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

Мова програмування Lazarus

Навчальний посібник
Частина 1

Укладач Паршукова Л. М.

Умань
2021

УДК 004(075.8)

Рецензенти:

Медведєва М.О., кандидат педагогічних наук, доцент, професор кафедри інформатики і інформаційно-комунікаційних технологій Уманського державного педагогічного університету імені Павла Тичини

Дякон В.М., кандидат фізико-математичних наук, доцент, директор Уманської філії Європейського університету

Рекомендовано до друку Вченою радою факультету фізики, математики та інформатики Уманського державного педагогічного університету імені Павла Тичини (протокол №5 від 25 січня 2021року).

Мова програмування Lazarus: навчальний посібник / МОН України, Уманський держ. пед. ун-т імені Павла Тичини ; уклад. Паршукова Л.М. – Умань : Візаві, 2021. - 161 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)

Зміст

Передмова
Історична довідка

Історію розвитку мов програмування, мабуть, можна почати з першого у світі програміста **Ади Лавлейс** (Августа Ада Кінг, графиня Лавлейс, математик). Ада Лавлейс народилася 10 грудня 1815 р. у Лондоні, була відома описом обчислювальної машини (механічна машина Ч. Бебіджа), у розробці якої вона брала участь, і створенням першої програми для неї. Ввела у вживання терміни «цикл» і «робочий осередок». На честь Ади Лавлейс в 1975 році була названа мова програмування **Ада**.

Реально мови програмування одержали розвиток в 1945-1955 р., коли з'явилися перші **ЕОМ** (Електронні Обчислювальні Машини), для яких програми складалися спочатку машинною мовою, а потім і на **Асемблері** - мнемонічному поданні машинної мови. І якщо «чистою» машинною мовою вже давно ніхто не користується, то Асемблер усе ще застосовується там, де потрібно або зверхмалий розмір програми, або більша швидкість її роботи, тобто, в основному, для створення критичних ділянок **ОС** (Операційних Систем) або драйверів, для програмування мікропроцесорів у різних платах, пристроях. Написати більшу сучасну програму на Асемблері - неймовірно складне, а те й нездійсненне завдання.

В 1954 року з'явилася перша мова програмування високого рівня **Фортран**, і почалася нова ера розвитку програмування.

Мова високого рівня (або високорівнева мова) - це мова програмування, найбільш наближена до людської мови. Вона містить значеннєві конструкції, описує структури даних, виконує над ними різні операції.

Сучасні мови високого рівня оперують уже цілими **об'єктами** - складними конструкціями, що володіють певним станом і поводженням.

Для навчання програмуванню й для рішення завдань загального призначення найбільше поширення одержала мова програмування високого рівня **Паскаль**, створений в 1968-1969 р. професором Ніклаусом Віртом, і названа на честь видатного французького математика Блеза Паскаля (між іншим, творця першої у світі механічної машини, що складає два числа). Ця мова вигідно відрізняється від інших мов програмування більш жорсткими правилами в описі й використанні даних різного типу. Паскаль - структурна мова, невелика й ефективна, сприятлива виробленню в

програміста гарного стилю програмування. У школах і закладах вищої освіти всіх країн у світі донині вивчають ту або іншу реалізацію Паскаля.

Оскільки **Lazarus** заснований на Pascal (точніше, на Об'єктному Pascal), історію інших високорівневих мов у рамках даного курсу ми розглядати не будемо.

В 1983 р. фірма Borland, відома розробкою Delphi - платного попередника **Lazarus**, випустила **Turbo Pascal** - інтегроване середовище розробки програм мовою Pascal. Turbo Pascal - це компілятор, компонувальник, редактор коду й налагоджувач в одному вікні. Він подібний до швейцарського ножа, де безліч різних інструментів вмонтовано в єдиний пристрій. Для програмістів Turbo Pascal примітний тим, що він став своєрідним прабатьком середовищ швидкої розробки програм.

В 1986 р. з'явилася мова **Object Pascal** (Об'єктний Паскаль), розроблений у фірмі Apple Computer. Цей діалект Pascal вже міг оперувати об'єктами.

В 1989 р. об'єктне розширення Pascal було додано й в Turbo Pascal фірми Borland.

В 1994 р. була випущена перша версія **Delphi** - Графічне інтегроване середовище швидкої розробки програм для Windows. Цей факт дав неймовірний поштовх розвитку таких середовищ, у яких розробка інтерфейсу програми для програміста замість нудотної рутини, перетворювалася в забавний конструктор форм. У сучасних середовищах можна створити програму, навіть не доторкаючись до клавіатури - винятково за допомогою миші. Правда, подібній до програми навряд чи можна буде додати будь-які корисні функції.

Всі ці мови й середовища були платними, й часто виявлялися недоступні освітнім установам у силу своєї дорожнечі. В 1993 р. почалися роботи над проєктом **Free Pascal** (**FPC** - Free Pascal Compiler). Перша версія FPC з'явилася лише в липні 2000 р., вона була повністю безкоштовна й підтримувала безліч платформ: Windows, Linux, FreeBSD, Mac OS X і т.п. FPC - це безкоштовний відкритий проєкт, його вихідні коди для вивчення або модифікації доступні кожному! Трохи пізніше з'явився **Lazarus** - єдине у світі безкоштовне графічне середовище для швидкої розробки програм, що використовує компілятор FPC. Як і FPC, Lazarus поширюється на умовах ліцензії **GNU GPL** (General Public License). Якщо не особливо вдаватися в юридичні подробиці, то GNU GPL - це ліцензія, що надає

користувачеві права вільно й безкоштовно копіювати, модифікувати й поширювати (у тому числі й на комерційній основі) даний продукт. По цій же ліцензії поширюються всі версії ОС (Операційної Системи) Linux - безкоштовний й досить серйозномий конкурент Windows.

Отже, ми з вами будемо говорити про останню (на момент написання курсу) версії **Lazarus** - 1.0.10, що працює з компілятором FPC 2.6. 2. **Lazarus** - молодий проєкт, що бурхливо розвивається, нові версії виходять досить часто, так що ви, ймовірно, будете користуватися більш свіжою версією. Проте, на курсі розглядаються фундаментальні питання програмування, які навряд чи будуть переглядатися. Так що можете вивчати наданий матеріал, користуючись версією **Lazarus** 1.0.10 або новішою версією.

У силу того, що переважна більшість користувачів як і раніше працює в операційній системі Windows, ми будемо розглядати роботу з **Lazarus** саме в цьому.

Втім, розробка програм під інші платформи має не так вже багато відмінностей, щоб створити вам непереборні труднощі при переході на іншу платформу.

Тема 1. Мова програмування Lazarus

Lazarus - це **IDE** (Integrated Development Environment) - Інтегроване Середовище Розробки програм, що використовує компілятор **FPC** (Free Pascal Compiler), редактори коду, форм, Інспектор Об'єктів, налагоджувач і багато інших інструментів.

Ще говорять, що середовище **Lazarus** - це **RAD** (Rapid Application Development) - середовище Швидкої Розробки Додатків.

Дотепер середовища розробки програм, подібні **Lazarus**, були винятково платними. **Lazarus** же став першою (і поки єдиною) IDE, доступною освітнім і державній установам зовсім безкоштовно. Більше того, **Lazarus** є проєктом **Open Source** - проєктом з відкритим вихідним кодом. Багато програмістів по усьому світі беруть участь у його розвитку, вихідний код **Lazarus** доступний для вивчення й модифікації. **Lazarus** має підтримку безлічі мов

Де знайти?

Lazarus, як уже говорилося, - безкоштовний і вільно розповсюджуваний продукт. Завдяки цьому, **Lazarus** все частіше використовують для вивчення програмування в школах і закладах вищої освіти, а також на багатьох підприємствах. Але де його взяти? На офіційному сайті виробника: http://lazarus.freepascal.org

У правій верхній частині сайту ви побачите наступну картинку:

Рис. 1.1. *Вибір і завантаження необхідної реалізації*

Тут ви зможете вибрати реалізацію саме під вашу платформу, від Windows до Mac OS X, як 32-х так і 64-х розрядну. При написанні курсу використовувася 32-х розрядний **Lazarus** для платформи Windows.

Натиснувши кнопку «**Download Now**» ви скачаєте останню версію **Lazarus**. Крім того, вибрати останню необхідну реалізацію й скачати її ви можете за адресою: http://sourceforge.net/projects/lazarus/files/

У цьому випадку, перейшовши за посиланнями, ви отримаєте доступ до завантаження декількох файлів, наприклад,

- lazarus-1.0.10-fpc-2.6.2-win32.exe
- lazarus-1.0.10-fpc-2.6.2-cross-arm-wince-win32.exe
- README.txt

Нам потрібний тільки перший файл із цього списку. Другий файл є розширенням для розробки програм під **Windows CE** (вона ж **WinCE**) - це варіант операційної системи Microsoft Windows для кишенькових комп'ютерів, смартфонів і вбудованих систем. На даному курсі цю можливість ми розглядати не будемо. Останній файл - простий текстовий файл із інформацією про версії, він нам теж не потрібний.

Як встановити?

Lazarus встановлюється досить просто. Властиво, нічого міняти нам не прийде, залишимо всі параметри, запропоновані установником за замовчуванням. Для початку виберемо російську мову установки, потім увесь час будемо натискати кнопки «**Далі**». Лише в передостанньому вікні встановлювача при бажанні можна поставити прапорець «**Створити значок на Робочому столі**». Коли вкажемо всі параметри, почнеться встановлення **Lazarus**. Потрібно почекати декілька хвилин, поки розпакується й скопіюється безліч файлів. І,

нарешті, кнопка «**Завершити**» для закриття вікна установника. Все, **Lazarus** у нас є! Ми можемо його завантажити.

На самому початку **Lazarus** виглядає трохи неохайно:

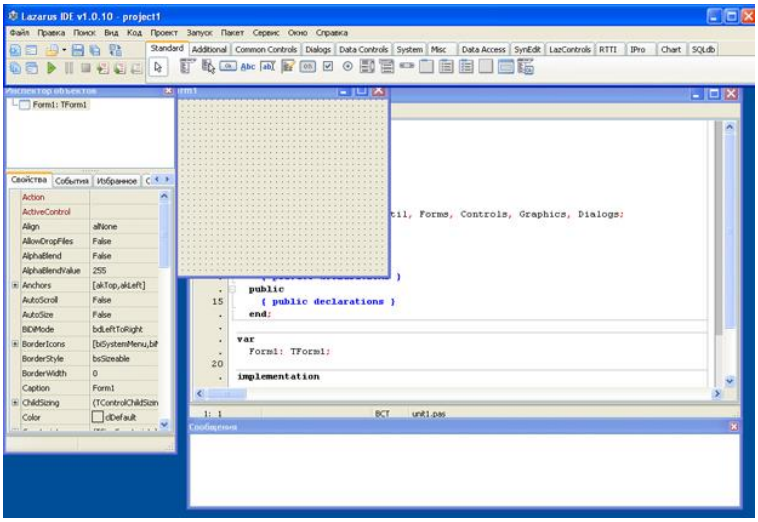

Рис. 1.2. *Перший запуск Lazarus*

Lazarus складається з декількох вікон (які варто підрівняти, щоб вони займали весь робочий стіл і не заважали один одному):

1. **Головне вікно**
2. **Інспектор об'єктів**
3. **Редактор форм**
4. **Редактор коду**
5. **Вікно повідомлень**

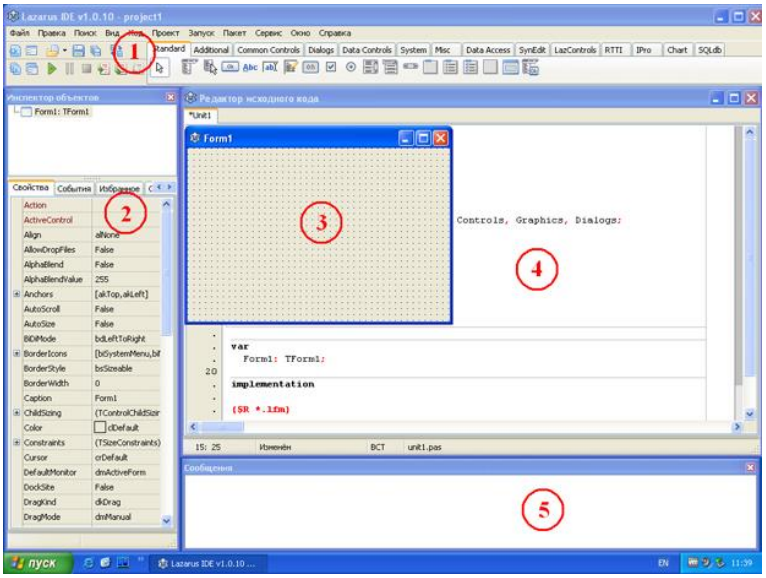

Рис. 1.3. *Вікна Lazarus*

Головне вікно

Головне вікно складається з наступних елементів:

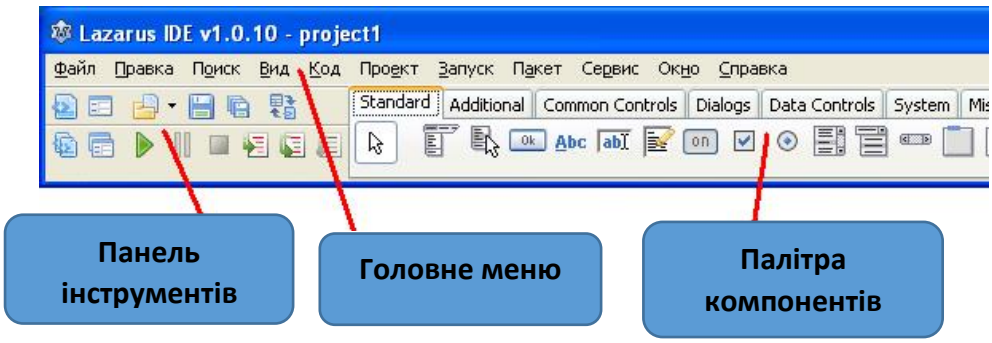

Рис. 1.4. *Головне вікно Lazarus*

1. **Головне меню** містить всі команди, необхідні для виправлення, компіляції, налагодження програми, для запуску різних допоміжних утиліт.

2. **Панель інструментів** містить кнопки найчастіше застосовуваних команд (цієї ж команди можна виконати й за допомогою **Головного меню**).

3. **Палітра компонентів** містить безліч вкладок, на яких утримується багатий вибір компонентів із власної бібліотеки компонентів **Lazarus - LCL** (Lazarus Component Library).

Інспектор об'єктів

Вікно **Інспектора об'єктів** складається із двох частин:

- **Дерево об'єктів**, у якому в деревоподібній формі розташовуються всі об'єкти, що використовуються в поточній формі.
- **Вікно із вкладками**, у якому можна набудовувати різні властивості поточного об'єкта. Незважаючи на те, що є 4 вкладки (**Властивості**, **Події**, **Обране**, **Обмеження**), найчастіше використовують тільки перші дві. Про властивості й події ми поговоримо детальніше у наступних лекціях.

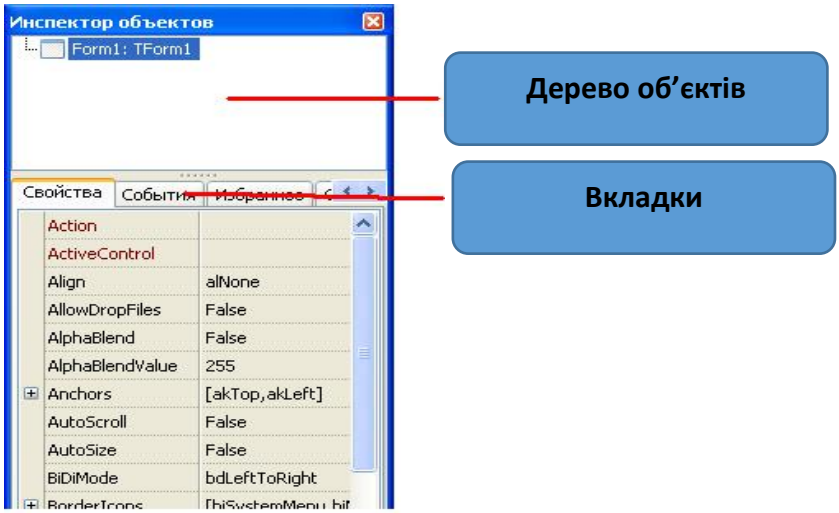

Рис. 1.5. *Інспектор об'єктів*

Редактор форм, Редактор коду й Вікно повідомлень

Останні три вікна простіші. **Редактор форм** призначений, відповідно, для редагування форми - положення й розмірів компонентів, розміщених на цій формі.

Незважаючи на явну схожість, форма й вікно додатка - не те саме. **Форма** - це те, що бачить програміст у процесі розробки проєкту, а вікно - це те, що побачить користувач, коли завантажить нашу програму.

Редактор коду містить вихідний код, що нам доведеться вводити й модифікувати. Редактор володіє рядом корисних умінь: підсвічує синтаксис команд, робить авто-відступ і авто-завершення команд, виводить необхідні підказки, загалом, сильно полегшує життя програмістові.

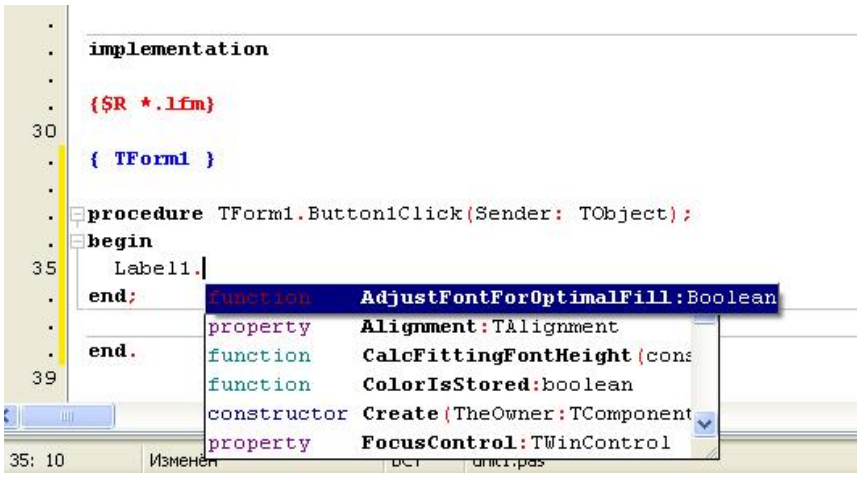

Рис. 1.6. *Редактор коду*

Нам доведеться часто перемикатися між **Редактором форм** і **Редактором коду**. Простіше всього це робити кнопкою **<F12>**.

І, нарешті, **Вікно повідомлень** виводить різні повідомлення: про знайдені помилки, про завершення компіляції, про наявність оголошених, але невикористовуваних змінних і т.п.

Схожість і відмінності з Delphi

Не можна не згадати про **Delphi**, що є платним аналогом **Lazarus**, і його попередником. І Lazarus, і Delphi підтримують код Об'єктного Pascal, використовують візуальні й невізуальні компоненти, мають схожий інтерфейс. Гляньте на малюнок:

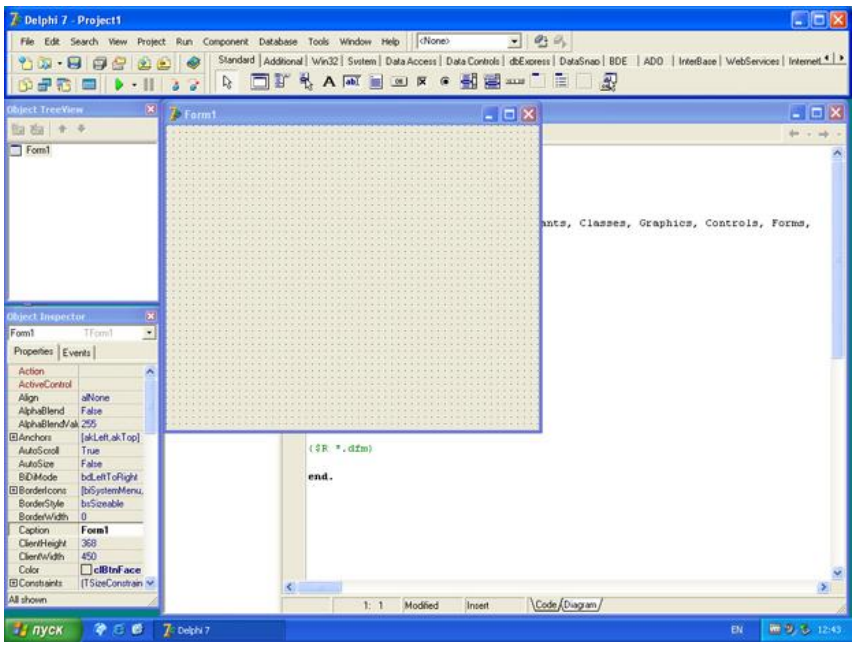

Рис. 1.7. *Вікна Delphi 7*

Як бачите, **Lazarus** дуже схожий на Delphi, однак є й відмінності.

- Lazarus, на відміну від Delphi, безкоштовний, і може вільно й легально застосовуватися в будь-якій навчальній, державній або виробничій установі, або будинку.
- Lazarus має власну бібліотеку компонентів LCL (Lazarus Component Library), а Delphi - VCL (Visual Component Library). Однак VCL і LCL багато в чому так схожі, що програміст при роботі з компонентами майже не відчуває різниці. Часто (але не завжди) проєкти, написані на Delphi можна без втрат компілювати на **Lazarus**.
- Lazarus крос-платформна IDE, тобто, підтримує різні операційні системи. Існує, щоправда, Kilyx - реалізація Delphi для Linux, однак Lazarus має реалізації для набагато більшого списку операційних систем, причому як 32-х, так і 64-х розрядних версій.
- Lazarus, у силу того, що молодший Delphi, поки має меншу підтримку: додаткові компоненти сторонніх розробників, сайти, присвячені мові й т.п.
- Lazarus має менш розвинені засоби для роботи з Базами Даних. Будемо сподіватися, це тимчасовий недолік.
- Деякі компоненти LCL в Lazarus ще «сирі» - іноді попадаються властивості, які не працюють. Найчастіше, це другорядні властивості, так що можна сміло використати компоненти й без них.

Незважаючи на всі відмінності, ці IDE так схожі, що можна сміло затверджувати - Delphi-програміст майже без зусиль зможе користуватися **Lazarus**, і навпаки. А бурхливий розвиток молодого

Lazarus гарантує, що в майбутньому його нечисленні недоліки будуть виправлені.

Перша програма

Насамперед, давайте створимо де-небудь загальну папку, у яку потім будемо складати всі наші проєкти. Нехай це буде

C:\Education
(education - освіта)

Кожний проєкт за правилами, повинен зберігатися в окрему папку. Щоб не заплутатися, будемо давати цим папкам імена, що відповідають номеру лекції, і номеру проєкту в ній (лекція-проєкт). Тобто, у першій лекції перший проєкт буде збережений у папку

C:\Education\01-01

Ви можете виробити й власні правила найменування папок - суть від цього не зміниться, аби тільки ви самі потім у них не заплуталися. Ми кілька разів згадали слово **проєкт**.

Проєкт - це те, що розробляє програміст. Коли проєкт готовий і **скомпільований**, виходить **програма**, з якої може працювати користувач. Проєкт - це набір зв'язаних файлів різного типу, а програма - це отриманий у результаті компіляції **виконуваний файл**. Детальніше про проєкти ми поговоримо в наступній лекції.

За традицією, перша створена програма повинна просто виводити повідомлення «Hello, world!» («Привіт, світ!»). Не будемо відступати від традицій, і зробимо це трьома різними способами. Отже, створимо на диску зазначені вище папки, куди збережемо наш перший проєкт. Завантажуємо **Lazarus**, якщо він ще не завантажений, і виділяємо редактор форм. У лівій частині, якщо ви не забули, перебуває **Інспектор об'єктів**, і в ньому виділена вкладка «**Властивості**» - це нам і потрібно. Серед властивостей знайдіть Caption, і замість тексту

Form1

який там перебуває за замовчуванням, впишіть

Hello, world!

По закінченні цієї процедури натисніть **<Enter>**. Як тільки ви це зробите, текст у заголовку форми зміниться:

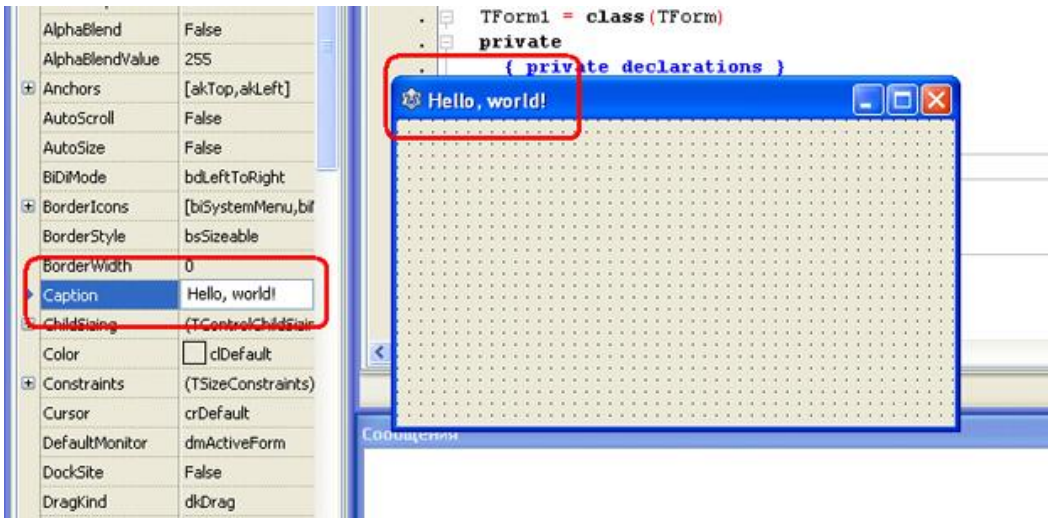

Рис. 1.8. *Перші кроки*

Тепер зверніть увагу на **Палітру компонентів**. На вкладці **Standard**, четвертий значок зображує кнопку з написом «**Ok**» на ній. Ця кнопка нам і потрібна.

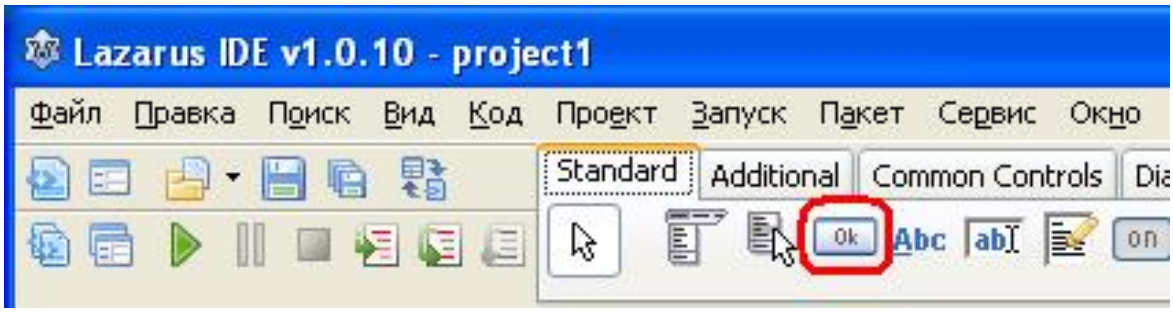

Рис. 1.9. *Кнопка TButton у Палітрі компонентів*

клацніть по ній мишею, а потім клацніть уже за формою, приблизно по центру. На формі відразу з'явиться кнопка, обрамлена рамочкою, з написом Button1 - таке ім'я **Lazarus** дав кнопці за замовчуванням. Рамочка навколо кнопки говорить про те, що схопивши мишею за одну з її сторін або кутів, ми зможемо міняти розміри кнопки. Схопивши за саму кнопку, ми зможемо переміщати її за формою.

Зліва, в **Інспекторі об'єктів**, список властивостей також змінився - деякі залишилися колишніми, інші додалися. Надійдемо, як і минулого разу - у властивості Caption замість Button1 впишемо Hello, world!. Потім мишею змінимо розміри й розташування кнопки, щоб у нас вийшло приблизно наступне:

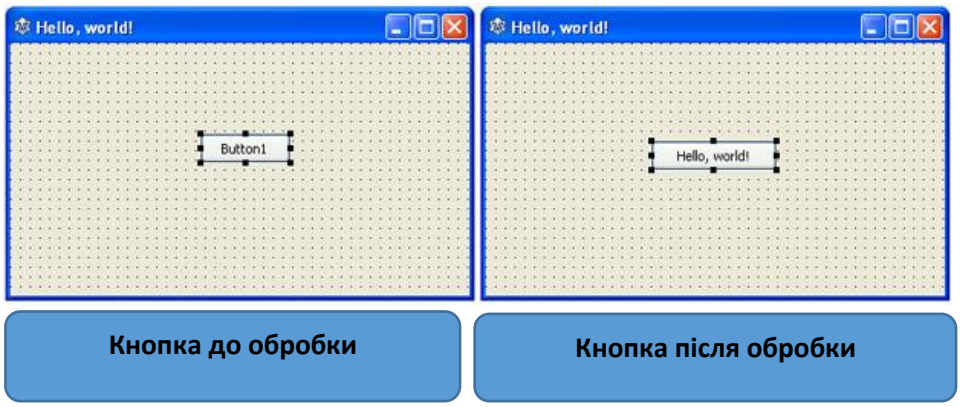

Рис. 1.10. *Кнопка до й після обробки*

Зверніть увагу: переміщати кнопку за формою можна не тільки мишею, але й клавішами **<Ctrl> + <Кнопки керування курсором>** (стрілки нагору, униз, уліво, вправо). А змінювати її розміри - клавішами **<Shift> + <Кнопки керування курсором>**. Цей спосіб зручніший для більш точного настроювання положення й розмірів будь-яких компонентів, не тільки кнопки.

Ми спробували два способи вивести необхідний текст. Тепер попрацюємо з вихідним кодом. Клацніть двічі по кнопці, і **Lazarus** відкриє **Редактор коду**, створивши оброблювач для цієї кнопки, і встановивши курсор всередину нього. Тут нам поки нічого розуміти не потрібно, просто впишіть текст, прямо туди, де перебуває курсор:

ShowMessage(’Hello, world!’);

Щоб вийшло так:

Рис. 1.11. *Код оброблювача кнопки*

На цьому наш проєкт закінчений. Залишилося зберегти його й скомпілювати в програму. Виберіть команду меню **Файл -> Зберегти**

все, або (що простіше) натисніть відповідну кнопку на **Панелі інструментів**:

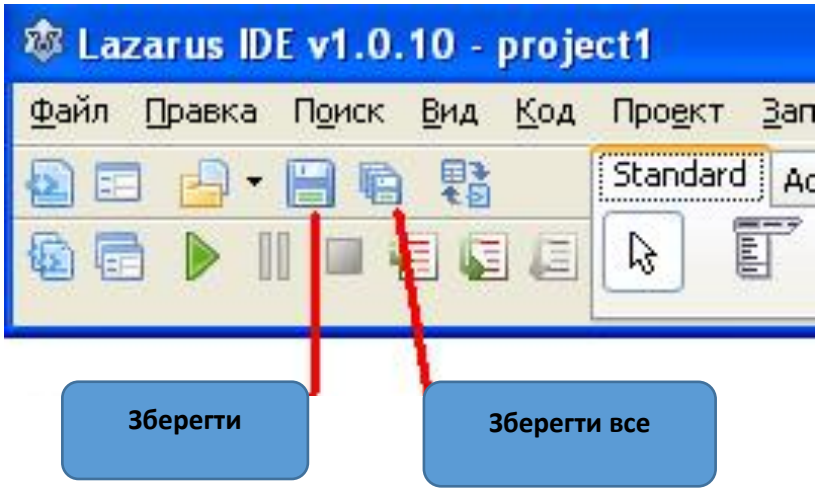

Рис. 1.12. *Кнопки збереження*

Кнопка «**Зберегти**» зберігає зміни поточної форми, кнопка «**Зберегти все**» - зміни всіх форм і модулів проєкту. Оскільки форма в нас одна, то можна нажати кожну з них. Як тільки ви це зробите, вийде вікно збереження проєкту. Ви пам'ятаєте, у яку папку ми будемо зберігати перший проєкт першої лекції? Туди й зберігайте. Назва проєкту залишіть без змін, детальніше про це ми поговоримо в наступній лекції.

