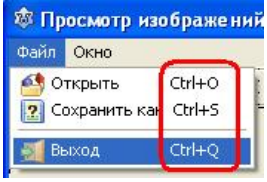


Рис. 16.12. *Гарячі клавіші в меню*

Тепер користувач зможе виконувати ці команди як за допомогою меню, так і за допомогою «гарячих клавіш». Робити те ж саме з пунктами спливаючого меню має сенс тільки в тому випадку, якщо його команди відрізняються від команд головного меню. У нашому випадку однаково будуть виконуватися команди головного меню, так що вказувати ці ж гарячі клавіші в спливаючому меню не потрібно.

5. Панель інструментів

Найпростіший спосіб організувати **Панель інструментів** - установити звичайну панель **TPanel** з настроюваннями властивостей Align = alTop, AutoSize = True, і вже на цю панель установити кнопки **TSpeedButton**. І панель, і кнопки ми з вами вивчали в лекції №3, тому зупинятися на цьому не будемо. Вивчимо іншої, більше професійний спосіб організації панелі інструментів.

Для цього нам буде потрібно компонент TToolBar із вкладки **Common Controls Палітри компонентів**:

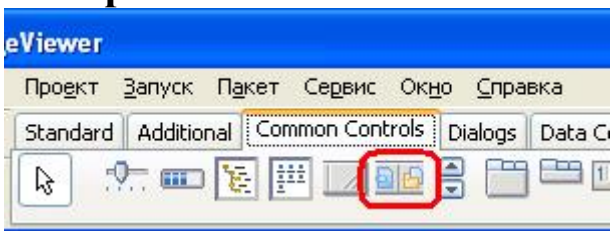


Рис. 16.13. *Компонент TToolBar*

Просто встановіть його на форму, і він сам займе потрібне місце - розтягнеться по всій верхній границі вікна. Зверну вашу увагу на деякі властивості компонента, які можуть представляти для нас інтерес.

Images	- посилання на список зображень для кнопок панелі. Виконує таку ж функцію, як відповідна властивість меню.
DisabledImages	- посилання на список зображень, які будуть прив'язані до неактивних кнопок. Найчастіше цю властивість залишають порожньою, у такому випадку в неактивних кнопок (у яких Enabled = False) буде таке ж зображення, як в активних, але тільки сірих кольорів. При бажанні для неактивних кнопок можна підібрати альтернативні зображення.
HotImages	- посилання на список зображень для кнопок, над якими в цей момент перебуває покажчик миші. Найчастіше цю властивість не заповнюють, і тоді

	зображення в таких кнопок не змінюється. При бажанні, можна підібрати альтернативні зображення, щоб виділяти поточну кнопку.
EdgeBorders	- установка бордюру по сторонах панелі. Має 4 підвластивості, що відповідають границям панелі, і якщо у відповідній підвластивості встановлене True, то дана межа буде виділена. За замовчуванням, включена тільки верхня межа, що відокремлює панель інструментів від рядка з головним меню. Якщо бажаєте, можете включити відображення всіх меж.
Flat	- якщо True, те кнопки плоскі, як у сучасних панелях Windows, якщо False - опуклі, як у старих програмах. За замовчуванням дорівнює True, так краще й залишити.
ShowHint	- якщо True, то спливаючі підказки будуть з'являтися, якщо False, то ні. З такою властивістю нам уже доводило зустрічатися. На панелі інструментів звичайно відображаються кнопки із зображеннями, але без тексту. Користувач, якщо тільки знайомиться із вашою програмою, спочатку може й не зрозуміти, яка кнопка для чого потрібна. Тому в панелі інструментів дану підказку краще включати (за замовчуванням ShowHint = False). Потім у властивості **Hin**t кнопок ви пропишіть потрібні підказки, і користувач зможе їх прочитати, коли підведе курсор миші до тієї або іншої кнопки.

Зробимо на панелі **ToolBar1** наступні установки:

EdgeBorders	- включаємо всі межі, якщо є бажання. А взагалі це необов'язково.
Images	- виберіть наш ImageList1. Як бачите, той самий список зображень ми використаємо й для головного меню, і для спливаючого, і для панелі інструментів.

ShowHint = True

Інші властивість можна не змінювати.

Займімося створенням кнопок. Спочатку зробимо кнопку «**Вихід із програми**», потім встановимо роздільник, потім кнопки «**Відкрити**

файл із зображенням» і «**Зберегти файл як**». Для команд пункту «**Вікно**» ми кнопок робити не будемо, тим більше, що в нас немає для них зображень. Отже, клацніть правою кнопкою миші по панелі інструментів, і виберіть команду «**Нова кнопка**». На панелі інструментів з'явилася кнопка, який Lazarus автоматично привласнили ім'я **ToolButton1**. Змініть її властивість Name на bExit, щоб назва відповідала дії. У властивості Hint напишіть наступну підказку: **Вихід із програми**

А у властивості ImageIndex виберіть зображення з індексом 2. Тепер перейдемо на вкладку **Події Інспектори об'єктів**, і у властивості OnClick виберемо FileExitClick - подія OnClick для пункту «**Вихід**» головного меню. Як бачите, для панелі інструментів нам теж не требба знову писати код.

Клацніть правою кнопкою по панелі інструментів, і виберіть команду «**Новий роздільник**». Тим самим ви вставите роздільник - вертикальну рису між попередньою кнопкою й наступними. Нічого із цим роздільником робити більше не потрібно, хіба що вам здається, що роздільник занадто (або навпаки, недостатньо) широкий. У цьому випадку виділите роздільник і змініть його властивість Width.

Вставте ще одну кнопку. Назвіть її **bOpen**, виберіть для неї зображення під індексом 0, у властивості Hint напишіть наступну підказку: **Відкрити файл із зображенням**

У події OnClick цієї кнопки виберіть відповідну подію з головного меню.

Вставимо ще одну, останню кнопку. Назвемо її **bSaveAs**, в ImageIndex виберемо картинку з індексом 1, у властивості Hint напишемо підказку: **Зберегти файл як**

Виберемо для кнопки відповідна подія OnClick.

У результаті у вас повинна вийти така **Панель інструментів**:

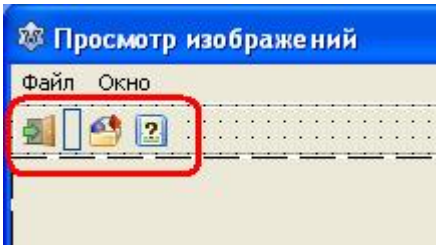


Рис. 16.14. *Панель інструментів у проєкті*

Це все. Збережіть проєкт і запустіть його на виконання. Якщо ви все зробили правильно, тепер у вас є правильно діюча панель керування із трьома кнопками й одним роздільником.

Тема 17. Модулі

1. Модулі
2. Структура модулів
3. Ім’я модуля
4. Створення модуля
5. Приклад включення модуля в проєкт

1. Модулі

Модуль (англ. Unit) - це автономно компільована програмна одиниця, що може містити в собі такі об’єкти програми, як типи, константи, змінні й підпрограми (процедури й функції).

Щораз, коли ми створюємо будь-яке вікно, для нього автоматично створюються два файли: файл описів *.lfm і модуль *.pas. У файлі описів у простому текстовому вигляді втримується опис форми - які компоненти на ній перебувають, як вони налаштовані, які параметри втримуються у властивостей кожного компонента. Редагувати цей файл вручну вкрай не рекомендується, краще все зробити в Lazarus, в Інспекторі об’єктів і Редакторі форм, так ви гарантовано уникнете помилок в описі форми.

У файлі модуля перебуває вихідний код - те, що робить програма, коли користувач взаємодіє з різними компонентами або із самою формою. Весь той код, всі процедури й функції, які ми вводили, перебувають у модулі того або іншого вікна. Однак значення модулів цим не обмежується. Модулі необов’язково повинні бути пов’язані з вікном, вони можуть існувати в проєкті й самі по собі! Найчастіше це роблять із двома цілями:

- Розробка додатків декількома програмістами (кожний пише свій код, що потім збирається й компілюється в єдиний проєкт).
- Програміст може збирати власну бібліотеку часто використовуваних функцій і процедур, поміщаючи їх у незалежні модулі, які потім він може підключити до будь-якого свого проєкту або навіть поділитися ними з іншими програмістами.

Як ви розумієте, найбільше нас цікавить друга мета, із цим рано або пізно зіштовхується будь-який практикуючий програміст. Крім того, на просторах Інтернету ви можете знайти безліч найрізноманітніших

інструментів, виконаних у вигляді модулів, які автори викладають у загальний доступ, їх можна скачати й використати у своїх програмах.

Якось раз один мий учень запитав, чи не є дурним тоном використати чужий код. Давним-давно, на зорі програмування це було не дуже прийняте. Однак тепер усе змінилося - програми стали набагато більше, функціональніше, проста програма містить дуже багато коду. Наприклад, ми з вами не раз застосовували функцію Length(), що повертає довжину рядка в символах або довжину масиву в елементах. У принципі, ми могли б піднапружитися, і самостійно реалізувати цю функцію стандартними засобами Паскаля. Однак це було б невиправданою витратою часу, адже ця функція вже реалізована й включена в бібліотеку **Lazarus**! Я відповів тому учневі, що дурним тоном буде винаходити велосипед, тобто, витрачати час на розробку того, що давним-давно реалізовано. Звичайно, якщо ви будете використати чужий код без ведена й дозволу автора, те це буде погано. Однак якщо автор сам виклав свій код, те чому б ним не скористатися? Зрозуміло, отут залишається ризик того, що автор - не дуже гарний програміст, і його код може «глючить» - працювати з помилками. У цьому випадку можу дати таку раду: шукайте джерела на пристойних, більших сайтах, які не закинуті й часто обновляються. У наступній лекції, на лабораторній роботі, ми використаємо один такий чужий модуль.

Зараз же нас більше цікавить, як створити модуль самостійно, як помістити в нього процедури й функції, які можуть знадобитися нам надалі, у різних проєктах. Ми реалізуємо модуль із «захистом від дурнів» - з перевіркою правильності числа, що вводить користувачем. Причому числа можуть бути цілі беззнакові (від 0 і вище), цілі зі знаком (можуть бути негативними) і речовинні, які все зі знаком.

2. Структура модулів

Модуль у Паскалі має наступну структуру:

Unit <ім’я модуля>;
interface
<інтерфейсна частина> //відкрита частина
implementation
<виконує частину, що (реалізація)> //закрита частина
initialization
<необов’язкова ініціююча частина>
finalization
<необов’язкова завершальна частина>

end.

2. Ім’я модуля

Ім’я модуля повинне точно відповідати імені файлу цього модуля. Тобто, якщо ви створюєте модуль

unit MyUnit;

те й зберігати його потрібно обов’язково у файл **MyUnit.pas**. При цьому намагайтеся вибирати для модулів зрозумилі імена, що не відповідають стандартним модулям. Імена стандартних модулів, що підключають, ви можете бачити в розділі uses **у Редакторі** коду.

Совет: якщо при редагуванні коду й утримуючи натиснутої клавішу **<Ctrl>** ви підведете покажчик миші до імені якого або підключати модуля, що, його ім’я перетвориться в гіперпосилання, клацнувши по який ви відкриєте цей модуль. Можете подивитися, як реалізуються професійні модулі, заглянути «у внутрішності» процедур і функцій.

Інтерфейсна частина

Інтерфейсна частина - це відкрита частина, починається зі службового слова interface. Тут можна повідомляти глобальні типи, константи, змінні, функції й процедури, які будуть доступні для всіх програм і модулів, до яких ви приєднаєте даний модуль. Тут же при бажанні ви можете вказати службове слово uses, після якого перелічити ті модулі, які бажаєте підключити до цього модуля. Приклад (виконувати не потрібно):

```
Unit MyUnit;
interface
uses windows;
const
  MyPI = 3.1415;
var
  MyGlobalVariable: Integer;
function MyGlobalFunc(a, b: string): boolean;
procedure MyClobalProc;
```

Тут ми створюємо модуль із ім’ям MyUnit, якому потрібно зберегти у файл із ім’ям **MyUnit.pas**. Далі, ми повідомляємо інтерфейсну частина. Отут за допомогою службового слова uses (англ. використати) ми вказуємо, що до цього модуля потрібно ще підключити модуль windows, що входить у стандартну поставку **Lazarus**.

Далі, ми повідомляємо константу MyPI, що відразу привласнюємо значення, і змінну MyGlobalVariable. І константою, і змінної можна буде користуватися ззовні - там, де буде підключатися цей модуль.

Робоча частина

Робоча частина (розділ реалізації) починається зі службового слова implementation. Це закрита частина. Тут описуються ті процедури й функції, які були оголошені в інтерфейсній частині, причому заголовки підпрограм повинні в точності відповідати попередньому оголошенню. Також тут можна повідомляти типи, константи й змінні, а також і підпрограми, які будуть видні тільки в цьому модулі й не зможуть використатися ззовні. Приклад:

```
implementation
uses MyOldUnit;
var
  MyLocalVariable: Real;

function MyGlobalFunc(a, b: string): boolean;
begin
  …
end;

procedure MyClobalProc;
begin
  …
end;
```

Як видно із прикладу, у робочій частині, також можна підключати зовнішні модулі. Тут же ми оголосили змінну MyLocalVariable, вона є глобальної усередині цього модуля, тобто, її можна використати у функції й процедурі, однак ззовні вона не буде видна. Далі в нас іде реалізація (опис) оголошених раніше підпрограм.

Ініціююча й завершальна частини

Ці частини модуля необов’язкові, їх можна не вказувати, як правило, їх і не використовують.

Ініціююча частина починається службовим словом initialization і містить код, що буде виконаний тільки один раз - до передачі керування в основну програму. Отут можна зробити підготовчу роботу - відкрити потрібний файл, привласнити значення будь-яким змінним і т.п.

Завершальна частина починається службовим словом finalization і містить код, що буде виконаний теж один раз - при завершенні роботи основної програми. Тут звичайно звільняють зайняті програмою ресурси, закривають відкриті файли, зберігають результати роботи й т.п.

Кінець модуля

Модуль завершується службовим словом end із крапкою наприкінці. Це єдиний випадок, коли після end ставиться не крапка з комою, а крапка. Весь подальший текст, якщо він є, компілятором буде ігнорований. Крім того, якщо подальший текст не відповідає синтаксису, компілятор виведе повідомлення про помилку, і не дасть скомпілювати програму. Так що, якщо вам потрібно вказати якісь коментарі, наприклад, з порадами як користуватися модулем, рекомендую робити їх у фігурних дужках на самому початку модуля, навіть до службового слова Unit. Так звичайно й роблять. А от після end із крапкою вже нічого писати не потрібно.