Як тільки ви збережете проєкт, вийде ще один запит - на збереження файлу модуля **Unit1**. Тут ми теж залишаємо назву за замовчуванням, натиснемо лише кнопку «**Зберегти**». Зверніть увагу - кнопки «**Зберегти**» і «**Зберегти всі**» стали неактивними - це означає, що ні у формі, ні в проєкті в цілому в нас змін немає.

Добре, проєкт ми зберегли, однак працюючої програми в нас поки немає. Щоб її одержати, потрібно **скомпілювати** проєкт. Причому зробити це можна трьома командами **Головного меню**:

- **Запуск -> Компілювати**
- **Запуск -> Зібрати**
- **Запуск -> Запустити**

Остання команда не тільки компілює проєкт і створює завантажувальний файл програми, але й відразу запускає його на виконання, так що в більшості випадків краща саме ця команда. Зручніше за все скористатися відповідною кнопкою на **Панелі інструментів**:

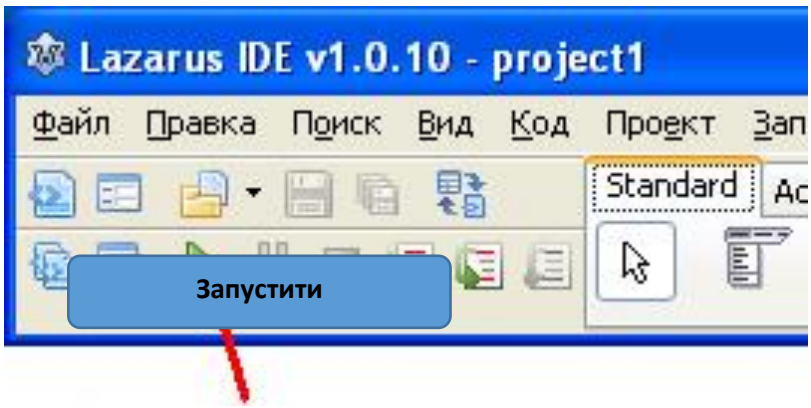

Рис. 1.13. *Кнопка запуску*

Натисніть цю кнопку, проєкт скомпілюється й відразу ж запуститься. Як тільки ми натиснемо кнопку «**Hello, world!**» у вікні програми, вискочить записане в **Редакторі коду** повідомлення:

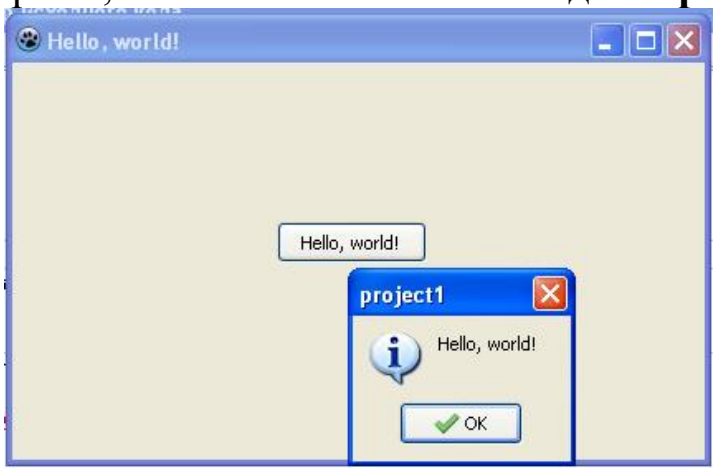

Рис. 1.14. *Робота нашої програми*

Натисніть кнопку «**Ок**», закривши повідомлення, після чого закрийте саму програму (не проєкт!). Якщо вийде подібне повідомлення:

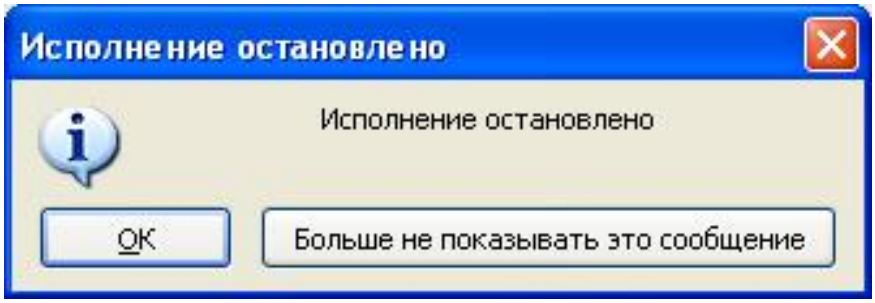

Рис. 1.15. *Повідомлення про зупинку виконання*

то раджу натиснути кнопку «**Більше не показувати це повідомлення**», щоб надалі воно вам не набридало. Після цього можете закрити й **Lazarus**. Файл, що **виконує, project1.exe** з нашою

програмою ви знайдете в папці, куди зберігали проєкт. Це вже цілком працездатна програма, її можна запустити прямо із цієї папки або переслати другові, щоб він позаздрив вашим новим знанням (для виконання програми потрібно тільки один файл **project1.exe**, інші файли не потрібні - це файли проєкту, які потрібні тільки програмістові).

Тема 2. Анатомія проєкту

1. Налаштування IDE

Щоб полегшити подальшу роботу з **Lazarus**, виконаємо деякі настроювання його IDE. Завантажуйте **Lazarus**. За замовчуванням включена опція завантаження останнього проєкту, тому у вас повинен був завантажитися проєкт із першої лекції. Це дуже зручно, якщо ви працюєте над яким-небудь більшим проєктом, на розробку якого йде не один день. Однак це буде незручно при вивченні **Lazarus** - адже ми будемо робити безліч невеликих програм, що виконують різні завдання, і для цього нам потрібно буде спочатку закрити старий проєкт, а потім створити новий. А це незручно.

Виберіть у головному меню команду «**Сервіс -> Параметри**», відкриється вікно «**Параметри IDE**». Нам потрібний розділ «**Файли**», розташований у гілці «**Оточення**»:

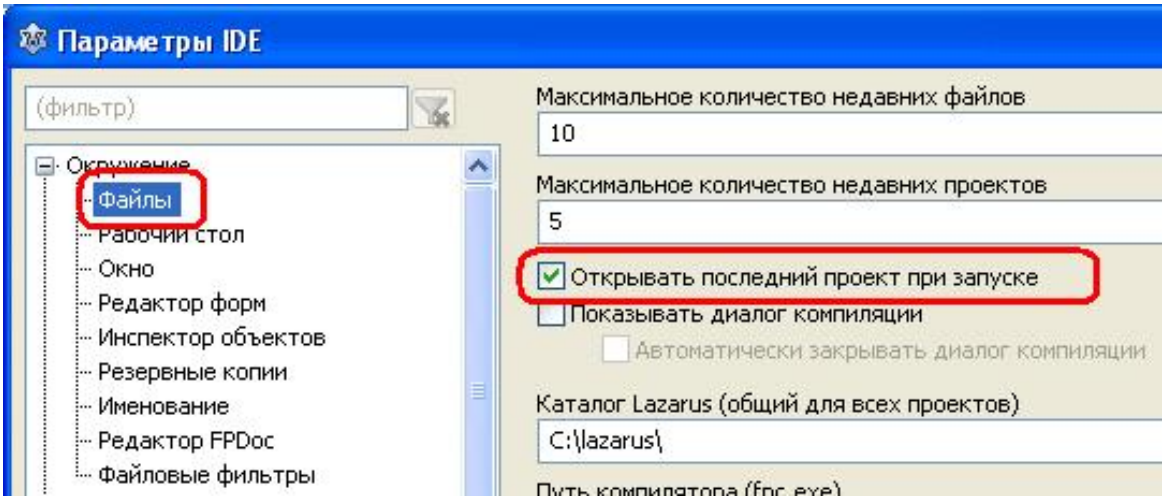

Рис. 2.1. *Параметри – Файли*

У цьому розділі є прапорець «**Відкривати останній проєкт при запуску**». Якщо цей прапорець включений, то щораз при запуску буде відкриватися останній проєкт. Якщо виключено - буде створюватися новий проєкт, що нам і потрібно. Виключимо його, натиснемо кнопку «**ОК**», закриємо **Lazarus** і знову запустимо. Ну от, зовсім інша справа,

створився новий проєкт. Нічого в проєкті не міняючи, просто збережемо його в папку

C:\Education\02-01

(ви пам'ятаєте про наш договір із приводу найменування папок із проєктами?). Натиснемо кнопку «**Запустити**» на **Панелі інструментів** (зелена стрілка вправо), щоб проєкт скомпіювався, і отримана програма відразу ж завантажилася на згадку. З'явиться просте віконце нашої програми, адже в ній нічого, крім форми, немає. Закрийте отриману програму, залишивши **Lazarus** із проєктом відкритим.

Потім відкрийте папку із проєктом за допомогою **Провідника** Windows або звичного для вас файлового менеджера (Total Commander, Far, і т.п.). У **Провіднику** встановіть вид відображення «**Таблиця**». Ви побачите приблизно таку картинку:

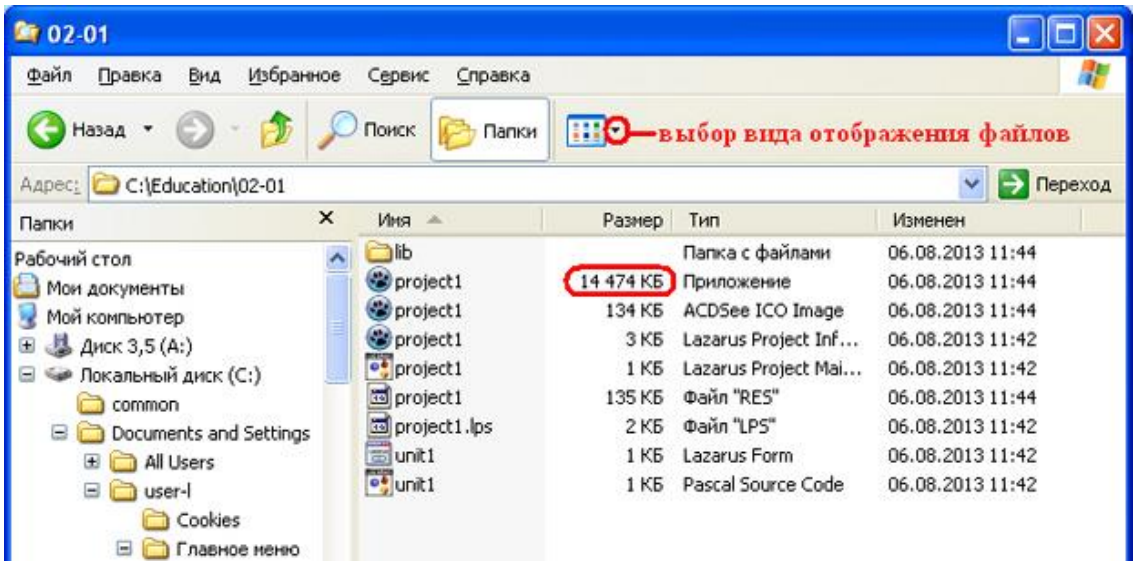

Рис. 2.2. *Папка проєкту з порожньою формою*

Наша програма одержала назву **project1.exe**, у колонці «**Тип**» значиться «**Додаток**». А от у колонці «**Розмір**» зазначено 14 474 кілобайта (у вас може ледве відрізнятися, це не страшно). Ого! Програма з порожнім віконцем, що зовсім нічого не робить, «важить» більше 14 мегабайт!!! Як таке може бути?!

Справа в тому, що при компіляції **Lazarus** за замовчуванням додає в програму всіляку налагоджену інформацію, у результаті чого й виходить такий великий розмір. Це корисно, коли ви налагоджуєте програму (про налагодження й тестування програм ми будемо говорити в іншій лекції), але зайве в нашому випадку. На жаль, відключити цю функцію можна тільки для поточного проєкту.

Робиться це так: виберіть команду **Головного меню** «**Проєкт -> Параметри проєкту**»:

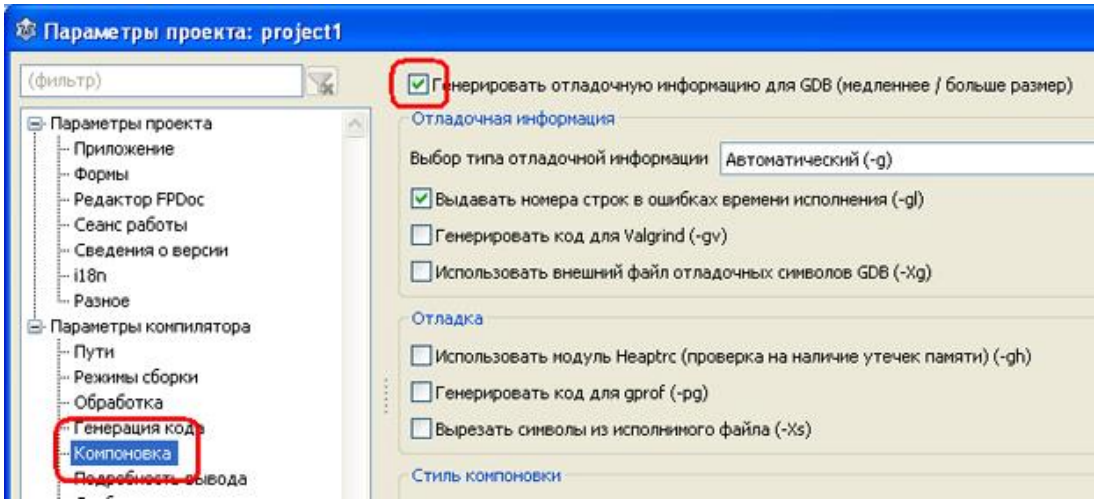

Рис. 2.3. *Налаштування проєкту*

У розділі «**Компонування**» зніміть прапорець «**Генерувати налагоджувальну інформацію для GDB**», натисніть «**ОК**» і заново запустіть програму. Після того, як програма запуститься, закрийте її й знову подивіться на отриманий розмір. Ну от, тепер порядок, розмір ледве більше півтора мегабайт. Теж чимало, якщо розібратися, але цілком нормально, з огляду на підключені до проєкту стандартні модулі з підтримкою кросплатформенності. Надалі ви можете або не звертати уваги на розмір навчальних програм - знімати цей прапорець (після налагодження, якщо воно потрібне) тільки для своїх готових реальних проєктів, або знімати його в кожному новому проєкті. Це на ваш розсуд. А якщо вам хочеться одержати зовсім малий розмір програми, то можна додатково скористатися яким-небудь програмним пакувальником. ASPack, наприклад, дозволяє стискати *.exe і *.dll файли до 80%! Такий пакувальник, до речі, не тільки стискає, але й шифрує вашу програму, створюючи їй додатковий захист. Сильного хакера це, щоправда, не зупинить, але зайвого головного болю йому додасть.

Також у налаштуваннях можна змінити розміри осередків у сітці із крапок, що відображається в **Редакторі форм**. Ця сітка потрібна для того, щоб програміст міг вирівнювати компоненти на формі відносно один одного. Виберіть команду **Головного меню** «**Сервіс -> Параметри**» і в розділі «**Оточення**» виберіть «**Редактор форм**». У правій верхній частині розташований розділ «**Сітка**», де можна змінювати її розміри (за замовчуванням, встановлено по 8 пікселів для

осі X і для осі Y), дозволяти або забороняти показ сітки, і т.д. Деякі програмісти воліють робити осередок сітки менше - 6 або навіть 4 пікселя по обох осях, мотивуючи це тим, що чим частіше сітка, тим точніше можна підрівняти компоненти. Може бути, вони й праві, однак майте на увазі, що розроблювачі **Lazarus** виставляли за замовчуванням найбільш зручні параметри. Загалом, ви можете встановити налаштування на свій смак, або залишіть їх за замовчуванням.

Складові проєкту

Ми раз у раз повторюємо слово «**Проєкт**». На першій лекції ми говорили, що **проєкт** - це набір зв'язаних файлів різного типу, з яких, зрештою, після компіляції, виходить **програма**.

З яких же файлів складається проєкт?

Виберіть команду **Головного меню** «**Сервіс -> Параметри**», і в гілці «**Оточення**» перейдіть на розділ «**Файлові фільтри**». Ви побачите 6 основних типів файлів, які можуть зустрічатися в проєкті:

- Модуль Lazarus (*.pas;*.pp)
- Проєкт Lazarus (*.lpi)
- Форма Lazarus або Delphi (*.lfm;*.dfm)
- Пакет Lazarus (*.lpk)
- Вихідний код проєкту Lazarus (*.lpr)
- Інший файл Lazarus (*.inc;*.lrs;*.lpl)

Якщо ми перейдемо в папку з нашим проєктом, то побачимо, що він складається з восьми файлів:

1. project1.exe (Виконує файл, що, програми).
2. project1.ico (Файл із «іконкою» проєкту - зображенням у вигляді лабети гепарда, що з'являється у верхньому лівому куті вікна програми).
3. **project1.lpi** (Інформаційний файл проєкту). Якщо ви бажаєте відкрити даний проєкт в **Lazarus**, то запускати потрібно саме цей, інформаційний файл.
4. **project1.lpr** (Вихідний файл проєкту). Запуск цього файлу також приведе до запуску **Lazarus** із завантаженням даного проєкту.
5. **project1.lps** (Конфігурація проєкту у вигляді xml-коду)
6. **project1.res** (Файл ресурсів, використовуваних у проєкті)
7. **unit1.lfm** (Файл форми модуля. У ньому в текстовому вигляді відображені настроювання всіх компонентів, використовуваних у модулі. Редагувати цей файл у ручну настійно не рекомендується, для редагування цих даних потрібно використати Редактор форм).

8. **unit1.pas** (Вихідний код модуля мовою Object Pascal).

Файли з ім’ям project1 - це файли всього проєкту в цілому, файли з ім’ям unit1 - це файли модуля.

Модуль> - це окрема одиниця вихідного коду, виконана у вигляді файлу з розширенням *.pas. Сукупність таких одиниць становить програму.

Коли ми створюємо вікно, то для нього створюється два файли: модуль - файл *.pas з вихідним кодом, і файл *.lfm, у якому утримуються настроювання використовуваних на формі компонентів. Текст модуля ми можемо бачити в **Редакторі коду**. Однак модуль не завжди має вікно, це може бути й просто текстовий файл із вихідним кодом. Про модулі і їхні розділи ми поговоримо детальніше в одній з наступних лекцій. У нашому проєкті всього один модуль, але взагалі їх може бути скільки завгодно. І кожний модуль буде представлений цією парою файлів.

Крім того, у папці проєкту перебуває папка **lib**, у якій розташовуються дані, що підключають до проєкту, і інформація про компіляції. Якщо ж ви змінювали проєкт, і зберігали ці зміни, то з’явиться також папка **backup**, у якій будуть зберігатися резервні копії старих варіантів проєкту.

Нерідко програміст додає в проєкт і свої типи файлів. Наприклад, у проєкті можна використати базу даних, який-небудь текстовий файл або **ini-файл** для збереження користувальницьких настроювань. Розумно розташовувати ці файли також у папці із проєктом.

Тепер декілька порад із приводу найменування проєкту й модулів. Проєкт варто називати так, як ми хочемо, щоб називалася наша програма. Наприклад, проєкту з першої лекції було б доцільно дати ім’я «Hello» замість нейтрального «project1».

Модулі ж потрібно називати, виходячи з їхнього значення. Завжди в проєкті є **головний модуль**. У наших проєктах поки що було по одному вікну. Модуль, створений для цього вікна, і буде головним. У навчальній літературі є безліч рекомендацій, як позначати модулі, зупинимося на одній з них. Давайте домовимося в майбутньому головний модуль називати Main (англ. main - головний), а іншим модулям давати значеннєві назви, наприклад, Options, Editor і т.п. Форму цього модуля (точніше, властивість Name форми) будемо називати також, але із приставкою f-, що позначає форму. Тобто, fMain, fOptions, fEditor і так далі. Закріпимо цей матеріал на практиці.

Відкрийте **Lazarus**, якщо він у вас закритий, або закрийте старий проєкт і почніть новий. У цей момент у **Редакторі форм** у нас порожня форма, у заголовку якої написане Form1 - це ім'я форми за замовчуванням. Ми домовилися називати головний модуль Main, а його формі додавати приставку f-. В **Інспекторі об'єктів** знайдіть властивість Name, і замість Form1 впишіть туди fMain. Як тільки ви натиснете **<Enter>**, заголовок форми зміниться на новий. Тепер збережемо проєкт і модуль головної форми. Натисніть кнопку «**Зберегти все**» на **Панелі інструментів**, або виберіть команду **Головного меню** «**Файл -> Зберегти все**».

У запиті замість імені проєкту project1 укажіть нове ім'я Hello, не забувайте, що ми домовилися зберігати проєкти в папки з ім'ям по номері лекції, і номеру проєкту в ній. У нашому прикладі це буде

C:\Education\02-02

Як тільки ви натиснете кнопку «**Зберегти**», вийде запит на збереження головного модуля. Форму ми назвали fMain, виходить, модулю дамо назву просто Main. В **Lazarus** малі та великі літери не розрізняються, однак для зручності читання коду краще привчитися використовувати великі літери, щоб виділяти назви. Наприклад, FileEdit, SaveAll і т.п.

У властивості Caption форми впишемо слово «Вітання» (зрозуміло, без лапок), це буде більш зрозумілим заголовком для вікна. Не забувайте після введення нових значень властивостей в **Інспекторі об'єктів** натискати **<Enter>**, щоб зміни набутили чинності. Тепер установимо на форму компонентів TLabel (мітку), що дозволить виводити на формі текст. Компонент перебуває на вкладці **Standard**:

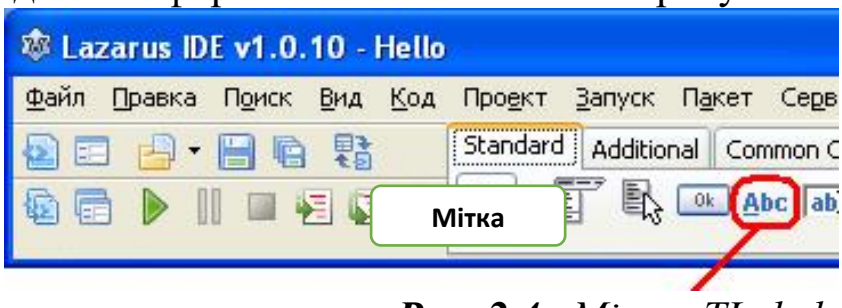

Рис. 2.4. *Мітка TLabel*

Підказка: якщо підвести курсор миші до компонента та не натискати кнопку, вийде спливаюча підказка з ім'ям компонента.

Клацніть мишкою по мітці, потім за формою, у верхній частині вікна. Оскільки мітка в нас одна, то можна залишити їй ім'я

(властивість Name) за замовчуванням - Label1. А от у властивості Caption мітки напишіть:

Як вас звуть?

Нижче мітки помістите компонент TEdit - текстове поле, що редагує, у якому користувач зможе написати щось:

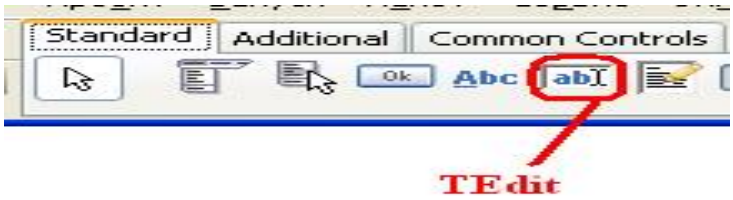

Рис. 2.5. *Текстове поле TEdit*

У цього компонента властивість Name також залишимо за замовчуванням - Edit1. Як ви можете помітити, у цього компонента властивості Caption немає, зате з'явилася властивість Text - саме отут і втримується текст, відображений у полі. За замовчуванням, він збігається з ім'ям компонента. Просто очистимо цю властивість, видаливши з нього старий текст (не забувайте про **<Enter>**).

Ще нижче встановимо кнопку **TButton**. Залишимо її ім'я за замовчуванням, а у властивості Caption напишемо

Виконати

Змінимо положення й розміри компонентів і самої форми так, щоб форма прийняла приблизно такий вигляд:

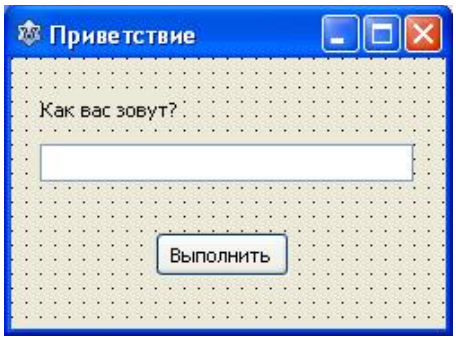

Рис. 2.6. *Остаточний вигляд головної форми*

Тепер запрограмуємо натискання кнопки. Клацніть по ній двічі, щоб згенерувалася подія, і автоматично відкрився **Редактор коду**. У місці, де мигає курсор, впишемо наступний код:

ShowMessage('Привіт, ' + Edit1.Text + '!');

Редактор коду повинен виглядати так:

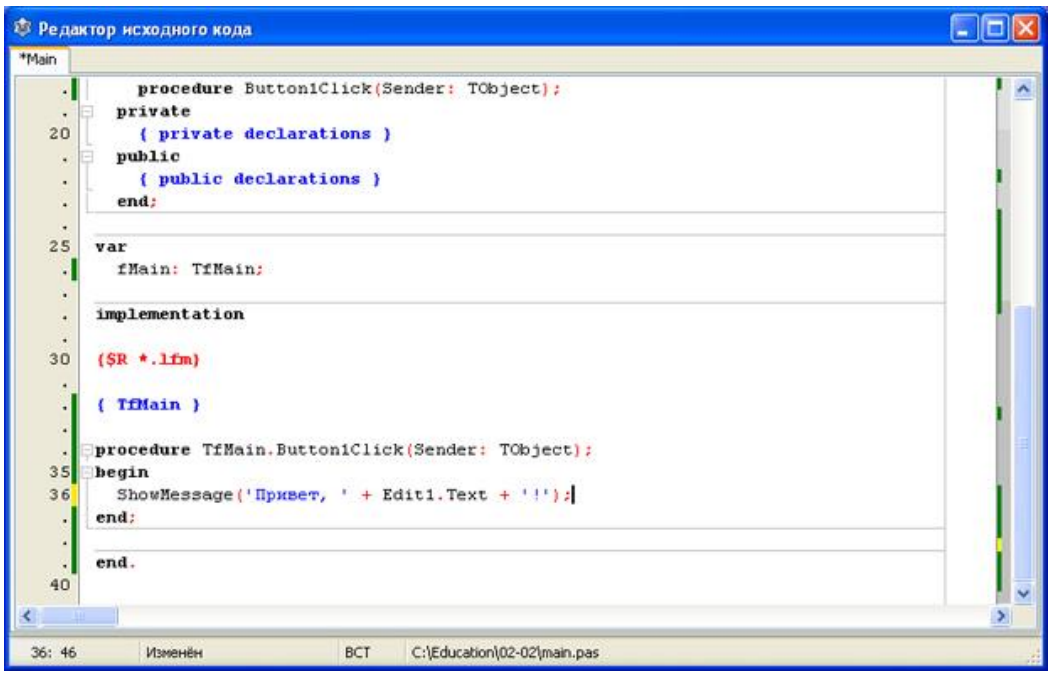

Рис. 2.7. *Введений код*

Збережіть проєкт і запустіть його. Коли програма завантажиться, впишіть у вікні **Edit1** своє ім'я й натисніть кнопку «**Виконати**». Ви повинні одержати приблизно такий результат:

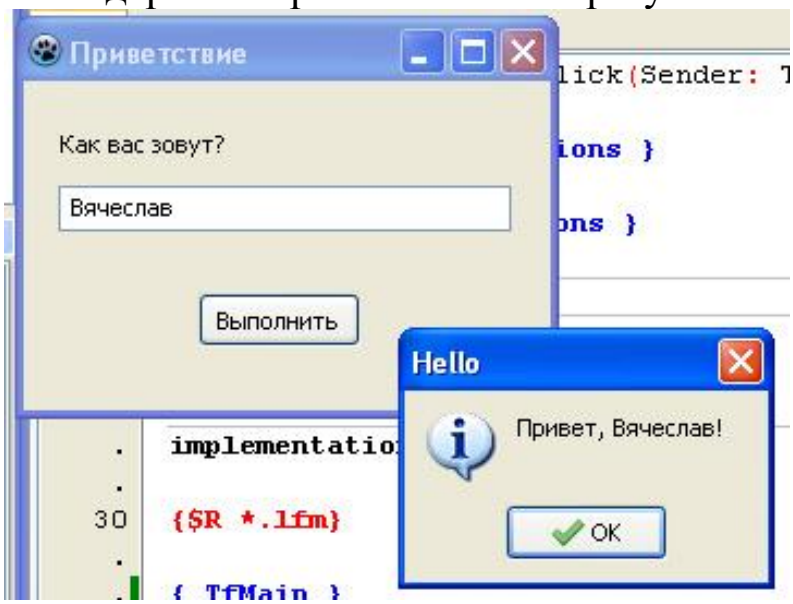

Рис. 2.8. *Програма Hello.exe у дії*

Ви можете скільки завгодно разів міняти ім'я в текстовому полі й знову натискати кнопку «**Виконати**», одержуючи вітання вже з іншим текстом. У нас вийшла дійсна **інтерактивна** програма, тобто, програма, взаємодіюча з користувачем.

У подальших лекціях ми не будемо так докладно зупинятися на тому, як зберігати проєкт або модуль, обмежуючись коротким

Збережіть проєкт під ім'ям ...

Як і куди зберігати проєкт, ви вже повинні знати.

Тема 3. Робота з компонентами

1. Форма

Ми багато разів вживали слово **компоненти**, але що це таке?

Компонент - це якийсь віртуальний об’єкт, що має свої властивості й подіями, перебуває на Палітрі компонентів, і який можна встановити на форму.

Форма не перебуває в **Палітрі компонентів**, однак, це теж компонент! Адже й у форми є властивості й події, а ми маємо можливість міняти перші й програмувати другі. Познайомимося з основними властивостями форми. Відкрийте **Lazarus** і новий додаток у ньому. Оскільки нічого, крім форми, у додатку поки немає, в Інспекторі об’єктів відображаються властивості форми. Причому всі властивості нам не видні, потрібно прокручувати **Інспектори об’єктів**, щоб розглянути інші властивості або знайти потрібне.

Властивість Name нам уже знайома - це ім’я компонента, у нашому випадку, форми (англ. name - ім’я). По цьому імені ми можемо програмно звертатися до його різних властивостей. Ми домовилися, що головну форму ми будемо називати fMain. Інакше кажучи, властивості Name головної форми ми привласнимо значення fMain. Так і зробимо. Як тільки ми перейменували форму, автоматично змінився і її заголовок.

Властивість Caption - заголовок форми (англ. caption - заголовок). У цю властивість впишемо новий текст:

Моя форма

Як тільки ми впишемо нове значення й натиснемо **<Enter>**, текст заголовка зміниться.

Оскільки нам доведеться сьогодні багато експериментувати, буде незайвим зберегти новий проєкт. Збережіть його під ім’ям **Proba** у папку

C:\Education\03-01

Ви ще не забули, що модуль ми називаємо так само, як форму, але без початкового «**f**»? У нашому випадку, це буде **Main**.

Тепер давайте прогуляємося по основних властивостях форми, від початку до кінця.

AlphaBlend - прозорість. Це властивість не назвеш основною, однак у деяких випадках вона може додати нашому додатку додаткову можливість. Це логічна властивість, у ній ми можемо вибрати або False (англ. false - хиба) - значення за замовчуванням, або True (англ. true -

істина). Інші значення в цю властивість ми вписати не зможемо. Якщо в цій властивості встановлене False, то формі заборонено бути прозорою, якщо ж True,- то дозволено. Давайте для проби встановимо True.

Порада. Поміняти False на True, або навпаки, можна або вибравши потрібне значення, або двічі клацнувши за змінюваним значенням.

AlphaBlendValue - значення прозорості. Просто встановити AlphaBlend в True недостатньо, щоб зробити форму прозорою. Потрібно ще вказати значення прозорості в AlphaBlendValue. Це значення може змінюватися від нуля (повна прозорість) до 255 (повна непрозорість). Інші значення ви просто не зможете встановити. Майте на увазі, що якщо ви зробите форму повністю прозорою, то побачити її в робочій програмі не зможете. Закривати прийдеться або за допомогою кнопки програми в рядку стану (праворуч від кнопки «**Пуск**»), або за допомогою **Диспетчера завдань** Windows, або (якщо ви скомпілювали й завантажили програму командою «**Запустити**» або аналогічною кнопкою на **Панелі інструментів**) виконавши команду **Lazarus** «**Запуск -> Скинути налагоджувач**». Але краще до цього не доводити. Встановіть значення в 125 (напівпрозорість), натисніть кнопку «**Запустити**» і подивіться, що вийшло:

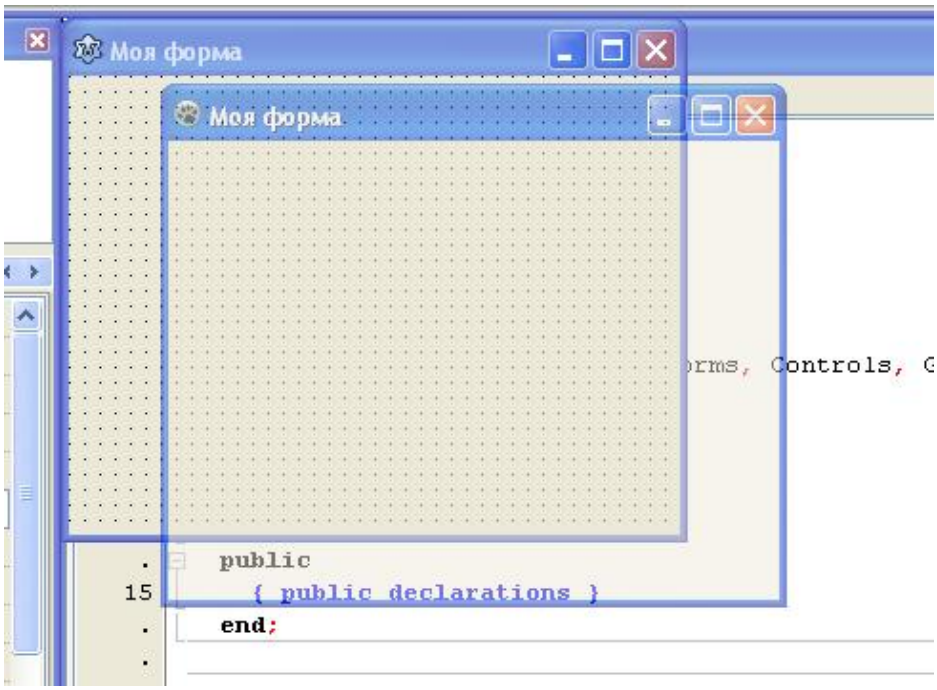

Рис. 3.1. *Напівпрозоре вікно*

Поверніть у властивість AlphaBlend значення False, тому що для подальших експериментів прозорість вікна буде нам заважати.

AutoScroll - автоматичне прокручування форми. Також логічна властивість. Якщо встановлено в True, то прокручування вікна форми дозволені. При цьому, якщо частині форми з'являється смуга прокручування, дозволяючи прокручивать форму. Залишимо деякі компоненти на формі,що стали невидимі, наприклад, при зменшенні розміру форми, то в правій значення за замовчуванням False - прокручування заборонене.

AutoSize - автоматичний розмір. Якщо встановлено в True, то форма автоматично буде приймати розмір, достатній тільки для встановлених компонентів. Майте на увазі, що побачити цей ефект ви зможете тільки в працюючому додатку, не у формі. Якщо, наприклад, встановити на форму кнопку, і запустити додаток, то вікно стиснеться по розміру кнопки. Не дуже передбачуваний ефект, тому дана властивість використовується рідко. Можете поекспериментувати імітуючи цю властивість, але наприкінці встановіть в ній значення за замовчуванням False.

BorderStyle - стиль обрамлення форми. Дуже цікава властивість, що сильно впливає на вид вікна. Може приймати наступні значення:

- bsDialog - Діалогове вікно з неможливістю змінювати розміри.
- bsNone - Вікно без рядка заголовка. Неможливо змінити розміри, неможливо переміщувати по екрані.
- bsSingle - Вікно, розміри якого не можна змінювати, але яке можна згорнути/розгорнути за допомогою кнопок рядка заголовка.
- bsSizeable - Звичайне вікно Windows.
- bsSizeToolWin - Рядок заголовка більш вузьке, кнопки згорнути/розгорнути відсутні, але можна змінювати розміри вікна мишею, схопивши за край або кут вікна.
- bsToolWindow - Як **bsSizeToolWin**, але не можна мишею змінювати розміри вікна.

Встановіть значення bsDialog і запустіть програму, подивітьсяя на зовнішній вигляд отриманого вікна.

Font - шрифт форми й всіх компонентів на ній. Коли ви клацаєте по властивості, у його правій частині з'являється кнопка із трьома крапками:

Рис. 3.2. *Розкриваюча властивість, Font*

Якщо клацнути по цій кнопці, відкриється діалог вибору шрифту. Якщо ви виберете, наприклад, шрифт Times New Roman, звичайний, розміром 11 пікселів, то це позначиться на всіх компонентах, розташованих на формі. Вони всі одержать цей шрифт «у спадщину», хоча для окремих компонентів потім можна буде вибрати й інший шрифт.

Height - Висота форми. Розмір у пікселях від рядка заголовка до нижньої межі форми. Коли ви мишею міняєте розміри форми, автоматично міняється й це значення. Змінити висоту вікна можна й вручну, вказавши в цій властивості потрібний розмір. Не працює, якщо вікно показується розгорнутим.

Left - Ліва границя форми. Відстань у пикселях від лівої границі робочого стола до лівої границі форми. Не працює, якщо встановлено положення форми по центрі робочого стола (екрана), або вікно розгорнуте.

Position - Положення форми. Може приймати наступні значення:

1. poDefault - положення вікна визначає Windows.
2. poDefaultPosOnly - розмір вікна відповідає проєкту, а його положення визначає Windows.
3. poDefaultSizeOnly - положення вікна відповідає проєкту, а його розмір визначає Windows.
4. poDesigned - значення за замовчуванням, і розмір, і положення визначає розроблювач.
5. poDesktopCenter - вікно буде розташовуватися по центрі робочого стола.
6. poMainFormCenter - дане значення застосовують для додаткових (неголовних) форм проєкту. Вікно буде розташовуватися по центрі головного вікна.
7. poOwnerFormCenter - це значення також використається для додаткових вікон, розташовуючи його по центрі батьківського вікна, тобто, вікна, з якого відбувся виклик даного вікна.
8. poScreenCenter - вікно розташовується по центрі екрана.

Порада: головне вікно розташовуйте по центру робочого стола, а інші вікна проєкту - по центру головної форми. Такі налаштування будуть кращі для переважної більшості проєктів.

ShowHint - показ підказок. Якщо ви підведете покажчик миші до якого-небудь компонента в **Палітрі компонентів** (або до кнопки на **Панелі інструментів**), але не будете натискати кнопку, то через якийсь час з’явиться спливаюча підказка (хінт, від англ. hint - натяк,

натякати) з ім’ям компонента (кнопки). Властивість логічна, якщо встановлено в True, то показ таких підказок дозволений для всіх установлених на формі компонентів, або для самої форми.

Hint - текст самої підказки. Спробуйте дозволити показ підказок, а в цій властивості напишіть який-небудь текст, наприклад, «Наша спливаюча підказка». Запустіть програму, наведіть на форму покажчик миші, але не натискайте на неї:

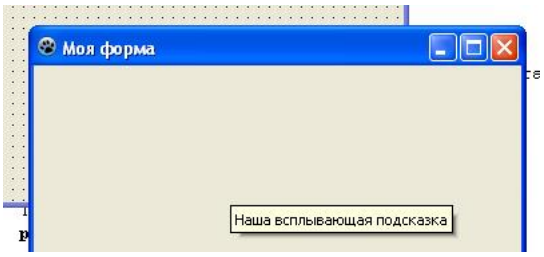

Рис. 3.3. *Спливаюча підказка*

Top - Верхня границя форми. Відстань у пікселях від верхньої границі робочого стола до рядка заголовка форми. Не працює, якщо встановлено положення форми по центру робочого стола (екрана), або вікно розгорнуте.

Width - Ширина форми. Розмір у пікселях від лівої до правої границі форми. Коли ви мишею міняєте розміри форми, автоматично міняється й це значення. Не працює, якщо вікно показується розгорнутим.

WindowState - Стан вікна. Може приймати наступні значення:

- wsFullScreen - у весь екран.
- wsMaximized - розгорнуто.
- wsMinimized - згорнуто.
- wsNormal - звичайний стан. Це значення за замовчуванням, найчастіше використовується саме воно.

Для подальших експериментів давайте встановимо наступні параметри форми:

BorderStyile = bsSizeable
Height = 400
Position = poDesktopCenter
Width = 600
WindowState = wsNormal

Примітка. Ми розглянули далеко не всі властивості форми, і зовсім не зачіпали її події. Одні з них ми й зовсім не будемо розглядати, тому

що в практичному програмуванні вони звичайно не використовуються, інші ми розглянемо на наступних лекціях.

Примітка. Ми розглядали властивості досить докладно. Але справа в тому, що багато компонентів мають схожі властивості. Так, у всіх компонентів є властивість Name, у багатьох з них є такі властивості, як Caption, Left, Top, Height, Width і т.п. Вивчивши ці властивості в одних компонентів, ми з легкістю зможемо ними користуватися й в інших.

2. Панель і TSplitter

Панель - це свого роду контейнер, для розміщення на ній зв'язаних по якійсь ознаці компонентів. Розміщені на панелі компоненти «прив'язуються» до неї. При переміщенні панелі за формою, будуть переміщатися й ці компоненти. Панель нерідко використовують і для прикраси форми. На формі може бути кілька панелей.

Після вивчення попереднього розділу, у вас повинен бути відкритий проєкт із порожньою формою, що налаштована певним чином. Якщо це так, установимо на форму панель. Для цього клацніть по компоненту **TPanel** у палітрі компонентів, а потім клацніть за формою, установивши на ній панель:

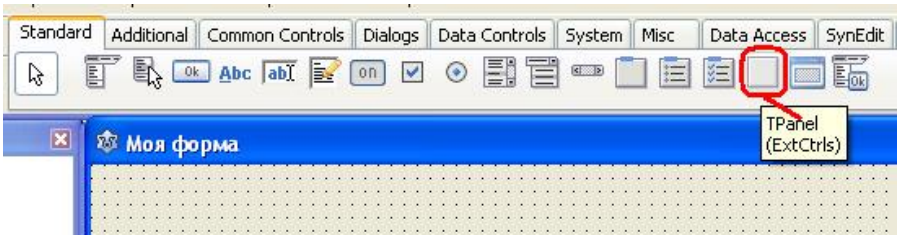

Рис. 3.4. *Компонент TPanel*

У вас на формі з'явиться якийсь прямокутник, в центрі якого буде написано Panell - ім'я компонента, що **Lazarus** привласнив за замовчуванням. Якщо ми будемо встановлювати інші панелі, **Lazarus** назве їх Panel2, Panel3 і т.д. Якщо на формі в нас буде тільки одна панель, то цифру в назві (властивість Name) і зовсім можна буде забрати. У цьому випадку в нас буде кілька панелей, так що залишимо панелі ім'я, що їй привласнив **Lazarus** за замовчуванням. Розглянемо деякі властивості панелей, якими нам часто прийдеться користуватися (компонент Panel1 при цьому повинен бути виділений).

Name	- ім’я панелі. Якщо будемо її перейменовувати, то в цій властивості.
Caption	- текст у центрі панелі, у нашому випадку, це Panel1. Цей текст дуже рідко використовується, звичайно його видаляють, щоб панель була порожня. Ми теж очистимо цей текст.
Left, Top	- відстань у пікселях, відповідно, від лівої й верхньої границь форми до панелі. Встановимо в обох властивостях значення 10.
Height	- висота панелі в пікселях, встановимо її в 320.
Width	- ширина панелі в пікселях, встановимо її в 580.

Тепер збережіть зміни в проєкті й запустіть його. Ви побачите вікно з опуклою панеллю, під якою залишилося місце для кнопки. Так звичайно оформляють діалогові вікна. Однак опуклість панелі можна й забрати, перетворивши її в акуратну рамочку. Закрийте вікно й поверніться до проєкту. Розглянемо три властивості, що впливають на опуклість панелі.

BevelInner - визначає стиль внутрішньої границі панелі. Може мати значення:

- bvNone - ніякого стилю. (за замовчуванням для BevelInner)
- bvSpace - порожня відмальовка границі, візуально не відрізняється від bvNone.
- bvRaised - піднята границя. (за замовчуванням для BevelOuter)
- bvLowered - утоплена границя.

BevelOuter - визначає стиль зовнішньої границі панелі. Може набувати такого ж значення, як і BevelInner.

BevelWidth - ширина окантовки в пікселях. За замовчуванням, має значення 1, що звичайно цілком достатньо. Щоб ще більше підкреслити границю, можна встановити значення 2, однак подальше збільшення ширини не зробить панель кращою, скоріше, навпаки. Втім, це залежить від вашого смаку й конкретного дизайну.

Порада: щоб панель зробити ввігнутою, властивість BevelOuter встановіть в bvRaised, а щоб зробити гарний кант, зробіть BevelInner =

bvLowered, а BevelOuter = bvRaised. Для ширини кантуцілком вистачить 1 пікселя.

Спробуйте «програти» із властивостями панелі: міняйте їх, запустіть проєкт на виконання, і дивиься, як буде мінятися зовнішній вигляд панелі. А коли награєтеся й запустите програму востаннє, розгорніть вікно на весь екран. От це вже неприємно: вікно програми збільшилося, а панель залишилася колишньою:

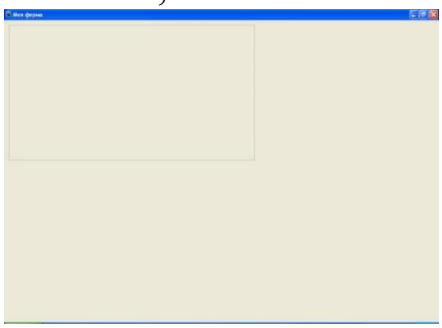

Рис. 3.5. *Перекручування дизайну вікна при зміні розмірів*

Справа в тому, що панель «прив'язується» до верхньої й лівої границі форми. А при зміні розмірів вікна, права й нижня границі панелі не змінюються. Щоб «прив'язати» до форми всі сторони панелі, слугують «якорі»:

Anchors (англ. якоря) - властивість, що дозволяє або забороняє прив'язку сторін до зовнішнього контейнера. Якщо панель розташовується прямо на формі, то й прив'язується вона до форми. А якщо панель розташовується на іншій панелі, то прив'язується вже до неї. Подібну властивість має більшість візуальних компонентів. Anchors - властивість, що розкривається. Якщо клацнути по кнопці «+» ліворуч від неї, то властивість розкриється, показуючи прив'язки сторін, а кнопка «+» перетвориться в «-»:

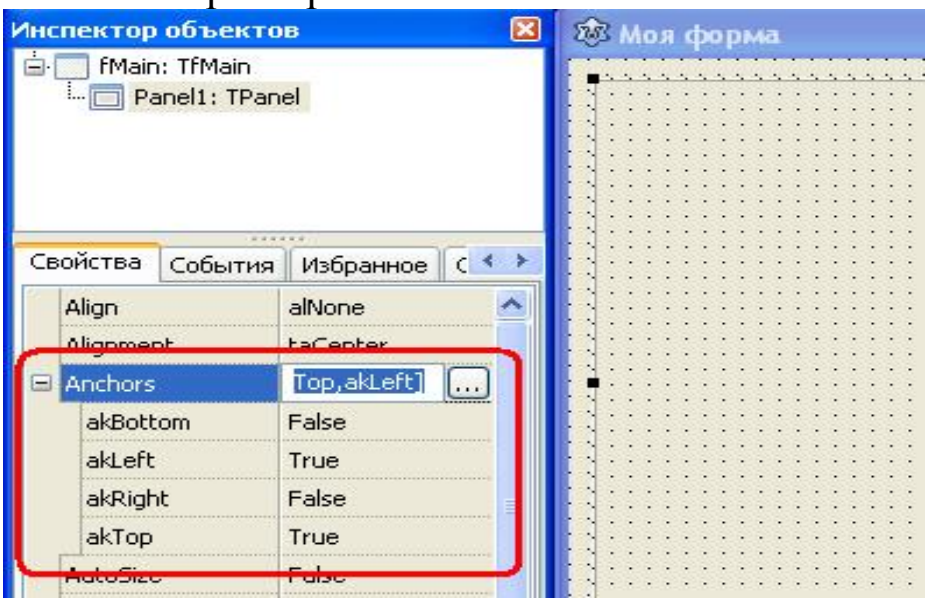

Рис. 3.6. *Розкрита властивість Anchors*

Як ми бачимо, за замовчуванням «прив'язані» тільки ліва й верхня границі (akLeft=True, akTop=True). Щоб дозволити прив'язку правої (akRight) і нижньої (akBottom) границь панелі, їх теж потрібно встановити в True. Зробіть це, запустіть програму на виконання й спробуйте змінювати розміри вікна. Тепер разом з вікном змінюються розміри й панель, що нам і було потрібно.

Однак є й інший спосіб прив'язки компонентів до різних сторін форми або контейнера, на якому компонент перебуває. Оскільки цей спосіб застосовується досить часто, розберемо його детальніше. Для початку видаліть панель - виділіть її на формі й натисніть **<Delete>**. Потім встановіть на форму нову панель. Таким чином, ми просто скинули всі зміни в налаштуваннях панелі, адже нова панель має всі значення за замовчуванням. А ім'я в неї таке ж - **Panel1**. Очистимо властивість Caption, і займемося вирівнюванням.

Align (англ. вирівнювання) - властивість, що дозволяє «приліпити» панель до однієї зі сторін зовнішнього контейнера, або до всього контейнера. Може мати наступні значення:

- alBottom - Панель займає весь низ контейнера (у нашому випадку контейнером є форма).
- alClient - Панель займає весь контейнер.
- alCustom - Користувальницьке вирівнювання, те ж саме, що alNone.
- alLeft - Панель займає всю ліву частину контейнера.
- alNone - Немає вирівнювання. Це значення за замовчуванням.
- alRight - Панель займає всю праву частину контейнера.
- alTop - Панель займає весь верх контейнера.

Якщо ми встановимо цю властивість в alClient, то наша панель займе всю форму. При зміні розмірів форми будуть змінюватися й розміри панелі. Якщо встановимо, приміром, в alTop, то панель займе всю верхню частину форми. Якщо потім на цій панелі встановимо кнопки, то одержимо **Панель інструментів**. А разом з іншим компонентом, TSplitter, що перебуває на вкладці **Additional Палітри компонентів** і являтиме собою переміщувану границю-роздільник, ми одержимо різні частини вікна, розміри якого користувач зможе міняти мишею під час виконання програми:

Рис. 3.7. *Роздільник TSplitter*

Давайте зробимо так. У нашій Panel1 установимо Align = alTop. Потім додамо на форму роздільник TSplitter, і також установимо в нього Align = alTop.

Тепер додамо на вільне місце форми ще одну панель. Очистимо в ній властивість Caption, і встановимо Align = alLeft. Додамо ще один TSplitter, переконаємося, що в нього також Align = alLeft. Тепер весь верх і всю ліву частину форми займають панелі, відгороджуючись від вільної частини сплітерами. На цю вільну частину форми ми встановимо третю панель. Очистимо в ній властивість Caption і встановимо Align = alClient. Панель зайняла всю частину форми, що залишилася. Тепер запустіть проєкт на виконання й переконайтеся, що мишею можна переміщати границі-роздільники, міняючи розміри панелей. При зміні розмірів вікна змінюються й розміри панелей:

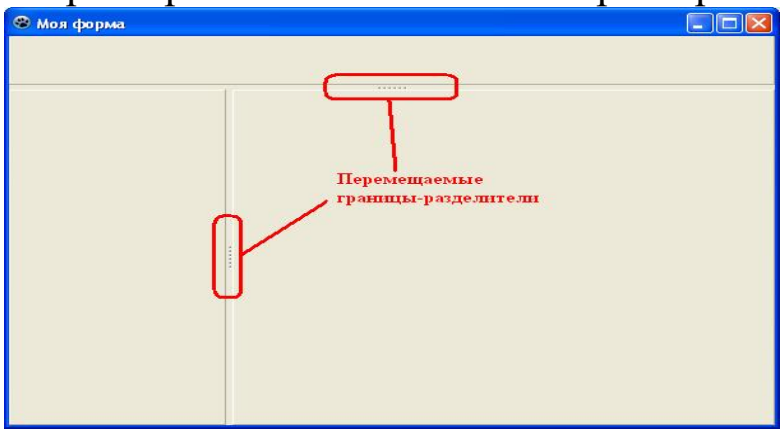

Рис. 3.8. *Три панелі й два роздільники на формі*

Не забудьте зберігати проєкт час від часу.

Ми упустили ще одну корисну властивість панелі - автоматичне настроювання розміру.

AutoSize - автоматичне настроювання розміру, логічна властивість. За замовчуванням, встановлене значення False - автоналаштування заборонене. Якщо встановити властивість в True, то автоматичне налаштування буде дозволено.

Встановіть на верхню панель кнопку **TButton**. Властивості Left і Top кнопки встановіть в 1. Тепер виділіть панель, на якій перебуває кнопка, і переведіть властивість AutoSize в True. Верхня панель як і раніше займає всю верхню частину форми, але тепер її висота зменшилася по висоті кнопки! Таким способом можна створювати прості панелі інструментів для додатка. Правда, якщо ви запустите програму на виконання, то побачите, що тепер верхнім роздільником неможливо міняти висоту верхньої панелі, адже вона в нас

автоматично приймає розмір під розташованою на ній кнопкою. Так що й потреба в першому сплітері в цьому випадку відпадає.

Кнопка TButton

Кнопки, мабуть, самий використовуваний компонент у програмах Windows. Вони всюди: у діалогових вікнах, у панелях інструментів, у повідомленнях. Познайомимося ближче із цими елементами.

Отже, проста кнопка **TButton** розміщується на вкладці **Standard Палітри компонентів**. Ми вже використали кілька разів цей компонент, і зараз у проєкті одна кнопка перебуває на верхній панелі. Виділіть кнопку, і подивимося на її властивості. З більшістю з них ви вже знайомі, познайомимося з деякими іншими.

Cancel - логічна властивість, за замовчуванням дорівнює False. Якщо дорівнює True, то натискання на кнопку **<Esc>** буде рівносильно натисканню на цю кнопку.

Default - логічна властивість, за замовчуванням дорівнює False. Якщо дорівнює True, то натискання на кнопку **<Enter>** буде рівносильно натисканню на цю кнопку. Однак якщо фокус перебуває на якій-небудь іншій кнопці, то натискання на **<Enter>** натисне її.

ModalResult - корисна в деяких випадках властивість, особливо в діалогових вікнах. Може мати значення:

- mrNone (за замовчуванням) - немає результату
- mrOk - ОК
- mrCancel - Скасування
- mrAbort - Перервати
- mrRetry - Повторити
- mrIgnore - Ігнорувати
- mrYes - Так
- mrNo - Немає
- mrAll - Всі
- mrNoToAll - Немає для всіх
- mrYesToAll - Так для всіх
- mrClose - Закрити

Будь-яка форма також має властивість ModalResult. Якщо кнопці в цій властивості привласнити одне із цих значень, і цією кнопкою закрити форму, то й у форми властивість ModalResult стане такою, як у кнопки. Припустимо, у вас є діалогове вікно із трьома кнопками: «**Так**», «**Ні**» і «**Скасування**». І кожній із цих кнопок у властивості ModalResult ви привласнили відповідне значення. Якщо кожна із цих кнопок буде закривати вікно, то при його закритті можна подивитися

властивість ModalResult форми й визначити, якою кнопкою користувач закрив вікно. І відповідно до результату, виконувати подальші дії.

Примітка. Просте присвоювання властивості ModalResult кнопці не закриє вікно, цю дію потрібно програмувати окремо.

Hint - текст спливаючої підказки. Ця підказка спливе, коли користувач підведе курсор миші до кнопки. Але підказка з'явиться, тільки якщо ShowHint = True.

Тепер будемо розбиратися з подіями. Однак якщо ви колись змініте властивість Caption кнопки, і впишите туди слово Вихід. Властивість Name залишиться за замовчуванням.

Подвійний клік по кнопці приведе до того, що **Lazarus** згенерує подію - натискання на цю кнопку. Відразу ж **Редактор коду** з'явиться нагорі, і ви побачите миготливий курсор там, де потрібно запрограмувати цю подію. Код, що ми туди впишемо, буде виконуватися щораз, коли користувач натисне на цю кнопку. Впишемо туди коротке

Close;

Саме так, із крапкою з комою наприкінці. Це оператор закриття вікна. А якщо це вікно головне, то виходить, закриється вся програма:

```
35  { TfMain }

    procedure TfMain.Button1Click(Sender: TObject);
    begin
39    Close;
40  end;

    end.
43
```

Рис. 3.9. *Програмування події натискання на кнопку*

Із процедурами ми будемо розбиратися пізніше, поки зверніть увагу на ім'я процедури: Button1Click. Button1 - це ім'я нашої кнопки, а подія, що ми програмували, називається OnClick - подія натискання на кнопку. В **Інспекторі об'єктів** перейдіть на вкладку **Події**, і знайдіть подію OnClick. Як бачите, там уже прописана саме ця процедура:

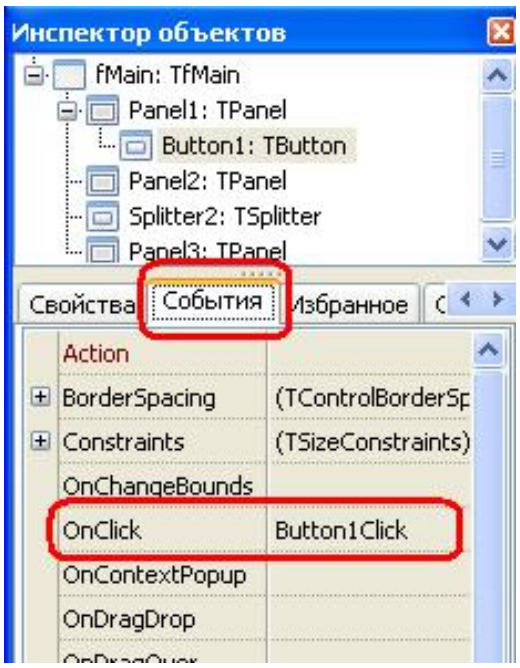

Рис. 3.10. *Подія OnClick кнопки*

Якщо ми виділимо цю подію, то праворуч від неї з'явиться кнопка «**…**»,натискання на яку приведе до генерування цієї події. Оскільки в нас вона вже запрограмована, то просто відкриється ця процедура й усередину її встановиться курсор. От вам другий спосіб генерувати потрібну подію, вибирайте, що вам зручніше. Нам не раз прийдеться перемикатися в **Інспекторі об'єктів** між вкладками **Властивості** й **Події**.

Інші події кнопок нам поки не потрібні. Збережіть проєкт і запустіть його. Переконаєтеся, що натискання на кнопку приводить до закриття програми.

Кнопкою **<F12>** поверніть на передній план **Редактор форм**, а в **Інспекторі об'єктів** поверніть вкладку **Властивості**. У **Палітрі компонентів** відкрийте вкладку **Additional**. Перші два компоненти в цій вкладці - це кнопки **TBitBtn** і **TSpeedButton**:

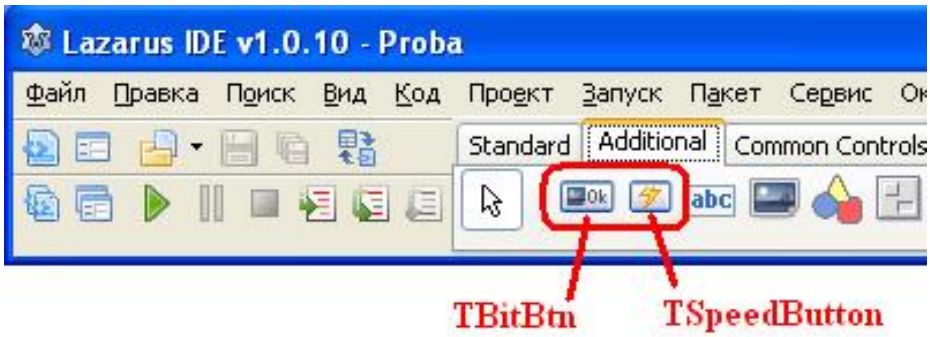

Рис. 3.11. *Кнопки TBitBtn і TSpeedBtn*

Поставте дві кнопки **TBitBtn** і одну **TSpeedButton** куди-небудь на нашу форму, наприклад, на верхню панель, правіше нашої **Button1**. Будемо розбиратися, чим вони відрізняються від звичайної кнопки.

Кнопка TBitBtn

Це більш просунута кнопка, що має деякі додаткові властивості. Насамперед, ви помітили, що вона вища, ніж звичайна кнопка. **TBitBtn** крім тексту може містити й зображення. Для цього використовуються кілька властивостей.

Glyph - Зображення. За допомогою цієї властивості можна знайти й завантажити зображення, що буде відображатися на кнопці.

Примітка: Більшість зображень зберігається в папці **Images** там, куди ви встановили Lazarus. Якщо ви залишили адресу за замовчуванням, це буде

C:\lazarus\images

Там зберігається кілька папок, де зображення розсортовані за тематиками. Однак необов'язково використовувати тільки ці зображення. В Інтернеті можна знайти величезну кількість безкоштовних колекцій зображень, які можна використовувати у своїх проєктах.

Виділітье властивість Glyph кнопки **BitBtn1** і клацніть по кнопці «**…**»праворуч від нього. Відкриється діалог вибору зображення. У цьому діалозі натисніть кнопку «**Завантажити**» і знайдіть папку

C:\lazarus\images\menu

(якщо ви вказували інший шлях при установці **Lazarus**, то вкажіть його). Відкриється список з картинками, які зберігаються в цій папці. При виборі вона буде відображатися в правій частині вікна. Знайдіть картинку **menu_exit** і натисніть кнопку «**Відкрити**», потім «**ОК**». Обране зображення з'явиться на кнопці, ліворуч від напису. До речі, щоб уже дати кнопці відповідний вигляд, змініть властивість Caption на Вихід. А заодно, згенеруйте подію натискання на цю кнопку, і впишіть код закриття, як це робили для простої кнопки. Збережіть проєкт і запустіть його; переконаєтеся, що натискання на цю кнопку також закриває програму.

Однак із цією кнопкою ми ще не закінчили. Виділіть її в **Редакторі форм** і зверніть увагу на властивості кнопки.

GlyphShowMode - політика показу або приховання зображення. Може бути:

- gsmAlways - завжди показувати зображення.
- gsmNever - ніколи не показувати зображення.

• gsmApplication - зображення буде показано під час виконання програми. Значення за замовчуванням для цієї властивості.

gsmSystem - буде показане чи ні зображення, залежить від поточних настроювань системи. Дану властивість краще не змінювати.

Layout - місце розташування зображення. Може бути:

• blGlyphBottom - зображення нижче тексту

• blGlyphLeft - зображення ліворуч від тексту (значення за замовчуванням)

• blGlyphRight - зображення праворуч від тексту

• blGlyphTop - зображення над текстом

Дані настроювання цілком залежать від конкретного стилю інтерфейсу, що ви вибрали для вашої програми. Але в більшості випадків, значення за замовчуванням - те, що потрібно.

Spacing	- відстань у пікселях від зображення до тексту. Цю властивість міняють також рідко.
TabOrder	- порядковий номер виділення компонентів. Є в багатьох візуальних компонентів. Визначає, у якому порядку компонента будуть одержувати фокус введення при натисканні клавіші <**Tab**>. Нумерація починається з 0, установити однаковий номер для двох компонентів неможливо, якщо вони перебувають на одному контейнері.
Enabled	- доступність компонента. Також є присутнім у багатьох візуальних компонентів. Логічна властивість. Якщо True, то компонент доступний (кнопку можна натискати), якщо False, те недоступний (кнопка сірих кольорів, нажати на неї неможливо). Змініть цю властивість, запустіть програму й подивіться, як змінилася кнопка. Потім поверніть значення за замовчуванням.
Visible	- видимість компонента. Якщо True, то компонент видимий, інакше компонент захований. «Пограйте» і із цією властивістю, не забудьте повернути її наприкінці в значення за замовчуванням.