4. Створення модуля

Ну от, з теорією покінчили, приступимося до практики. Як уже говорилося вище, зараз ми з вами реалізуємо «захист від дурнів». Допустимо, у програми в нас є компоненти TEdit, у які користувач повинен вписати якесь число - ціле без знака, ціле зі знаком або речовинне, котре ще називають дійсним числом. Повинен те він повинен, але хто може припустити, що він туди насправді введе? Може бути, слово, дужки, знак відсотка - користувач непередбачений! Як сказав один мудрий програмист, «Якщо є хоч найменша можливість увести в програму неправильні дані, користувач цю можливість неодмінно знайде». Наша з вами завдання - передбачити всі й позбавити його цієї можливості. Адже нам надалі прийде перетворювати це число з текстового формату в дійсні числа, використовуючи StrToInt(), StrToFloat() і т.п. функції. Якщо ж користувач уведе помилкові дані, то програма викличе помилку.

Оскільки робити такі перевірки програмистові доводиться досить часто, має сенс реалізувати їх у вигляді окремого модуля, що потім можна буде підключити до будь-якої програми. І цей модуль буде початком нашої бібліотеки модулів, що, як я сподіваюся, ви будете постійно поповнювати як своїм, так і чужим кодом.

Модуль - це текстовий файл із розширенням *.pas, якому можна створити в будь-якому текстовому редакторі. Однак буде зручніше скористатися **Редактором коду** Lazarus - він підсвічує синтаксис коду,

виводить підказки, при натисканні **<Ctrl+J>** виводить список із часто використовуваними командами. Так що для початку відкриємо **Lazars**. Якщо там завантажився попередній або новий проєкт, то командою меню **Файл->Закрити** закрийте його. Далі, виберіть команду **Файл->Створити модуль**.

Відразу ж з'явиться фрагмент модуля з наступним кодом:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
implementation
end.
```

Ім'я модуля за замовчуванням Unit1 - воно ні про що нам не говорить, замінимо його на numbers. Далі, ідуть директиви компіляторові, потім інтерфейсна частина із двома підключеними модулями - їх відключати не будемо, оскільки там можуть зберігатися описи глобальних змінних, констант, підпрограм, які можуть нам знадобитися. Зокрема, для перевірки правильності переведення роздільника в дійсне число, ми використаємо системну змінну DecimalSeparator, що описана в одному із цих модулів, і яку без нього не вийде використати.

Вцілому код модуля наступний (можете просто скопіювати його звідси):

```
//-i-i-i Початок модуля -i-i-i-i-
```

У модулі описані дві функції, які перевіряють правильність що вводить користувачем символу. Функція TrueIntKeys перевіряє цілі числа й має параметри: key - введений користувачем символ; str - рядок в TEdit;

sign - якщо True, то число зі знаком, якщо False - без знаку. Якщо користувач ввів правильний символ, функція його не змінює, інакше повертає

символ #0 - тобто, нічого. Функція TrueFloatKeys аналогічним способом робить перевірку дійсних чисел. Приклад використання:

```
Key:= TrueIntKeys(Key, Edit1.Text, True); //ціле зі знаком
Key:= TrueIntKeys(Key, Edit1.Text, False); //ціле без знаку
Key:= TrueFloatKeys(Key, Edit1.Text); //дійсне
```

```
unit Numbers;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
//перевірка правильності символу в цілому числі:
function TrueIntKeys(key:char; str:string; sign:boolean):char;
//перевірка правильності символу в дійсному числі:
function TrueFloatKeys(key:char; str:string):char;

implementation

{перевірка правильності символу в цілому числі}
function TrueIntKeys(key:char; str:string; sign:boolean):char;
begin
  //спочатку вкажемо, що повертається той же символ, який ввів користувач:
  Result:= key;
  //далі робимо перевірку на правильність символу. якщо символ не правильний,
  //ми його заборонимо:
  case key of
    //всі числа дозволяємо:
    '0'..'9': ;
    //backspace дозволяємо:
    #8: ;
    //якщо знакове, то дозволяємо мінус за умови, що мінус - перший
    //символ у рядку:
    '-': if sign and (Length(str) = 0) then Result:= key
         else Result:= #0; //якщо беззнакове, або мінус не перший, забороняємо
  //всі інші символи забороняємо:
  else Result:= #0;
  end; //case
end;

{перевірка правильності символу в дійсному числі}
function TrueFloatKeys(key:char; str:string):char;
begin
```

```
    //спочатку вкажемо, що повертається той же символ, який ввів користувач:
    Result:= key;
    //далі робимо перевірку на правильність символу. якщо символ не правильний,
    //ми його заборонимо:
    case key of
     //всі числа дозволяємо:
     '0'..'9': ;
     //backspace дозволяємо:
     #8: ;
     //якщо роздільника ще немає - виводимо правильний роздільник,
     //інакше нічого не виводимо
     ',' , '.' : if Pos(DecimalSeparator, str)= 0 then
                   Result := DecimalSeparator
                 else Result := #0;
     //дозволяємо мінус за умови, що мінус - перший символ у рядку:
     '-': if Length(str) = 0 then Result:= key
          else Result:= #0; //якщо мінус не перший, забороняємо
    //всі інші символи забороняємо:
    else Result:= #0;
    end; //case
  end;

  end.
  //-i-i-i- Кінець модуля -i-i-i-i-
```

Лістинг . (html, txt)

```
Рядка
//-i-i-i Початок модуля -i-i-i-i-
//-i-i-i- Кінець модуля -i-i-i-i-
```

Копіювати в модуль не обов'язково, вони тільки позначають границі лістинга цього коду.

Першою справою ми змінили ім'я модуля з Unit1 на Numbers. Тепер цей модуль потрібно зберегти в яку-небудь тимчасову папку, у файл **Numbers.pas**. Разом з модулем буде зберігатися й проєкт, не звертайте на це увагу - проєкт нам не потрібний, його потім можна буде видалити, залишивши лише файл із модулем.

Виберіть команду **Файл->Зберегти**. Спочатку буде запит на збереження проєкту, його ім’я можна не міняти, а папку вибрати тимчасову, наприклад, **C:\Temp**. Потім буде запропоновано зберегти модуль, причому Lazarus сам підставить ім’я numbers. Просто натисніть «**Зберегти**». Тепер відкрийте ту папку **Провідником** Windows або файловим менеджером, і побачите файл **numbers.pas** - це і є наш модуль, інші файли в цій папці нам не потрібні.

Перш, ніж рухатися далі, давайте розберемося з кодом модуля.

У первісній заготівлі модуля ми тільки змінили його ім’я. Потім повернулися на самий початок, у вигляді багаторядкового коментаря описали роботу із цим модулем. Робити це досить корисно, тому що модуль прослужить вам не рік і не два - надалі ви можете просто забути, як він був реалізований. Щоб знову не вивчати його вихідний код, простіше описати реалізовані тут інструменти й рекомендації з їхнього використання. А якщо ви плануєте викласти свій модуль в Інтернет, має сенс також додати в коментар рядка

Автор: ПІБ автора

Дата: Дата реалізації

Потім в інтерфейсній частині ми оголосили дві функції:

```
function TrueIntKeys(key:char; str:string; sign:boolean):char;
function TrueFloatKeys(key:char; str:string):char;
```

Виходимо ось із чого. У компонента TEdit є подія OnKeyPress, що виникає до того, як введений користувачем символ потрапить у рядок Edit1.Text. У події є змінна Key, що і містить цей символ. Як правило, у цій події перевіряють коректність уведеного користувачем символу. Якщо символ правильний, то змінну залишають без змін. Якщо неправильний, його можна виправити - у коді ми порівнюємо роздільник не із крапкою або комою, а зі змінною DecimalSeparator, що гарантує правильний роздільник, незважаючи на те, крапку натиснув користувач, або кому. Ну а якщо вже символ зовсім некоректний (наприклад, буква в числі), то змінній Key у події OnKeyPress можна привласнити #0 - нульовий символ, тобто, ніщо. У цьому випадку, користувач може натискати на неправильні клавіші хоч тиждень, в TEdit ці символи не з’являться. І, звичайно ж, у перевірці обов’язково треба врахувати, що користувачеві потрібно надати можливість редагувати неправильно уведені символи. Так, клавіша **<Backspace>**,

що затирає символ ліворуч від курсору, має код #8 - цю клавішу потрібно дозволити.

Реалізацію самих функцій ми зробили в робочому полі. Спочатку змінній Result ми привласнили те значення, що вже зберігається в параметрі key. Адже якщо символ правильний, те простіше нічого не робити, чим щораз писати

Result:= key;

Потім для цілого числа ми влаштовуємо наступну перевірку:

```
case key of
  //всі числа дозволяємо:
  '0'..'9': ;
  //backspace дозволяємо:
  #8: ;
  //якщо знакове, то дозволяємо мінус за умови, що мінус - перший
  //символ у рядку:
  '-': if sign and (Length(str) = 0) then Result:= key
       else Result:= #0; //якщо беззнакове, або минус не перший, забороняємо
  //всі інші символи забороняємо:
  else Result:= #0;
  end; //case
```

Коментарі досить докладні, але все-таки розберемо код «по цеглинках». Перемикач case виконує один з операторів, залежно від значення параметра key. Рядок умов

```
'0'..'9': ;
```

означає, що якщо в key зберігається символ будь-якої цифри, від 0 до 9, то нічого робити не потрібно, тому що Result уже містить символ, введений користувачем, і цей символ коректний. Якби ми, наприклад, хотіли б зробити перевірку на латинський рядковий символ, то вказали б діапазон 'a'..'z'.

Далі, у нас перевірка

```
#8: ;
```

Як ми вже з'ясували, код #8 має клавіша **<Backspace>**, що ми теж дозволяємо, тому нічого не робимо.

А от далі цікавий код. Число може бути знаковим (параметр sign=True) - у цьому випадку нам потрібно дозволити символ мінуса. Однак все одно, ми повинні перевірити, що якщо це мінус - він повинен бути першим символом у рядку, і що він єдиний мінус у рядку. Погодитеся,

12--34

це неправильне ціле число. Якщо ж число беззнакове, то ми й зовсім забороняємо цей мінус. Все це реалізується наступною перевіркою:

```
'-': if sign and (Length(str) = 0) then Result:= key
     else Result:= #0; //якщо беззнакове, або мінус не перший, забороняємо
```

Цей оператор виконується, якщо в параметрі key у нас символ мінуса. Умовний оператор if виконується, тільки якщо в параметрі sign у нас True, і якщо рядок порожній (Length(str)=0). Погодитеся, якщо в рядку вже щось є, то мінус не буде першим символом! Якщо ж число беззнакове або вводить символ, що, - не перший, виконується частина після else - функція повертає код #0, тобто, результат буде такий, начебто користувач нічого й не натискав.

І на закінчення цієї перевірки ми також повертаємо код #0, сюди підпадуть всі інші символи. У результаті, користувач не зможе ввести в TEdit ніякого неправильного символу.

Перевірка правильності дійсного числа у функції TrueFloatKeys така ж, як і для цілого числа, але з двома виключеннями: по-перше, всі дійсні числа знакові (можуть бути негативними), тому параметр sign отут не потрібний, і перевірки на знаковість немає. По-друге, дійсне число може містити роздільник цілої й дробової частини, а в рядку не може бути двох роздільників. Це реалізується наступною перевіркою:

```
',' , '.' : if Pos(DecimalSeparator, str)= 0 then
              Result := DecimalSeparator
            else Result := #0;
```

Цей оператор виконується, якщо в key крапка або кома. Функція Pos(DecimalSeparator, str)

поверне нуль тільки в одному випадку: якщо правильний роздільник, що зберігається в системної зминної DecimalSeparator, не виявлений у рядку str. У цьому випадку ми повертаємо правильний роздільник, що

може бути як крапкою, так і комою, залежно від мови операційної системи. Як бачите, це універсальна перевірка, що не залежить від мови, що до того ж рятує користувача від необхідності стежити за тим, крапкою або комою він відокремлює цілу й десяткову частину числа. Користувачеві буде зручно користуватися правою частиною клавіатури для уведення чисел. Якщо ж у рядку правильний роздільник уже є, то ми повертаємо нульовий код, забороняючи другий такий же роздільник.

Тепер про те, як використати даний модуль в інших проєктах. Це можна зробити двома способами. Перший полягає в тому, що модуль потрібно не тільки зберегти, але й скомпілювати. У цьому випадку, у папці із проєктом з'явиться папка **lib\i386-win32** (для Windows), у якій будуть два файли модуля - **numbers.o** і **numbers.ppu**. Це - наш модуль у скомпільованому (виконуючому) виді. Тепер потрібно знайти папку, у якій в інших папках зберігаються бібліотеки поставляють із Lazarus різних модулів. Для Windows за замовчуванням це папка

C:\lazarus\fpc\2.6.2\units\i386-win32

Адреса може бути й іншою, якщо у вас інша версія FPC. Тут ми можемо створити свою папку, наприклад, **MyModules** або **MyUnits**, куди й будемо копіювати наші модулі у вигляді пари файлів, а потім підключати їх до програм, включаючи ім'я модуля в розділ uses. Однак цей варіант не дуже гарний - файли двійкові, прочитати їх простим текстовим редактором ми не зможемо. Якщо таких файлів багато, як довідатися, де потрібні нам інструменти, як доповнювати ці модулі новими функціями й процедурами?

Я пропоную інший спосіб. Створіть де-небудь на диску, де зберігаються ваші важливі документи, папку з будь-яким ім'ям, куди ви будете збирати ваші (і чужі) модулі, і скопіюйте туди файл **numbers.pas** - наш модуль у вихідному, не скомпільованому виді. А потім, якщо в якомусь проєкті нам потрібно буде робити перевірку на правильність числа, ми просто будемо копіювати **numbers.pas** у папку із цим проєктом! Таким чином, він завжди буде доступний у текстовому варіанті, по коментарях перед модулем ми легко зможемо знайти саме те, що нам потрібно.

Отже, ви створили папку, що будете використати, як бібліотеку ваших модулів, і скопіювали туди **numbers.pas**? Відмінно, тепер займімося самим проєктом.

Приклад включення модуля в проєкт

Відкрийте **Lazarus** з новим проєктом. Як звичайно, форму назвіть **fMain**, в Caption пропишіть Перевірка чисел, збережіть проєкт у папку **17-01** під ім'ям CheckNum, модуля головної форми, як звичайно, дайте ім'я Main.

Тепер відкрийте ваш улюблений файловий менеджер, з його допомогою знайдіть наш модуль **numbers.pas** і скопіюйте його в папку із проєктом **17-01**. Далі, у **Редакторі коду** знайдіть розділ uses в інтерфейсній частині, поставте кому після останнього зазначеного модуля, додайте туди ім'я