Ми не даремно встановили на форму дві кнопки **TBitBtn**. Для вивчення наступної властивості виділіть BitBtn2.

Kind - різновид кнопки. При виборі якогось значення на кнопці з’являється відповідний текст і картинка, а кнопка відразу має потрібні

властивості, що залежать від ОС - закриває вікно (у випадку якщо Kind = bkClose), привласнює формі потрібне значення ModalResult. Правда, у працюючій програмі (принаймні, для Windows XP Professional SP3) текст на кнопці буде англійською мовою. Може бути:

- **bkAbort** - Перервати
- **bkAll** - Всі
- **bkCancel** - Скасування
- **bkClose** - Закрити
- **bkCustom** - Користувальницькі настроювання - текст і напис встановлює програміст. Це значення за замовчуванням.
- **bkHelp** - Довідка
- **bkIgnore** - Пропустити
- **bkNo** - Немає
- **bkNoToAll** - Немає для всіх
- **bkOK** - ОК
- **bkRetry** - Повтор
- **bkYes** - Так
- **bkYesToAll** - Так для всіх

Встановіть Kind = bkClose, збережіть проєкт і запустіть його. Переконаєтеся, що тепер і кнопка **BitBtn2** закриває вікно. Для цього нам навіть не довелося її програмувати!

Кнопка TSpeedButton

Наостанок попрацюємо із цим типом кнопок. Вони найчастіше застосовуються для організації панелей інструментів. Справа в тому, що ця кнопка не має фокуса введення! Що я маю на увазі? Запустіть програму на виконання й клавішею **<Tab>** переміщайтеся по кнопках. Ви побачите, що можна виділити будь-яку кнопку, крім **SpeedButton1**. На таку кнопку можна нажати тільки мишею, при цьому, якщо вікно не закриється, фокус повернеться до того компонента, що був виділений раніше.

Як бачите, кнопка квадратна, має малий розмір і розрахована тільки на те, щоб за допомогою властивості Glyph в неї завантажили зображення. Однак вона має такої ж властивості, як TBitBtn: Caption, Layout, Space і т.д., тому в даній кнопці також можна використати текст разом із зображенням, або тільки текст, або тільки зображення.

Оскільки кнопка не має фокуса введення, властивості TabOrder у неї також немає. Відсутня і властивість Kind. Але в іншому вона схожа на **TBitBtn**.

За допомогою властивості Glyph завантажте картинку **menu_exit**, як у минулому прикладі. Однак програмувати подію OnClick ми не будемо - навіщо писати той самий код? Можна спрацюватии простіше. Перейдіть на вкладку «**Події**» **Інспектори об'єктів**, і, не натискаючи в події **OnClick** на кнопку «**…**»,відкрийте доступний список:

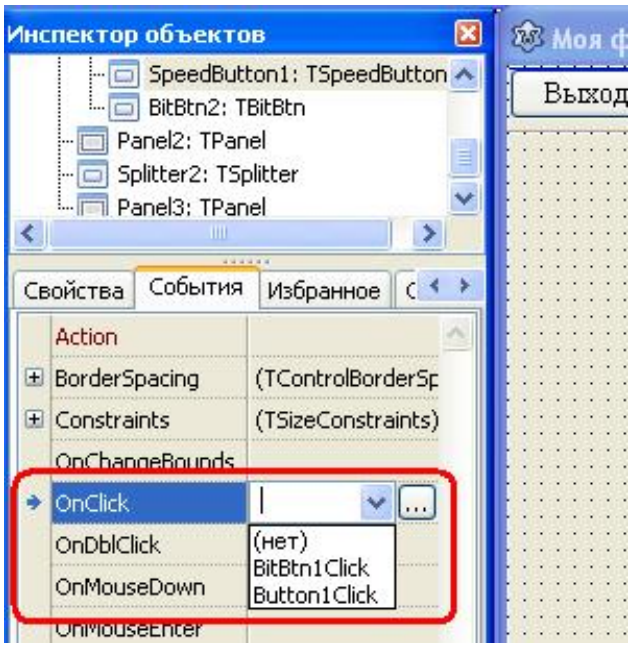

Рис. 3.12. *Вибір події OnClick*

Виберіть у списку Button1Click, збережіть проєкт і запустіть його на виконання. Переконаєтеся, що наша SpeedButton1 закриває програму - це спрацьовує подія першої кнопки.

Наостанок виправимо один недолік. Якщо ви звернули увагу, у нас на формі 4 кнопки, але при наведенні курсора на кожну з них спливає та сама підказка, що ми писали у властивості Hint форми: «Наша спливаюча підказка». А це не дуже добре. Потрібно у властивості Hint кнопок написати відповідні тексти, наприклад:

- Кнопка Button1
- Кнопка BitBtn1
- Кнопка BitBtn2
- Кнопка SpeedButton

Звичайно, у реальних додатках у властивості Hint ви будете писати необхідний текст підказки, наприклад, «Вихід із програми». Але в нас всі кнопки виконують вихід із програми, і щоб відрізнити їх олдин від одного, ми вкажемо відповідний текст. Збережіть проєкт і запустіть його на виконання; переконаєтеся, що текст спливаючих підказок міняється.

Порада. Текст у властивості Hint форми нам, загалом кажучи, і не потрібний, його можна очистити. Однак щоб одержати доступ до властивостей форми, нам потрібно виділити саму форму, а це тепер проблемно, адже її повністю перекривають три панелі! Виділити форму можна або в **Дереві об'єктів** у верхній частині **Інспектора об'єктів**, або виділивши яку-небудь панель, і натиснувши клавішу **<Esc>**. При цьому виділення перейде на зовнішній компонент, тобто, на форму.

Тема 4. Основи коду

1. Типи даних

Напевно, ви розумієте, що вміння користуватися компонентами - це ще не програмування? Розробка користувальницького інтерфейсу - настроювання форми, розміщення на ній компонентів, організація меню, додання всьому привабливого виду й зручності використання - все це, скоріше, дизайн. Програмування - це знання мови, уміння писати вихідний код. Цим ми сьогодні й займемося.

Будь-які дані, якими доводиться оперувати програмістові, мають свій тип. Паскаль, що лежить в основі багатьох мов і середовищ розробки, у тому числі й в основі Lazarus, відрізняється більше жорсткими правилами використання даних різного типу. Зокрема, у Паскалі будь-яку змінну будь-якого типу потрібно попередньо **оголосити** - вказати ім'я змінної і її тип, привласнити початкові значення (у багатьох інших мовах змінна оголошується прямо там, де вона була потрібна, а в деяких мовах обходяться й зовсім без оголошення й вказівки типу). Все це привчає програміста до більш якісного стилю програмування, робить програму більш зручною для читання й зрозумілою компіляторові.

У Паскалі існує всього кілька стандартних типів, з якими може працювати програміст:

- Цілі числа (числа без коми: 5; -12; 0)
- Дійсні числа (числа з комою: 5,3; -3,14; 0,0)
- Символи (окремі букви або інші символи: «Б», «z», «&», «/»)
- Рядки (групи символів: «Це рядок», «This is a text»)
- Логічний тип (може приймати значення або Хиба - False, або Істина - True)

Lazarus заснований на Object Pascal, що має безліч додаткових типів даних. Наприклад, тип Дата-Час, колекції рядків, файли, записи,

класи й так далі. З багатьма типами ми будемо знайомити при вивченні курсу, у порядку зростання складності матеріалу.

2. Елементи програми

Програма мовою Object Pascal може містити наступні стандартні елементи:

- Службові (зарезервовані) слова
- Ідентифікатори
- Типи
- Змінні
- Константи
- Коментарі
- Мітки
- Підпрограми

Службові (зарезервовані) слова - це такі слова, які є частиною мови й застосовуються для позначення різних інструкцій або елементів програми. Приклади службових слів: as, class, if, for, var, array, unit, interface, uses.

Службові слова в **Редакторі коду** завжди виділяються **жирним шрифтом**. Програміст не може використовувати службові слова як ідентифікатори: для позначення змінних, констант, записів, користувальницьких констант і т.п.

Ідентифікатори - це слова (імена), якими програміст позначає будь-який інший елемент мови, крім службових слів, коментарів і властивостей, ідентифікаторів. Наприклад, коли ми даємо ім’я змінній або константі, ми **ідентифікуємо** її.

Правило. Для ідентифікаторів дозволено використовувати тільки символи латинського алфавіту, цифри й знак підкреслення. Причому першим символом ідентифікатора не може бути цифра. Рядкові й заголовні символи не розрізняються, тобто myvar, MyVar і MYVAR - це той самий ідентифікатор.

Приклади правильних ідентифікаторів:

- MyVar
- _perem
- a2
- MinimalnayaZarplata

Приклади неправильних ідентифікаторів:

- Зарплата (не латинські символи)
- 2a (перший символ - цифра)
- My perem (в ідентифікаторі недозволений символ - пробіл)

Типи - це спеціальні конструкції мови, що вказують компіляторові, скільки пам’яті потрібно зарезервувати для оголошуваного елемента, наприклад, змінній, масиву, константи. Типами можуть бути стандартні типи Паскалю, розширені типи Об’єктного Паскалю або типи даних, оголошені користувачем. Ми будемо знайомити з різними типами даних протягом усього курсу.

Приклади типів: integer (ціле число), real (число з коми), string (рядок) і т.п.

3. Змінні

Змінні є елементом мови, що використовується повсюдно. Нам неодноразово буде потрібно використовувати змінні різних типів, різного призначення, різної області видимості. Тому даний розділ варто розглянути детальніше.

Всі дані, з якими працює програма, розташовуються в **ОЗУ** (Оперативний Запам’ятовувальний Пристрій, Оперативна пам’ять) комп’ютера. Коли користувач завантажує програму, то в оперативну пам’ять зчитуються всі інструкції програми й дані, які в ній використовуються. Дані розташовуються в пам’яті за різними адресами, які фактично, являють собою цифри. Так, перший байт пам’яті розташовується під номером 1, другий під номером два й т.д. Однак не всіх дані займають тільки один байт, є й двобайтні, і чотирибайтові дані, і навіть більше. Саме тому так важливо вказувати тип даних - компілятор повинен знати, скільки байт у пам’яті буде займати ваш елемент, скільки місця йому потрібно виділити. Припустимо, нам потрібний якийсь елемент даних в 2 байти, і комп’ютер помістить його в ОЗУ за адресою, наприклад, 1000000. Виходить, коли буде потрібно довідатися вміст цього елемента, комп’ютер повинен звернутися до комірки пам’яті під номером 1000000, і взяти відтіля два байти.

У процесі роботи програми ці дані можуть мінятися. Тому нам потрібний якийсь контейнер, скринька певного розміру, де ми будемо зберігати дані. Ці дані будуть мінятися, а адреса скриньки в пам’яті залишиться незмінною, поки працює наша програма. Така скринька і називається **змінною** - елементом, чиї дані можуть змінюватися. Отже,

Змінна - це комірка оперативної пам’яті, що займає відповідному її типу кількість байт. Дані в цій комірці можуть змінюватися, а адреса розміщення в ОЗУ залишається незмінною, поки працює програма.

Щоб програміст не мучився з адресацією цих комірок, в Паскалі змінним привласнюється ім’я - ідентифікатор. Коли ми повідомляємо

змінну з певним ім’ям, комп’ютер запам’ятовує це ім’я й асоціює його з реальною адресою в пам’яті, і кількістю займаних змінною байт. А коли ми зчитуємо дані змінної MyPerem, то комп’ютер сам знаходить потрібну ділянку пам’яті й зчитує відтіля необхідну кількість байт.

Змінні оголошуються в розділі var (англ. variable - змінна). Оголошення має наступний синтаксис (конструкцію):

```
var
  Ім'я_перем_1: Тип_1;
  Ім'я_перем_2: Тип_2;
  ...
  Ім'я_перем_n: Тип_n;
```

Тобто, в оголошенні ми вказуємо ім’я створюваної змінної, потім символ «двокрапка», а потім тип даних цієї змінної. Якщо ж нам потрібно оголосити декілька змінних одного типу, то ми можемо перелічити їх в одному операторі через кому:

```
var
  перем_1, перем_2,... перем_n: Тип;
```

Оголошення змінних у різних рядках - це неправильно, весь опис потрібно вмістити на одному рядку:

```
var перем_1: Тип_1; перем_2: Тип_2;
```

Але такий код буде складно читати, він важкий для сприйняття. Тому попередні приклади все-таки кращі, коли кожнен оператор описується на своєму рядку, програма стає більше «читабельною».

Порада. Давайте змінним і іншим елементам і конструкціям, які ви описуєте, **осмислені імена**, а не просто незрозумілі набори символів!

Наприклад, якщо ми хочемо зберігати в змінній мінімальну заробітну плату, то розумно назвати таку змінну MinZarPlat, для змінної з вашим ім’ям - MyName або MoeImya - залежить від вашої фантазії, аби тільки вам самим було зрозуміло, для чого ви взагалі створювали цю змінну. Адже це тільки спочатку ви будете пам’ятати, що за змінні ви використовуєте. А якщо ви відкриєте свій проєкт через рік? Або його відкриє ваш друг або колега? Що зберігається в змінній? Для чого створена змінна d3? Такі незрозумілі ідентифікатори

ускладнюють сприйняття програми й категорично не рекомендуються до використання.

Виключення можуть скласти змінні, які звичайно створюються як лічильники для циклів, наприклад, або для якихось коефіцієнтів. Такі змінні традиційно називають i, k, l (або ін. символами англійської мови).

Давайте спробуємо використати змінні на практиці. Відкриємо другий проєкт із другої лекції, той, де користувач вписує своє ім'я, а програма виводить йому вітання. Якщо ви додержувалися моїх рекомендацій, у вас цей проєкт повинен зберігатися за адресою:

C:\Education\02-02

Завантажите файл проєкту **Hello.lpi** - це приведе до завантаження **Lazarus** і відкриттю в ньому даного проєкту. На передньому плані повинен виявитися **Редактор коду** (якщо це не так, натисніть **<F12>**, щоб перемкнутися з форми на код). Ви можете відразу помітити, що одна змінна в нас уже оголошена:

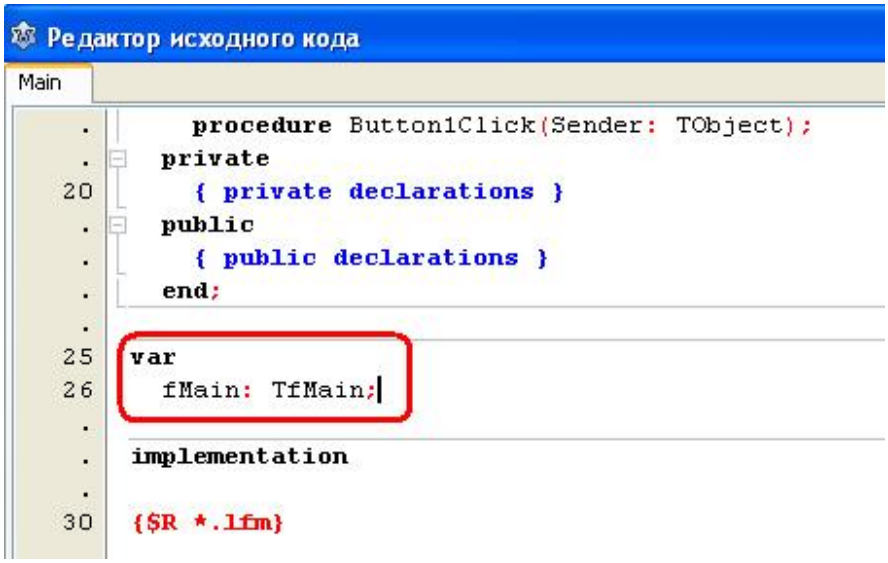

Рис. 4.1. *Змінна fMain*

Цікаво: у нас є форма **fMain**, так ще й змінна з таким же ім'ям! Справа в тому, що компілятор FPC розглядає нашу форму, як змінну. Звертання до цієї змінної - це звертання до самої форми.

Тепер опустимо курсор до обробки події натискання на кнопку **Button1** і трішки змінимо код цієї події:

```
25  var
      fMain: TfMain;

    implementation

30  {$R *.lfm}

    { TfMain }

    procedure TfMain.Button1Click(Sender: TObject);
35  var
      MyName: String;
    begin
      MyName:= 'Привет, ' + Edit1.Text + '!';
      ShowMessage(MyName);
40    MyName:= 'Рады вас видеть, ' + Edit1.Text + '!';
41    ShowMessage(MyName);
    end;

    end.
45
```

Рис. 4.2. *Новий код події OnClick кнопки Button1*

Надалі, щоб не захаращувати лекції зайвими ілюстраціями, я буду вказувати вихідний код простим текстом з похилим шрифтом, наприклад, так:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  MyName: String;
begin
  MyName:= ‘Привіт, ‘ + Edit1.Text + ‘!’;
  ShowMessage(MyName);
  MyName:= ‘Ради вас бачити, ‘ + Edit1.Text + ‘!’;
  ShowMessage(MyName);
end;
```

Як бачите, тут ми використали змінну MyName з типом String (рядок). Ми двічі привласнювали їй різні значення, а потім зчитували їх, щоб вивести в повідомленні. Де саме потрібно повідомляти змінні, ми поговоримо в лекции №8 (Підпрограми), там же обговоримо таке важливе поняття, як область видимості змінної. Поки лише врахуйте, що при використанні подій, змінні потрібно повідомляти **до** службового слова begin, а використати саму змінну потрібно **після** цього службового слова.

4. Оператор присвоювання значення

На прикладі вищенаведеного коду ми могли помітити, що значення змінним привласнюються за допомогою оператора присвоювання

:=

Тобто, двокрапка й знак «дорівнює», що випливає відразу за ним, без пробілів. Присвоєння відбувається ліворуч: спочатку обчислюється значення, зазначене ПІСЛЯ цього оператора, потім воно привласнюється змінній, зазначеній ДО оператора присвоювання. Наприклад:

```
var
  i: integer;  //оголосили змінну з типом integer - ціле число
begin
  i:= 3;  //привласнили цієї змінної значення 3
```

Значення, що ми привласнюємо змінній, може бути й складним. Ми можемо вказати не просто значення, а цілий вираз, причому використати в ньому попереднє значення самої змінної! Гляньте на приклад:

```
var
  i: integer;
begin
  i:= 3;
  i:= 2 * 5 + 36 * i;  //складний вираз
```

Отут ми оголосили цілу змінну *i*. Цій змінній ми привласнили значення 3. Потім ми привласнили нове значення, що спочатку вираховується комп'ютером зі складного виразу, а потім тільки привласнюється змінній. Яке ж значення тепер в i? Давайте рахувати. З уроків математики ми знаємо, що спочатку обчислюються такі оператори, як множення й ділення, а тільки потім - додавання й віднімання. Тобто, спочатку рахуємо 2 * 5 = 10 (для комп'ютера знак «*» - це множення, якщо хтось не знає). Потім обчислюється 36 * i, а i - це 3, ми робили це присвоювання рядком вище. Так що одержуємо 36 * 3 = 108. Потім складаємо ці значення: 10 + 108 = 118. Саме стільки й потрапить, зрештою, у змінну *i*.

Тут у когось може виникнути питання: а чому потрібно давати змінним осмислені імена, а тут призначено просте ім'я *i*? Питання справедливе, відповім так. Це - не реальна програма з безліччю підпрограм і змінних, це простий приклад з однією змінною, і ми в жодному разі, не заплутаємося. Крім того, змінна була призначена не для якоїсь конкретної мети, а лише для демонстрації присвоювання їй значення. Тому й ім'я в змінної просте. У більш складних прикладах і в лабораторних роботах постараємося дотримуватися своїх же рекомендацій.

Якщо ви помітили, то в нашій програмі ми також привласнювали змінній не просто значення, а складений вираз:

```
MyName:= 'Привіт, ' + Edit1.Text + '!';
```

Тут ми рядок «Привіт, « з'єднували з тим, що зберігалося в полі введення Edit1, додавали знак оклику, а результат уже привласнювали змінній MyName. Зверніть увагу, що після слів «Привіт» ми вписали не просто кому, а кому з наступним пробілом. Якби ми не вказали цей пробіл, то в результаті текст зливався б, а це не дуже красиво:

```
Привіт,Дмитро!
```

4. Константи

Константи використаються рідше, ніж змінні, однак у деяких випадках вони можуть бути дуже корисними.

У математиці **константа** (лат. constanta - постійна, незмінна) - деяка величина, що не змінює своє значення в рамках розглянутого процесу. У програмуванні константа має те ж значення. Причому константи бувають двох типів: **прості** неіменовані) та **іменовані**.

Проста константа являє собою готове значення, наприклад,

```
i:= 3; //тут 3 - це числова константа
s:= 'рядок'; // 'рядок' - строкова константа
```

Звичайно, число 3 завжди й скрізь буде дорівнює 3, тому дане значення - константа. Такі прості константи ми застосовуємо повсюдно, не особливо замислюючись.

Іменована константа - трохи більш складне поняття. Фактично, це такий же комірка оперативної пам'яті, як і змінна, але значення в цій комірці не можуть мінятися. Застосування таких констант корисно у двох випадках:

1. Коли не хочеться запам’ятовувати реального значення, або коли воно занадто довге. Наприклад, число ПІ дорівнює 3,141 592 653 589 793 238 462 643 383 279 502 88. Хочеться вам запам’ятовувати таке значення? Чи не простіше визначити константу *pi* і записати в неї це число? Тоді надалі, замість вказівки цього числа ми будемо вказувати константу *pi*. Компілятор сам підставить потрібне значення замість імені цієї константи.
2. Коли якесь значення використовується в програмі без змін, але згодом воно може стати іншим. Наприклад, мінімальна зарплата (МРОП - Мінімальний Розмір Оплати Праці) на 2013 рік прийнята в розмірі 943 грн. Наша програма, наприклад, обраховує зарплату співробітників, для цієї МРОП може множитися на різні коефіцієнти, наприклад, коефіцієнт вислуги років. Оскільки таке обчислення буде багаторазовим - для кожного співробітника окремо, так ще з різними умовами (відпустка, лікарняний, премія й т.п.), то розумно оголосити константу mrop зі значенням 943, і надалі використовувати її замість реального значення. Чому такий підхід зручний? По-перше, наша програма може потрапити в інший регіон, а там може виявитися свій розмір МРОП. По-друге, на 2014 рік розмір МРОП напевно збільшиться, а виходить, його значення зміниться. В обох цих випадках досить буде змінити значення константи на початку модуля, і у всіх місцях коду, де ця константа застосовується, автоматично буде використано нове значення. Нам не доведеться копітко вишукувати по всьому коді, де ж ми використали при розрахунках розмір МРОП, щоб вписати туди нове значення.

Отже, правило:

Константа - область оперативної пам’яті, значення в якій залишається незмінним, поки програма працює.

Варто відзначити, що коли програма завершує свою роботу, всі комірки, виділені під змінні й під константи, очищаються. Вірніше, перестають враховуватися операційною системою. Тобто, якщо інша програма захоче їх використати, Windows це дозволить. Але доти, поки ця програма не запише в комірки вже свої значення, там буде «сміття» - набір незрозумілих даних. От чому дуже важливо не тільки повідомляти змінні (константи), але й привласнювати їм початкові значення.

Константи оголошуються в розділі const (англ. constants - постійні, константи), причому цей розділ повинен розташовуватися **до** розділу

var. Оголошення константи й присвоєння їй значення на відміну від змінних, відбувається одночасно:

```
const
  mrot = 5205;
```

Тут ми створили розділ констант, у якому оголосили константу mrop, і відразу ж привласнили їй значення 943. Як бачите, замість оператора присвоювання «:=«, як це було в змінних, тут використовується простий знак «дорівнює», а тип константи не вказується. Компілятор сам, залежно від зазначеного значення, привласнює константі найбільш зручніший тип.

Повернемося до нашого прикладу, і знову доробимо код події натискання на кнопку:

```
procedure TfMain.Button1Click(Sender: TObject);
const
  priv = 'Привіт, ';
var
  MyName: String;
begin
  MyName:= priv + Edit1.Text + '!';
  ShowMessage(MyName);
  MyName:= 'Ради вас бачити, ' + Edit1.Text + '!';
  ShowMessage(MyName);
end;
```

У цьому прикладі ми оголосили константу priv і привласнили їй текст - початок вітання. Причому розділ const ми помістили до розділу var. У рядку

```
MyName:= priv + Edit1.Text + '!';
```

компілятор замість «priv» підставить значення «Привіт, « і ми одержимо колишній текст вітання.

5. Коментарі

Коли коду дуже багато, програміст легко може в ньому заплутатися. Щоб цього не відбулося, у Паскалі передбачені **коментарі**.

Коментар - це пояснювальний текст, що додається програмістом до вихідного коду й ігнорується компілятором.

Тобто, програміст сам собі залишає коментарі> - замітки, що вказують, що відбувається в даній ділянці коду, для чого створений даний елемент, що буде відбуватися далі й т.п. При зборці програми компілятор FPC ігнорує весь цей текст і не включає його в програму. Коментар залишається лише у вихідних кодах.

Коментарі поліпшують сприйняття коду програмістом, а якщо проєкт розробляється не одним програмістом, а групою, отут без коментарів і зовсім не обійтися. Говорять, у компанії Microsoft, коли приймають на роботу нового програміста, дивляться на стиль його коду. І якщо там коментарів менше, ніж 1/3 від всієї програми, йому відмовляють у вакансії. Це й зрозуміло - якщо хтось інший буде підключати ваш модуль до загальної програми, без коментарів йому буде важко розібратися, що у вас до чого. Навіть якщо ви самі повернетеся до вашої програми через якийсь час, без коментарів вам буде важко згадати, де й що у вас зберігається, буває простіше написати таку програму заново. Тому використання коментарів обов'язково для кожного програміста, що поважає свій власний час, і час колег.

Коментарі бувають однорядковими й багаторядковими.

Однорядковий коментар починається із символів «//» і може розташовуватися як на окремому рядку, так і **після** діючого оператора. Приклади ми вже бачили:

```
i:= 3; //привласнили цієї змінної значення 3
```

Тут текст після «//» - однорядковий коментар. Ми могли б розташувати його й на окремому рядку:

```
//привласнюємо змінної нове значення:
i:= 10;
```

Майте на увазі, що якщо ви використаєте якийсь оператор **після** однорядкового коментаря на тім же рядку, для FPC це буде вважатися продовженням коментаря, і оператор **не буде виконаний**:

```
//привласнюємо змінній нове значення:   i:= 10;
```

Іноді однорядкові коментарі застосовуються для візуального відділення однієї значеннєвої частини коду від іншої:

```
//**********************************************…який
сь код//**********************************************
```

Або так:

//-і-і-і-і-і ПОЧАТОК БЛОКУ КОДУ -і-і-і-і-і-і-і-і-і-і-і-і-і-
…якийсь код//-і-і-і-і-і КІНЕЦЬ БЛОКУ КОДУ -і-і-і-і-і-і-і-і-і-і-і-і-і-

Загалом, для поліпшення читабельності програми, після «//» ви можете використати власні роздільники.

Багаторядковий коментар. Коли коментар містить багато пояснювального тексту, його роблять багаторядковим. Для цього його поміщають у фігурні дужки «{...}». Усе, що перебуває усередині таких дужок, вважається коментарем:

Також в Паскалі замість фігурних дужок для багаторядкового коментаря можна застосовувати символи коментарів мови З: «(*...*)»:

(*Приклад багаторядкового коментаря у стилі З/З++*)

Які коментарі застосовувати - справа смаку. Їх можна й комбінувати між собою, у серйозних проєктах часто так і роблять.

Порада: коментарі - це добре, однак не перестарайтеся й не давайте зовсім вже очевидних коментарів.

Вище були наведені такі очевидні коментарі, але там це було необхідно, щоб пояснити нову тему:

```
var
  i: integer;  //оголосили змінну з типом integer - ціле число
begin
  i:= 3;  //привласнили цієї змінної значення 3
```

Тепер ви й без коментарів зможете розібратися в коді цього прикладу, а виходить, коментарі тут можна й не давати.

Тема 5. Символи й рядки

1. Поняття «символ» і «рядок»

Для рядового користувача ці поняття досить абстрактні. Коли він вводиться з клавіатури «А», то вважається це символом, літерою. Коли вводиться «4», то це - цифра. А про всякі там пробіли, розділові знаки або арифметичні знаки він і зовсім не думає. Але програміст повинен розуміти, як всі ці знаки сприймаються комп'ютером. Тому наберіться терпіння, ми вивчимо ці поняття досить докладно.

Насправді, все, що ми вводимо із клавіатури - це символи. Літера «z», цифра «3», пробіл, знак помножити, знак відсотка й т.д. - все це символи. Комп'ютер же може оперувати тільки цифрами, причому двійковими - такими, які містять лише 0 або 1. Всі ці літери, десяткові

цифри та інші знаки для нього не значать рівно, нічого. І для того, щоб ми могли якось обробляти текст, цифри й іншу інформацію, потрібно було вигадати спеціальну систему для перекладу інформації в зрозумілу комп'ютеру, і назад. Так з'явилися **кодові сторінки**.

Кодова сторінка (англ. code page) - спеціальна таблиця, що зіставляє кожному значенню байта деякий символ.

Не дуже зрозуміло? Давайте розбиратися. Ми знаємо, що інформація виміряється байтами, і що в одному байті 8 біт. Біт - це мінімальна одиниця інформації, з якою може працювати комп'ютер. У біті може зберігатися або 0, або 1.

На зорі розвитку комп'ютерів була розроблена кодова сторінка **ASCII** (англ. American Standard Code for Information Interchange - Американський кодовий стандарт для обміну інформацією). Перша версія цього стандарту з'явилася в 1963 р. Ця сторінка містила 7-ми бітні символи, у кожному байті один біт був не задіяний. Мінімальне двійкове число, що могло зберігатися в 7-ми бітах - це нуль. Максимальне - 1111111.

Натисніть кнопку «**Пуск**», виберіть команду «**Виконати**» і у вікні «**Запуск програми**» вкажіть

```
calc
```

і натисніть **<Enter>**. Завантажиться стандартний калькулятор Windows. У головному меню програми, у розділі «**Вид**» виберіть «**Інженерний**». У лівій частині калькулятора, нижче поля введення чисел, ви побачите перемикачі різних систем вирахування:

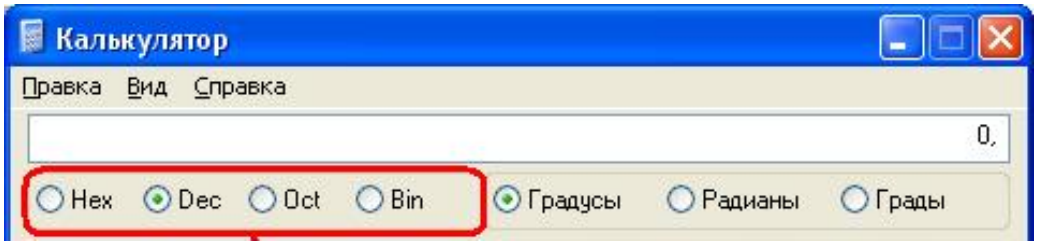

Рис. 5.1. *Перемикачі калькулятора*

Тут ми маємо можливість перемикатися на чотири системи вирахування:

- **Hex** - 16-кова. Використовується в основному, у мові Асемблер або при налагодженні програм.
- **Dec** - 10-кова. Наша звичайна система, використовується за замовчуванням.

- **Oct** - 8-кова. Майже не використовується.
- **Bin** - 2-кова. Хоч вона і є для комп'ютера єдино зрозумілою, але внаслідок громіздкості записів, у програмуванні її майже не застосовують.

Давайте подивимося, скільки символів могло втримуватися в кодовій сторінці ASCII. Перемкніться на двійкову систему (Bin), введіть 7 одиниць, потім перемкніться назад на десяткову систему (Dec). У нас вийшло 127. Саме стільки символів утримувалося в першій ASCII таблиці. Крім латинських літер, таблиця містила й інші символи - цифри, арифметичні знаки, розділові знаки, символи пробілу, дужки й т.п. Кожному символу відповідав власний номер у таблиці. Якщо ми вводили англійську літеру «A», то в комп'ютер попадав номер цього символу в таблиці - 65. Або, у двійковому вигляді, 100 0001. Таким чином, символи можна було порівнювати між собою. Англійська літера «B» перебувала під номером 66 і, отже, була більшою, ніж «A». Малі літери мали інші номери, наприклад, «a» ішла в таблиці під номером 97 і вважалася більшої, ніж «A». Ми вводимо символи, які автоматично перетворюються у цифри, з якими вже оперує комп'ютер.

Отут потрібно зробити одне важливе пояснення. Якщо ми вводили число «65», то для ПК це не було числом 65, або латинською літерою «A», це було двома символами «6» і «5». Символу «6» відповідає номер 54 кодової таблиці, а символу «5» - номер 53. Таким чином, числа, які ми вводимо в ПК, насправді не числа, а текстові символи! Перетворення таких символів у числа й назад звичайно виконується програмою автоматично. Такі перетворення, наприклад, постійно виконує стандартний калькулятор Windows. А коли ми будемо вивчати числа, нам самим потрібно виконувати такі ж перетворення.

Усе б добре, але крім англійської мови, у світі існує безліч інших мов! Ми, наприклад, пишемо кирилицею. 127 символів було явно недостатньо, щоб можна було вводити текст і на інших мовах. Тому таблиця ASCII розвивалася, рік у рік з'являлися нові стандарти. Кожен символ став уже 8-ми бітним. Подивіться на калькуляторі - вісім біт можуть містити максимальне число 1111 1111, при перекладі в десяткову систему ми одержимо 255 символів. Перша половина таблиці залишалася незмінною, зате другу половину таблиці можна було задіяти для символів інших мов і псевдографіки, за допомогою якої програмісти часів MS-DOS малювали віконця, панельки, таблиці й меню. Однак і цього було занадто мало, щоб закодувати символи всіх

мов на Землі. Для кожної мови доводилося розробляти свій стандарт, несумісний з іншими. Причому, для однієї мови могло бути розроблене кілька стандартів! У хаосі стандартів треба було якось розбиратися, удосконалювати їх.

На зміну ASCII прийшло кодування **ANSI** (англ. American National Standards Institute - Американський Національний Інститут Стандартів). Так, в MS Windows кодова сторінка ANSI, що містить кирилицю - це Windows-1251 (або CP1251), що з'явилася в 1990-1991 р.

Однак і цього було недостатньо, адже для кожної мови як і раніше було потрібне власне кодування, а мов на Землі багато. Назріла необхідність переходити до «широких» стандартів, у яких символ займає більше одного байта. Так, в 1991 р. був запропонований стандарт **Юнікод** (англ. Unicode) - універсальна система кодування символів, що вміщує знаки практично всіх мов. У цьому стандарті в одному документі можна використати символи кирилиці, китайські або японські ієрогліфи, знаки математичних формул, музичні знаки й т.п.

Перша версія Юникоду мала фіксований розмір символів 2 байти (16 біт). В одному кодуванні вже можна було використати 65 535 символів! Однак виявилося, що й цього недостатньо. Юникод одержав подальший розвиток, і рік у рік стали з'являтися нові версії й стандарти, засновані на Юникоді. Є такі стандарти, як **UTF-8** (англ. Unicode Transformation Format, 8 bit - 8-ми бітний формат перетворення Юникоду), **UTF-16**, **UTF-32**.

В Lazarus, в основному, використовується формат UTF-8, так що розберемо його. UTF-8 з'явився 2 вересня 1992 року. Основна його відмінність від первісного Unicode у тім, що в UTF-8 символи мають **не фіксований** розмір! Якщо використовувались символи з номером меншим, ніж 128, то вони займають 1 байт, як звичайний ASCII-текст. Символи з номером від 128 і більше можуть займати від 2 до 6 байт (реально максимальний розмір символу - 4 байти, тому що в Юникоді немає символів з більшим номером). Символи кирилиці займають, наприклад, по 2 байти. Так що ланцюжок символів в 5 байт в UTF-8 не завжди означає рядок з п'яти символів.

Такий підхід робить UTF-8 самим економічним кодуванням для сумісності зі старими стандартами, однак є й мінуси. На жаль, для всіх не англомовних користувачів Windows в Lazarus прийдеться зіштовхнутися з деякими проблемами застосування різних кодувань: в

Lazarus використовується кодування UTF-8, ОС Windows використовує UTF-16, а в консольних додатках Windows використовується системне кодування CP866 (тобто, стандарт ANSI). Його ж використовують деякі функції компілятора FPC. Так що в деяких випадках нам доведеться користуватися функціями перетворення кодувань, наприклад, UTF8ToConsole(), CP866ToUTF8() і т.п. Ми познайомимося з ними пізніше, у свій час.

Отже, підведемо деякі підсумки.

- **Символ** - це графічне зображення літери, цифри, арифметичного знака, розділового знака або якого-небудь іншого знака, що відповідає якому-небудь стандарту кодування символів.
- Кожному символу відповідає його номер у кодовій таблиці символів.
- Кодувань (code page) існує безліч.
- **Рядок** - це ланцюжок символів.
- Цифри, які ми набираємо на клавіатурі - це символи. Щоб обробляти їх, як цифри, програма виконує автоматичне перетворення із символів у цифри, і назад, коли виводить нам результати обчислень.
- **UTF-8** - це одне з подань Юникоду, що використовується в Lazarus.
- В одному рядку можуть зустрічатися символи, які займають 1, 2 або навіть більше байт.

2. Символьні типи даних

В Lazarus є символьний тип Char. Змінна такого типу займає рівно один байт пам'яті й може містити будь-який ASCII - символ. Українській мові із цим типом не пощастило - оскільки символи кирилиці займають по 2 байти, використовувати цей тип з українськими літерами не можна. Давайте відкриємо **Lazarus** з новим проєктом. Якщо Lazarus у вас уже завантажений, закрийте старий проєкт командою **Файл -> Закрити**, і створіть новий проєкт командою **Проєкт -> Створити проєкт -> Додаток**. Відразу ж збережіть проєкт у папку 05-01 там, де ви зберігаєте всі проєкти лекцій, проєкту дайте ім'я MyStrings, і не забудьте назвати модуль головної форми Main, а саму форму fMain. Нехай це ввійде у вас у звичку. Як все це робиться, ви повинні знати по попередніх лекціях. Не здумайте давати своїм проєктам імена, використовуючи ключові (зарезервовані) слова. Наприклад, не можна давати проєкту або модулю імена Strings, оскільки це слово зарезервоване, але MyStrings можна.

Прямо посередині форми встановіть кнопку **TButton**, двічі натисніть на неї, щоб згенерувати оброблювач OnClick. В оброблювачі створимо розділ змінних var, зазначимо там одну змінну типу Char, і запрограмуємо наступний код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  ch1: Char; //змінна символьного типу
begin
  ch1:= 'Z';
  ShowMessage(ch1);
end;
```

Як бачимо, тут ми змінній ch1 привласнили значення - символ «Z». Запам'ятайте правило:

Символи й рядки в Lazarus повинні бути вміщені в одинарні лапки. Якщо потрібно ввести в текст одинарні лапки (наприклад, у слові **can't**), то варто вказати двоє таких одинарних лапок, які також вміщені в одинарні лапки.

Наприклад:

```
Mystring:= 'I can' + '''' + 't';
```

Цей приклад виконувати не потрібно, він просто демонструє використання лапок у рядкових виразах, і можливість у виразі з'єднувати кілька рядків в один за допомогою знака «+». Однак повернемося до нашого проєкту. Після того, як ми привласнили символьній змінній велику англійську літеру «Z», ми вивели вміст змінної за допомогою рядка

```
ShowMessage(ch1);
```

Збережіть проєкт, скомпілюйте його й виконайте. Коли програма із кнопкою з'явиться на екрані, натисніть нашу єдину кнопку - вийде повідомлення з літерою «Z»:

Рис. 5.2. *Висновок символу*

Таким чином, ми можемо працювати з окремими символами. Однак якщо ви виправите код, і замість літери «Z» у рядку

```
ch1:= 'Z';
```

вкажіть український символ, то при спробі скомпілювати й запустити проєкт вийде помилка «Error: Incompatible types: got «Constant String» expected «Char»« - несумісність типів String (рядок) і Char (символ). Оскільки українські літери займають по два байта, Lazarus їх вважає рядком символів. Однак це не виходить, що ми зовсім не маємо можливості працювати з окремими символами кирилиці.

Раніше згадувалося, що в Lazarus використовується формат UTF8. Для підтримки цього формату були розроблені розширені типи даних, у тому числі й символьні. Так, є тип TUTF8Char, що дозволяє працювати з **будь-якими** окремими символами, у тому числі й українськими. Має сенс завжди використовувати його замість стандартного Char. Але для цього нам потрібно буде підключити модуль LCLType, де цей тип описаний.

Перейдіть у редактор коду й пролистайте код модуля нагору. Там ви побачите розділ uses (англ. use - використати), де перераховані підключені модулі. Нам потрібно поставити після останнього модуля кому й додати наш модуль підключення:

```
uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, StdCtrls,
  LCLType;
```

Якщо рядок виходить довгий, то після коми можна переносити текст на інший рядок.

Тепер повернемося до оброблювача натискання на кнопку й переробимо його:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  ch1: Char;
  ch2: TUTF8Char;
begin
  ch1:= 'Z';
  ch2:= 'Я';
  ShowMessage(ch1);
  ShowMessage(ch2);
```

```
end;
```

Тепер все спрацює як треба, і український символ «Я» вийде так само, як англійський «Z».

Однак, це ще не все. Символи можна вводити, вказуючи їхній номер у таблиці символів. Для цього перед номером потрібно вказати символ «#», наприклад:

```
ch1:= #90;
```

Під номером 90 перебуває літера Z у таблиці символів, так що результат не зміниться. Також у строкових виразах можна вказувати й спеціальні символи. Наприклад, символ під номером 13 - це символ переходу на інший рядок. Давайте знову трішки змінимо код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  ch1: Char;
  ch2: TUTF8Char;
begin
  ch1:= #90;  //буква Z
  ch2:= 'Я';
  ShowMessage(ch1 + #13 + ch2); //висновок дворядкового повідомлення
end;
```

Збережіть проєкт, скомпілюйте й запустіть на виконання. Тепер вміст двох символьних змінних виходить в одному повідомленні, але кожне на своєму рядку.

3. Строкові типи даних

В Lazarus основним рядковим типом є String (англ. string - рядок). На форму нашого проєкту розмістіть ще одну кнопку. Її оброблювач буде виглядати так:

```
procedure TfMain.Button2Click(Sender: TObject);
var
  s: String;
begin
  s:= 'Це рядок з п'яти слів!';
  ShowMessage(s);
end;
```

Як бачите, оскільки рядок в Lazarus має формат UTF8, то ніяких особливих хитрощів для роботи з рядками отут не потрібно, у змінну типу String можна записати будь-який, у тому числі й український текст. Розмір рядка String необмежений, однак є можливість жорстко задати розмір. Такий спосіб використовується, коли ви точно знаєте, що більше даного розміру рядок не буде. У цьому випадку розмір вказується після ключового слова String у квадратних дужках, наприклад:

```
var
  MyStr: String[50];
```

У змінну MyStr можна записати до 50 символів. Максимальний розмір рядка, який можна жорстко задати таким способом - 255 символів. Однак маються на увазі символи ASCII, тобто англійські, однобайтові. Приклад:

```
var
  s: string[7];
begin
  s:= 'Привіт!';
  ShowMessage(s);
end;
```

Даний приклад помилки не викличе, однак повідомлення вийде не повністю, а обрізаним: «При». Тобто, три перших літери зайняли 6 байт, четверта вже не вмістилася. У цьому випадку буде правильним вказати розмір не 7, а 14 - за подвоєною кількіст букв. Втім, на практиці звичайно застосовують тип String без обмежень, у цьому випадку рядок обробляється правильно.

Майте на увазі, що тип String є динамічним, тобто, заздалегідь пам’ять для нього не виділяється. Строго говорячи, виділяється пам’ять на покажчик рядка, а не на сам рядок. Фізично рядку виділяється необхідна пам’ять тільки в момент присвоєння їй якогось значення. Однак нерідко виникає необхідність очистити такий рядок. Для цього досить привласнити їй порожні лапки, без пробілів і без яких-небудь інших символів:

```
s:= '';
```

У цьому випадку, рядку привласнюється значення Nil, тобто нуль, нічого, і рядок стає порожнім.

Рядковій змінній можна привласнювати значення символьних змінних або констант, наприклад:

```
var
  c: Char;
  s: String;
begin
  c:= 'Z';
  s:= c;
```

Крім String в **Lazarus** є й інші рядкові типи.

ShortString	Короткий рядок, що може містити максимум, 255 ASCII-символів. Тобто, ShortString - це String[255].
AnsiString	Рядок необмеженої довжини з ANSI-символів. Фактично все, що ми говорили про String, відноситься саме до AnsiString. Окремого типу String як такого, не існує, це просто скорочений запис AnsiString. Однак це залежить від настроювань. Є в Паскалі такі перемикачі, які ще називають директиви компілятора. Такі директиви подібно коментарям, беруть у фігурні дужки, а першим символом повинен бути символ долара. Тип, що буде використаний у якості String, залежить від директиви {$H}. Якщо вона виключена {$H-}, то при вказівці типу String буде матися на увазі тип ShortString. Якщо вона включена {$H+} - то AnsiString. За замовчуванням перемикач включений, ви зможете це побачити в третьому рядку модуля, пролиставши його код нагору: unit Main; {$mode objfpc}{$H+} На практиці короткий рядок використовують вкрай рідко, адже якщо потрібен короткий рядок, то його розмір можна вказати явно. Тому даний перемикач виправляти вручну звичайно не доводиться. Давайте вважати, що String і AnsiString - це те саме.
PChar	Рядок у стилі C/C++ з нульовим символом наприкінці. Такий тип рядків використається у функціях **Windows API**, тому має підтримку й в **Lazarus**. Що це за рядок

	такий? Справа в тому, що звичайні рядки динамічні, тобто фактично змінні типу String - це вказівник на рядок. Ця змінна зберігає фізичну адресу рядка в пам'яті, і кількість займаних нею байт. Рядок типу PChar влаштований трохи інакше. Це теж вказівник на рядок, але він не зберігає його розмір. А як тоді компіляторові знати, скільки байт потрібно зчитувати при звертанні до такої змінної? Справа в тому, що в рядках типу PChar завершальним завжди буде нульовий символ, тобто, символ #0. Якщо такий символ помістити всередині рядка, то компілятор не буде читати весь рядок. Зустрівши символ #0, він вирішить, що рядок закінчений. Ми будемо використовувати PChar, коли дійдемо до функцій WinAPI.

Як правило, рядкам одного типу можна привласнювати значення рядків іншого типу, компілятор автоматично перетворить текст. Виключення становлять рядки PChar, отут нам доведеться робити перетворення вручну, за допомогою функції PChar(). Налагодимо оброблювача другої кнопки:

```
procedure TfMain.Button2Click(Sender: TObject);
var
  s1: String; //він же AnsiString
  s2: ShortString;
  s3: PChar;
begin
  s1:= 'Це рядок з п'яти слів!';

  //Тепер привласнимо той же текст в ShortString:
  s2:= s1;

  //Той же текст в PChar. Для перетворення рядка в цей тип
  //використаємо функцію PChar():
  s3:= PChar(s1);

  //Зберемо повідомлення із трьох різнотипних рядків. Кожний тип
  //виведемо в окремому рядку:
  ShowMessage(s1 + #13 + s2 + #13 + s3);
end;
```

Збережітье проєкт, скомпілюйте й запустіть. При натисканні на другу кнопку у вас вийде повідомлення із трьох рядків

Є ще типи UnicodeString і WideString, обидва типи містять двохбайтові символи. Але працювати з такими типами незручно - виникають проблеми з кирилицею, для нас набагато зручніший простий String. Тому розглядати дані типи ми не будемо.

Резюмуємо деякі положення:

- Три основних строкових типи: AnsiString, ShortString і PChar.
- Звичайно вказують тип String. Якщо перемикач {$H} включений (за замовчуванням), то використовується тип AnsiString. Інакше - ShortString.
- Рядку можна привласнити або текст, вкладений в одинарні лапки, або вміст іншої рядкової змінної, наприклад:

```
s1:= 'Текст';
s2:= s1;
```

- Рядку типу PChar можна привласнити або текст, або вміст інший рядкової змінної. Але якщо тип цієї змінної відрізняється, то потрібно зробити перетворення функцією PChar():

```
var
  s: String;
  pc: PChar;
begin
  s:= 'Текст';
  pc:= 'Текст'; //привласнили текст
  pc:= PChar(s); //привласнили перетворене значення змінної s
```

4. Компоненти для введення рядків

Тепер ми знаємо, що таке рядок, саме час розібрати способи надання користувачеві можливостей ці самі рядки вводити в нашу програму. Згадаємо другий проєкт другої лекції? Там ми пропонували користувачеві ввести своє ім'я, що потім використали у вітанні. Для цього ми використали компонент TEdit. Розберемо його можливості, і познайомимося з іншими аналогічними компонентами.

Для цього закрийте старий проєкт і створіть новий. Як завжди, головну форму назвіть fMain, і збережіть проєкт. При цьому проєкту дайте ім'я EditControls (контролами програмісти між собою називають компоненти, за допомогою яких програма взаємодіє з користувачем),

а модуль цієї форми назвіть Main. Проєкт збережіть в папку 05-02 там, де в нас зберігаються всі інші проєкти.

4.1. Компонент TEdit

На форму встановіть компонент TEdit (якщо не пам'ятаєте, де він перебуває, дивіться рис. 2.5 в темі №2). За замовчуванням, він має ширину 80 пікселів (властивість Width), збільште його до 200. Тепер давайте розбиратися з основними властивостями цього компонента, які можуть нам знадобитися, для цього він повинен бути виділений.

Почнемо із самого початку. Рекомендую прочитати пояснення до властивості, а потім «програти» з його значеннями, міняючи їх і запускаючи програму на виконання, щоб подивитися результат. Потім повернути значення за замовчуванням, якщо не буде запропоноване інше, після чого переходити до наступної властивості. Таким чином, ви познайомитеся з компонентом більш докладно.

Align	Вирівнювання компонента. Із цією властивістю ми вже знайомі, вона дозволяє компонент зайняти весь верх, низ, ліву або праву сторону, або ж всю форму або панель, на якій компонент перебуває. У компоненті TEdit ця властивість теж є, але застосовується дуже рідко - важко придумати ситуацію, у якій цей компонент буде потрібно в такий спосіб вирівняти. Звичайно цю властивість не чіпають. Значення цієї властивості повинні бути вам знайомі по лекції №3.
Alignment	Вирівнювання тексту. От ця властивість використовується частіше. Вона дозволяє вирівняти текст у компоненті по центру, по лівому або правому краї. Як вирівнювати текст, звичайно кожний вирішує для себе сам, але є деякі рекомендації: простий текст вирівнюють звичайно по лівому краю, цифри - по правому, а такі дані, як дата-час можна вирівняти по центру. Отже, значення цієї властивості: • taCenter - по центру. • taLeftJustify - по лівому краю. Це значення використовується за замовчуванням. • taRightJustify - по правому краю.
Anchors	Прив'язка компонента до країв форми або панелі, на якій компонент перебуває. Із цією властивістю ви також знайомі по третій лекції.

AutoSelect	Автоматичне виділення тексту. Якщо властивість дорівнює True (за замовчуванням), то коли компонент стає активним, виділеним, або як говорять, одержує фокус введення, текст у компоненті, якщо він є, автоматично виділяється. Інакше текст буде невиділений.
CharCase	Регістр символів тексту в компоненті. Може мати наступні значення: • ecLowerCase - Всі символи рядкові (звичайні), навіть якщо набирати текст із натиснутою клавішею **<Shift>** або із включеної **<Caps Lock>**. • ecNormal - Звичайний текст, у якому можуть бути й рядкові, і прописні символи. Використовується за замовчуванням. • ecUpperCase - Всі символи прописні (заголовні).
Color	Дозволяє вибрати кольори тла в компоненті.
Font	Дозволяє вибрати шрифт, використовуваний у компоненті. Взагалі кожен вирішує сам, як буде виглядати дизайн його програми, однак бажано для всіх компонентів використовувати той самий шрифт. Якщо вас не влаштовує шрифт за замовчуванням, поміняйте його для всієї форми, і він автоматично буде використовуватися для всіх компонентів, які ви на цій формі встановите! Якщо ж у вашому проєкті буде багато форм, то виробіть для них якісь одні стандарти (шрифт, кольори тла, обрамлення й т.п.), і дотримуйтеся їх у всіх формах.
Hint	Текст підказки, що буде спливати, коли користувач підведе до компонента курсор миші.
MaxLength	Максимальний розмір тексту в символах. За замовчуванням дорівнює нулю, що знімає обмеження в розмірах. Якщо текст не вміщається в поле введення, він буде прокручуватися. Обмеження на кількість символів іноді буває необхідним, наприклад, якщо текст буде записаний у базу даних або запис, які мають фіксований розмір рядка. У всіх інших випадках властивість можна не змінювати.

PasswordChar	Символ, що приховує пароль при введенні. Дуже цікава властивість. Мабуть, усім доводилося колись вводити пароль. І звичайно замість потрібних літер і цифр у цьому полі відображалися зірочки або кружечки. Насправді, текст залишається незмінним, але він ховається від тих, хто може виявитися у вас за спиною й підглянути ваш пароль. Реалізується ця можливість саме даною властивістю. Тут потрібно вказати номер символу в таблиці символів. За замовчуванням, це #0 - нульовий символ, що означає, що текст виводиться, як є. Але можна поставити й інший символ, причому вказати саме символ, а не його номер. Давайте в цій властивості вкажемо символ «*». Як тільки ви натиснете **<Enter>**, відразу ж зміниться й властивість EchoMode - обидві ці властивості взаємозалежні. Ми розберемо її трохи нижче, а поки збережіть проєкт і запустіть на виконання. Тепер будь-який текст, включаючи пробіли й інші знаки, буде виводитися на екран у вигляді зірочок. Однак це не все.

Ми маємо можливість маскувати пароль і інші символи. Виділіть компонент TEdit і змінітье його властивість Font, вибравши там шрифт Wingdings (якщо у вас ОС Windows). Це той рідкий випадок, коли має сенс вибрати інший шрифт для окремого компонента. Тепер нам потрібно вибрати символ. Відкрийте процесор MS Word. Там виконайте команду **Вставка -> Символ**:

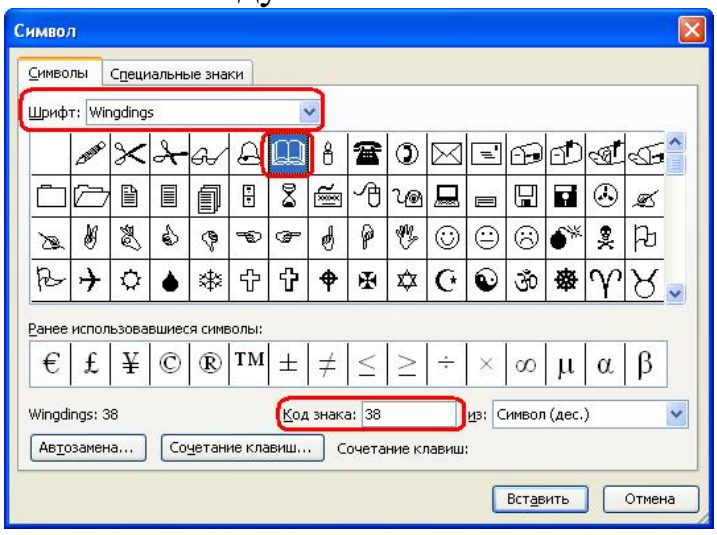

Рис. 5.3. *Визначення коду символу*

У полі «**Шрифт**» виберіть Wingdings, потім виберіть підходящий символ, наприклад, книгу. Потім подивіться код обраного знака, у цьому випадку це 38. Натисніть «**Скасування**» і закрийте MS Word - він нам більше не потрібний. Тепер поверніться до проєкту й у властивості PasswordChar укажіть #38 - код обраного нами знаку. Коли натиснете **<Enter>**, то значення #38 зміниться на & - не звертайте уваги. Збережіть проєкт і запустіть його на виконання - текст буде ховатися за піктограмою книги. Використовуючи різні шрифти, ви можете підібрати для себе й інші підходящі символи.

EchoMode	Тип повідомлення. Пов'язаний з PasswordChar, і звичайно встановлюється автоматично, але можна встановити його й вручну. Має наступні значення: • emNone - замість символів виводяться пробіли. Сам текст не змінюється. • emNormal - звичайний тип, використовується за замовчуванням. Встановлюється, якщо в PasswordChar установлене значення #0. • emPassword - тип пароля. Встановлюється, якщо в PasswordChar ви вкажете інше значення, не #0. Сам текст не змінюється.
ReadOnly	Тільки для читання. Дозволяє або забороняє користувачеві введення тексту. Якщо False, то користувач може й читати, і вводити текст, інакше він може тільки читати його. Однак у кожному разі ви можете змінювати там текст програмно. Пізніше ми розберемо таку можливість.
Text	Основна властивість цього компонента. Містить текст, введений користувачем, або встановлений у компоненті програмно. Причому навіть якщо ви його сховаєте, як пароль, сам текст змінюватися не буде. До цієї властивості можна ставитися, як до рядкової змінної, котру може змінювати користувач. За замовчуванням, у цю властивість відразу ж прописується ім'я компонента, у нашому випадку, edit1. На практиці такий текст навряд чи знадобиться, тому програміст звичайно очищає цю властивість. Очистіть його й ви. Як тільки для підтвердження натиснете **<Enter>**, текст у компоненті на формі зникне.

Інші корисні властивості повинні бути вам знайомі по попередніх лекціях. Отже, встановіть для компонента Edit1 шрифт Times New Roman, розмір шрифту нехай буде 11. У властивості PasswordChar нехай буде значення за замовчуванням - #0. Властивість Text очищена, а властивість ReadOnly установлена в True. Тепер нижче TEdit встановіть кнопку, згенеруйте для неї оброблювач натискання, і введіть там наступний код:

```
procedure TfMain.Button1Click(Sender: TObject);
begin
  Edit1.Text:= 'Привіт!';
end;
```

Тут ми програмно змінюємо текст у компоненті. Для цього ми звертаємося до нього за іменем - Edit1, після чого ставимо крапку. Зверніть увагу - якщо поставити крапку й трішки почекати, спливе віконце з усіма властивостями, методами й подіями, які ми можемо використати. Нам потрібно властивість Text. Як тільки ми введемо першу «T», список відфільтрується, залишивши там тільки команди на букву «T». Як тільки ми введемо наступну букву «e», то в списку залишиться тільки одна підходяща властивість - «Text». Те, що потрібно. Можна просто натиснути **<Enter>**, не вводячи інших символів, і властивість сама вставиться в код. Причому разом з наступним оператором присвоювання. Дріб'язок, а приємно. Залишиться тільки вказати рядок, що ми хочемо привласнити.

Таким чином, можна програмно змінювати багато які (але не всі) властивості компонентів, їхні розміри й положення, текст і кольори, і багато чого іншого. Збережіть проєкт і запустіть його на виконання. Якщо ви все зробили правильно, то користувач не зможе вписувати текст в Edit1, а при натисканні кнопки **Button1**, текст там поміняється програмно. Тепер давайте трішки змінимо команду присвоювання:

```
Edit1.Text:= Edit1.Text + 'Привіт! ';
```

З виразами ми вже працювали, так що тут труднощів бути не повинно. Ми просто беремо текст, що вже був у властивості Text (при першому натисканні там буде порожній рядок), і додаємо до нього рядок 'Привіт! ' - не забудьте після знака оклику поставити пробіл, щоб текст не злипався один з одним. Збережіть проєкт і запустіть його.

Натискаючи кнопку, ми зможемо багаторазово додавати слово ‘Привіт! ‘ у компонент.

4.2Компонент TLabelEdit

На вкладці **Additional Палітри компонентів** є дивний компонент TLabelEdit:

Рис. 5.4. *Компонент TLabelEdit*

Встановіть його на вільне місце на форму, і подивіться на його властивості в **Інспекторі об’єктів**. Взагалі ж, це гібрид мітки TLabel і текстового поля TEdit. Властивості майже всі знайомі, але є й деякі розходження.

EditLabel - розкриваюча властивість, всередині якої перебуває свій набір підвластивостей для мітки. Ліворуч від цієї властивості ми бачимо кнопку «+», клацнувши по який розкриємо підвластивості. Тут нам знадобиться, насамперед, властивість Caption, у якій ми можемо записати текст мітки. Цей текст буде пояснювати призначення текстового поля. Напишемо в цій властивості Прізвище, текст мітки зміниться, а користувач відразу зрозуміє, що сюди він повинен буде вписати своє прізвище:

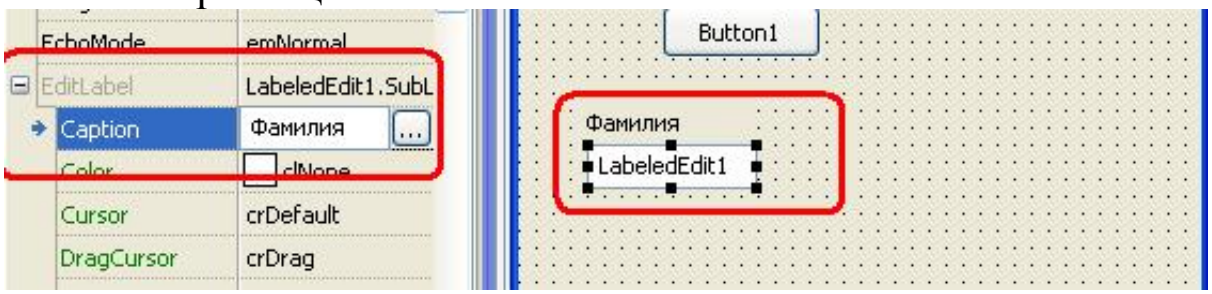

Рис. 5.5. *Компонент TLabelEdit в Інспекторі об’єктів і в Редакторі форм*

Можете клацнути по кнопці «-» ліворуч від властивості, закривши список підвластивостей, більше нічого цікавого там немає. Подивимося, які властивості ще нам знадобляться.

LabelPosition - місцезнаходження мітки. Може бути:

- lpAbove - зверху текстового поля. Це значення за замовчуванням.
- lpBelow - знизу текстового поля.
- lpLeft - ліворуч

• lpRight - праворуч

LabelSpacing - відстань у пікселях між міткою й текстом. За замовчуванням дорівнює 3 пікселі. Змінюючи це значення, можна наближати мітку до поля або відсувати її.

В іншій, цей компонент майже повністю ідентичний звичайному TEdit.

TLabelEdit буде корисний при створенні форм, де користувач повинен вписувати різні дані, для цього раніше застосовувалися окремо TLabel і TEdit. Даний гібрид, безумовно, зручніший.

4.3 Компонент TMaskEdit

На тій же вкладці **Additional** перебуває ще один корисний компонент для введення рядків - TMaskEdit:

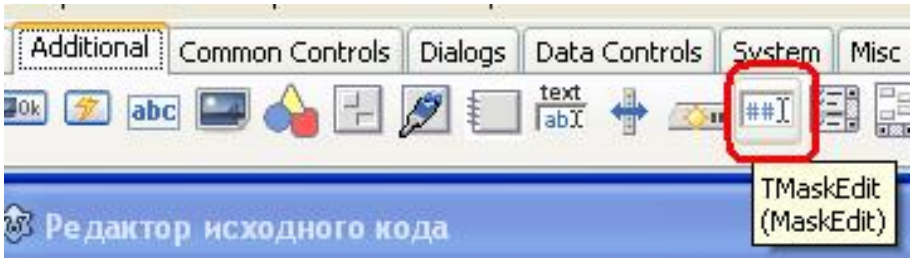

Рис. 5.6. *Компонент TMaskEdit*

Цей компонент призначений для введення тексту по певних шаблонах - масках. Коли користувачеві потрібно буде ввести в цей компонент якийсь текст, він буде змушений вводити його саме в тому вигляді, у якому нам потрібно.