```
numbers:
uses
 Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs,
numbers;
```

Все, наш модуль ми включили в проєкт, тепер можемо реалізувати перевірку. Проєкт буде найпростіший, з мінімальним набором компонентів - тільки щоб перевірити роботу нашого модуля. Встановіть на форму три компоненти TLabel, їхні імена миняти не потрібно, а в Caption компонентів напишіть відповідно, «Ціле зі знаком:», «Ціле без знака:» і «Речовинне:». Правее миток встановіть три компоненти TEdit, їхні імена також залишіть без змин, у кожному TEdit очистите властивість Text. Ну й, нарешті, у нижній частині форми встановите кнопку TBitBtn із вкладки **Additional**, у властивості Kind якої виберіть значення bkClose. Підрівняйте компоненти так, щоб у вас вийшла приблизно така форма:

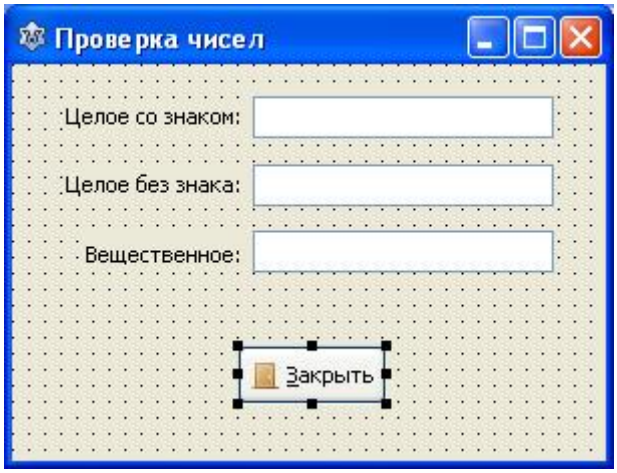


Рис. 17.1. *Форма перевірочного проєкту*

Тепер виділите Edit1, що призначений для введення цілого числа зі знаком. В Інспекторі об'єктів перейдіть на вкладку **Події** й згенеруйте для компонента подія OnKeyPress. Його код:

```
procedure TfMain.Edit1KeyPress(Sender: TObject; var Key: char);
begin
  //перевірка правильності символу, що вводить, у ціле число зі знаком:
  Key:= TrueIntKeys(key, Edit1.Text, true);
end;
```

Для Edit2 код події OnKeyPress буде таким:

```
procedure TfMain.Edit2KeyPress(Sender: TObject; var Key: char);
begin
  //перевірка правильності символу, що вводить, у ціле число без знака:
  Key:= TrueIntKeys(key, Edit2.Text, false);
end;
```

І, нарешті, для Edit3 код події OnKeyPress буде таким:

```
procedure TfMain.Edit3KeyPress(Sender: TObject; var Key: char);
begin
  //перевірка правильності символу, що вводить, у дійсне число:
  Key:= TrueFloatKeys(key, Edit3.Text);
end;
```

Всі, збережете проєкт, скомпілюйте й запустите його на виконання. Тепер, як би ви не намагалися, ввести некоректні значення в рядки введення у вас не вийде:

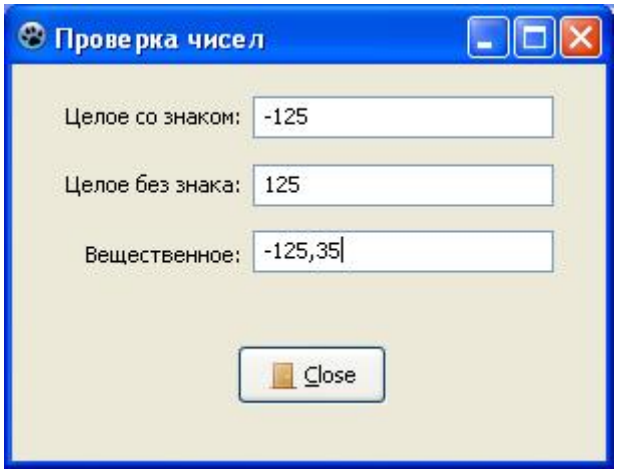


Рис. 17.2. *Працююча програма*

Надалі, точно в такий же спосіб ви зможете використати модуль numbers.pas для перевірки правильності введення чисел у будь-яких ваших наступних проєктах.

Тема 18. Блокнот - шифратор
План

1. Постановка завдання
2.

1. Постановка завдання

Написати простий **Блокнот**, нехай навіть із усіма можливостями стандартного Блокнота Windows, було б нуднувато. От якщо ще навчити його шифрувати/дешифрувати текст, так ще й зі змінним ключем-паролем - зовсім інша розмова! Ідея така: робимо **Блокнот**, додаємо до нього модуль стороннього розробника, у якому реалізовані процедури шифрації й дешифрации тексту. Причому для шифрации/дешифрации потрібно ввести пароль - слово або навіть фраза. І якщо хоч один символ пароля буде невірний, текст не розшифрується! Потім ви зможете роздати такий **Блокнот** всім бажаючим, але прочитати ваш зашифрований текст зможе лише той, хто знає пароль. Приступимо?

Реалізація

Підготовка форми дуже проста, набір компонентів теж не дуже великий. Для початку відкрийте **Lazarus** з новим проєктом. Швидше за все, ви цю програму будете передавати друзям, тому має сенс відразу ж відключити від проєкту вставку налаштувальної інформації. Якщо ви пам’ятаєте, потрібно вибрати команду «**Проєкт -> Параметри проєкту**», у розділі «**Параметри компілятора**» перейти на підрозділ «**Компонування**» і забрати прапорець «**Генерувати налаштувальну інформацію для GDB**».

Далі, як звичайно, форму називаємо **fMain**, зберігаємо проєкт у папку **18-01** під ім’ям **CodeBook**, модулю головної форми привласнюємо ім’я Main.

Тепер займемося самою формою. Блокнот Windows має можливість змінювати свій розмір, мінімізувати й максимізувати вікно, залишимо таку можливість і наш **Блокнот**. Однак за замовчуванням, Lazarus виводить занадто мале вікно. Давайте зробимо вікно середнього розміру. Нехай буде Height = 500, а Width = 750, і розташуйте його

приблизно посередині вікна. У властивості Caption напишіть «Блокнот - шифрувальник».

Далі, встановимо на форму компонентів TMemo, ім’я залишимо за замовчуванням - Memo1, властивість Align встановіть в alClient. Текст у вікні занадто дрібний, відкрийте кнопкою «**…**»редактор властивості Font і виберіть шрифт Times New Roman з розміром 12 (або будь-який інший, який вам сподобається більше, тільки не забувайте, що не у всіх користувачів гарний зір!). Тепер кнопкою «...» зайдіть у редактор властивості Lines і зітріть там весь текст - нам зовсім не потрібно, щоб при відкритті програми відображався текст «Memo1», правда?

Займемося іншими компонентами. Нам будуть потрібні діалоги: TOpenDialog, TSaveDialog, TFontDialog і TColorDialog. Оскільки нам доведеться в коді звертатися до них за іменем, змініть його властивості Name відповідно, на OD, SD, FD і CD.

Крім того, для TOpenDialog і TSaveDialog зробіть наступні налаштування:

Filter=Текстові файлы|*.txt|все файли|*.*

DefaultExt=.txt

Ну, і наостанок, нам буде потрібно головне меню TMainMenu. Ім’я самого компонента можна залишити без змін - MainMenu1, а от пункти цього меню будемо перейменовувати. Деяким підпунктам меню будемо привласнювати «гарячі клавіші». Першим у таблиці йдуть параметри розділів (виділяються жирним шрифтом), потім параметри їхніх підрозділів. Підпункти-роздільники, у яких буде відбиватися розділова лінія, будемо називати нейтрально: N1, N2, і т.д. Отже, створюємо наступні пункти й підпункти:

Таблиця 18.1. Параметри розділів і підрозділів головного меню

Name	Caption	ShortCut
Розділ «Файл»		
FileMenu	**Файл**	
FileCreate	Створити	**Ctrl+N**
FileOpen	Відкрити	**Ctrl+O**
FileSave	Зберегти	**Ctrl+S**
FileSaveAs	Зберегти як…	

N1	-	
FileExit	Вихід	
Розділ «Виправлення»		
EditMenu	**Виправлення**	
EditCancel	Скасувати	**Ctrl+Z**
N2	-	
EditCut	Вирізати	**Ctrl+X**
EditCopy	Копіювати	**Ctrl+C**
EditPaste	Вставити	**Ctrl+V**
EditDelete	Видалити	
N3	-	
EditSelectAll	Виділити всі	**Ctrl+A**
Розділ «Формат»		
FormatMenu	**Формат**	
FormatFont	Шрифт	
FormatColor	Кольори	
N4	-	
FormatWordWrap	Перенос за словами	
Розділ «Кодування»		
CoderMenu	**Кодування**	
CoderCode	Шифрувати	
CoderDecode	Дешифрувати	
Розділ «Довідка»		
HelpMenu	**Довідка**	
HelpAbout	Про програму	

В результаті у вас має вийти таке меню:

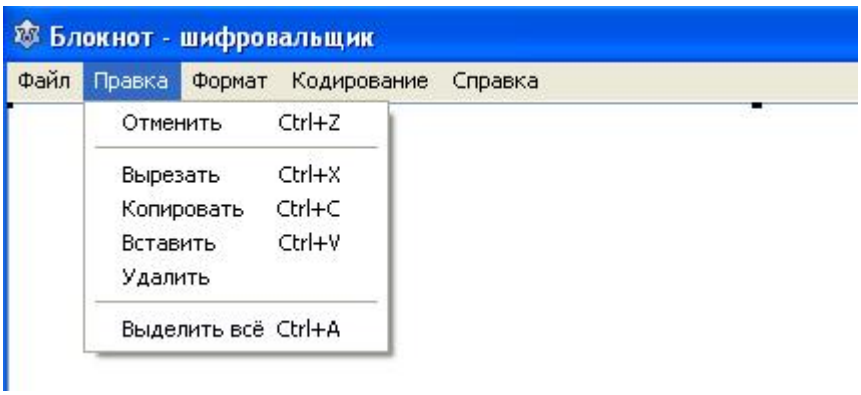


Рис. 18.1. *Відкритий розділ меню «Виправлення»*

Нам потрібно, щоб за замовчуванням, наш **Блокнот** робив перенос за словами. Для цього знадобиться зробити три речі:

1. Встановити в Memo1 вертикальну смугу прокручування (ScrollBars=ssVertical).
2. Переконатися, що його властивість WordWrap (перенос за словами) встановлене в True.
3. Поставити прапорець на пункт меню «**Формат->Перенос за словами**». Для цього в **Редакторі меню** виділіть даний пункт, і встановіть True у його властивості Checked. Результат можна буде подивитися, закривши редактор меню, і відкривши саме головне меню:

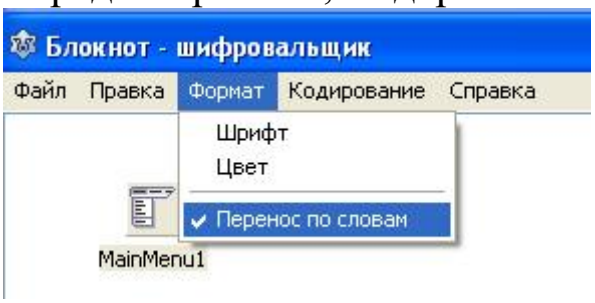


Рис. 18.2. *Виділений прапорцем пункт меню*

Оскільки стандартний Блокнот Windows не має картинок у головному меню, ми їх теж прив'язувати не будемо. Втім, якщо є бажання, ви можете самостійно додати TImageList і прикрасити меню картинками. Як це робити, ми обговорювали в лекции №16.

Ну от, усі компоненти ми підготували, і налаштували, як потрібно. Але перш ніж перейдемо безпосередньо до кодування, підготуємо модуль. Існує цей модуль у проєкті DelphiWorld багато років, і ми використаємо його, щоб зашифрувати ключі активації програм. Основна частина модуля написана мовою Асемблер, однак для використання цього модуля знання Асемблера нам не потрібно. Оскільки автор виклав модуль у відкритий доступ (спасибі йому

велике за це), ми з вами можемо використати його у своїх програмах із чистою совістю.

Відкрийте будь-який текстовий редактор, хоча б той же стандартний **Блокнот** Windows. Скопіюйте в нього нижченаведений рядок лістингу

```
//-i-i-i Початок модуля -i-i-i-i-
//-i-i-i- Кінець модуля -i-i-i-i-
```

копіювати в модуль не потрібно вони тільки позначають межі листинга цього коду:

```
//-i-i-i Початок модуля -i-i-i-i-
{ **** UBPFD ********** by delphibase.endimus.com ****
>> Шифрування й дешифрування текстів за принципом S-Coder зі схованим ключем

Після підключення модуля ключове слово (фраза) будуть в MyCript.MyPassword;
Шифрувати приблизно так:
  Memo1.Text:= MyCript.Write(MyCript.Encrypt(Memo1.Text));
Дешифрувати приблизно так:
  Memo1.Text:= MyCript.Decrypt(MyCript.Read(Memo1.Text));

Copyright:   EFD Systems http://www.mindspring.com/~efd
Дата:        1 лютого 2021р.
********************************************************** }

unit MyCript;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

//Це просто інтерфейс функцій EnCipher і Crypt для шифрування рядка тексту:
function Encrypt(text: Ansistring): Ansistring;
//Це просто інтерфейс функцій EnCipher і Crypt для дешифрування рядка тексту:
```

```
function Decrypt(text: Ansistring): Ansistring;

function Write(text: Ansistring): Ansistring;
function Read(text: Ansistring): Ansistring;

var
  MyPassword: AnsiString; //тут буде пароль

implementation


procedure EnCipher(var Source: AnsiString);

{Low order, 7-bit ASCII (char. 32-127) encryption designed for database use.
 Control and high order (8 bit) characters are passed through unchanged.

 Uses a hybrid method...random table substitution with bit-mangled output.
 No passwords to worry with (the built-in table is the password). Not industrial
 strength but enough to deter the casual hacker or snoop. Even repeating char.
 sequences have little discernable pattern once encrypted.