Встановіть TMaskEdit на форму й подивіться на його властивості в **Інспекторі об'єктів**. В принципі, всі властивості нам знайомі. Інтерес викликає лише одна властивість EditMask. Це складна розкриваюча властивість, тобто, клацнувши по кнопці із трьома крапками правіше властивості, можна її розкрити:

Рис. 5.7. *властивість, Що Розкривається, EditMask*

Коли вона розкрита, ми бачимо наступну картину:

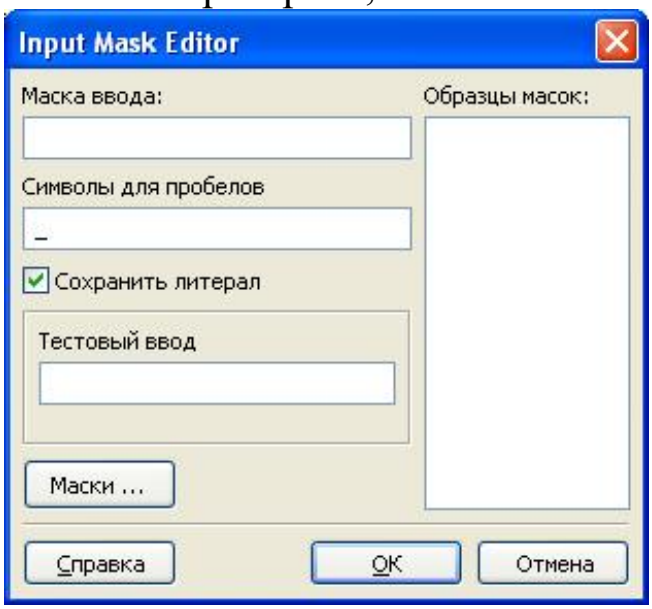

Рис. 5.8. *Редактор масок*

У рядку «**Маска введення**» ми можемо вказати маску, що буде впливати на формат заповнюваного користувачем рядка. У масці можна застосовувати деякі символи, що вводяться у кожній позиції, і символи, що додаються самою маскою. Маска може мати наступні символи:

Таблиця 5.1. Символи маски компонента TMaskEdit	
0	У даній позиції повинна бути цифра. Якщо користувач введе не всі цифри маски, буде створена виняткова ситуація, і він не зможе продовжувати роботу, не заповнивши всі необхідні цифри. Якщо така ситуація виникла у вас під час тестового виконання програми, можете закрити програму командою «Запуск->Скинути отладчик».
9	У даній позиції може бути або цифра, або нічого.
#	У даній позиції може бути знак «+», «-», цифра, або нічого.
L	У даній позиції повинна бути літера.
l	У даній позиції повинна бути літера або нічого.
A	У даній позиції повинна бути літера або цифра.
a	У даній позиції повинна бути літера або цифра, або нічого.
C	У даній позиції повинен бути будь-який символ.
c	У даній позиції повинен бути будь-який символ, або нічого.

Крім того, у масці можна використати різні роздільники:

Таблиця 5.2. Символи-роздільники маски компонента TMaskEdit	
:	Звичайно застосовується для поділу годин, хвилин, секунд.
/	Звичайно застосовується для поділу року, місяця, дня.
-	Звичайно застосовується для поділу цифр, наприклад, у телефонних номерах.

Приклади масок:

- 999-99-99 (телефон)
- 99/99/9999 (дата)
- 00:00:00 (час)
- 999999,00 р. (грошовий формат)

Давайте вкажемо в рядку «**Маска введення**» маску телефону:
000-00-00

Використання нулів як маски означає, що в цьому місці обов'язково повинна бути цифра. Це розумно, адже інакше номер телефону буде неповний, і ми не зможемо додзвонитися до абонента. Символи «-» служать роздільниками, і додаються автоматично. Введіть маску, натисніть «**ОК**», збережітье проєкт і запустіть його на виконання. Тепер ми зможемо вводити в маску тільки цифри. Причому якщо ми введемо не всі цифри, і спробуємо вийти з рядка редагування, згенерується помилка. Нам доведеться або повернутися й заповнити номер до кінця, або зупинити програму, скинувши налаштування.

В **Редакторі масок** крім рядка «**Маска введення**» є й інші рядки. У рядку «**Символи для пробілів**» ми можемо вказати символ, що буде виводитися маскою в тому місці, де користувач повинен буде щось ввести. За замовчуванням, це символ підкреслення «_». Користувач буде бачити порожню маску телефону, як

___-__-__

Ми можемо використати й інші символи для заповнення пробілів.

Нижче в **Редакторі масок** є прапорець «**Зберегти літерал**». Цей прапорець включає або виключає можливість збереження в тексті символів-роздільників. Якщо прапорець включений, то в тексті

збережуться символи маски разом із символами-роздільниками, наприклад:

123-45-67

Якщо прапорець виключений, то в кінцевому тексті залишаться тільки символи маски:

1234567

У рядку «**Тестове введення**» ми можемо випробувати отриману маску, не запускаючи програму на виконання.

Отже, для нашого прикладу в рядку «**Маска введення**» вкажіть

000-00-00

В «**Символи для пробілів**» залишіть «_». Прапорець «**Зберігати літерал**» нехай буде включений. Як тільки натиснете «**ОК**», властивість EditMask зміниться, тепер у ній буде текст:

000-00-00;1;_

Як бачите, значення цієї властивості складається із трьох частин, розділених крапкою з комою. Перша частина - це зазначена нами маска. У другій частині може бути або 0 (прапорець «**Зберігати літерал**» виключений), або 1 (прапорець включений). У третій частині маски вказується символ для пробілів.

Таким чином, ми могли б і не користуватися **Редактором масок**, а вказати маску вручну. Наприклад, ввівши значення

00-00-0000 р.;1;_

ми створимо маску, де користувач буде зобов’язаний ввести всі цифри місяця, дня й року, де в тексті збережуться символи-роздільники «-», і де на місці пробілів буде знак підкреслення. У результаті, ми одержимо дату, наприклад у такому виді: «11-11-2013 р.».

Тема 6. Стандартні строкові функції й повідомлення

1. Функції для роботи з рядками

На попередній лекції ми досить докладно познайомилися рядковими та й символьними типами даних, вивчили три компоненти, за допомогою яких користувач може вводити в програму рядка. Однак цих інструментів найчастіше буває недостатньо для обробки рядків. У практиці програмування над рядками раз у раз потрібно виконувати якісь дії: порівнювати рядки між собою, перетворювати рядок у рядкові або в прописні літери, знаходити частину рядка й т.п. Сьогодні ми розберемо найбільш затребувані рядковіі функції.

А для того, щоб подивитися роботу функцій на практиці, створимо новий додаток. Це буде простий додаток з єдиною кнопкою посередині вікна. Збережіть проєкт у папку **06-01**, оскільки проєкт буде зовсім простий, імена модуля й проєкту можна залишити без змін. Згенеруйте подію натискання на кнопку, у цій події ми й будемо експериментувати над рядками.

2. Об'єднання (конкатенація) рядків

Програмістові досить часто потрібно в коді об'єднувати декілька рядків в один. Ми вже робили подібне об'єднання на попережніх лекціях. Для об'єднання слугуєь оператор «+»:

```
string_1 + string_2 + ...string_n;
```

Як рядок можна використати й окремі символи, наприклад, символ переходу на інший рядок. Напишемо наступний код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s1: String;
begin
  s1:= 'Рядок №1' + #13 + 'Рядок №2' + #13 + 'Рядок №3';
  ShowMessage(s1);
end;
```

Тут, у змінній s1 ми зібрали рядок з 5 шматочків, причому два з них - символи переходу на новий рядок. Збережіть проєкт, скомпілюйте його й переконаєтеся, що отримане повідомлення буде трирядковим.

Функція Concat() робить те же саме - об’єднує рядки. Її синтаксис такий (у квадратні дужки прийнято поміщати необов’язкові параметри функцій):

```
Concat(S1 [, S2,...Sn]);
```

Тут S1, S2 і т.д. - рядка. Попередній приклад можна було б записати й так:

```
sl:= Concat('Рядок №1', #13, 'Рядок №2', #13, 'Рядок №3');
```

3. Довжина рядка

Оскільки ми надаємо користувачам можливість вводити різні рядки, нерідко виникає необхідність з’ясувати довжину цих самих рядків. Для цього існує функція Length(). Її синтаксис:

```
Length(S);
```

де S - рядок. Функція повертає нам розмірну кількість символів у цьому рядку. Однак в Lazarus не все так просто. Давайте змінимо попередній код налаштовувача кнопки на наступний:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  sl: String;
begin
  sl:= 'Hello';
  ShowMessage(IntToStr(Length(sl)));
end;
```

Оскільки результатом роботи функції Length() буде ціле число - кількість символів, то для підсумку цих даних на екран нам потрібно перетворити отримане число в рядковий тип, що ми й робимо в коді

```
ShowMessage(IntToStr(Length(sl)));
```

Подібні перетворення ми будемо розглядати пізніше, поки лише потрібно зрозуміти от що: даний рядок коду реалізує складний вираз, що виконується комп'ютером у три етапи:

1. Length() - обчислення розміру рядка s1.
2. IntToStr() - перетворення отриманого в №1 результату із цілого числа в рядок.
3. ShowMessage() - підсумок перетвореного в №2 результату на екран.

Якщо ви відкомпілюєте проєкт і запустите його на виконання, то отримаєте результат: 5. Стільки символів у рядку «Hello». Однак змінітье цей рядок на «Привіт» і знову виконаєте програму:

```
s1:= 'Привіт';
ShowMessage(IntToStr(Length(s1)));
```

Ми очікуємо одержати число 6 - кількість символів у слові «Привіт», але зненацька одержуємо 12! Справа в тому, що функція працює з ANSI-рядками, тобто з тими, що займають по 1 байту на символ. А кирилиця - це не ANSI, а UTF8, і вимагає по 2 байти на символ! Як бути? У подібних випадках використайте UTF8-аналоги рядкових функцій. Тобто, замість Length() використайте UTF8Length(). Ця функція гарантовано поверне кількість символів у рядку, на якій би мові ці символи не вводилися. Але для використання UTF8Length() потрібно підключити модуль LCLProc, де й реалізовані всі UTF8-функції.

Прогорніть код вище, на початок модуля, і додайте LCLProc у розділ uses:

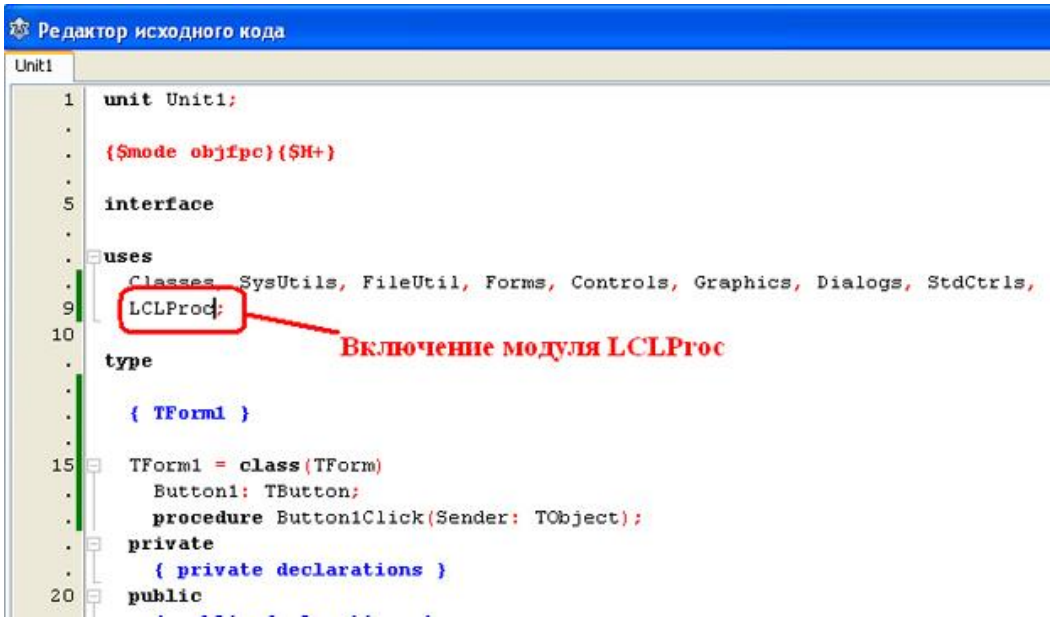

Рис. 6.1. *Додавання модуля в розділ uses*

Тепер ми можемо змінити наш оброблювач:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s1: String;
begin
  s1:= 'Привіт';
  ShowMessage(IntToStr(UTF8Length(s1)));
end;
```

Аж тепер ми одержимо в результаті число 6.

Таким чином, якщо ви точно знаєте, що будете мати справу тільки з англійськими символами, то для визначення кількості символів у рядку, і займаних цим рядком байт, використайте функцію Length(). У всіх інших випадках Length() можна використати для одержання розміру рядка в байтах, а UTF8Length() - для одержання кількості символів у рядку.

4. Пошук у рядку

Іноді необхідно з'ясувати, є чи в рядку необхідний текст. Для цього служать функції

- Pos() - для ANSI-символів
- UTF8Pos() - для кирилиці (UTF8-символи) - вимагає підключення модуля LCLProc

Їхній синтаксис:

```
Pos(Substr, Str);
UTF8Pos(Substr, Str);
```

Тут Substr - шуканий текст; Str - рядок, у якій шукається текст. Функції повертають як результат ціле число. Якщо шуканий текст у рядку відсутній, обидві функції повернуть нуль. Інакше - номер символу рядка, з якого починається знайдена підстрічка. Переробимо код налаштовувача кнопки (модуль LCLProc у нас уже підключений):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s1: String;
begin
  s1:= 'Привіт';
  ShowMessage(IntToStr(UTF8Pos('верб', s1)));  //результат: 3
```

```
end;
```

Запустивши програму на виконання, переконаємося, що функція UTF8Pos('верб', s1)

повертає результат 3 - номер першого символу шуканої підстрічки в рядку «Привіт».

Якщо в рядку утримується декілька шуканих підстрічок, функції повернуть номер першого такого входу. А якщо для рядка з українськими літерами використати функцію Pos() замість UTF8Pos(), результат буде невірний.

5. Одержання підстрічки

Щоб з рядка одержати його частину (підстрічку), застосовують функції

- Copy() - для ANSI-символів
- UTF8Copy() - для кирилиці - вимагає підключення модуля LCLProc

Їхній синтаксис:

Copy(Str, StartCharIndex, Count);

UTF8Copy(Str, StartCharIndex, Count);

Тут Str - рядок вихідного тексту; StartCharIndex - номер першого символу підстрічки; Count - кількість символів підстрічки. Знову переробимо налаштовувач кнопки:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s1: String;
begin
  s1:= 'Привіт';
  ShowMessage(UTF8Copy(s1, 3, 2));
end;
```

У результаті виконання програми ви переконаєтеся, що UTF8Copy(s1, 3, 2)

з рядка «Привіт» повернеться «верб» - підстрічка, починаючи із третього символу, розміром два символи.

Якщо для кирилиці замість UTF8Copy() використати Copy(), результат буде невірним.

6. Видалення частини рядка

Для видалення частини рядка застосовують функції Delete() і UTF8Delete().

Їхній синтаксис:

```
Delete(Str, StartCharIndex, Count);
UTF8Delete(Str, StartCharIndex, Count);
```

Тут Str - рядок вихідного тексту; StartCharIndex - номер першого символу що видаляє підстрічки; Count - кількість символів, що видаляється. Ця функція відрізняється від інших тим, що вона змінює вихідний рядок Str, обрізає його. Інші функції залишали цей рядок без змін. Отже, наш новий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s1: String;
begin
  s1:= 'Привіт';
  UTF8Delete(s1, 3, 2); //обрізаємо рядок
  ShowMessage(s1);
end;
```

Функцією UTF8Delete(s1, 3, 2) ми ампутували рядок s1, вирізали з нього, починаючи із третього символу, підстрічку розміром 2 символи. У результаті в нас залишився рядок «Пре». Як і з попередніми функціями, застосування з кирилицею не UTF8-варіанта спотворить результат.

7. Перетворення символів рядка в малі та великі

Іноді потрібно порівняти два рядки, або знайти якусь підстрічку. Але не завжди відомо, у якому регістрі користувач ввів текст. Приміром, нам потрібно довідатися, чи ввів користувач слово «Київ», або щось інше. Користувач знає, що він має ввести назву нашої столиці, однак він може полінуватися нажати **<Shift>**, і в результаті введе «київ». Або натисне **<Caps Lock>** і введе «КИЇВ». А оскільки слово він написав без помилок, то впевнений, що все зробив правильно. Але ми очікуємо не просто слово, а слово з чітким описом!

На жаль, користувачі далеко не завжди вводять то, що необхідно, і програмістам доводиться йти на усілякі хитрості, щоб домогтися потрібного результату. Наприклад, ми можемо заздалегідь перетворити введене користувачем слово у верхній регістр (тобто, зробити всі літери великими), і порівняти його зі словом «КИЇВ». І яку б літеру користувач спочатку не вводив, результат однаково буде вірним.

Аналогічно ми можемо всі літери зробити малими, і порівняти рядок з текстом «київ». Для подібних перетворень слугують наступні функції:

UpperCase(Str)	- Перетворить ANSI-рядок Str у верхній регістр
UTF8UpperCase(Str)	- Перетворить UTF8-рядок Str у верхній регістр
LowerCase(Str)	- Перетворить ANSI-рядок Str у нижній регістр
UTF8LowerCase(Str)	- Перетворить UTF8-рядок Str у нижній регістр

Змінимо код оброблювача:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s1: String;
begin
  s1:= 'Москва';
  ShowMessage(UTF8UpperCase(s1));
  ShowMessage(UTF8LowerCase(s1));
end;
```

Збережіть проєкт і запустіть його на виконання. Ми одержимо два повідомлення: великими і малими літерами.

8. Функції-повідомлення

Нам раз у раз потрібно виводити користувачеві різні повідомлення. І в цій лекції, і в попередніх лекціях, ми вже неодноразово користувалися однією такою функцією - ShowMessage(). Її синтаксис дуже простий:

```
ShowMessage('Текст повідомлення');
```

У результаті буде виведене вікно з повідомленням, що не дасть користувачеві продовжувати роботу, поки він не натисне кнопку «**ОК**»:

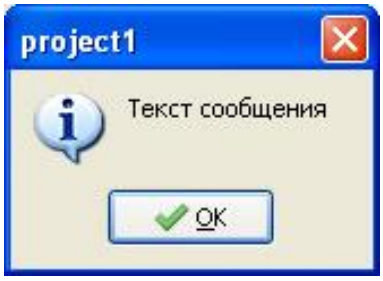

Рис. 6.2. *Повідомлення ShowMessage()*

Application.MessageBox()

Ще один варіант повідомлення - функція Application.MessageBox(). Дана Windows API-функція дозволяє вивести повідомлення з різними налаштуваннями. Синтаксис функції наступний:

Application.MessageBox(Text: PChar; Caption: PChar; Flags: LongInt): Integer;

Як аргументи ми повинні вказати текст повідомлення Text, заголовок вікна з повідомленням Caption і прапорці - значок повідомлення й використовуваних кнопок. І Text, і Caption мають тип PChar. Прапорці мають цілочисельний тип, однак нам потрібно запам’ятати тільки символьні позначення цих прапорці:

Таблиця 6.1. Прапорці функції Application.MessageBox()	
Значки повідомлення	
MB_ICONERROR	Білий хрестик у червоному колі. Такий значок звичайно використають у повідомленнях про помилку.
MB_ICONHAND	
MB_ICONSTOP	
MB_ICONQUESTION	Синій знак питання в білому винесенні. Таким способом позначають питання, звернений до користувача.
MB_ICONWARNING	Чорний знак оклику в жовтому трикутнику. Таким знаком привертають увагу користувача до ймовірної небезпеки, що може піти в результаті дій користувача.
MB_ICONEXCLAMATION	

MB_ICONINFORMATION	Синя буква «і» у білому винесенні. Таким способом позначають якусь інформацію для користувача.
MB_ICONASTERICK	
Кнопки повідомлення	
MB_OK	Кнопка «**OK**» у середині вікна.
MB_OKCANCEL	Кнопки «**OK**» і «**Cancel**».
MB_ABORTRETRYIGNORE	Кнопки «**Abort**», «**Retry**» і «**Ignore**».
MB_YESNOCANCEL	Кнопки «**Yes**», «**No**» і «**Cancel**».
MB_YESNO	Кнопки «**Yes**» і «**No**».
MB_RETRYCANCEL	Кнопки «**Retry**» і «**Cancel**».

Примітка: Щоб користуватися функцією Application.MessageBox(), потрібно в розділі uses підключити модуль LCLType.

Як видно з таблиці, деякі значки повідомлень збігаються. Вказувати потрібно якесь одне з них. Переробимо код налаштовувача кнопки:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Application.MessageBox('Повідомлення', 'Заголовок',
                  MB_ICONINFORMATION + MB_ABORTRETRYIGNORE);
end;
```

В результаті отримаємо таке вікно повідомлення:

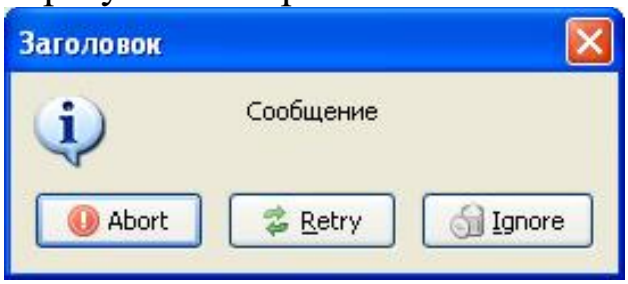

Рис. 6.3. *Повідомлення Application.MessageBox()*

Ви помітили, що як прапорець використовується сума значка й кнопок? І те, і інше, насправді, ціле число. Але нам зручніше запам’ятати ці значення в символьному варіанті.

А як ми довідаємося, яку кнопку натиснув користувач? Ця функція повертає значення - кнопці, натиснутій користувачем, що може бути:

- IDOK
- IDCANCEL
- IDABORT
- IDRETRY
- IDIGNORE
- IDYES
- IDNO
- IDCLOSE
- IDHELP

Тобто, якщо користувач натиснув кнопку «**Abort**», функція поверне значення IDABORT. А раз так, ми можемо робити перевірку на натиснуту кнопку. Снову змінимо код оброблювача:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Application.MessageBox('Повідомлення', 'Заголовок', MB_ICONQUESTION +
     MB_YESNOCANCEL) = IDYES then ShowMessage('Ви нажали кнопку Yes');
end;
```

Логічну конструкцію if ми будемо вивчати в наступній лекції, тут лише помітимо, що повідомлення ShowMessage() буде виведено тільки в тому випадку, якщо користувач натиснув кнопку «**Yes**». У всіх інших випадках це повідомлення не вийде.

MessageDlg()

Схожим образом діє Windows Функці-API-функція MessageDlg(), що описан у модулі Dialogs. Цей модуль включається в розділ uses автоматично, тому нам не потрібно навіть щось туди додавати. Синтаксис цієї функції трохи відрізняється від попередньої:

```
MessageDlg(Caption, Text, MessageType, MessageButton, HelpKeyword): Integer;
```

Тут Caption і Text відповідно, текст заголовка й самого повідомлення, тип PChar. Далі йде тип повідомлення MessageType, - значок повідомлення, що може бути:

mtWarning	- Знак оклику, подібний MB_ICONWARNING функції Application.MessageBox()
mtError	- хрестик
mtInformation	- літера «i»
mtConfirmation	- знак питання
mtCustom	- користувальницьке вікно без значка.

Далі потрібен тип кнопок MessageButton, що може бути:

mbYes
mbNo
mbOK
mbCancel
mbAbort
mbRetry
mbIgnore
mbAll
mbNoToAll
mbYesToAll
mbHelp
mbClose

Назви кнопок говорять самі за себе, коментарі тут зайві. Кнопки вказують у квадратних дужках, відокремлюючи один від одного комами, наприклад: [mbYes, mbNo, mbIgnore].

Останній параметр HelpKeyword визначає екран контекстної довідки, що буде з'являтися, якщо користувач натисне **<F1>**. Звичайно в цьому параметрі вказують значення 0 - немає довідки.

Функція MessageDlg() повертає значення - натиснуту кнопку, що відповідає типу кнопок, зазначеному вище. Значення, що повертає, замість «mb» починається на «mr», тобто, якщо користувач натиснув кнопку «**Yes**» (тип mbYes), те буде повернуте значення mrYes.

Знову змінимо код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if MessageDlg('Підтвердження', 'Ви дійсно хочете закрити програму?',
```

```
    mtConfirmation, [mbYes, mbNo, mbIgnore], 0) = mrYes then Close;
end;
```

У цьому випадку користувачеві виводиться запит на підтвердження закриття програми, і якщо він натискає кнопку «**Yes**», програма закривається (виконується оператор Close). Тому що функції MessageDlg() і Application.MessageBox() схожі, вибирайте кожну, на свій смак.

9. Функція-запит

Іноді потрібно отримати від користувача якісь дані (згадайте програму Hello лекції №2). У цьому випадку можна використати компонент TEdit, але можна зробити простіше - скористатися функцією InputQuery().

Функція InputQuery() виводить вікно запиту. Синтаксис функції такий:

```
InputQuery(Caption, Message, StrVar);
```

Тут, Caption і Message - відповідно, текст заголовку й текст повідомлення всередині вікна. StrVar - змінна рядкового типу, що повинна бути оголошена заздалегідь. Якщо в цій змінній є текст, він виводиться в рядку редагування як текст за замовчуванням. Якщо змінна порожня, то порожнім буде й рядок. Якщо користувач щось введе в рядок і натисне кнопку «**ОК**», цей текст запишиться в змінну StrVar, а функція поверне значення True (Істина). У протилежному випадку функція поверне значення False (Хиба). Переробимо код налагоджувача:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  YourName: String;
begin
  YourName:= 'Невідомий';
  if InputQuery('Хто ви?', 'Укажіть ваше ім'я', YourName) then
     ShowMessage('Привіт, ' + YourName + '!');
end;
```

Нагадаю, що з логічною конструкцією if ми познайомимося в наступній лекції, а поки розберемо код. Спочатку ми привласнюємо

рядковій змінній YourName текст «Невідомий». Потім ми викликаємо функцію InputQuery(). Ми вказуємо заголовок вікна «Хто ви?», текст повідомлення «Вкажіть ваше ім’я» і текстову змінну YourName. Текст, що ми раніше помістили в цю змінну, буде виходити в рядку редагування:

Рис. 6.4. *Результат роботи функції InputQuery()*

Якщо користувач введе якийсь інший текст, він потрапить у змінну YourName, функція поверне True і в результаті буде виведено повідомлення з вітанням

```
ShowMessage(‘Привіт, ‘ + YourName + ‘!’);
```

Інакше вітання не буде виведено. Функція InputQuery() часто буває корисна, так що запам’ятаєте її.

Тема 7. Логічні типи, конструкції й компоненти

1.Логічний тип даних

На попередніх лекціях ми вже зустрічалися з таким типом - деякі властивості компонентів могли приймати тільки одне із двох значень: або True (Істина), або False (Хиба). Це і є **логічний тип даних**. У Паскалі це тип Boolean, що займає один байт оперативної пам’яті. В Lazarus, крім того, є типи ByteBool (теж 1 байт), WordBool (2 байти) і LongBool (4 байти). Всі ці типи можуть приймати лише одне із двох значень: true або false. На практиці, щоправда, таке різноманіття не застосовується, програмісти цілком обходяться тільки одним стандартним типом boolean. Ви можете створити змінну такого типу й привласнити їй true або false. Крім того, нерідко зустрічаються вирази, які в результаті теж дають або Істину, або Хибу, вони також

вважаються логічними. Наприклад, якщо ви привласните логічній змінній вираз 3>2, то в змінну потрапить True, тому що три дійсно, більше двох.

Отже, що ж можна робити з логічними даними, крім присвоювання їм значень True або False? Їх можна порівнювати між собою, використовуючи для цього оператори порівняння:

= **(Дорівнює)**
<> **(Не дорівнює)**
> **(Більше)** - Тут варто мати на увазі, що компілятор, насправді, замість False записує двійкове значення 0, а замість True - 1. Таким чином, True завжди буде більше, ніж False.
< **(Менше)**
>= **(Більше або дорівнює)**
<= **(Менше або дорівнює)**

Операції порівняння можна ускладнити, застосувавши наступні додаткові логічні оператори:

- NOT (Логічне **НЕ**)
- AND (Логічне **И**)
- OR (Логічне **АБО**)
- XOR (Логічне, що виключає **АБО**).

Ці оператори порівнюють між собою два логічних значення (операнда) і залежно від цих значень повертають як результат true або false. Наступна таблиця демонструє результат цих операцій (порівнюються логічні змінні a і b):

Таблиця 7.1. Логічні операції					
A	**B**	**not A**	**A and B**	**A or B**	**A xor B**
true	true	false	true	true	false
false	true	true	false	true	true
true	false	false	false	true	true
false	false	true	false	false	false

Давайте розберемо дії цих операцій детальніше.

Операція NOT (НЕ), на відміну від інших логічних операцій, працює тільки з одним операндом. Це – операція - перевертиш: якщо операнд містить True, те повертає результат, що, буде False, і навпаки. Цю операцію дуже часто застосовують для того, щоб поміняти значення логічної змінної (або властивості) на протилежне, наприклад,

A:= not A;

Операція AND (И) повертає True тільки в тому випадку, якщо обидва порівнюваних операнда містять True. Якщо перший операнд містить False, подальша перевірка вже не проводиться (сенсу немає), і AND повертає False. Якщо перший операнд містить True, то AND робить перевірку другого операнда - якщо він також містить True, тоді повертається результат, що, буде True, інакше повертається False.

Операція OR (АБО) діє схожим образом, але повертає True у тому випадку, якщо хоч один з операндів містить True. OR перевіряє перший операнд. Якщо він True, подальша перевірка вже не виконується, і OR повертає True. Інакше OR перевіряє другий операнд - якщо він True, тоді повертається True, у протилежному випадку повертається False.

Операція XOR (яка виключає АБО) завжди перевіряє обидва операнда, і повертає True тільки тоді, коли один з операндів True, а інший обов'язково False. Правда, на практиці XOR звичайно не використовують, оскільки завжди простіше виконати перевірку A <> B, що повертає тотожний результат.

Отже, приклади роботи з логічними даними (в Lazarus їх виконувати не потрібно):

```
var
  A, B, C: Boolean;
begin
  A:= True;  //результат - True
  B:= not A;  //результат - False
  C:= A and B;  //результат - False
  C:= A or B;  //результат - True
  C:= A <> B;  //результат - True
  C:= 3 > 5;  //результат - False
  ...
```

2. Керуюча конструкція IF

Від логічного типу даних було б мало користі, якби ми не могли застосовувати різні дії залежно від результатів перевірки - якщо в змінній перебуває Істина, тоді виконати один код, інакше виконати інший код. Саме таку можливість надає керуюча конструкція IF.

Найпростіший синтаксис цієї конструкції такий:

if Умова then Дія;

Англійське IF перекладаться як ЯКЩО, а THEN - ТОДІ. Тут все доволі зрозуміло: якщо якийсь умовний вираз (логічна змінна або логічна властивість компонента) має значення True, тоді виконується зазначена дія. Тепер ми можемо вивчати приклади на практиці. Завантажте **Lazarus** з новим проектом. Посередині вікна встановіть кнопку **TButton**. Збережіть проект у папку **07-01** під ім'ям **MyBool**; не забудьте головну форму назвати **fMain**, та збережіть модуль, Потім згенеруйте подію натисканням на кнопку, що оформиться в такий спосіб:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  b: boolean;
begin
  b:= True;
  if b then ShowMessage('Істина');
end;
```

Тут все гранично просто: повідомляємо логічну змінну b, потім привласнюємо їй значення True (Істина). У рядку

if b then ShowMessage('Істина');

ми перевіряємо: якщо b має значення True (тобто, якщо b дійсне), тоді ми виводимо повідомлення «Істина». Збережіть й запустіть проект. Натискання на кнопку викличе підсумок повідомлення. Можете для прикладу привласнити b значення False - у цьому випадку натискання на кнопку не викличе ніякого повідомлення.