 NOTE: When displaying encrypted strings, remember that some characters
       within the output range are interpreted by VCL components; for
       example, '&'.}
begin
{$asmmode intel}
asm
   Push ESI //Save the good stuff
   Push EDI

   Or EAX,EAX
   Jz @Done
   Push EAX
   Call UniqueString
```

```
    Pop EAX
    Mov ESI,[EAX] //String address into ESI
    Or ESI,ESI
    Jz @Done
    Mov ECX,[ESI-4] //String Length into ECX
    Jecxz @Done //Abort on null string
    Mov EDX,ECX //initialize EDX with length
    Lea EDI,@ECTbl //Table address into EDI
    Cld //make sure we go forward
@L1:
    Xor EAX,EAX
    Lodsb //Load a byte from string
    Sub AX,32 //Adjust to zero base
    Js @Next //Ignore if control char.
    Cmp AX,95
    Jg @Next //Ignore if high order char.
    Mov AL,[EDI+EAX] //get the table value
    Test CX,3 //screw it up some
    Jz @L2
    Rol EDX,3
@L2:
    And DL,31
    Xor AL,DL
    Add EDX,ECX
    Add EDX,EAX
    Add AL,32 //adjust to output range
    Mov [ESI-1],AL //write it back into string
@Next:
    Dec ECX
    Jnz @L1
// Loop @L1 //do it again if necessary

@Done:
    Pop EDI
    Pop ESI

    Jmp @Exit
// Ret //this does not work with Delphi 3 - EFD 971022
```

```
@ECTbl: //The encipher table
   DB 75,85,86,92,93,95,74,76,84,87,91,94
   DB 63,73,77,83,88,90,62,64,72,78,82,89
   DB 51,61,65,71,79,81,50,52,60,66,70,80
   DB 39,49,53,59,67,69,38,40,48,54,58,68
   DB 27,37,41,47,55,57,26,28,36,42,46,56
   DB 15,25,29,35,43,45,14,16,24,30,34,44
   DB 06,13,17,23,31,33,05,07,12,18,22,32
   DB 01,04,08,11,19,21,00,02,03,09,10,20
@Exit:

end;//asm
end;

procedure DeCipher(var Source: AnsiString);

{Decrypts a string previously encrypted with EnCipher.}
begin
{$asmmode intel}
asm
   Push ESI //Save the good stuff
   Push EDI
   Push EBX

   Or EAX,EAX
   Jz @Done
   Push EAX
   Call UniqueString
   Pop EAX
   Mov ESI,[EAX] //String address into ESI
   Or ESI,ESI
   Jz @Done
   Mov ECX,[ESI-4] //String Length into ECX
   Jecxz @Done //Abort on null string
   Mov EDX,ECX //Initialize EDX with length
   Lea EDI,@DCTbl //Table address into EDI
   Cld //make sure we go forward
@L1:
   Xor EAX,EAX
```

```
    Lodsb //Load a byte from string
    Sub AX,32 //Adjust to zero base
    Js @Next //Ignore if control char.
    Cmp AX,95
    Jg @Next //Ignore if high order char.
    Mov EBX,EAX //save to accumulate below
    Test CX,3 //unscrew it
    Jz @L2
    Rol EDX,3
@L2:
    And DL,31
    Xor AL,DL
    Add EDX,ECX
    Add EDX,EBX
    Mov AL,[EDI+EAX] //get the table value
    Add AL,32 //adjust to output range
    Mov [ESI-1],AL //store it back in string
@Next:
    Dec ECX
    Jnz @L1
// Loop @L1 //do it again if necessary

@Done:
    Pop EBX
    Pop EDI
    Pop ESI

    Jmp @Exit
// Ret Does not work with Delphi3 - EFD 971022

@DCTbl: //The decryption table
    DB 90,84,91,92,85,78,72,79,86,93,94,87
    DB 80,73,66,60,67,74,81,88,95,89,82,75
    DB 68,61,54,48,55,62,69,76,83,77,70,63
    DB 56,49,42,36,43,50,57,64,71,65,58,51
    DB 44,37,30,24,31,38,45,52,59,53,46,39
    DB 32,25,18,12,19,26,33,40,47,41,34,27
    DB 20,13,06,00,07,14,21,28,35,29,22,15
    DB 08,01,02,09,16,23,17,10,03,04,11,05
```

```
  @Exit:
  end;//asm
  end;

  procedure Crypt(var Source: Ansistring; const Key: AnsiString);

  {Encrypt AND decrypt strings using an enhanced XOR technique similar
to
    S-Coder (DDJ, Jan. 1990). To decrypt, simply re-apply the procedure
    using the same password key. This algorithm is reasonably secure on
    it's own; however,there are steps you can take to make it even more
    secure.

        1) Use a long key that is not easily guessed.
        2) Double or triple encrypt the string using different keys.
           To decrypt, re-apply the passwords in reverse order.
        3) Use EnCipher before using Crypt. To decrypt, re-apply Crypt
           first then use DeCipher.
        4) Some unique combination of the above

    NOTE: The resultant string may contain any character, 0..255.}
  begin
  {$asmmode intel}
  asm
     Push ESI //Save the good stuff
     Push EDI
     Push EBX

     Or EAX,EAX
     Jz @Done
     Push EAX
     Push EDX
     Call UniqueString
     Pop EDX
     Pop EAX
     Mov EDI,[EAX] //String address into EDI
     Or EDI,EDI
     Jz @Done
     Mov ECX,[EDI-4] //String Length into ECX
```

```
    Jecxz @Done //Abort on null string
    Mov ESI,EDX //Key address into ESI
    Or ESI,ESI
    Jz @Done
    Mov EDX,[ESI-4] //Key Length into EDX
    Dec EDX //make zero based
    Js @Done //abort if zero key length
    Mov EBX,EDX //use EBX for rotation offset
    Mov AH,DL //seed with key length
    Cld //make sure we go forward
  @L1:
    Test AH,8 //build stream char.
    Jnz @L3
    Xor AH,1
  @L3:
    Not AH
    Ror AH,1
    Mov AL,[ESI+EBX] //Get next char. from Key
    Xor AL,AH //XOR key with stream to make pseudo-key
    Xor AL,[EDI] //XOR pseudo-key with Source
    Stosb //store it back
    Dec EBX //less than zero ?
    Jns @L2 //no, then skip
    Mov EBX,EDX //re-initialize Key offset
  @L2:
    Dec ECX
    Jnz @L1
  @Done:
    Pop EBX //restore the world
    Pop EDI
    Pop ESI
  end;//asm
  end;

  function Encrypt(text: Ansistring): Ansistring;
  //Це просто інтерфейс функцій EnCipher і Crypt для шифрування рядка тексту
  begin
    {шифруємо текст}
```

```
  EnCipher(Text);
  {зашифровуємо ключем}
  Crypt(Text, MyPassword);
  Result := Text;
end;

function Decrypt(text: Ansistring): Ansistring;
//Це просто інтерфейс функцій EnCipher і Crypt для дешифрування рядка тексту
begin
  {розшифровуємо ключем}
  Crypt(Text, MyPassword);
  {розшифровуємо результат}
  DeCipher(Text);
  Result := Text;
end;

function Write(text: Ansistring): Ansistring;
var
  i: integer;
begin
  Result := ‘‘;
  for i := 1 to Length(text) do
    {одержуємо hex код з тексту}
    Result := Result + InttoHex(ord(text[i]), 2);
end;

function Read(text: Ansistring): Ansistring;
var
  i: integer;
begin
  Result := ‘‘;
  for I := 1 to Length(text) do
    if odd(i) then
      {одержуємо текст із hex коду}
      Result := Result + Chr(StrtoInt(‘$’ + text[i] + text[i + 1]));
end;

end.
```

```
//-i-i-i- Кінець модуля -i-i-i-i-
```

Лістинг . (html, txt)

Збережіть модуль у файл **MyCript.pas** туди, куди ви вирішили збирати свою колекцію модулів. Не забудьте скопіювати цей файл у папку з нашим поточним проєктом.

Також у **Редакторі коду** нашої програми додайте модуль MyCript у розділ uses, у кінець списку підключених модулів.

Теперь можемо приступити безпосередньо до кодування команд. Ви пам’ятаєте, що щоб згенерувати команду OnClick пункту меню, досить вибрати цю команду? Тоді почнемо з більш простих пунктів меню. Найпершим опрацюємо пункт «**Формат->Перенос за словами**»:

```
procedure TfMain.FormatWordWrapClick(Sender: TObject);
begin
  //змінюємо меню:
  FormatWordWrap.Checked:= not FormatWordWrap.Checked;
  //привласнюємо настроювання Memo1:
  Memo1.WordWrap:= FormatWordWrap.Checked;
  //якщо перенос за словами включений, потрібна тільки вертикальна
  //лінійка прокручування, інакше потрібні обидві лінійки:
  if Memo1.WordWrap then Memo1.ScrollBars:= ssVertical
  else Memo1.ScrollBars:= ssBoth;
end;
```

Ми з вами встановили за замовчуванням, що текст в Memo1 переноситься за словами, а даний пункт меню відзначений прапорцем. При виборі цієї команди відбувається зворотна дія: стан Checked пункту меню змінюється на протилежний, стан WordWrap компонента Memo1 стає таким самим. Тобто, якщо прапорець встановлений, то і перенос буде працювати, і навпаки. На завершення ми встановлюємо потрібний стан смуг прокручування. Якщо перенос працює, то горизонтальна смуга не потрібна, тільки вертикальна. Якщо ж переносу немає, то рядок може вийти дуже довгим, тут знадобляться обидві смуги прокручування.

Далі опрацюємо команди меню «**Виправлення**». Вони дуже прості, всього на команду необхідно ввести по одному рядку коду, так що додаткові коментарі не потрібні. Меню «**Виправлення->Скасувати**»:

```
procedure TfMain.EditCancelClick(Sender: TObject);
begin
```

```
  Memo1.Undo;
end;
```

«Виправлення->Вирізати» не складніше:

```
procedure TfMain.EditCutClick(Sender: TObject);
begin
  Memo1.CutToClipboard;
end;
```

«Виправлення->Копіювати»:

```
procedure TfMain.EditCopyClick(Sender: TObject);
begin
  Memo1.CopyToClipboard;
end;
```

«Виправлення->Вставити»:

```
procedure TfMain.EditPasteClick(Sender: TObject);
begin
  Memo1.PasteFromClipboard;
end;
```

«Виправлення->Видалити»:

```
procedure TfMain.EditDeleteClick(Sender: TObject);
begin
  Memo1.ClearSelection;
end;
```

«Виправлення->Виділити всі»:

```
procedure TfMain.EditSelectAllClick(Sender: TObject);
begin
  Memo1.SelectAll;
end;
```

З пунктом «**Виправлення**» розібралися, повернемося до пункту «**Формат**». У нас залишилися неопрацьованими два підпункти - «**Шрифт**» і «**Кольори**». Процедура OnClick для пункту «**Формат->Шрифт**»:

```
procedure TfMain.FormatFontClick(Sender: TObject);
begin
```

```
  //спочатку діалогу привласнюємо шрифт як в Memo:
  FD.Font:= Memo1.Font;
  //якщо діалог пройшов успішно, змінюємо шрифт в Memo:
  if FD.Execute then Memo1.Font:= FD.Font;
end;
```

Тут ми спочатку властивості Font діалогу привласнили такий же шрифт, як і в Memo1. Навіщо потрібно було це робити? Якби ми цього не зробили, діалог не знав би, який шрифт вважається «поточним», там був би порожній рядок. І користувачеві складно було б розібратися, який шрифт, якого розміру він зараз використовується. Тепер же, при виклику діалогу, відразу буде виділений наш шрифт Times New Roman, 12. Якщо ж користувач змінить цей шрифт, то властивості Font компонента Memo1 буде привласнений новий шрифт.

Приблизно так само опрацьовуємо й пункт «**Формат->Кольори**», тільки тут ми звертаємося до властивості Color і діалогу CD:

```
procedure TfMain.FormatColorClick(Sender: TObject);
begin
  //спочатку встановлюємо кольори діалогу, як у Memo:
  CD.Color:= Memo1.Color;
  //якщо діалог пройшов успішно, змінюємо кольори в Memo:
  if CD.Execute then Memo1.Color:= CD.Color;
end;
```

Тепер перейдемо до більш складного пункту меню «**Файл**». Спочатку найпростіша команда, «**Файл->Вихід**»:

```
procedure TfMain.FileExitClick(Sender: TObject);
begin
  Close;
end;
```

Команда «**Файл->Зберегти як**» трішки складніша:

```
procedure TfMain.FileSaveAsClick(Sender: TObject);
begin
  {Переписуємо заголовок вікна діалогу, інакше він буде англійською.
  Якщо збереження відбулося, то властивість Modified в Memo переводимо в false,
  тому що всі зміни уже збережені}
```

```
  SD.Title:= ‘Зберегти як’;
  if SD.Execute then begin
    Memo1.Lines.SaveToFile(Utf8ToSys(SD.FileName));
    Memo1.Modified:= false;
  end; //if
end;
```

Тут ми спочатку змінили заголовок. Чомусь діалог, хоч і має за замовчуванням текст в Title ук4раїнською мовою, у робочій програмі все одно виходить англійською, хоча діалоги шрифту й кольори виводять нормальний український заголовок. Може бути, у майбутніх версіях Lazarus ця проблема буде усунута? Поки ж будемо змінювати заголовки всередині коду. Далі ми зберігаємо текст із Memo1 у файл, причому робимо це не так, як звичайно:

```
Memo1.Lines.SaveToFile(SD.FileName);
```

а так:

```
Memo1.Lines.SaveToFile(Utf8ToSys(SD.FileName));
```

Навіщо такі ускладнення? Справа в тому, що ім’я файлу (властивість FileName діалогу) зберігається у форматі UTF8, а процедура SaveToFile вимагає формату ANSI. Це не проблема, якщо ви будете давати файлу ім’я винятково латинськими буквами або цифрами. Але якщо ви спробуйте зберегти файл із ім’ям під українськими символами (або будь-якими неанглійськими), то вийде те, що називають «кракозябрами» - незрозуміла абракадабра замість імені. Такий файл і відкрити потім не вийде. Якщо ж перетворити ім’я файлу в потрібний формат, це гарантує правильне збереження (завантаження) цього файлу, на якій би мові ви не написали його ім’я.

Далі ми встановлюємо в False властивість Modified компонента Memo1. Це важливо! Властивість Modified стає True, якщо в Memo1 є якісь зміни. Якщо ж ми ці змини зберегли у файл, будемо вважати, що інших змін поки немає. Нам доведеться ще перевіряти стан цієї властивості.

Тепер обробимо команду «**Файл->Зберегти**»:

```
procedure TfMain.FileSaveClick(Sender: TObject);
begin
  {якщо ім’я файлу відомо, то не потрібно викликати діалог SaveDialog,
```

```
    просто викликаємо метод SaveToFile. }
   if SD.FileName <> ‘‘ then begin
    Memo1.Lines.SaveToFile(Utf8ToSys(SD.FileName));
    //установлюємо Modified в false, тому що змини вже зберегли:
    Memo1.Modified:= false;
   end //if
   //інакше ім’я файлу не відомо, викликаємо Зберегти як...:
   else FileSaveAsClick(Sender);
  end;
```

Дивіться, що тут відбувається. Ми дивимося, є чи який-небудь текст у властивості FileName діалогу SD. Якщо текст є (властивість FileName не дорівнює порожньому рядку), виходить, діалог SD уже викликався, або ж відкривався існуючий файл. Але ми знаємо ім’я файлу, куди потрібно зберігати даний текст. Зберігаємо, переводимо в False властивість Modified нашого Memo1. Якщо ж там тексту немає, виходить, текст новий, ще жодного разу не зберігався. У цьому випадку, нам потрібно викликати метод OnClick команди «**Файл->Зберегти як**». Зверніть увагу, як ми це робимо:

```
FileSaveAsClick(Sender);
```

Справа в тому, що як параметр у метод потрібно передати об’єкт, звідки цей метод був викликаний. Змінна Sender має тип TObject і звичайно вказує на цей об’єкт. Ми могли б вказати і який це конкретний об’єкт, наприклад, так:

```
FileSaveAsClick(Memo1);
```

У цьому випадку не буде ніякої різниці, команда однаково спрацює однаково.

Команда «**Файл->Створити**» трішки складніша:

```
procedure TfMain.FileCreateClick(Sender: TObject);
begin
  {Якщо є зміни тексту, запитаємо користувача, чи не хоче він зберегти їх перед створенням нового тексту}
   if Memo1.Modified then begin
    //якщо користувач згодний зберегти зміни:
    if MessageDlg(‘Збереження файлу’,
                ‘Поточний файл був змінений. Зберегти зміни?’,
        mtConfirmation, [mbYes, mbNo, mbIgnore], 0) = mrYes then
```

```
        FileSaveClick(Sender);
    end; //if
    //тепер очищаємо Memo, якщо є текст:
    if Memo1.Text <> ‘‘ then Memo1.Clear;
    //в SaveDialog забираємо ім’я файлу. це буде означати, що файл не збережений:
    SD.FileName:= ‘‘;
  end;
```

Коли користувач вибирає команду «**Файл->Створити**», в Memo уже міг бути якийсь текст. Він потрібний користувачеві, або нам просто потрібно очистити Memo, і підготувати його до нового тексту? Для цього, у випадку, якщо в Memo є зміни, ми виводимо запит-повідомлення MessageDlg(). Якщо користувач бажає зберегти старий текст, ми викликаємо команду «**Файл->Зберегти**». І кожен раз, потім ми очищаємо Memo, якщо там щось є, та очищаємо властивість FileName у діалогу SD (TSaveDialog). Адже текст новий, і ми ще не знаємо, куди користувач захоче його зберегти, вірно?

Код команди «**Файл->Відкрити**» самий довгий, хоча й в ньому нічого складного немає:

```
  procedure TfMain.FileOpenClick(Sender: TObject);
  begin
    //перевірка необхідності збереження файлу, як у Файл->Створити:
    if Memo1.Modified then begin //змини є
      //якщо користувач згоден зберегти зміни:
      if MessageDlg(‘Збереження файлу’,
                ‘Поточний файл був змінений. Зберегти зміни?’,
          mtConfirmation, [mbYes, mbNo, mbIgnore], 0) = mrYes then
          FileSaveClick(Sender);
    end; //if

    //очищаємо ім’я файлу в діалогу OpenDialog, змінюємо заголовок, і
    //викликаємо метод LoadFromFile, якщо діалог відбувся
    OD.FileName:= ‘‘;
    OD.Title:= ‘Відкрити існуючий файл’;
    if OD.Execute then begin
      //очищаємо Memo, якщо є текст:
      if Memo1.Text <> ‘‘ then Memo1.Clear;
      //читаємо з файлу
```

```
    Memo1.Lines.LoadFromFile(Utf8ToSys(OD.FileName));
    //копіюємо ім'я файлу в діалог SaveDialog, щоб потім знати,
    //куди зберігати:
    SD.FileName:= OD.FileName;
  end; //if
end;
```

У першій частині процедури ми дивимося, чи немає змін в Memo, адже користувач міг редагувати один файл, а потім захотів відкрити інший! Потрібно чи зберігати ці зміни, якщо вони є? Як і в минулому прикладі, для цього ми виводимо запит MessageDlg(). Зберігаємо, якщо користувач цього хоче.

Далі ми очищаємо властивість FileName у діалогу відкриття файлу. Навіщо? Добре, якщо цей діалог викликається перший раз, тоді ця властивість буде порожньою. А якщо це не перший файл, що відкривається користувачем? Якщо властивість не очистити, то при виклику діалогу колишній файл буде там за замовчуванням. Але користувач же хоче відкрити інший файл! Тому ми попередньо очищаємо цю властивість, а вже потім викликаємо діалоги. У коді ми змінюємо заголовок і в цього діалогу, інакше він теж вийде англійською мовою.

Потім ми чистимо Memo, якщо там є старий текст, і завантажуємо новий текст із нового файлу, після чого копіюємо ім'я цього файлу в діалог збереження, щоб знати, куди зберігати зміни при виборі команди «**Файл->Зберегти**».

Тепер ми з вами повинні передбачити ще от що. Припустимо, користувач щось записав у блокнот, а потім вирішив його закрити. Зміни є, а чи треба їх зберігати? Потрібно запитати про це в користувача. Це можна було б зробити в команді «**Файл->Виходи**», але ми цього не зробили. Чому? А якщо користувач закриє програму не цією командою, а кнопкою із хрестиком у верхній правій частині вікна? Або кнопками **<Alt+F4>**? От щоб опрацювати закриття програми, яким би чином користувач її не закривав, скористаємося подією OnClose форми головного вікна, що виникає ПЕРЕД закриттям програми. Проблема в тім, що компонент Memo1 закриває всю форму, а нам потрібно її виділити. Виділити форму **fMain** можна або у верхній частині вікна **Інспектори об'єктів**, де всі об'єкти відображені у вигляді дерева - форма там розташована в самому верху, як головний, батьківський компонент. Або можна зробити простіше - виділите

Memo1 і натисніть **<Esc>**, при цьому виділення перейде на зовнішній до Memo1 компонент, тобто, до форми. Потім перейдіть на вкладку **Події Інспектори об'єктів**, і згенеруйте подію OnClose. Його код вам уже знайомий по попередньому коді:

```
procedure TfMain.FormClose(Sender: TObject; var CloseAction: TCloseAction);
begin
  {Якщо є зміни тексту, запитаємо користувача, чи не хоче він зберегти їх перед створенням нового тексту}
  if Memo1.Modified then begin
    //якщо користувач згоден зберегти зміни:
    if MessageDlg('Збереження файлу',
              'Поточний файл був змінений. Зберегти зміни?',
        mtConfirmation, [mbYes, mbNo, mbIgnore], 0) = mrYes then
          FileSaveClick(Sender);
  end; //if
end;
```

Лістинг . (html, txt)

Тепер яким би способом користувач не закривав програму, буде перевірятися - чи немає змін в Memo, і якщо є, вийде запит - чи зберігати їх. Практично все, що вміє робити стандартний Блокнот, уміє тепер і наша програма. Самий час зайнятися шифруванням.

Код «**Кодування->Шифрувати**» дуже простий:

```
procedure TfMain.CoderCodeClick(Sender: TObject);
begin
  //спочатку очистимо ключ:
  MyCript.MyPassword:= '';
  if InputQuery('Введення ключа', 'Введіть ключове слово (фразу):',
    MyCript.MyPassword) then
      Memo1.Text:= MyCript.Write(MyCript.Encrypt(Memo1.Text));
end;
```

Спочатку ми очищаємо ключ. Робиться це про всяк випадок, раптом це вже не перше шифрування, тоді в глобальній змінній MyPassword буде зберігатися попередній ключ. Або уявіть таку ситуацію: користувач уводив особистий текст, йому знадобилося відлучитися, і він його зашифрував. Якщо ключ у змінній зберігається, кожен може

підійти й розшифрувати його. Елементарна безпека вимагає, щоб ми завжди очищали ключ.

Потім, за допомогою InputQuery() ми виводимо запит, у якому користувач може ввести пароль - ключове слово або фразу. Цей ключ ми розміщеємо в змінній MyPassword, після чого викликаємо шифрування тексту Memo так, як було зазначено в рекомендаціях модуля MyCript. Дешифрування («**Кодування->Дешифрувати**») відбувається схожим образом:

```
procedure TfMain.CoderDecodeClick(Sender: TObject);
begin
  //спочатку очистимо ключ:
  MyCript.MyPassword:= ‘‘;
  if InputQuery(‘Введення ключа’, ‘Введіть ключове слово (фразу):’,
    MyCript.MyPassword) then
      Memo1.Text:= MyCript.Decrypt(MyCript.Read(Memo1.Text));
end;
```

От і весь шифратор. Збережіть його, скомпілюйте й випробуйте в роботі. Майте на увазі, що якщо ввести неправильний пароль, то текст буде дешифрований неправильно й не прочитається. Зворотної дії не передбачено в тих же цілях безпеки. І ще: залишилася неопрацьованою команда меню «**Довідка->Про програму**», ми повернемося до неї в іншій лекції, де будемо вивчати багатовіконні додатки, так що не видаляйте поки цей проєкт. Він нам також знадобиться в лекціях 28 і 29, де ми будемо створювати для цього проєкту довідкову систему й інсталятор.

Тема 19. Дерева

Мета лекції: розглянути застосування компонента TTreeView для роботи з деревоподібними ієрархічними даними, на практиці показані всі основні дії з компонентом.

План

1. Дерево TTreeView
2. Властивості й методи TTreeView.Items

1. Дерево TTreeView

Компонент TTreeView розташований на вкладці **Common Controls Палітри компонентів**, і призначений для відображення різних деревоподібних ієрархічних структур:

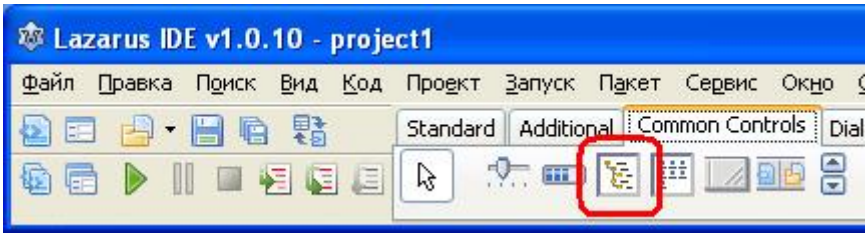


Рис. 19.1. *Компонент TTreeView*

Ми часто зіштовхуємося з ієрархічною інформацією. Це може бути **Дерево каталогів** у стандартному Провіднику Windows, що відображає у вигляді галузей дерева зовнішні й вкладені папки, і файли, які там зберігаються. Це може бути структура підприємства, бібліотечний каталогізатор, структура книги, реферату, курсовий... Загалом, сама різна інформація, у якій присутні батьківські й дочірні (вкладені) об’єкти. Всі подібні об’єкти можна відобразити за допомогою компонента TTreeView.

Об’єкти, які втримуватися в даному компоненті називаються вузлами (англ. node), а сам компонент являє собою список вузлів. Подібно спискам TListBox і TComboBox, з якими ми вже знайомі, TTreeView має властивість Items - індексований список вузлів. Кожен вузол - це об’єкт, що має тип TTreeNode. І батьківські, і вкладені в них дочірні елементи - все це вузли (об’єкти) типу TTreeNode. Причому дочірній елемент одночасно може бути батьком стосовно іншого вузла, рівень вкладеності необмежений.

Але давайте все по-черзі. Розглянемо роботу компонента на прикладі бібліотечного каталогу. Відкрийте **Lazarus** з новим проєктом. Як звичайно, форму назвіть **fMain**, проєкт збережете під

ім’ям MyLibrary у папку **19-01**, модулю форми дайте ім’я **Main**. У властивості Caption форми напишіть **Бібліотечний каталог**. Саму форму трішки розтягніть, нехай у нас буде висота 350, а ширина 500 пікселів.

Далі, встановимо на форму просту панель TPanel, з неї ми зробимо своєрідну **Панель інструментів програми**. У панелі очистіть властивість Caption, у властивості Align встановать alLeft, у властивості Width 127 пікселів.

Далі, на отриману **Панель інструментів** одну за одною встановіть 7 простих кнопок **TButton**. У всіх кнопок у властивості Left встановіть значення 1, а у властивості Width - значення 125. Верхню кнопку розташуєте вище (Top = 1), інші - трохи нижче, щоб між кнопками була зовсім невелика відстань. Нам потрібно перейменувати кнопки й зробити на них відповідні написи. Зробіть наступні настроювання кнопок:

Таблиця 19.1. Настроювання властивостей кнопок

Name	**Caption**
bNewNode	Новий розділ
bNewChildNode	Новий підрозділ
bDelete	Видалити
bEdit	Перейменувати
bSort	Сортувати
bCollapse	Згорнути список
bExpand	Розгорнути список

Тепер, правее панелі, із вкладки **Common Controls** встановіть компонент TTreeView. Оскільки дерево в нас одне, перейменовувати його ми не будемо. У властивості Align дерева також встановіть значення alLeft, у властивості Width встановіть 360 пикселей. Поверх TTreeView із цієї ж вкладки встановите список зображень TImageList, його теж перейменовувати не будемо. У результаті в нас повинна вийти от така форма:

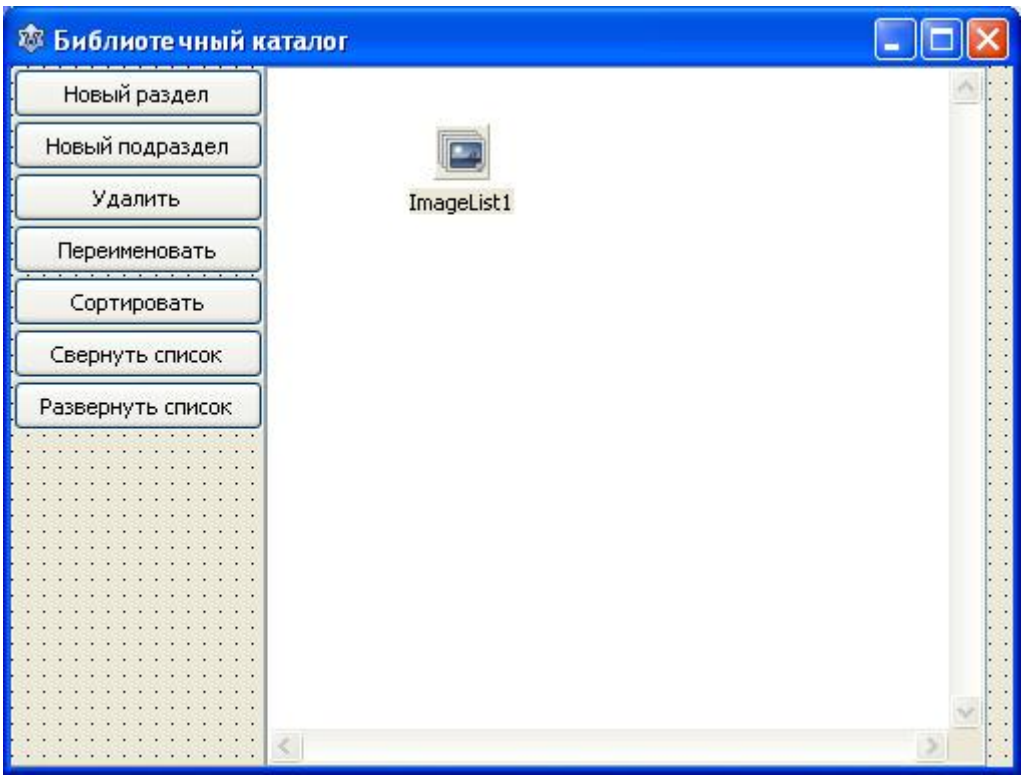


Рис. 19.2. *Отримана форма*

Як бачите, у правій частині форми після TreeView1 залишилося небагато мисця. Зараз ми навчимося тільки робити різні дії з компонентом TTreeView, а пізніше, в іншій лекції, ми повернемося до цього проєкту, і в правій частині зробимо відображення книг, які ставляться до даного розділу бібліотеки.

Отже, виділите дерево TreeView1 і зверніть увагу на його властивості. Ми розглянемо тільки основні, з якими ще не зіштовхувалися в інших компонентах.

2. Властивості TTreeView

AutoExpand	- якщо True, то дерево автоматично розгортається, коли програма починає працювати, інакше список залишається згорнутим. Нам бажано, щоб дерево було розгорнутим, тому встановіть тут значення True.
HideSelection	- якщо True, то коли компонент губить фокус (активним стає інший компонент), виділення поточного вузла буде приховано. Ну, це справа смаку, можна залишити значення за замовчуванням - True.
Images	- отут потрібно вибрати список зображень, якщо бажаєте, щоб у батьківських і дочірніх вузлів були різні піктограми. Виберемо отут ImageList1, а підходящі картинки підберемо пізніше.

Indent	- відстань у пикселях від лівого вузла до його подузлов. За замовчуванням дорівнює 15 пикселей, і я рекомендую цю відстань не зменшувати. Інакше вузли дерева будуть розташований близько друг від друга, як би злипнутися, що утруднить читання списку.
Items	- сам список, основна властивість компонента. При натисканні на кнопку «…»виходить **Редактор елементів дерева**:

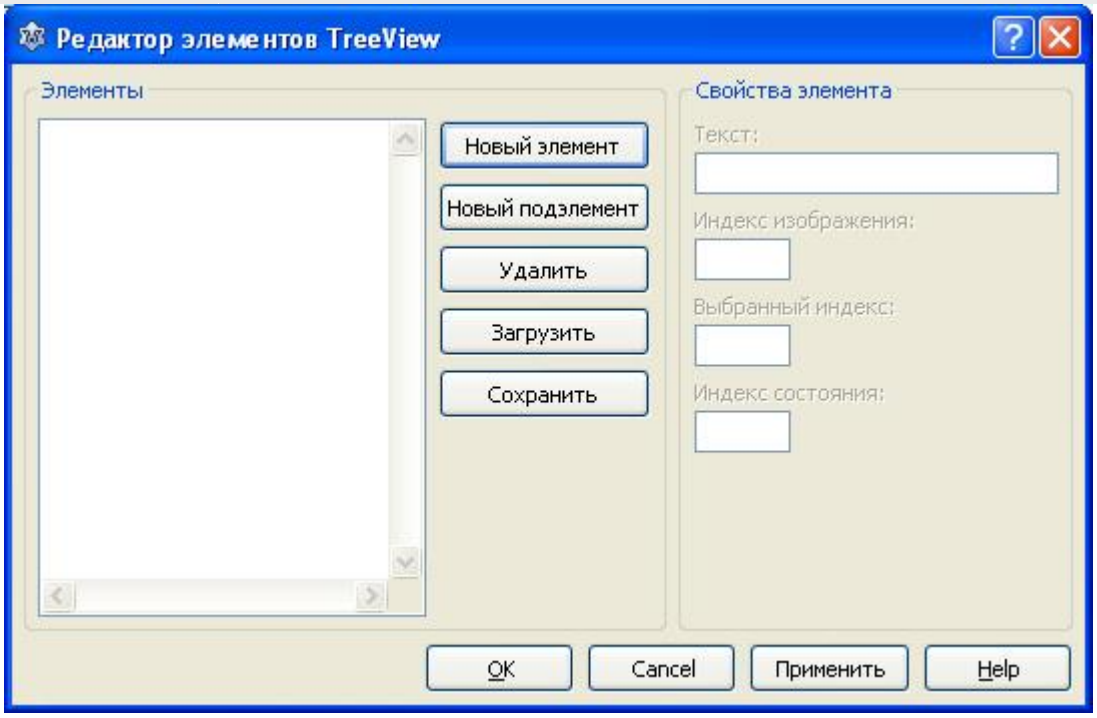


Рис. 19.3. *Редактор елементів компонента TTreeView*

Як бачите, кнопки практично ідентичні тим, що ми встановили на форму, і працюють приблизно також. Так що ми не будемо розглядати роботу із цим редактором, оскільки майже завжди дерево заповнюють не вручну, а програмно. Звернутися до окремого вузла дерева можна через властивість Items, вказавши індекс елемента; індексація починається з нуля. Наприклад, до першого вузла в списку звертаються так: TreeView1.Items[0].

MultiSelect	- якщо True, те дозволяє одночасне виділення декількох вузлів; якщо False - забороняє.
MultiSelectStyle	- перемикачі, які визначають спосіб виділення декількох вузлів. Перемикачів чотири, якщо вони мають значення True, то даний спосіб працює, інакше не працює:

	• msControlSelect - з натиснут і утримуваної **<CTRL>**, коли клацають по елементах списку в довільному порядку. • msShiftSelect - з натиснут і утримуваної **<SHIFT>**, коли вибирають відразу діапазон елементів, клацаючи спершу по першому, потім по останньому елементі. • msSiblingOnly - як msShiftSelect, але в діапазон включаються тільки вузли одного рівня. • msVisibleOnly - як msShiftSelect, але в діапазон не включаються нерозкриті дочірні вузли.
RightClickSelect	- при True вузли виділяються як лівої, так і правою кнопкою миші; при False - тільки лівої.
RowSelect	- при True виділяється весь рядок елемента, при False - тільки його ім’я. Марна властивість, треба сказати.
ScrollBars	- смуги прокручування, з ними ви знайоми по компоненті TMemo. Рекомендую встановлювати в цій властивості значення ssAutoBoth.
ShowButtons	- дозволяє (при True) або забороняє показ кнопок біля вузлів. Звичайно дозволяє, тоді ліворуч від закритого вузла буде кнопка із плюсом, від відкритого - з мінусом.
ShowLines	- дозволяє показ ліній галузей дерева. Бажано залишати True.
ShowRoot	- дозволяє показ лінії, що йде від кореня дерева. Якщо ShowLines = False, то ніяких ліній не буде показано в жодному разі.
SortType	- спосіб сортування списку. До цієї властивості ми ще повернемося в програми. Може бути: • stNone - немає сортування. • stText - сортування по тексту. • stData - сортування за даними. • stBoth - сортування й по тексту, і за даними.

TreeLineColor і TreeLinePenStyle відповідають за кольори й тип ліній галузей дерева.

Методи TTreeView

AlphaSort	- ще один спосіб відсортувати список. Викликається так: TreeView1.AlphaSort; Метод повертає True, якщо сортування пройшло успішно, і False в іншому випадку.
FullCollapse	- звертає дерево, приховуючи всі його дочірні вузли.
FullExpand	- навпаки, розвертає дерево.
LoadFromFile	- метод зчитує інформацію про вузли з текстового файлу. Файл зберігається у форматі UTF-8. Дочірні вузли перебувають нижче батьківських, і зрушені вправо символом табулятора. Перед використанням бажано перевіряти реальна наявність файлу.
SaveToFile	- навпаки, зберігає структуру дерева в текстовий файл.

Події TTreeView

Компонент має кілька специфічних подій, які можуть виявитися корисними.

OnChange	- виникає після переміщення на інший вузол.
OnChanging	- виникає перед переміщенням на інший вузол. При бажанні, можна згенерувати ця подія, і заборонити користувачеві цей перехід, наприклад, якщо не виконане якась умова.
OnCollapsed	- виникає після згортання вузла. Користувач може викликати цю подію, клацнувши по кнопці «-» поруч із вузлом.
OnCollapsing	- виникає після згортання вузла.
OnCompare	- виникає, коли оброблювач при сортуванні списку порівнює два вузли й вирішує, що повинен бути вище.
OnDeletion	- виникає при видаленні вузла.
OnEdited	- виникає після редагування користувачем напису вузла.
OnEditing	- виникає під час редагування напису вузла.
OnExpanded	- виникає після розгортання вузла.
OnExpanding	- виникає після розгортання вузла.

OnGetImageIndex	- виникає при визначенні індексу зображення у вузла зі зв’язаного TImageList.
OnGetSelectedIndex	- виникає при визначенні індексу поточного вузла.

Властивості й методи TTreeView.Items

Як уже згадувалося, при програмній обробці дерева (а найчастіше, така обробка й використовується) доводиться користуватися властивістю Items, що має тип TTreeNodes і саме є об’єктом, а тому має власний набір властивостей і методів. Розберемо основні з них:

Властивості

Count	- Кількість вузлів, що входять у дерево, включаючи й вкладені.
Item[Index:Integer]	- Індексований доступ до вузлів. Як завжди, індексація починається з нуля, тому, наприклад, змінити текст першого вузла дерева можна так: TreeView1.Item[0].Text:= ‘Новий текст’;

Нерідко виникає необхідність обійти все дерево, від першого до останнього вузла. Наприклад, змінити текст всіх вузлів можна так (приклад виконувати не потрібно):