До речі, у цьому випадку як умову ми вказали просто ім'я змінної b. Адже вона вже містить значення True! Так що рядок

```
if b then ShowMessage('Істина');
```

буде повністю аналогічний рядку з повною умовою

```
if b = True then ShowMessage('Істина');
```

Якщо ми будемо розширювати умову за допомогою умовних операцій, то всі операнди варто взяти в дужки. Виключення становить операція NOT, що працює тільки з одним операндом. Змінимо код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  a,b: boolean;
begin
  a:= False;
  b:= True;
  if (a = b) or b then ShowMessage('Істина');
end;
```

Тут ми перевіряємо вже дві умови. І якщо перше (a = b) поверне нам False, то друге b поверне True. Ми використовуємо операцію OR, нам досить, щоб хоч один з операндів був True. Виходить, повідомлення буде виконано. Якби в другому операнді ми порівнювали b ще із чимсь, то його теж варто було б помістити в дужки. Наприклад,

```
if (a = b) or (b > a) then ShowMessage('Істина');
```

У цьому прикладі повідомлення «Істина» буде виведено, тому що в другому операнді b дійсно більше, ніж a. Якщо ви забудете про дужки, компілятор не стане вказувати вам на помилку, і програма буде запущена. Однак результат може виявитися невірним. Наприклад, якщо ми заберемо дужки в першому прикладі:

```
if a = b or b then ShowMessage('Істина');
```

То повідомлення «Істина» не буде виконано. Адже оператор IF робить тільки одну перевірку, виходить, він перевірить тільки a = b.

Частина умови з OR буде пропущена, тому що ми не помістили операнди в дужки. Тому якщо ви в умові застосовуєте операції AND, OR або XOR, не забувайте вміщати операнди в дужки. Дужки можна не використовувати, якщо в якості операндів ви вказуєте по одній умовній змінній або властивості, наприклад, код:

```
if a  or b then ShowMessage('Істина');
```

виведе повідомлення «Істина», тому що в b утримується True.

Керуючий оператор IF не даремно називають конструкцією, його синтаксис може складатися з декількох частин:

```
if Умова then Дія1
else Дія2;
```

Англійське ELSE перекладається як ІНАКШЕ. Тут, ЯКЩО будь яка умова поверне True, то буде виконана Дія1, ІНАКШЕ буде виконана Дія2. Зверніть увагу, що перед ELSE крапку з комою НЕ СТАВЛЯТЬ! Змінимо код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  b: boolean;
begin
  b:= True;
  if b then ShowMessage('Істина')
  else ShowMessage('Хиба');
end;
```

Натискання на кнопку викличе повідомлення «Істина», однак якщо потім ви виправити код, і в змінну b замість True помістити False, то вийде повідомлення «Хиба» - перевірте це самі.

Однак конструкція IF може бути й складнішою. Якщо нам потрібно перевірити не одне, а дві умови? Або більше? Повний синтаксис цієї конструкції такий:

```
if Умова1 then Дія1
else if Умова2 then Дія2
...
```

```
else if Умова then Дія
else Діяінакше;
```

Давайте розбиратися. У цій конструкції, якщо Умова1 поверне True, то буде виконано Дію1, і на цьому конструкція закінчиться. Якщо ж воно поверне False, перевірка буде продовжена. Якщо наступна Умова2 поверне True, то буде виконано Дію2, і так далі, подібних перевірок може бути скільки завгодно. Якщо ж всі перевірки умов повернуть False, то буде виконан Діюінакше. Змінимо код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  a,b: boolean;
begin
  a:= False;
  b:= True;
  if a then ShowMessage('Дія №1')
  else if b then ShowMessage('Дія №2')
  else if a or b then ShowMessage('Дія №3')
  else ShowMessage('Дія №4');
end;
```

Як ви думаєте, яке повідомлення вийде в цьому випадку? І скільки повідомлень вийде? Якщо ви сказали, що вийде тільки одне повідомлення «Дія №2», то ви засвоїли конструкцію **IF**. Інакше раджу повернутися назад, й уважніше вивчити все вищезазначене.

3. Дужки оператора BEGIN...END

Логічний оператор IF, як і багато інших операторів, які нам ще має тотрібно вивчити, виконує тільки одна дію. Глянемо на наступний код:

```
if a then
   ShowMessage('Дія №1');
   ShowMessage('Дія №2');
   ShowMessage('Дія №3');
```

У цьому прикладі повідомлення «Дія №1» вийде тільки в тому випадку, якщо a = True. Інші повідомлення вийдитимуть кожного разу, тому що вони до оператора IF уже не відносяться, а виконуються як

самостійні оператори. Однак що робити, якщо в рамках конструкції IF нам потрібно виконати не один оператор, а цілий блок коду? У такому випадку, цей блок поміщають в дужки оператора begin...end, у результаті чого блок операторів виконується, як єдиний оператор. Змінимо код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  a: boolean;
begin
  if a then begin
    ShowMessage('Дія №1');
    ShowMessage('Дія №2');
    ShowMessage('Дія №3');
  end; //кінець if
end;
```

У даному прикладі всі три повідомлення розміщено в дужках begin...end і виконуються, як один оператор. Виходить, вони всі будуть виведені, тільки якщо a = True, інакше не буде виведено жодне повідомлення.

Тут варто сказати декілька слів про стиль програмування. Єдиних стандартів тут немає, є тільки рекомендації.

По-перше, код повинен бути зручним для читання. Порівняймо:

```
if a then ShowMessage('Дія №1') else ShowMessage('Дія №2');
```

Синтаксис тут правильний, компілятор помилок не знайде. І код цей буде виконаний вірно, однак як же його складно читати! Всі частини конструкції IF зливаються в один рядок, відразу й не розбереш, що тут до чого, але ж конструкція ще дуже проста! Що, якби ми застосовували кілька операцій AND або OR, так ще з безліччю операндів? Зовсім інакше виглядає такий код:

```
if a then
ShowMessage('Дія №1')
```

```
else
ShowMessage('Дія №2');
```

Тут конструкція розбита на кілька невеликих логічних частин, кожній частині відведено окремий рядок. Код став явно більш зрозумілим, та зручним для читання. Однак його можна ще поліпшити, якщо змістити вкладені оператори вправо на 2-3 пробіли, щоб показати залежності:

```
if a then
  ShowMessage('Дія №1')
else
 ShowMessage('Дія №2');
```

Гарний програміст саме так і оформлює свій код. Отут відразу видно, що від чого залежить, конструкція стає зрозумілішою.

По-друге, якщо ви застосовуєте дужки оператора, то в навчальній літературі ви можете зустріти два способи оформлення коду. Такий:

```
if a then begin
 ShowMessage('Дія №1');
 ShowMessage('Дія №2');
 ShowMessage('Дія №3');
end; //кінець if
```

і такий:

```
if a then
begin
 ShowMessage('Дія №1');
 ShowMessage('Дія №2');
 ShowMessage('Дія №3');
end; //кінець if
```

Обидва цих способи вважаються класичними, обидва добре виглядають, який з них застосовувати - ваш вибір. Краще переносити BEGIN на окремий рядок (хіба що рядок виходить занадто довгим),

тому віддаємо перевагу першому способу. Ви, якщо маєте бажання, можете вибрати й другий варіант, він теж правильний.

4. Прапорці та радіокнопки

У програмах, особливо в налаштуваннях, нерідко використовуютьють прапорці (які інакше називають галочками) та радіокнопки. Прапорці TCheckBox мають логічну залежність ввімкнений/вимкнений, тому вивчимо їх у цій лекції. Радіокнопки **TRadioButton** також мають таку залежність. І прапорець, і радіокнопка перебувають на вкладці **Standard Палітри компонентів**:

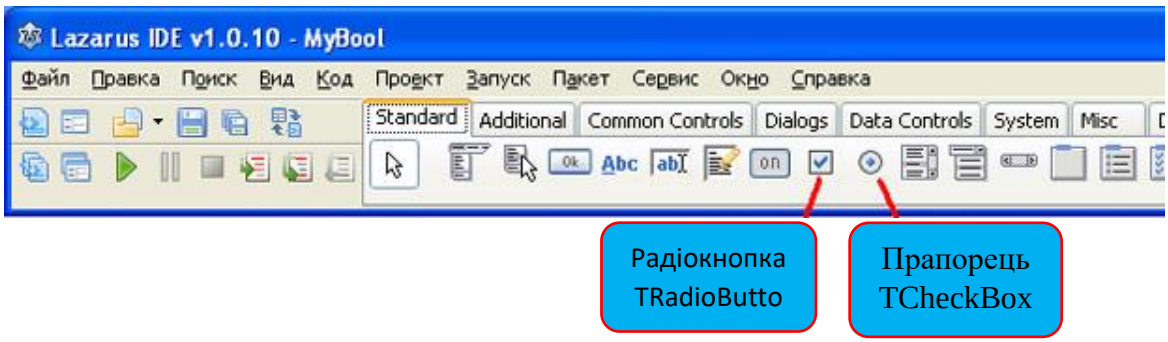

Рис. 7.1. *TCheckBox і TRadioButton*

Встановіть на форму один прапорець. Подивимося на його властивості.

Name задає ім’я компонента. Caption - пояснювальний текст праворуч від прапорця. Найважливіша властивість для нас - Checked (англ. checked - перевірений). Вона має логічний тип. Якщо в цій властивості значення True, то прапорець включений, інакше він виключений. Значення міняється автоматично, хоча його можна змінити й програмно.

Властивість Name змінимо на Ch1, щоб простіше до неї звертатися. У властивості Caption вкажіть текст

Наш прапорець

Ви можете помітити, що по мірі введення тексту у властивість Caption ширина компонента збільшується, щоб вмістився весь текст. Тепер змінимо код обробки натискання на кнопку:

```
procedure TfMain.Button1Click(Sender: TObject);
begin
  if Ch1.Checked then
    ShowMessage('Наш прапорець включений!')
  else
    ShowMessage('Наш прапорець виключений!');
end;
```

Ви й без коментарів повинні зрозуміти, що якщо у властивості Checked нашого прапорця Ch1 утримується значення True, то вийде одне повідомлення, інакше - інше. Збережіть проект і запустіть його на виконання. Вмикайте або вимикайте прапорець, після чого натискайте кнопку. Повідомлення скаже, ввімкнений він чині.

State (стан) - ще одна корисна властивість прапорця. Ця властивість визначає первісне значення прапорця, крім того, можна перевірити його й у процесі роботи програми. Може мати наступні значення:

- cbChecked - ввімкнений
- cbGrayed - невизначений (від англ. gray - сірий)
- cbUnchecked - вимкнений

Ще нас можуть цікавити дві події, які доступні на вкладці **Події Інспектори об'єктів**.

OnChange - При зміні. Подія виникає щораз, коли користувач вмикає або вимикає прапорець. Ви можете згенерувати цю подію так само, як і подію натискання на кнопку - двічі клацнувши по ній в **Інспекторі об'єктів**. Потім можете скопіювати код з події кнопки, а подію кнопки очистити:

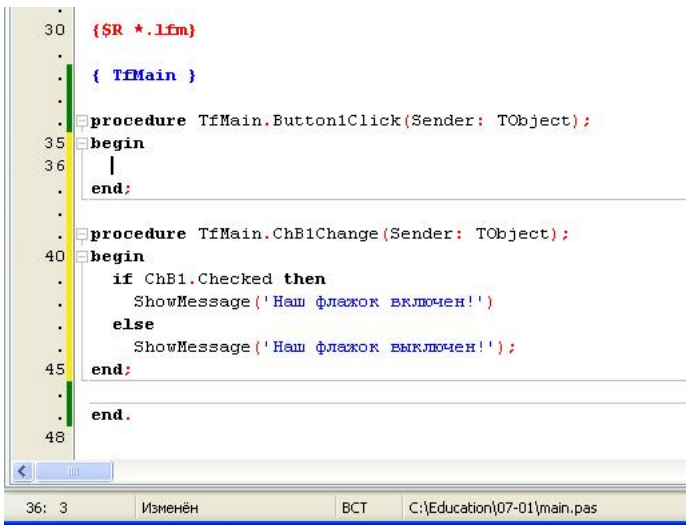

Рис. 7.2. *Подія OnChange прапорця*

Тепер повідомлення будуть виходити відразу ж при зміні стану прапорця. Кнопка ж ніяких дій виконувати не буде. Однак не поспішайте її видаляти, вона ще знадобиться.

OnClick - подія також виникає, коли користувач клацне мишею по компоненту, ввімкнувши прапорець, або вимкнувши його.

Різниця між цими подіями в тому, що OnChange буде виконаний у кожному разі - якщо користувач мишею ввімкнув/вимкнув прапорець, або якщо ми зробили це програмно, наприклад, вписавши в подію кнопки Button1Click (між begin і end) код:

```
Ch1.Checked:= not Ch1.Checked;
```

Тоді при натисканні на кнопку стан прапорця зміниться на протилежний - спрацював наш перевертень NOT. OnChange при цьому спрацює, а OnClick - ні, адже мишею по компоненту не клацали!

Тепер займемося радіокнопками. Якщо прапорець можна використати в одиничному варіанті, то радіокнопок повинно бути як мінімум дві. Радіокнопки дозволяють користувачеві вибрати один з можливих варіантів, але для вибору однієї такої кнопки буде явно недостатньо.

Давайте зімітуємо вибори президента, встановивши 5 таких радіокнопок одну під одною. Властивості Name цих кнопок перейменуємо, відповідно, в RB1, RB2, RB3, RB4 і RB5. Ну, а у властивостях Caption цих кнопок запишемо, відповідно

Порошенко
Янукович
Кравчук
Кучма
Ющенко

Тепер нам потрібно включити одну із цих радіокнопок. Оскільки нинішній президент у нас Порошенко, переведемо в True властивість Checked радіокнопки RB1.

Тепер змінимо код натискання на кнопку:

```
procedure TfMain.Button1Click(Sender: TObject);
begin
  if RB1.Checked then
```

```
    ShowMessage('Ви вибрали Порошенка')
  else if RB2.Checked then
    ShowMessage('Ви вибрали Януковича')
  else if RB3.Checked then
    ShowMessage('Ви вибрали Кравчука')
  else if RB4.Checked then
    ShowMessage('Ви вибрали Кучму')
  else
    ShowMessage('Ви вибрали Ющенка');
end;
```

Отут все доволі прозоро, обробка радіокнопок практично не відрізняється від обробки прапорців. Єдине розходження: користувач не зможе відзначити декілька радіокнопок, тільки одну з них. А прапорці можуть бути включені всі, деякі, або жоден. Оскільки ми створюємо імітацію виборів, непогано б і у властивості Caption форми написати Вибори. Збережіть проект, запустіть на виконання й грайте на здоров'я у вибори президента.

5. Контейнери для прапорців та радіокнопок

Можна звичайно, користуватися й одиничними компонентами TCheckBox і TRadioButton, однак куди простіше використовувати для цього контейнери, особливо коли прапорців та радіокнопок багато. Для прапорців використовується контейнер TCheckGroup, а для радіокнопок - TRadioGroup. Обидва компоненти розташовуються на вкладці **Standard Палітри компонентів**. Щоб не заморочуватись зі зміною дизайну й коду нашого проекту, буде простіше цей проект закрити, і створити новий.

Властивість Name форми перейменуйте відразу в fMain, у властивості Caption впишіть текст

Вибори

Властивість BorderStyle має сенс перевести в bsDialog, а Position - в poDesktopCenter. Далі, збережіть проект під ім'ям **election** у папку **07-02**, модулю головної форми дайте ім'я **Main**.

Тепер можемо зайнятися контейнерами:

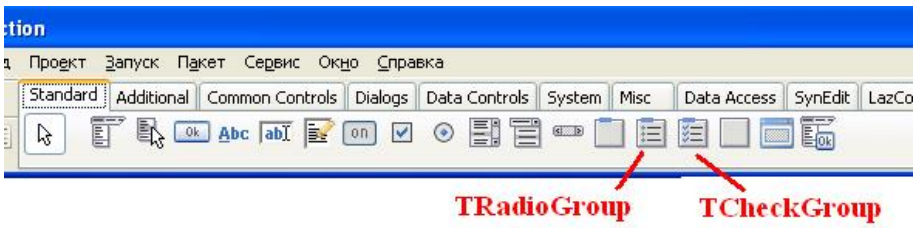

Рис. 7.4. *Контейнери для радіокнопок та прапорців*

Ліворуч на форму встановіть контейнер для радіокнопок TRadioGroup. Виділіть його, і змініть властивість Name на RG1. Властивість Caption контейнера формує загальний заголовок, впишемо там слово Кандидати.

Тепер зверніть увагу на властивість Items. Це складна властивість, містить текст всіх радіокнопок (набір рядків). Клацніть по кнопці «**…**»правіше властивості:

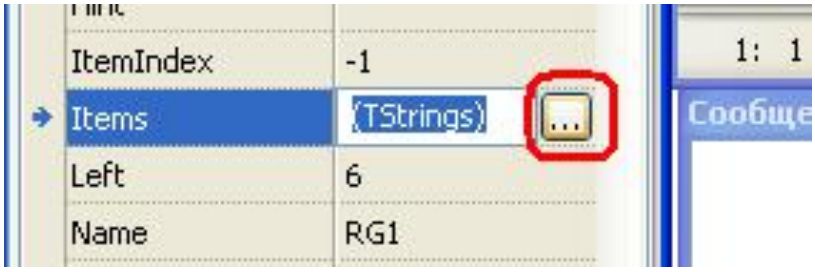

Рис. 7.5. *Кнопка Діалогу введення рядків*

Відкриється простий текстовий редактор, де ви можете вказати текст ваших радіокнопок. Текст кожної радіокнопки повинен розташовуватися на окремому рядку. Як і в минулому випадку, введіть такі рядки:

Порошенко
Янукович
Кравчук
Кучма
Ющенко

Зверніть увагу: якщо ви будете збільшувати висоту контейнера, відстань між отриманими радіокнопками буде збільшуватися. І навпаки. Підберіть найбільш оптимальну відстань між ними.

Правіше цього контейнера встановіть контейнер для прапорців TCheckGroup. Його властивість Name перейменуйте в Ch1, а у властивості Caption напишіть Міста. Цей контейнер також має складну властивість Items, що містить текст всіх прапорців. Клацніть по кнопці «**…**»правіше цієї властивості, і в **Діалозі введення рядків** запишіть такі рядки:

Київ
Харків
Львів
Одеса
Вінниця

У нашому проекті це будуть міста, що проголосували. Нижче контейнерів встановіть кнопку, у властивості Caption якої напишіть Результат.

Тепер, перш ніж ми приступимо до програмування події натискання на кнопку, розберемо ще дещо. Контейнер прапорців TCheckGroup має властивість Checked, що буде доступно при роботі програми. Це властивість логічного типу, вона вказує, чи ввімкнений прапорець. А щоб розібратися, який саме прапорець ввімкнений, після властивості, у квадратних дужках, вказують індекс прапорця. Індекси починаються з нуля, тому, щоб довідатися, чи ввімкнений перший прапорець, потрібно виконати код

```
if Chl.Checked[0] then ...
```

Не забувайте - індекси прапорців завжди починаються з нуля!

Контейнер радіокнопок TRadioGroup має властивість ItemIndex. Ця властивість вказує, яку саме радіокнопку користувач вибрав, індексація в них також починається з нуля. За замовчуванням, властивість дорівнює -1. Це означає, що ніяку радіокнопку не обрано. Таким чином, щоб виконати дію для першої радіокнопки, ми повинні перевірити, чи обрана вона:

```
if RGl.ItemIndex = 0 then ...
```

Щоб повторити попередній приклад, вкажемо 0 у властивості ItemIndex (тут індекси теж починаються з нуля) - радіокнопка **Порошенко** виявиться обраною за замовчуванням. Не забувайте після введення у властивості якихось значень натискати **<Enter>**, щоб підтвердити зміни.

Тепер, коли ми знаємо, як визначити, які прапорці включені і яка радіокнопка обрана, займемося програмуванням події натискання на кнопку.

```
procedure TfMain.Button1Click(Sender: TObject);
```

```
var
  s: String; //для формування рядка з містами, що проголосували
begin
  //спочатку вкажемо переможця:
  if RG1.ItemIndex = 0 then
    ShowMessage('Ви вибрали Порошенка')
  else if RG1.ItemIndex = 1 then
    ShowMessage('Ви вибрали Януковича')
  else if RG1.ItemIndex = 2 then
    ShowMessage('Ви вибрали Кравчука')
  else if RG1.ItemIndex = 3 then
    ShowMessage('Ви вибрали Кучму')
  else
    ShowMessage('Ви вибрали Ющенка');

  //тепер потрібно щось привласнити змінній s, щоб ініціалізувати
  //їй. привласнимо просто порожній рядок:
  s:= '';
  //тепер будемо збирати рядок s, залежно від включених прапорців
  //після кожного рядка будемо вставляти знак переходу на новий рядок
  if Ch1.Checked[0] then s:= s + 'Київ проголосував' + #13;
  if Ch1.Checked[1] then s:= s + 'Харків проголосував' + #13;
  if Ch1.Checked[2] then s:= s + 'Львів проголосував' + #13;
  if Ch1.Checked[3] then s:= s + 'Одеса проголосувала' + #13;
  if Ch1.Checked[4] then s:= s + 'Вінниця проголосувала' + #13;

  //Якщо хоч один прапорець був ввімкнений, то рядок s не порожній.
  //покажемо його в цьому випадку:
  if s <> '' then ShowMessage(s);
end;
```

Коментарі вичерпні, яких то утруднень із кодом у вас виникнути не повинно. Єдине, помічу, що якщо не проініціалізувати спочатку

змінну s, тобто, не привласнити їй якесь значення, хоч і порожній рядок, у вікні повідомлень при компіляції може вийти нагадування:

Warning: Local variable «s» does not seem to be initialized

Справа в тому, що далі ми будемо намагатися використовувати значення рядка s:

… s:= s + ‘Москва проголосувала’ + #13;

А компілятор не знає, чи є там щось, або просто сміття? Помилки не буде, але однаково, змінні бажано спочатку ініціалізувати: літерним привласнювати порожній рядок, числовим - значення 0 (обнуляти). Логічні за замовчуванням самі приймають значення False.

Збережіть проект і запустіть його на виконання.

Тема 8. Числа

1. Цілі числа

В Lazarus (а точніше, в Free Pascal), як і в будь-якій іншій мові програмування, числа грають досить важливу роль. Важко уявити собі програму, у якій не використовувалися б числа. Навіть коли ви просто встановите якийсь компонент на форму, автоматично починають діяти безліч налаштувань. Left, Top, Height, Width - всі ці властивості є в будь-якому візуальному компоненті, і вони містять числа. Числа бувають цілі й дійсні, знакові й беззнакові.

У цьому розділі поговоримо про цілі числа, як знакові, так і беззнакові. Що таке **ціле число**? Це число без коми, тобто, без десяткової частини. Знаковим називають число зі знаком: -1, наприклад. Беззнакове число - це число від нуля й більше.

У програмуванні базовим цілим числом є integer, що ми вже не раз використовували. Але ви, ймовірно, здогадалися, що це не єдиний можливий цілий тип? Є різні типи цілих чисел, вони можуть бути зі знаком і без нього, мають різний діапазон можливих значень і, відповідно, займають різний розмір оперативної пам’яті. Розберемо ці типи:

Таблиця 8.1. Цілі числа

Тип	Діапазон	Розмір у байтах
Byte	0..…255	1
ShortInt	-128..…127	1

Word	0...65…65535	2
Smallint	-32 768…32767	2
LongWord	0...4…4294967295	4
Cardinal	0...4…4294967295	4
LongInt	-2 147 483 648…2147483647	4
Integer	-2 147 483 648…2147483647	4
Int64	$-2^{63} \ldots 2^{63}$	8

Зверніть увагу, тут діапазон і розмір Integer збігається з LongInt. Взагалі ж, це залежить від режиму компілятора **FPC**. Проект можна скомпілювати в різних режимах, з підтримкою Delphi, наприклад, або TP (Turbo Pascal). За замовчуванням, виставлений режим **Object Pascal**, це можна перевірити, виконавши в середовищі **Lazarus** команду меню **Проект -> Параметри проекту**, потім у розділі **Параметри компілятора** вибрати **Обробка**. У верхній частині там зазначений **Режим синтаксису**, за замовчуванням це **Object Pascal**, але при необхідності його можна й поміняти.

Отож, якщо там виставлений режим **Object Pascal** або **Delphi**, тоді Integer має розмір 32 біта, або 4 байти. Якщо ж виставлений старий режим **Turbo Pascal** або **Free Pascal**, то Integer буде мати розмір в 16 біт або 2 байти, і буде відповідати типу Smallint.

Навіщо потрібно така розмаїтість цілих типів? У колишні часи оперативна пам’ять була досить маленькою. У ті часи програмісти боролися за кожний байт пам’яті, переписуючи й мінімізуючи код, вибираючи самі маленькі з можливих типи даних. Це називалося **оптимізацією коду**. Припустимо, вам потрібно виконати якийсь цикл 10 разів. Для підрахунку кроків циклу вам доведеться створити змінну цілого типу. Але навіщо використовувати змінну Integer в 4 байти, коли цілком можна обійтися однобайтовим Byte? Зараз звичайно, це не грає такої великої ролі, як колись, але однаково, оптимізація коду - це ознака гарного програміста, це гарний тон у програмуванні. Так що намагайтеся не витрачати дарма зайву пам’ять.

Рекомендації тут наступні: якщо ви знаєте, що число буде без знаку, то й обирайте беззнакові типи. Якщо ви точно знаєте, що максимальне число в змінній буде маленьким, вибирайте типи менші. Якщо вам невідомо, якого розміру число потрапить у змінну, то вибирайте Integer - це універсальний тип, придатний для більшості випадків. Ну а якщо ви впевнені, що число буде дуже великим, то використовуйте 4-х або навіть 8-ми байтові типи.

2. Дійсні числа

Дійсними називаються числа із дробовою частиною, причому, якщо дробова частина дорівнює нулю, її однаково потрібно вказати. Наприклад:

3.5

-10.427

8.0

Такі числа ще називають **числами із плаваючою крапкою**, оскільки кількість цифр після крапки може бути різною. Записуються дійсні числа за певними правилами. Якщо в математиці ми дробову частину відокремлюємо комою, в Lazarus для цього використовують крапку. Якщо число дуже велике можна вибрати скорочену форму. Якщо в математиці для цього число множать на десятковий степінь, наприклад,

$0,25 * 10^5$

то в Lazarus замість 10 вказують букву E (від англ. exponent - показник степеня):

$0.25E5$

Степені можуть бути й негативними:

$0.7E-23$

Речовинних типів теж багато. У характеристиці речовинних чисел роль грає не тільки розмір, займаний у пам’яті, але й кількість значущих цифр:

Таблиця 8.2. Дійсні числа

Тип	Діапазон	Кількість значущих цифр	Розмір у байтах
Single	1.5E-45…3...4E38	7-8	4
Real	5.0E-324…1...7E308	15-16	8
Double	5.0E-324…1...7E308	15-16	8
Comp	-2E64+1...2…2E63-1	19-20	8
Currency	-922 337 203 685 477.5808 … 922 337 203 685 477...5807	19-20	8
Extended	1.9E-4932…1...1E4932	19-20	10

Як бачимо, дійсні числа куди більше цілих, процесорного часу на обробку таких чисел витрачається теж більше. Тому дійсні числа має сенс застосовувати тільки по необхідності, коли цілими числами явно не обійтися. Не слухайте тих, хто пропонує на всі випадки життя використати тип Real - і для цілих, і для дійсних чисел.

Рекомендації тут такі ж, як і для цілих чисел - вибирайте типи по необхідності. Особливо виділю тип Currency - його створили спеціально для фінансових розрахунків, тому для всякого роду бухгалтерських розрахунків краще вибирати саме цей тип, як найбільш точний. Але найчастіше обходяться типом Real (або Double).

3. Операції над цілими та дійсними числами

Цілі числа можна додавати (+), віднімати (-) і множити (*) один на одного. З діленням справа складніша. Припустимо, нам потрібно 10 розділити на 3. Вийде 3,33333..., а це вже не ціле число. Тому для цілих чисел у Паскалі передбачено ділення націло. Операція div забезпечує длілення націло, і повертає цілу частину, відкидаючи дробову. Наприклад, 10 розділити на 8 буде дорівнювати 1,25. Якщо застосувати ділення націло, то 10 div 8 = 1. Щоб дізнатися залишок від такого ділення, застосовують операцію mod. 10 mod 8 = 2.

Арифметика над дійсними числами ще простіша, тут застосовують наступні стандартні операції: + (додавання), - (віднімання), * (множення), / (ділення).

Крім того, як цілі, так і дійсні числа можна порівнювати між собою, використовуючи для цього логічні оператори: = (дорівнює), <> (не дорівнює), > (більше), < (менше), >= (більше або дорівнює), <= (менше або дорівнює).

Дуже часто доводиться використовувати більші й складні вирази, де разом з арифметичними використовуються й логічні оператори. Тут головне - не забувати про пріоритети. Візьмемо вираз

```
r := 3 + 4 * 2;
```

Що потрапить у змінну r? Якщо ви відповіли 11, то ви праві. Щоб спочатку виконати додавання, його потрібно помістити в дужки, які мають вищий пріоритет:

```
r := (3 + 4) * 2;
```

У цьому випадку, у змінну r потрапить число 14.

Таблиця 8.3. Пріоритети

Порядок	Операції

1	( ) - Те, що в круглих дужках, обчислюється в першу чергу
2	NOT
3	*, /, DIV, MOD, AND
4	+, -, OR, XOR
5	=, <>, >, <, >=, <=

Примітка: Якщо у виразі зустрічаються операції однакового пріоритету, вони виконуються ліворуч - праворуч.

Примітка: Ні ціле, ні дійсне число НЕ МОЖНА ДІЛИТИ НА НУЛЬ! Якщо таке відбудеться, виникне Виняткова ситуація, і буде виведена помилка. Програмісти звичайно завжди перевіряють, що й на що користувач ділить. Якщо він намагається ділити на нуль, виводиться зрозуміле повідомлення про помилку, а саме ділення не виконується. Це називається «захист від дурня»!!!

4. Перетворення типів

Нерідко виникають ситуації, коли потрібно перетворити один тип даних в інший. Наприклад, ціле число в дійсне або в рядок, і назад.

Ціле число можна відразу ж привласнити дійсному - перетворення відбудеться автоматично. Наприклад,

```
Var

  r: real;
begin
  r:= 3;  //результат: 3.0
  ...
```

Якщо є бажання, ви можете діяти аналогічнояк у минулих лекціях: створити новий проект із однією кнопкою, згенерувати для кнопки подію натискання, і всі приклади тестувати в цій події. Розділ змінних і початок події begin зазначені, не забувайте наприкінці події залишити end;

Дійсне число привласнити цілій змінній не можна - компілятор видасть помилку. У цьому випадку ми можемо діяти двома способами: або просто відкинути дробову частину, або заокруглити дійсне число до найближчого цілого. Математична функція Trunc() відтинає дробову частину, і повертає ціле число:

```
var
```

```
  i: integer;
begin
  i:= Trunc(3.65); //результат = 3
  ...
```

Математична функція Round() заокруглює дійсне число до найближчого цілого:

```
var
  i: integer;
begin
  i:= Round(3.65); //результат = 4
  ...
```

Якщо ціле число потрібно перетворити в рядок, використовують функцію IntToStr(), нам уже доводилося це робити. У дужках вказують ціле число будь-якого цілочисельного типу, а функція повертає це ж число у вигляді рядка. Приклад:

Для зворотного перетворення використовують функцію StrToInt(), програміст var

```
  i: integer;
  s: string;
begin
  i:= 10;
  s:= IntToStr(i); //тепер в s рядок '10'
  ShowMessage(s);  //вміст s можна вивести на екран
  ...
```

має стежити, щоб у цю функцію потрапляло дійсно ціле число у вигляді рядка. Приклад:

```
var
  i: integer;
  s: string;
begin
  s:= '35';  //завантажили число у вигляді рядка
```

```
i:= StrToInt(s);  //перетворили рядок '35' у число 35
i:= i * 2;  //тепер тут 70
s:= IntToStr(i);  //одержали рядок '70'
ShowMessage(s);  //вміст s виводимо на екран
...
```

Якщо рядок потрібно перетворити в дійсне число, використовують функцію StrToFloat():

```
var
  r: real;
  s: string;
begin
  s:= '3,14';
  r:= StrToFloat(s);
```

Зверніть увагу - тут у рядкову змінну ми внесли дійсне число у вигляді рядка. При цьому десяткову частину ми розділили не крапкою, а комою: «3,14». Так заведено вносити числа в операційній системі, якщо встановлено російську версію Windows. В англійській версії роздільником є крапка. Дане налаштування залежить від глобальної змінної DecimalSeparator. Змінну повідомляти не потрібно - вона вже оголошена, і містить символ системного роздільника дробової частини.