```
var
  i: integer;
begin
  for i:= 0 to TreeView1.Items.Count-1 do
    TreeView1.Item[i].Text:= ‘Вузол №’ + IntToStr(i+1);
```

У цьому випадку ми одержимо дерево з текстом
Вузол №1
Вузол №2

і так далі.

Методи

Add(SiblingNode:TTreeNode; const S):TTreeNode	- додає новий вузол у кінець списку SiblingNode, з текстом S. Якщо додається кореневий (не має батька) вузол, то SiblingNode = NIL.

AddChild(ParentNode:TTreeNode; const S):TTreeNode	- додає новий дочірній (вкладений) вузол у кінець батьківського списку ParentNode, з текстом S.
AddFirst(SiblingNode:TTreeNode; const S):TTreeNode	- додає новий вузол у початок списку SiblingNode, з текстом S. Якщо додається кореневий (не має батька) вузол, то SiblingNode = NIL.
AddChildFirst(ParentNode:TTreeNode; const S):TTreeNode	- додає новий дочірній (вкладений) вузол у початок батьківського списку ParentNode, з текстом S.
Clear	- очищає список дерева від всіх вузлів.
Delete(Node:TTreeNode)	- видаляє вузол Node.

Основні властивості вузла TTreeNode

До окремого вузла можна одержати доступ через властивість Item, наприклад, TreeView1.Item[0].

ImageIndex	містить індекс піктограми зі зв'язаного TImageList, що буде відображатися в даному вузлі.
Text	містить текст, відображуваний у гілці даного вузла.

Будь ласка, не плутайте - властивість компонента Items має тип TTreeNodes, що являє собою індексований список вузлів. А окремий вузол має тип TTreeNode - це не те саме!

Повернемося до програми. Знайдемо дві підходящі піктограми для вузлів дерева. Клацніть двічі по ImageList1, щоб відкрити його редактор. Натисніть кнопку «**Додати**». Якщо ви встановлювали Lazarus у папку за замовчуванням, то різні стандартні картинки будуть розташовані за адресою

C:\Lazarus\Images

Не будемо копатися по вкладених папках - прямо отут перебувають дві підходящі піктограми. Першим додайте зображення folder.png, це буде піктограма для батьківських вузлів. Піктограма одержала індекс 0. Другим додайте зображення template.png, це буде зображення вкладених підрозділів. Картинка встала під індексом 1. Можна закрити редактор **ImageList** кнопкою «**ОК**».

Тепер приступимо до програмування кнопок. Згенеруйте подію OnClick для кнопки «**Новий розділ**». Її код буде наступним:

```
procedure TfMain.bNewNodeClick(Sender: TObject);
var
  NodeCaption: string; //для одержання заголовка нового вузла
  NewNode: TTreeNode; //для створення нового вузла
begin
  //спочатку очистимо заголовок:
  NodeCaption:= ‘‘;
  //тепер, якщо користувач не ввів заголовок нового вузла, виходить:
 if not InputQuery(‘Введення заголовку’, ‘Введіть заголовок розділу’,
                NodeCaption) then exit;
  //якщо ми тут, то заголовок є. створюємо батьківський вузол:
  NewNode:= TreeView1.Items.Add(nil, NodeCaption);
  //привласнюємо йому картинку під індексом 0:
  NewNode.ImageIndex:=0;
end;
```

Як видно з коду й коментарів, ця кнопка додає в кінець списку вузлів новий батьківський вузол. Функцією-запитом InputQuery() ми одержуємо в користувача заголовок для майбутнього вузла. Якщо користувач закрив діалог, не ввівши цього заголовка, то ми просто виходимо з події, нічого не вживаючи. Але якщо він щось туди ввів, то цей заголовок попадає в змінну NodeCaption, і ми приступаємо до створення батьківського вузла, що й робить код:

```
  NewNode:= TreeView1.Items.Add(nil, NodeCaption);
```

Те, що вузол батьківський, говорить параметр nil - ніщо, що вказує, що в нового вузла немає батька.

На закінчення ми привласнюємо цьому вузлу картинку під індексом 0, якщо пам’ятаєте , там зображення папки.

Код для кнопки «**Новий підрозділ**» дуже схожий на попередній:

```
procedure TfMain.bNewChildNodeClick(Sender: TObject);
var
  NodeCaption: string;
  NewNode: TTreeNode;
begin
  NodeCaption:= ‘‘;
```

```
    if not InputQuery('Введення заголовка', 'Введіть заголовок підрозділу',
                      NodeCaption) then exit;
    NewNode:= TreeView1.Items.AddChild(TreeView1.Selected, NodeCaption);
    if NewNode.Parent = nil then NewNode.ImageIndex:=0
    else NewNode.ImageIndex:=1;
  end;
```

Різниці отут дві. По-перше, цього разу ми використаємо метод Items.AddChild, що додає саме дочірній вузол. У параметрі замість nil ви бачите вже TreeView1.Selected, що означає посилання на виділений у цей момент розділ. Саме для цього розділу буде створюватися підрозділ.

По-друге, ми вказуємо не конкретний індекс картинки, що буде отут відображатися, а робимо перевірку:

```
    if NewNode.Parent = nil then NewNode.ImageIndex:=0
    else NewNode.ImageIndex:=1;
```

Якщо в нового вузла все-таки немає батька (NewNode.Parent = nil), то привласнюємо вузлу зображення 0, інакше це буде зображення 1.

Для кнопки «**Видалити**» код буде наступним:

```
  procedure TfMain.bDeleteClick(Sender: TObject);
  begin
    if TreeView1.Selected <> nil then
      TreeView1.Items.Delete(TreeView1.Selected);
  end;
```

Код дуже простенький. Якщо виділений вузол не дорівнює nil (тобто, якщо взагалі якийсь вузол виділений), те ми видаляємо зі списку цей виділений вузол.

Код для кнопки «**Перейменувати**»:

```
  procedure TfMain.bEditClick(Sender: TObject);
  var
    NodeCaption: string;
  begin
    NodeCaption:= '';
    if not InputQuery('Уведення заголовка', 'Уведіть новий заголовок',
                      NodeCaption) then exit;
```

```
  TreeView1.Selected.Text:= NodeCaption;
end;
```

Тут ми точно також намагаємося одержати в користувача новий заголовок. Якщо він його ввів, то цей новий заголовок ми привласнюємо властивості Text виділеного в цей момент вузла.

Тепер напишемо код для кнопки «**Сортувати**». Її код зовсім простій:

```
procedure TfMain.bSortClick(Sender: TObject);
begin
  TreeView1.AlphaSort;
end;
```

Метод AlphaSort повертає істину, якщо сортування пройшло успішно. Але нам нема рації перевіряти цю успішність, тому сам метод ми викликаємо, а на значення, що повертає їм, не обертаємо уваги.

Далі на черзі в нас кнопка «**Згорнути список**». Її код не складніше:

```
procedure TfMain.bCollapseClick(Sender: TObject);
begin
  TreeView1.FullCollapse;
end;
```

Для «**Розгорнути список**»:

```
procedure TfMain.bExpandClick(Sender: TObject);
begin
  TreeView1.FullExpand;
end;
```

Методи FullCollapse і FullExpand ми вивчали вище.

Кнопки ми запрограмували. Однак поки користі від нашої програми - нуль. Користувач витратить час, заповнюючи список розділів і підрозділів бібліотеки, але коштує йому тільки вийти із програми, і вся ця робота втратиться. Нам потрібно навчити програму цей список зберігати у файл, і завантажувати його з файлу. Файл списку назвемо **MyLibrary.dat**. **MyLibrary** - тому, що так називається наша програма, **dat** - таке розширення традиційно мають файли з даними. Де найкраще завантажувати цей список? Звичайно, у події OnCreate головної форми! Ця подія виникає один раз, коли завантажується програма, але перед її відображенням на екрані. Якщо вам потрібно зробити яку то підготовчу роботу перед відкриттям вашої програми, то OnCreate для

цього - саме місце. Виділите форму **fMain** - це можна зробити, клацнувши по маленькій вільній ділянці правее компонента TreeView1, або вибравши **fMain** у верхній частині **Інспектора об'єктів**. Потім перейдіть на вкладку «**Події**» **Інспектори об'єктів**, знайдіть і згенеруйте подію OnCreate. Код буде наступним:

```
procedure TfMain.FormCreate(Sender: TObject);
var
  i: integer;
begin
  //якщо файл існує, завантажимо його:
  if FileExists('MyLibrary.dat') then
    TreeView1.LoadFromFile('MyLibrary.dat');
  //тепер пройдемося за списком, і кожному вузлу привласнимо
  //потрібну піктограму:
  for i:= 0 to TreeView1.Items.Count-1 do
    if TreeView1.Items[i].Parent=nil then
TreeView1.Items[i].ImageIndex:=0
    else TreeView1.Items[i].ImageIndex:=1;
end;
```

Тут ми спочатку за допомогою функції FileExists() перевіряємо, є чи взагалі в поточній папці файл **MyLibrary.dat**? Функція поверне істину, якщо такий файл є. У цьому випадку ми його завантажуємо в дерево TreeView1. Але дерево вийде без піктограм, їх ще потрібно завантажити. Це ми робимо в циклі for, обходячи всі вузли дерева. Якщо вузол батьківський (TreeView1.Items[i].Parent=nil), ми привласнюємо йому картинку з індексом 0, інакше - картинку з індексом 1.

Залишилося навчити програму зберігати список. Робити це найкраще при виході із програми, у події OnClose головної форми. У цій події звичайно роблять всі завершальні дії - закривають відкриті ресурси, зберігають параметри програми, і т.п. Код події наступний:

```
procedure TfMain.FormClose(Sender: TObject; var CloseAction:
TCloseAction);
begin
  TreeView1.SaveToFile('MyLibrary.dat');
end;
```

Отут ми просто зберігаємо список у файл. Збережете проєкт, запустите його й спробуйте заповнити розділи й підрозділи бібліотеки. У мене вийшло приблизно так:

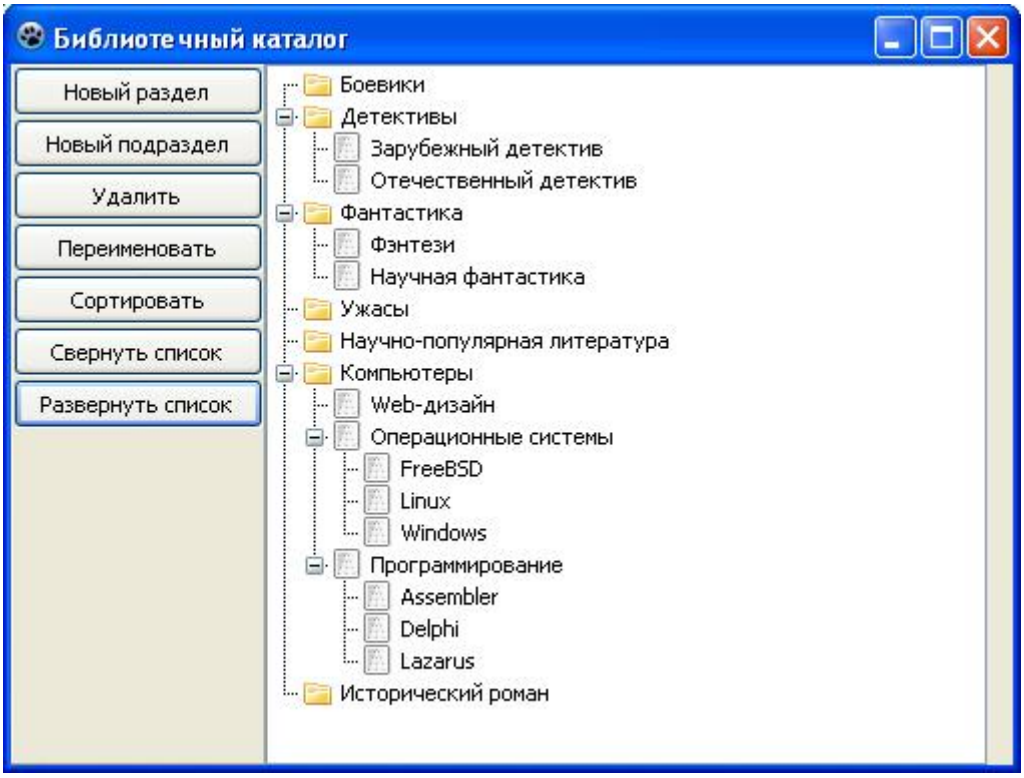


Рис. 19.4. Працююча програма

Поэкспериментируйте з роботою всіх кнопок. При виході із програми список повинен зберігатися, а при вході - зчитуватися. Не видаляйте проєкт - ми до нього ще повернемося!

Тема 20. Багатовіконні додатки. SDI- і MDI-інтерфейси

1. Багатовіконні додатки
2. Модальні вікна
3. Немодальні вікна

1.Багатовіконні додатки

Дотепер ми з вами всі додатки робили з одним єдиним вікном. А тим часом, у сучасному програмуванні рідко зустрічаються програми, що мають тільки одне вікно. Навіть прості стандартні утиліти, начебто Калькулятора calc.exe або гри «Сапер» - winmine.exe мають по нескольку вікон. Я недавно закінчив проєкт для однієї організації,

проєкт цей можна вважати середньої складності, а він містить 102 вікна! У цій лекції ми з вами навчимося робити многооконные додатка.

Є два типи інтерфейсів: **SDI** (**Single Document Interface** - однодокументный інтерфейс) і **MDI** (**Multi Document Interface** - многодокументный інтерфейс). SDI-додатка працюють одночасно з одним документом, MDI-додатки призначені для одночасної роботи з безліччю однотипних документів. При цьому всі документи розташовуються усередині одного контейнера, яким служить, як правило, головна форма. Компанія Microsoft не рекомендує використати MDI-інтерфейси, хоча сама використає їх у різних службових програмах, наприклад, у консолях начебто Диспетчера пристроїв. Крім того, розробка MDI-додатків в Lazarus поки не реалізована, так що докладно розглядати MDI-інтерфейси ми не будемо, хоча коротенько й торкнемося цієї теми. Але спочатку - SDI.

SDI

В SDI-додатках вікна можуть бути двох видів - **модальні** й **немодальні**. Створюються вони однаково, різниця полягає тільки в способі висновку цих вікон на екран. Модальне вікно блокує програму, не дає з нею працювати, поки ви це вікно не закриєте. Стандартний приклад модального вікна - вікно «**Про програму**», що є присутнім майже в будь-якому додатку. Як правило, таке вікно перебуває в меню «**Довідка**». Поки ви не натиснете «**ОК**», закривши це вікно, ви не зможете працювати з основною програмою.

Немодальні вікна дозволяють перемикатися миж ними, і програмою, і працювати одночасно й там, і там. Типовий приклад - вікна **Lazarus** - ви можете перемикатися миж **Інспектором об'єктів**, **Редактором коду**, **Редактором форм**, і іншими вікнами - вони не заважають один одному, тому що всі вони немодальні.

Вивчимо роботу з різними вікнами на прикладах.

2. Модальні вікна

Якщо ви помнете, у темі №18 ми робили лабораторну роботу - Блокнот-шифрувальник. Там ми передбачили в меню «**Довідка**» підминю «**Про програму**», але саме вікно не робили. Прийшов час виправити цей недогляд. Ви повинні були зберегти проєкт у папку **18-01** під ім'ям **CodeBook**. Переконаєтеся, що **Lazarus** закрито, і завантажите файл **CodeBook.lpi** - це інформаційний файл проєкту. У результаті, завантажиться Lazarus із цим проєктом (з тим же успіхом можна було б завантажити файл **CodeBook.lpr**).

Виберіть команду меню «**Файл -> Створити форму**» або натисніть однойменну кнопку на **Панелі інструментів**. З'явиться нова форма з ім'ям за замовчуванням **Form1**. Ми з вами домовлялися називати всі форми зрозумилими іменами, і на початку імені ставити префікс f, що означає форму. Тому у властивості Name цієї форми напишіть fAbout, потім натисніть кнопку «**Зберегти всі**» (або виберіть «**Файл -> Зберегти всі**»), і модулю цього нового вікна дайте ім'я About. Перемкнетеся клавішею **<F12>** у **Редактор коду** - ви побачите вкладки модулів:

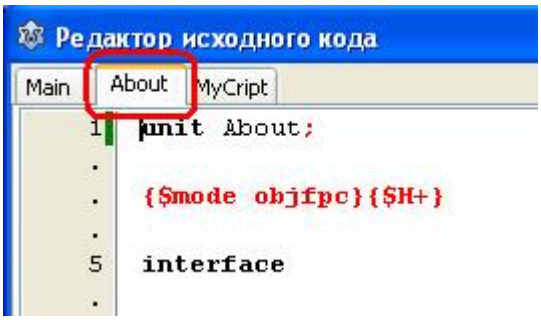


Рис. 20.1. Вкладки модулів у Редакторі коду

Два із цих модулів - Main і About мають форми. Переходячи по цих вкладках можна перемикатися миж модулями. Але нам спочатку потрібно зробити саме вікно «**Про програму**». Так що клавішею **<F12>** перемкнетеся назад у **Редактор форм**. Насамперед, у властивості BorderStyle форми **fAbout** виберіть значення bsDialog, тому що нам не потрібно, щоб користувач мав можливість зминювати розмири вікна, розвертати або звертати його. Потім у властивості Position виберіть poMainFormCenter, щоб вікно з'являлося по центрі головного вікна. Дотепер ми не встановлювали це значення у вікон, тому що всі наші додатки мистили єдине, воно ж головне вікно. Вікно «**Про програму**» - не головне, тому його бажано виводити на екран по центрі головного вікна. Головним у проєкті вважається вікно, створене першим, його ми звичайно називаємо **fMain**.

В властивості Caption форми напишіть «**Про програму**».

Встановіть на форму просту панель **TPanel**, очистите її властивість Caption. Щоб зробити з панелі гарну рамку, встановіть в її властивості BevelInner значення bvLowered.

Далі, на панель встановіть дві митки TLabel і один TMemo. В TMemo ми будемо виводити многострочный текст із поясненням про призначення програми. Оскільки нам не потрібно, щоб користувач миг редагувати цей текст, властивість ReadOnly компонента Memo1

встановіть в True. Текст в Memo1 прийде вводити убудованим **Редактором** через властивість Lines.

Нижче панелі встановите кнопку **TBitBtn** із вкладки **Additional Палітри компонентів**, у властивості Kind кнопки виберіть значення bkOK.

Для економії мисця я не буду докладно розписувати, як уводити в митки текст, миняти в компонентів шрифти й розмири - ви прекрасно повинні вмити робити це сами. У результаті у вас повинна вийти приблизно така форма:

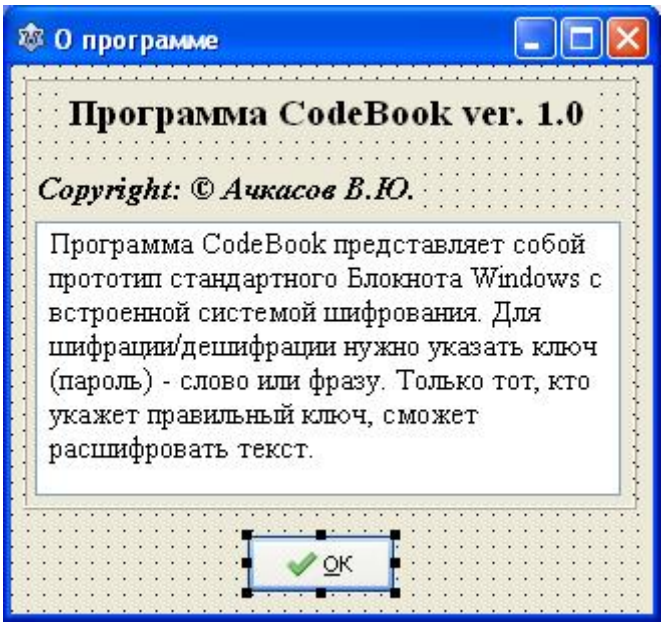


Рис. 20.2. Форма fAbout

Зазначу, що у всіх компонентах обирався стандартний шрифт - Times New Roman - ви ж можете вибрати свій, тільки підберіть зручний розмір і накреслення.

Окремо зупинюся на рядку Copyright. Слово Копірайт (англ. Copyright) означає авторське право. Причому авторське право може бути двох видів - майнове й немайнове. Якщо ви робите програму на замовлення, то майнове авторське право належить замовникові - він може встановлювати цю програму на скільки завгодно комп'ютерів, продавати або дарувати її. Немайнове право в кожному разі належить авторові програми, тобто, вам. Воно має на увазі, що програму не можна перейменовувати або зминювати її код без вашої згоди, і що в програми ви обов'язково повинні згадуватися, як автор. Таким чином, якщо ви робите програму на замовлення, ви не зобов'язані разом із програмою віддавати вихідний код вашого проєкту! Інакше вийде, що

ви передаєте замовникові не тільки майнове, але й немайнове право, а це вже буде ціна продукту на порядок вище.

Отож, у рядку Copyright вказується майновий правовласник. Якщо ви робите програму на замовлення, тут ви повинні вказати замовника. Себе ж ви можете згадати рядком нижче, встановивши ще одну мітку, і почавши її текст, як «**Автор:** «Але оскільки в даному проєкті замовника в нас немає, то всі авторські права належать нам. До речі, вказуйте своє прізвище, адже я добровільно викладаю цей код для загального використання, а проєкт по моїх описах робили все-таки ви!

І ще. Знак авторського права © має в таблиці символів код 0169. Щоб вставити його в Caption мітки, при уведенні тексту натисніть **<Alt>**, і втримуючи його, наберіть 0169. Потім відпустите **<Alt>**. Символ повинен з'явитися в мітці. Замість цього знака іноді вказують спрощений варіант: «(c)». Зробіть, як уважаєте потрібним.

Вікно ми зробили, потрібно тепер навчити програму виводити його на вимогу користувача. Кнопкою **<F12>** перейдіть у **Редактор коду**, потім, клацнувши по вкладці **Main**, перейдіть на модуль головної форми. От так, сходу, ми ще не зможемо викликати форму **fAbout**, спочатку потрібно підключити її модуль до головної форми. У розділі uses головної форми, після модуля MyCript через кому додайте модуль нової форми About. Тепер ми зможемо викликати це вікно!

Згенеруйте подію OnClick для команди меню «**Довідка -> Про програму**» (якщо ви ще помнете, для цього досить просто вибрати дану команду). Її код дуже простій:

```
procedure TfMain.HelpAboutClick(Sender: TObject);
begin
  fAbout.ShowModal;
end;
```

Метод ShowModal, зазначений у коді, викликає на екран вікно **fAbout** у модальному режими. Поки вікно не закриється, із програмою працювати буде не можна. Як тільки воно закриється, керування передасться назад у програму. Збережете проєкт, запустите його на виконання й переконаєтеся, що вікно «**Про програму**» викликається на вимогу користувача й закривається кнопкою «**ОК**». Однак не поспішаєте закривати проєкт, він нам ще знадобиться.

4. Немодальні вікна

Як уже згадувалося, немодальні вікна можуть бути відкриті одночасно з основною програмою, і не заважають її роботі. Щоб

продемонструвати роботу немодальних вікон, постачимо наш проєкт **Блокнота-шифрувальника** ще однією корисною функцією - статистикою. Створимо ще одне маленьке вікно, у якому будемо виводити кількість рядків у тексті, кількість слів і символів.

Створіть нову форму командою «**Файл -> Створити форму**» або однойменною кнопкою на панелі інструментів Lazarus. Форму (властивість Name) назвіть **fStats**, а її модуль збережете під ім'ям Stats. Далі, у властивості Caption форми напишіть «Статистика». Властивість BorderStyle установимо в bsDialog, щоб користувач не миг миняти розмири форми, а Position - в poMainFormCenter, щоб вікно з'являлося по центрі головного вікна.

Тепер нам знадобляться шість миток TLabel. Перші три встановите в лівій частині форми, одну під іншою. Ці митки ми перейменовувати не будемо, тому що нам не прийде звертатися до них програмно. У першій напишіть «Рядків:», у другий «Слів:» і в третьої «Символів:». Не забувайте ставити двокрапку після цих слів. Потім у правій частині, так само одну під іншою, установимо ще три митки, їх уже перейменуємо. У властивості Name цих миток напишіть, відповідно, LinesCount, WordsCount і CharsCount, а у властивості Caption всіх трьох миток встановіть «0» (нуль).

У центрі нижньої частини форми із вкладки **Additional Палітри компонентів** встановіть кнопку **TBitBtn**, у властивості Kind якої виберіть значення bkOK. У результаті, у вас повинна вийти приблизно така форма:

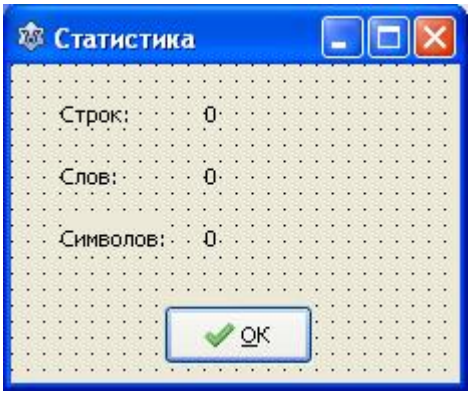


Рис. 20.3. Форма fStats

Згенеруйте наступну подію OnClick для кнопки «**ОК**»:

```
procedure TfStats.BitBtn1Click(Sender: TObject);
begin
  Close;
end;
```

Тобто, коли користувач натисне на цю кнопку, вікно статистики закриється.

Далі перейдемо в модуль головного вікна **fMain**. Щоб ми могли відтіля викликати нове вікно, потрібно через кому додати його модуль Stats у кінець роздягнула uses, як це ми робили з модулем About у минулому прикладі. Сюди ж додайте ще два модулі: LCLProc і StrUtils. У першому реалізовані UTF8-функції, одна з них нам знадобиться для визначення кількості символів. У другому реалізована безліч корисних строкових функцій, включаючи й ту, котра нам потрібна для підрахунку кількості слів у тексті:

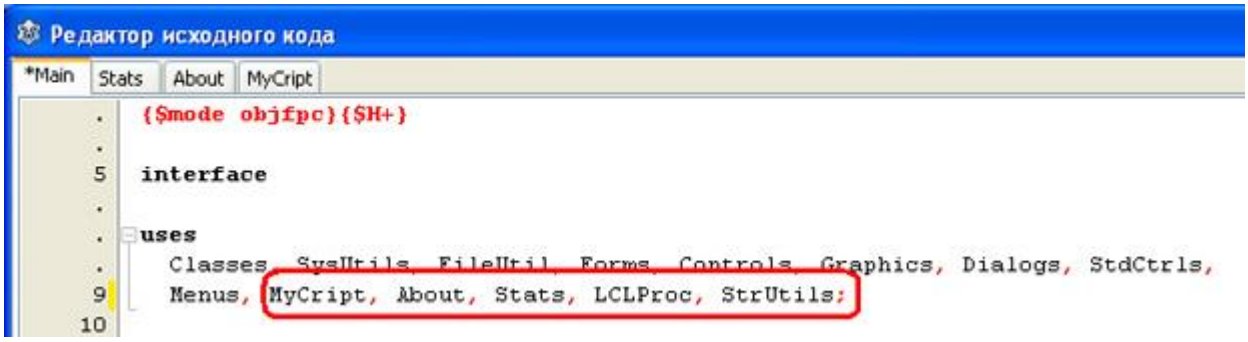


Рис. 20.4. Приєднані модулі

Тепер подумаємо, яким чином ми будемо викликати вікно статистики. Найкраще помістити виклик статистики в розділ меню «**Файл**». Двічі клацніть по **MainMenu1**, щоб викликати редактор меню. У розділі меню «**Файл**» виділіть підрозділ «**Зберегти як…**», клацніть по ньому правою кнопкою й виберіть команду «**Вставити новий пункт (після)**». У властивості Name нового підпункту введіть FileStat, а у властивості Caption напишіть «Статистика»:

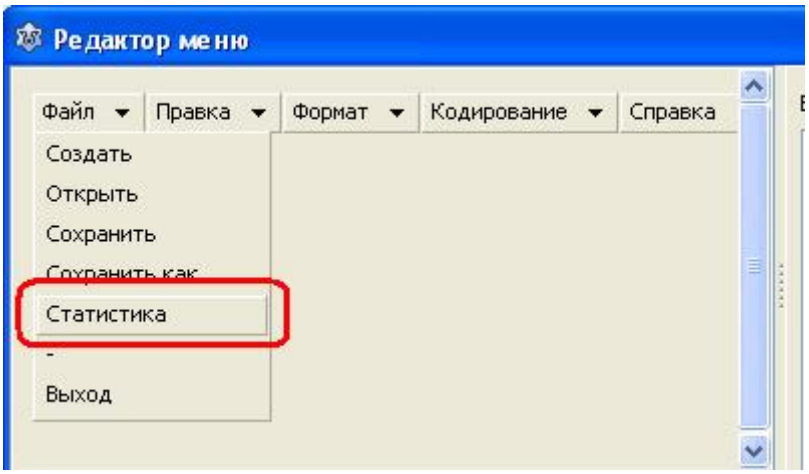


Рис. 20.5. Новий підпункт у меню

Тепер закрийте **Редактор меню**; вибравши команду «**Файл -> Статистика**» згенеруйте подію OnClick для нового підпункту. Код події наступний:

```
procedure TfMain.FileStatClick(Sender: TObject);
begin
  fStats.Show;
end;
```

Як бачите, тут ми викликаємо форму **fStats**, але не методом ShowModal, як у минулому прикладі, а методом Show. У результаті вікно буде показано, як немодальне. Однак цього недостатньо, щоб вікно показувало статистику. Нам потрібно ще підрахувати й вивести на екран кількість рядків, слів і символів.

Виділите компонент Memo1. В Інспекторі об'єктів перейдіть на вкладку «**Події**» і згенеруйте для нього подія OnChange, двічі клацнувши по ньому. Ця подія виникає щораз при змині тексту в **Memo1**, отут ми й будемо вважати статистику. Код події наступний:

```
procedure TfMain.Memo1Change(Sender: TObject);
begin
  //уважаємо символи:
  fStats.CharsCount.Caption:= IntToStr(UTF8Length(Memo1.Text));
  //слова:
  fStats.WordsCount.Caption:= IntToStr(WordCount(Memo1.Text,
StdWordDelims));
  //рядка:
  fStats.LinesCount.Caption:= IntToStr(Memo1.Lines.Count);
end;
```

Отут для нас багато нового, так що розберемо код подробней. Спочатку функцією UTF8Length ми одержали кількість символів у тексті Memo1, включаючи службові символи переходу на новий рядок. Це - ціле число, що нам довелося перетворити в строкову форму за допомогою функції IntToStr, щоб ми могли привласнити цей рядок властивості Caption митки CharsCount. Зверніть увагу, що мітка перебуває не на цієї, а на іншій форми, тому нам довелося спочатку вказати ім'я цієї форми, потім ім'я мітки потім властивість Caption.

Слова в тексті змінювати складніше, для цього ми скористалися новою для нас функцією WordCount. Функція описана в модулі

StrUtils, що ми підключили, вона повертає кількість слів у зазначеному тексті, і має наступний синтаксис:

function WordCount(const S: String; const WordDelims:TSysCharSet):integer;

Константа S містить текст, у якому підраховуються слова, у нашому випадку, це Memo1.Text. Константа WordDelims містить список символів-роздільників, якими може відокремлювати одне слово від іншого. Простіше всього в цьому параметрі скористатися системною константою StdWordDelims, що вже описана в цьому ж модулі, і оголошена в такий спосіб:

const StdWordDelims = [#0..’ ‘, ‘,’, ‘.’, ‘;’, ‘/’, ‘\’, ‘:’, ‘“‘, ‘«‘, ‘“‘] + Brackets;

Тобто, у цій константі перераховані основні символи-роздільники слів. Функція повертає ціле число - кількість слів у зазначеному тексті. Перетворивши його в символьну форму, ми привласнюємо це значення властивості Caption митки WordsCount.

А от кількість рядків навіть уважати не потрібно, вона втримується у властивості Memo1.Lines.Count. Нам залишається лише перетворити його в символьну форму, і привласнити відповідній митці. Тепер щораз, як в Memo1 зміниться текст (користувач увів символ, скопіював текст в Memo або видалив текст), ця подія буде перераховувати кількість слів, символів і рядків, і виводити їх у форму **fStats**, навіть якщо її не видно. Вибравши команду меню «**Файл -> Статистика**», користувач зможе вивести вікно статистики. Більше того, він може, не закриваючи цього вікна, повернутися в головну форму й продовжувати набирати текст. Вікно статистики при цьому буде паралельно відображати результати:

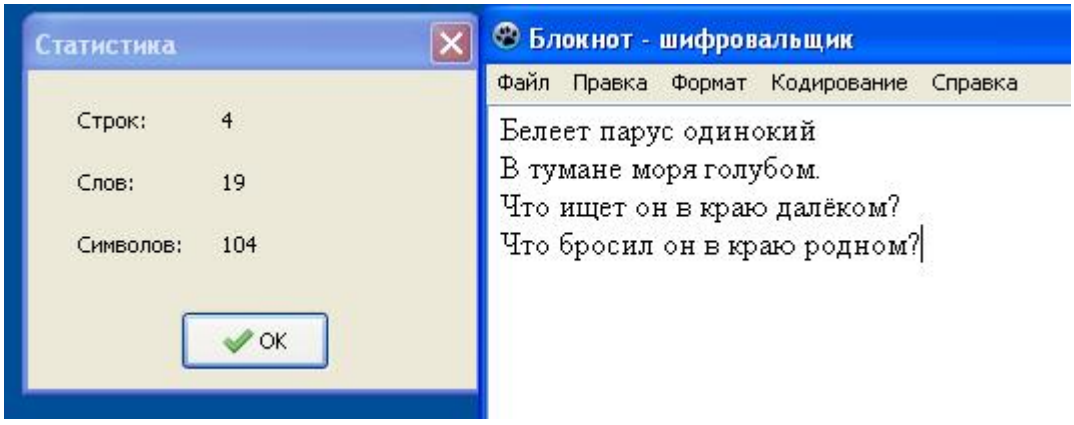


Рис. 20.6. Немодальне вікно статистики в дії

Збережіть проєкт, запустіть його на виконання й переконайтеся, що вікно статистики не заважає роботі головного вікна. Проєкт **Блокнота-шифрувальника** нам буде потрібно ще для наступних тем, так що не видаляйте його.

MDI-додатка

На відміну від Delphi, в Lazarus поки не реалізована можливість створення MDI-додатків, а оскільки MDI-інтерфейси вважаються застарілими, то можливо, вона й не буде реалізована. Але знати про ці інтерфейси потрібно, тому коротенько торкнемося цієї теми, тим більше, що я можу помилятися, і в наступних версіях Lazarus розроблювачі цю можливість все-таки реалізують. Принцип створення MDI-додатків наступний:

1. Спочатку ви створюєте головне, воно ж батьківське вікно. Це вікно буде служити своєрідним контейнером для дочірніх вікон, тому основна, робоча частина головної форми повинна бути вільної. У властивості FormStyle (стиль форми) батьківського вікна варто вибрати значення fsMDIForm.
2. Потім ви створюєте дочірнє вікно. Під час роботи програми дочірнє вікно буде створюватися усередині батьківського, і не зможе покинути його межі. У властивості FormStyle дочірнього вікна варто вибрати значення fsMDIChild.
3. Форму дочірнього вікна ми конструюємо тільки один раз, але в програми це вікно можна викликати скільки завгодно багато разів, відкриваючи в ньому різні документи. Всі ці дочірні вікна можуть бути відкриті одночасно.
4. MDI-інтерфейс додатка не забороняє вам створювати також модальні й немодальні вікна. Наприклад, в MDI-додатку ви з таким же успіхом і в такий же спосіб можете створити вікно «**Про програму**». Якщо ви будете створювати окреми модальні й немодальні вікна, то у властивості FormStyle цих форм варто залишити значення за замовчуванням fsNormal.

Для реалізації MDI-інтерфейсу сами розроблювачі рекомендують установити додатковий компонент MultiDoc, що реалізує псевдоMDI-интерфейс.

Список використаних джерел та корисні посилання

1. http://lazarus.freepascal.org/ - Офіційний сайт Lazarus. Тут ви зможете й нові версії знайти, і новини почитати.
2. http://wiki.freepascal.org/Main_Page/ua - документація по Lazarus і Free Pascal. Якщо постаратися, отут можна знайти відповіді майже на всі питання, які неодмінно у вас будуть виникати.
3. http://www.cyberforum.ua/lazarus/ - Великий форум по Lazarus. Форуми взагалі дуже корисні й інформативні, а форуми про програмування особливо. Навіть якщо ви й не змогли знайти в Інтернеті рішення вашої проблеми, на форумі неодмінно знайдеться який-небудь «гуру», що вам допоможе порадою.
4. http://www.freepascal.ua/ - інформаційний портал про Free Pascal і Lazarus. Теж досить корисний сайт.