Для перетворення дійсного числа в рядок можна використовувати дві функції. Найпростіше скористатися функцією FloatToStr():

```
var
  r: real;
  s: string;
begin
  r:= 3.14;  //тут ми вводимо число, а не рядок, тому роздільник - крапка
  s:= FloatToStr(r); //тепер в s рядок '3,14'
  ShowMessage(s);
```

Якщо ми спробуємо ввести в дійсну змінну r число, використовуючи як роздільник кому, то буде помилка. Числа потрібно

розділяти крапкою, а числа у вигляді рядка – комою (у російськомовній версії Windows).

Але найкраще для перетворення числа в рядок використовувати функцію FormatFloat(). Ця функція не тільки перетворить дійсне число в рядок, але й виведе його в потрібному форматі. Бажаний формат задається маскою - рядком зі спеціальними символами. Синтаксис функції:

FormatFloat(Формат, Значення);

Формат - це рядок з маскою, Значення - вихідне дійсне число. Функція повертає як результат зазначене дійсне число у вигляді рядка, відформатованого відповідно до маски.

Таблиця 8.4. Символи формату

Символ	Опис
0	У дану позицію записується цифра. Якщо у вихідному числі в даній позиції була цифра - записується вона. Інакше буде записаний ‘0’.
#	У дану позицію записується цифра. Якщо у вихідному числі в даній позиції була цифра - записується вона. Інакше в дану позицію нічого не записується.
.	Даний символ, що зустрічається в масці перший раз, визначає розташування десяткового роздільника. Будь який наступний символ ‘.’ ігнорується. Символ, що буде використаний як десятковий роздільник у результуючому рядку, зберігається в глобальній змінній DecimalSeparator.
,	Якщо маска містить даний символ, то у відформатованому рядку, між кожною групою із трьох цифр ліворуч від десяткової крапки, будуть вставлені роздільники тисяч. Положення й кількість символів ‘,’ у масці на результат не впливає. Символ, що буде використаний як роздільник тисяч, визначається глобальною змінною ThousandSeparator.
E+	Експонентний формат. Якщо кожна з рядків: ‘E+’, ‘E-’, ‘e+’, ‘e-’, утримується в рядку формату, то результуюче значення буде представлено в експонентному вигляді. За одним із цих рядків можуть випливати символи ‘0’, що визначають мінімальне число цифр в експоненті.

;	Відокремлює в масці варіанти форматування для додатнього, від'ємного й нульового значення.

Примітка: Якщо в масці після коми мають вийти дві цифри, а в дійсному числі їх більше, дробова частина буде заокруглена.

Приклад:

```
var
  r: real;
  s: string;
begin
  r:= 123.789;
  s:= FormatFloat('#.##', r);
  ShowMessage(s);  //результат - '123,79' - дробова частина округлена
  r:= 789.123;
  s:= FormatFloat('#.##', r);
  ShowMessage(s);  //результат - '789,12'
```

Для грошових розрахунків можна застосовувати маску '#.00' - це гарантує дві цифри після коми. Якщо цифр немає, будуть виведені нулі, наприклад

123,00

5. Практичне застосування

Тепер, коли ви знаєте про цифри майже все, саме час закріпити матеріал на практиці. Ми напишемо програму, що буде чи визначати нормальна у нас вага, надлишкова, або навпаки, недостатня.

Загальне визнання одержав так званий індекс маси тіла (ІМТ). Його розрахунок такий: розділіть свою вагу в кілограмах на ріст у метрах у квадраті. Приклад: ІМТ = 68 кг / (1,72 м * 1,72 м) = 23. Ця формула гарна тим, що працює й для «малят», і для «гуліверів», і для жінок, і для чоловіків. Нормою вважається ІМТ від 19 до 25. ІМТ менше 19 - дефіцит ваги, 25-30 - надлишкова вага, 30-40 - ожиріння, більше 40 - сильне ожиріння.

Завантажите **Lazarus** з новим проектом. Якщо він у вас уже завантажений - закрийте проект і створіть новий. Збережіть проект у

папку **08-01** під ім'ям **IMT**, модуль головної форми назвіть **Main**, а властивість Name форми перейменуйте в fMain. Зробимо попередні налаштування форми. У властивості Caption напишіть:

IMT - Індекс маси тіла

У властивості BorderStyle встановіть bsDialog, а в Position - poDesktopCenter.

Від користувача нам потрібно отримати вагу й зріст, причому зріст має бути дійсним числом, а вага - цілим. Наприклад, вага 68, ріст 1,72. Щоб гарантувати правильність введення, застосуємо компонент TMaskEdit із вкладки **Additional Палітри компонентів**. Нам знадобляться два таких компоненти, дві мітки TLabel і одна кнопка для підсумку результатів. Оформіть форму за схожою схемою:

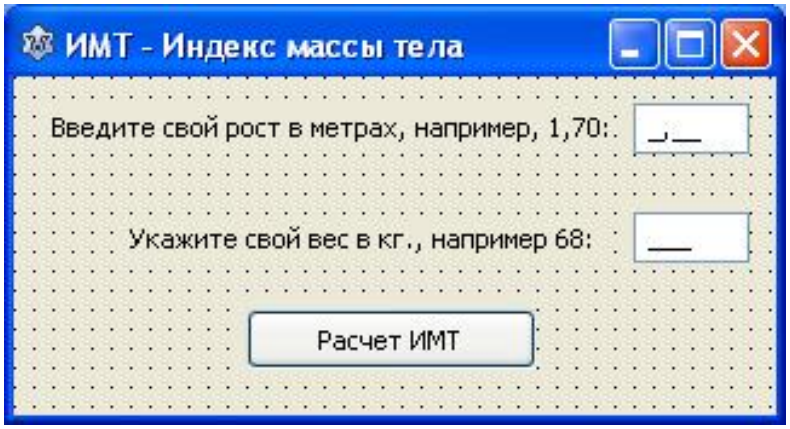

Рис. 8.1. *Зовнішній вигляд програми IMT*

Змініть властивість Name верхнього TMaskEdit на ME1, нижнього - на ME2. Імена інших компонентів можна залишити за замовчуванням. Нам ще знадобиться змінити властивості EditMask обох компонентів. Для ME1 встановіть EditMask:

0,00;1;_

Це нам гарантує правильне введення користувачем свого зросту. Однак не забувайте, що насправді, користувач буде вводити не число, а рядок! Приміром, він введе «1,82» - це не дійсне число, як можна було б подумати, а рядок із цифрових символів. І нам ще потрібно перетворити її в дійсне число. Для ME2 маска буде

###;0;_

З огляду на те, що вага може бути й тризначним цілим числом.

Тепер нам залишилося тільки зробити розрахунок. Правда, поки ми не будемо робити перевірку - чи ввів взагалі користувач в ME1 і ME2 що-небудь? Залишимо це на совісті користувача. Про те, як робити «захист від дурнів», ми ще поговоримо в одній з наступних лекцій.

Згенеруйте подію натискання на кнопку, і оформіть її в такий спосіб:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  s: string; //для формування звіту
  rost: real; //для одержання зросту
  ves: Byte; //для одержання ваги
  imt: real; //для розрахунку ІМТ
begin
  //спочатку перетворимо зріст із рядка в дійсне число:
  rost:= StrToFloat(ME1.Text);
  //тепер вага:
  ves:= StrToInt(ME2.Text);
  //тепер розраховуємо ІМТ:
  imt:= ves / (rost * rost);
  //залежно від результату формуємо рядок звіту:
  s:= 'Ваш ІМТ = ' + FormatFloat('#.##', imt) + #13;
  if imt < 19 then s:= s + 'У вас дефіцит ваги!'
  else if (imt >= 19) and (imt <= 25) then s:= s + 'У вас нормальна вага!'
  else if (imt > 25) and (imt <= 30) then s:= s + 'У вас надлишкова вага!'
  else if (imt > 30) and (imt <= 40) then s:= s + 'У вас ожиріння!'
  else if (imt > 40) then s:= s + 'Кошмар! У вас сильне ожиріння!'
  else s:= 'Щось пішло не так, результат не вдалося розрахувати';
  //виводимо результат на екран:
  ShowMessage(s);
end;
```

Програма вийшла невелика, але досить корисна. Коментарі тут досить докладні, щоб ви зрозуміли, що тут до чого.

Тема 9. Підпрограми

1.Підпрограми

Спочатку мови програмування були простіші, вони виконувалися чітко вниз, один оператор за іншим. Такі мови ще називали **лінійними**. Типовий приклад лінійних мов - Бейсик. Єдину можливість організувати хоч якусь логіку в таких мовах надавав оператор безумовного переходу GOTO, що залежно від умови, «перестрибував» на заздалегідь розставлені мітки. У сучасних мовах програмування GOTO теж залишився, напевно, для аматорів антикваріату. Але його застосування може привести до складних логічних часових помилок, які важко відслідкувати (run-time errors). Використання GOTO у сучасному програмуванні вважається дурним тоном. Ми не будемо вивчати цю можливість, оскільки для організації логіки є куди більш «просунуті» засоби! Одним з таких засобів є **підпрограми**.

Підпрограма - це частина коду, яку можна викликати з будь-якого місця програми невизначену кількість разів.

Інакше кажучи, підпрограми подібні до будівельних цеглинок, з яких, зрештою, виходить будинок - програма. Без підпрограм можна обійтися, якщо ви пишете невелику навчальну програму на парі десятків рядків коду. А якщо це серйозний додаток, з парою сотень модулів, у кожному з яких можуть бути тисячі рядків коду? Як таку програму написати, не розбиваючи завдання на окремі частини? Підпрограми допомагають поліпшити код, структурувати його. Тому мови високого рівня, які дозволяють використовувати підпрограми, називають ще **процедурно-орієнтованими** мовами. І наш компілятор FPC теж відноситься до таких мов.

Підпрограми бувають двох типів: **процедури** й **функції** - перші просто виконують свою роботу, другі ще й повертають результат цієї роботи.

2. Процедури

Насправді, ми вже неодноразово використовували процедури. Наприклад, коли генерували подію натискання на кнопку. Ця подія - процедура. Процедура починається із ключового слова procedure і має наступний синтаксис:

```
procedure <ім'я процедури>(<список параметрів>);
const
  <оголошення констант>;
type
  <оголошення нових типів>;
```

```
var
  <оголошення змінних>;
<опис вкладених процедур і функцій>;
begin
  <тіло процедури>;
end;
```

Більша частина зазначеного синтаксису є необов'язковою - процедура може не мати параметрів, констант, користувальницьких типів даних, змінних і т.п. - вона може бути дуже простою, наприклад:

```
procedure ErrorMessage;
begin
  ShowMessage('Помилка!' +#13 + 'На нуль ділити не можна!');
end;
```

Таку процедуру можна викликати з будь-якого місця програми, але процедура обов'язково повинна бути описана вище - інакше компілятор не буде знати про неї. Є ще можливість попередньо оголосити процедуру, але про це трохи пізніше. Отже, якщо ця процедура описана вище, то ми можемо викликати її, просто вказавши її ім'я:

```
ErrorMessage;
```

Компілятор перейде до процедури й виконає її код (у цьому випадку - виведе повідомлення про помилку). Після цього компілятор повернеться назад, і виконає наступний за викликом процедури оператор.

3. Параметри

Параметри підпрограм - досить важлива тема, поговоримо про це детальніше. У процедуру (або у функцію) можна передати будь-які вихідні дані, щоб процедура їх обробила. Такі дані називаються **параметрами**, або **формальними параметрами**. Приклад - користувач ввів якесь число (а насправді, рядок із цифрових символів), нам потрібно подвоїти його, а результат повідомити користувачеві. Описати подібну процедуру можна в такий спосіб:

```
procedure Udvoenie(st: string);
var
  r: real;
begin
  //отриманий рядок перетворимо в число:
  r:= StrToFloat(st);
  //тепер подвоїмо його:
  r:= r * 2;
  //тепер виведемо результат у повідомленні:
  ShowMessage(FloatToStr(r));
end;
```

Цей приклад уже складніше, правда? Насправді, все просто. Давайте розберемо. Отже, рядок оголошення процедури:

```
procedure Udvoenie(st: string);
```

повідомляє процедуру Udvoenie з параметром рядкового типу st. Це означає, що тепер ми можемо викликати процедуру, передавши їй як параметр якийсь рядок. Параметр st умовно можна вважати внутрішньою змінною процедури, у яку компілятор скопіює переданий процедурі рядок. Цей спосіб передачі даних у підпрограму називається **параметром за значенням**. Допустимо, надалі ми викликали процедуру в такий спосіб:

```
Udvoenie('123.4');
```

Компілятор зробить виклик процедури, передавши в параметр st зазначене значення '123.4'. Або ж ми можемо викликати процедуру інакше, передавши в неї значення, що зберігається в ту чи іншу рядкову змінну:

```
myst:= '123.4';
Udvoenie(myst);
```

Результат буде таким же. Отут важливо пам'ятати, що тип переданого значення обов'язково повинен збігатися з типом параметра. Якщо параметр у нас string, то й передавати йому потрібно значення типу string. Компілятор копіює це значення в параметр.

Інакше кажучи, якщо всередині процедури ми змінимо значення параметра st, це ніяк не відіб'ється на змінній myst, оскільки ми змінимо копію даних, а не самі дані.

Підемо далі. А далі ми повідомляємо дійсну змінну r:

```
var
  r: real;
```

Тут вона необхідна, адже нам потрібно помножити значення параметра на два, тому ми змушені будемо перетворити рядкове подання числа в дійсне число - адже рядок на два не помножиш! Результат помістимо в r:

```
begin
  //отриманий рядок перетворимо в число:
  r:= StrToFloat(st);
```

Службовим словом begin ми починаємо тіло процедури. Стандартною функцією StrToFloat(st) ми перетворимо рядкове значення параметра st у число, і привласнимо це число змінній r. Далі все просто:

```
  //тепер подвоїмо його:
  r:= r * 2;
  //тепер виведемо результат у повідомленні:
  ShowMessage(FloatToStr(r));
end;
```

Ми подвоюємо значення r, результат цього поміщаємо знову в r, потім стандартною функцією FloatToStr(r) перетворимо отримане число в рядок, і виводимо цей рядок у повідомленні ShowMessage(). От, власне, і все.

Тепер ми зможемо викликати цю процедуру, коли необхідно, і передавати їй різні числа у вигляді рядка. А вже функція сама подбає про всі необхідні перетворення, про подвоєння числа й виведення результатів на екран.

До речі, самі дані, які ми передаємо в підпрограму, називаються **аргументами** або **фактичними параметрами**. У прикладі виклику процедури

```
myst:= '123.4';
Udvoenie(myst);
```

змінна myst - аргумент.

Як параметри в процедурі можна використати не одну, а безліч змінних. Якщо вони мають однаковий тип, то їхні імена розділяють комами, а тип вказується наприкінці відразу для всіх параметрів. Наприклад:

```
procedure MyStrings(st1, st2, st3: string);
```

Якщо параметри мають різні типи, їх розділяють крапкою з комою:

```
procedure MyProc1(st: string; r1:real);
procedure MyProc2(st1, st2, st3:string; r1:real);
```

Однак, розбавимо теорію практикою, і попрацюємо із процедурами на реальному прикладі. Відкрийте **Lazarus** з новим проектом. Як завжди, назвемо головну форму (властивість Name) fMain, збережемо проект у папку **09-01**, при цьому назвемо проект, наприклад, **MyPodprog**, а модулю дамо ім'я **Main**.

У властивості Caption форми напишемо

Приклади роботи з підпрограмами

Наше завдання: одержати від користувача дійсне число, подвоїти його, і результат вивести на екран. Користувач може ввести й ціле число, але процедура обробить його як дійсне наприклад, якщо користувач введе 3, то процедура одержить 3.0. У результаті обчислення вийде 6.0, але FloatToStr() кінцеві нулі не виводить, так що користувач побачить на екрані просто 6.

Добре, зараз потрібно вирішити, як одержати в користувача число. Для цього використаємо компонент TEdit, що нам уже знайомий по минулих лекціях. Для початку встановимо мітку TLabel з пояснювальним текстом

Введіть будь-яке число:

а поруч встановимо TEdit. Імена в TLabel і TEdit залишимо за замовчуванням, TEdit буде називатися Edit1. Не забудьте очистити в ньому властивість Text.

Нижче встановіть просту кнопку **TButton**, в Caption якій напишіть текст:

Приклад подвоєння №1

Зрозуміло, будуть і інші приклади. Підрівняйте компоненти, при необхідності змініть їхні розміри. Наша форма повинна виглядати приблизно так:

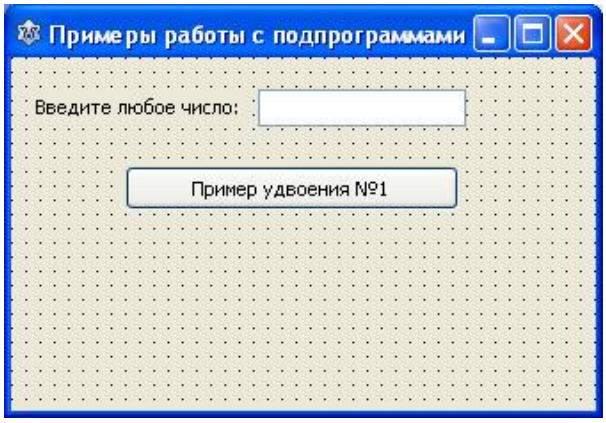

Рис. 9.1. Вікно програми MyPodprog

Поки що ми будемо змушені довіряти користувачеві, що він введе в поле **Edit1** число, і нічого більше. Але на минулій лекції вам було обіцяно показати реалізацію «захисту від дурнів», так що трохи пізніше ми й це зробимо.

Згенеруйте подію натискання на кнопку, вона буде такою:

```
procedure TfMain.Button1Click(Sender: TObject);
begin
  Udvoenie(Edit1.Text);
end;
```

Тепер нам потрібно створити процедуру Udvoenie **вище** нашої події, відразу після коментаря

```
{ TfMain }
```

Текст процедури наведений вище.

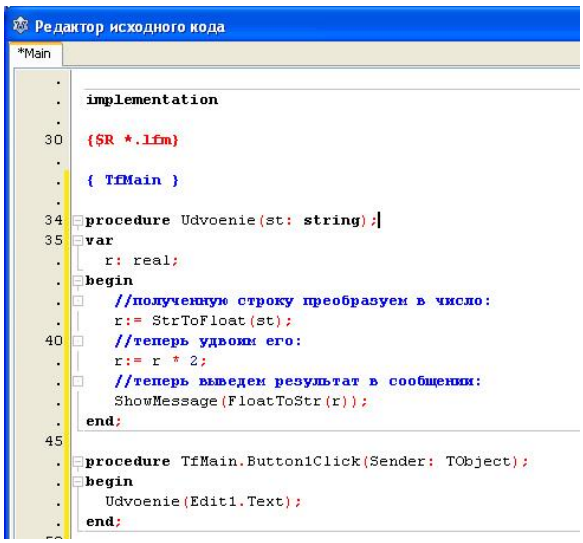

Рис. 9.2. Реалізація підпрограми Udvoenie

Зверніть увагу, ми передаємо в підпрограму значення, що ввів користувач, і яке зберігається у властивості Text компонента Edit1:

```
Udvoenie(Edit1.Text);
```

Ніяких додаткових змінних у цьому випадку створювати не потрібно. Збережіть проект і запустіть його на виконання. Спробуйте ввести ціле число. Потім дійсне. Зверніть увагу: якщо у вас установлена російська версія Windows, то як роздільник дійсного числа нам потрібно вводити кому, а не крапку! Пам'ятаєте про глобальну змінну DecimalSeparator?

Якщо ж ви випадково або навмисно ввели крапку, то вийде повідомлення про помилку, подібне цьому:

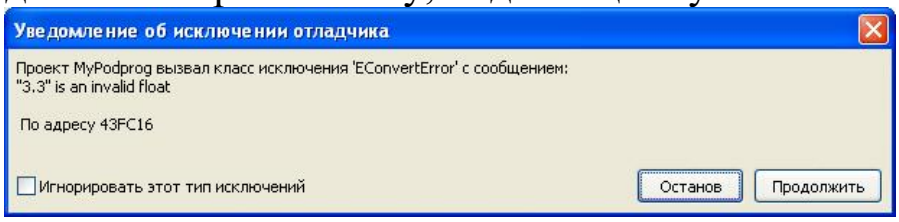

Рис. 9.3. Повідомлення про помилку

Нічого страшного, натисніть кнопку «**Останов**», потім виберіть команду головного меню «**Запуск -> Скинути отладчик**». Lazarus закриє завислий проект, і ви зможете запустити його знову. Схожа помилка виникне, якщо ви спробуєте подвоїти порожній рядок. Якщо ж ви ввели числа правильно, то програма буде працювати як потрібно в незалежності, ціле це було число, або дійсне. Не закривайте поки проект, він нам ще знадобиться.

4. Функції

Функції є такими ж підпрограмами, як і процедури, з однією різницею: вони повертають значення зазначеного типу. Функція починається із ключового слова function і має наступний синтаксис:

function <ім'я функції>(<список параметрів>): <тип поверненого значення,>;

```
const
  <оголошення констант>;
type
  <оголошення нових типів>;
var
  <оголошення змінних>;
<опис вкладених процедур і функцій>;
begin
 <тіло процедури>;
end;
```

У тілі функції повинен бути оператор, що привласнює повернене значення, стандартної змінної Result, що й приводить до повернення функцією значення. Result повідомляти не потрібно, він уже є в кожній функції. Зрозуміло, в Result потрібно привласнювати значення тільки зазначеного в заголовку типу. Приклад:

```
function MyFunc(i: integer): integer;
begin
  Result:= i * 2;
end;
```

Тут функція MyFunc приймає як параметр якесь ціле число, подвоює його й привласнює результат змінної Result. Це приводить до того, що функція повертає цей результат. Тепер ми можемо подвоїти ціле значення, викликавши цю функцію, наприклад:

```
myperem:= 5;
myperem:= MyFunc(myperem);
```

Що буде в результаті в змінної myperem? Якщо ви відповіли 10, то ви праві. Тут ми в першому кроці привласнюємо зміній цілого типу myperem значення 5. У другому кроці ми викликаємо функцію

MyFunc, передаючи їй як параметр значення myperem. У третьому кроці ми привласнюємо отриманий від функції результат подвоєння знову в змінну myperem.

До речі, замість системної змінної Result можна використати ім'я функції:

```
function MyFunc(i: integer): integer;
begin
  MyFunc:= i * 2;
end;
```

Даний приклад дасть точно такий же результат. Який спосіб використати - справа вибору. Особисто я волію використовувати Result, це виглядає якось більш стандартно. У кожному разі, ви повинні знати про обидва способи.

На відміну від багатьох інших мов, в Lazarus змінній Result (або імені функції) значення можна привласнювати неодноразово, якщо цього вимагає логіка підпрограми. Наприклад, функція одержує два дійсні числа числа. Перше потрібно розділити на друге, і результат повернути. Але на нуль ділити не можна, тому щоб уникнути зависання програми, якщо друге число - нуль, то й повернути потрібно нуль. Таку функцію можна реалізувати в такий спосіб:

```
function Delenie(r1, r2: real): real;
begin
  if r2 = 0 then Result:= 0
  else Result:= r1 / r2;
end;
```

Добре, випробуємо функції на практиці. Нижче **Button1** додайте **Button2** з текстом в Caption:

Приклад подвоєння №2

Згенеруйте для другої кнопки події OnClick, трішки вище цієї події опишіть функцію подвоєння FuncUdvoenie. У результаті у вас вийде наступне:

```
function FuncUdvoenie(st: string): string;
var
```

```
  r: real;
begin
  //отриманий рядок спочатку перетворимо в число:
  r:= StrToFloat(st);
  //тепер подвоїмо його:
  r:= r * 2;
  //тепер повернемо результат у вигляді рядка:
  Result:= FloatToStr(r);
end;

procedure TfMain.Button2Click(Sender: TObject);
begin
  ShowMessage(FuncUdvoenie(Edit1.Text));
end;
```

Зверніть увагу, тут ми пішли трішки іншим шляхом. Ми створили функцію, що лише повертає результуюче число у вигляді рядка, а висновок цього рядка на екран організували при виклику функції, у рядку

```
ShowMessage(FuncUdvoenie(Edit1.Text));
```

Отут у першому кроці ми викликаємо функцію FuncUdvoenie, передаючи їй як параметр текст із Edit1. У другому кроці відпрацьовує функція - перетворить цей текст у число, подвоює його, знову перетворить у рядок, і результат повертає компіляторові. А в третьому кроці за допомогою функції ShowMessage() ми виводимо цей результат на екран. Реалізовано інакше, але працювати буде точно також. Спробуйте. І поки не закривайте проект.

5. Параметри за посиланням

Параметри за посиланням дозволяють змінювати самі аргументи, оскільки в процедуру або функцію передається **не копія** аргументу, а посилання на самі аргументи. Щоб у функцію або процедуру передати параметр за посиланням, потрібно перед параметром вказати ключове слово var, наприклад:

```
procedure MyProc(var myparam: integer);
```

Якщо ви передаєте в підпрограму параметри як за значенням, так і за посиланням то параметри за посиланням повинні йти в описі останніми. От приклад оголошення процедури з безліччю параметрів:

```
procedure MyProc2(a,b: integer; c: real; var s1, s2: string);
```

Тут ми оголосили три параметри за значенням: a і b - цілі числа, c - дійсне; і два параметри за посиланням - s1 і s2, обидва рядкові типи. Якщо всередині процедури ми змінимо всі ці параметри, то вихідні цілі числа й дійсне число не зміняться, але зміни в обох рядках будуть збережені й після виходу із процедури. Давайте спробуємо попрацювати з параметрами за посиланням на практиці. Зробіть у проекті третю кнопку, аналогічно першим двом, і розташовану нижче. У властивості Caption цієї кнопки напишемо:

Приклад подвоєння №3

Згенеруйте подію OnClick для неї, і трішки вище опишіть процедуру, що буде приймати й змінювати параметр за посиланням, у такий спосіб:

```
procedure UdvoeniePoSsilke(var r: real);
begin
  r:= r * 2;
end;

procedure TfMain.Button3Click(Sender: TObject);
var
  myReal: real;
begin
  myReal:= StrToFloat(Edit1.Text);
  UdvoeniePoSsilke(myReal);
  ShowMessage(FloatToStr(myReal));
end;
```

Цей приклад відрізняється від попередніх, але виконує ту ж роботу, і з тим же результатом. У процедурі UdvoeniePoSsilke усього один рядок

```
r:= r * 2;
```

Цей рядок подвоює не просто параметр-копію, вона подвоює сам аргумент! Оскільки перед параметром r зазначене ключове слово var, то цей параметр - не копія аргументу, а посилання на нього. І подвоюючи r, ми подвоюємо аргумент. Що й було продемонстровано наступною подією натискання на третю кнопку. Тут, у першому кроці ми перетворимо рядкове подання числа з Edit1 у число, і результат привласнюємо дійсній змінній myReal. Потім, у другому кроці, ми передаємо цю змінну в процедуру UdvoeniePoSsilke, що змінює значення myReal. Ну й, нарешті, у третьому кроці, ми виводимо результат на екран, попередньо перетворивши його в рядкове подання.

6. Опис підпрограм з їхнім попереднім оголошенням

Застосування процедур і функцій вищеописаними способами має деякі недоліки. По-перше, ми змушені описувати процедури й функції вище того місця, де будемо їх застосовувати. Це пов’язано з тим, що компілятор не буде знати про ці підпрограми, якщо описати їх нижче, і не зможе їх виконати. Важливо, що ви не зможете навіть скомпілювати такий проект. По-друге, у таких підпрограмах ми не зможемо звертатися до властивостей компонентів. Якби ми спробували з підпрограми звернутися до тієї ж Edit1.Text, то не змогли б це зробити, хоча в події натискання на кнопку ми робимо це без зусиль. Справа в тому, що події належать до самої нашої форми, це видно за назвою події:

```
procedure TfMain.Button3Click(Sender: TObject);
```

Бачите, компілятор звертається спочатку до форми, і тільки потім до самої події? А от у випадку підпрограм ми звертаємося прямо до процедури або функції. Але є спосіб оголошення підпрограми, як частини форми. Такий спосіб надає нам наступні переваги:

1. Попередньо оголошену підпрограму можна описувати в будь-якому місці програми, не обов’язково вище місця її застосування.
2. У такій підпрограмі можна звертатися до властивостей і подій компонентів.
3. Підпрограму можна описати як приватну, або як публічну. Приватну можна використати тільки в поточному модулі (pas-файлі). Публічну можна використати в будь-якому іншому модулі, до якого підключається поточний.

Давайте подивимося, як це робиться на практиці. Додайте четверту кнопку нижче попередніх трьох, з написом в Caption

Приклад подвоєння №4

Згенеруйте для неї подію натискання на кнопку, у якій просто вкажіть виклик процедури MyPrivat:

```
procedure TfMain.Button4Click(Sender: TObject);
begin
  MyPrivat;
end;
```

Це нічого, що самої процедури ще немає, зараз ми її оголосимо, а потім створимо. Підніміть курсор на початок модуля. Ви побачите розділ type, у якому є оголошення класу TfMain - нашої форми. А там оголошені всі компоненти й згенеровані нами раніше події OnClick. Нижче розташовуються підрозділи private і public, де ми можемо повідомляти підпрограми відповідно, приватні й публічні. Давайте оголосимо процедуру MyPrivat у підрозділі private:

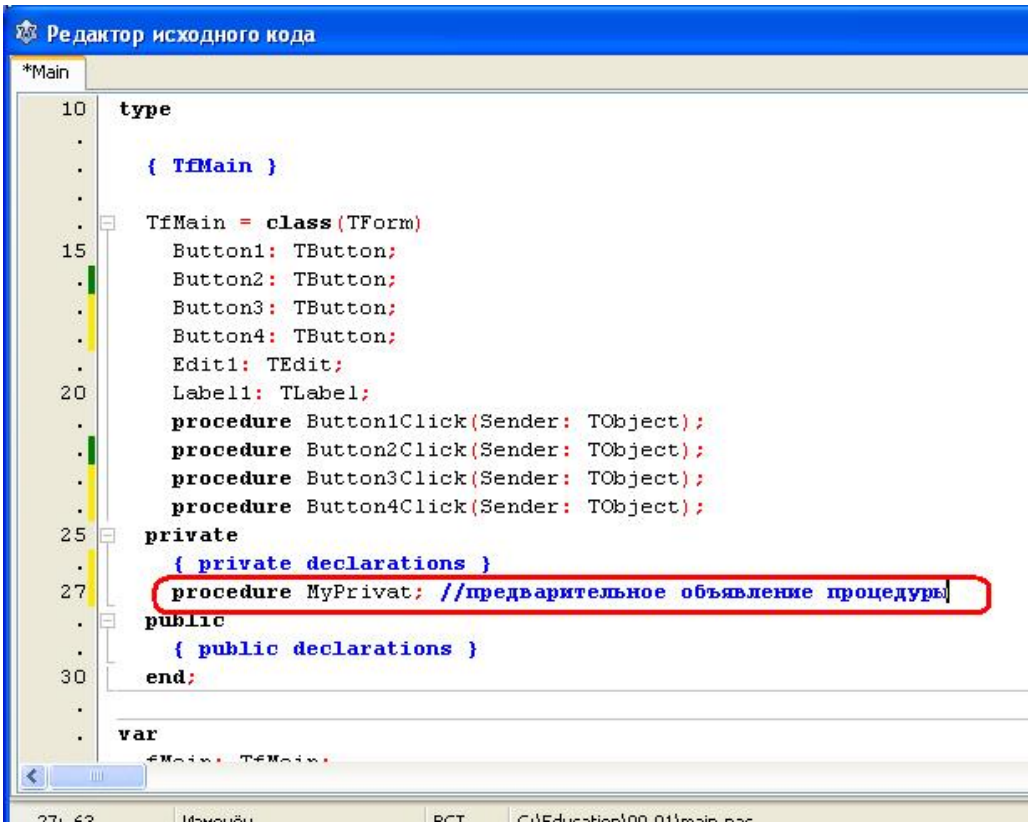

Рис. 9.4. Попереднє оголошення процедури в підрозділі privat

Тепер, не забираючи курсору із цього рядка, натисніть клавіші **<Ctrl + Shift + C>**. Це приведе до того, що аж унизу модуля (але перед

завершальним «end.») згенерується опис процедури MyPrivat. Насправді, вона могла б бути згенерована десь і в іншому місці, все залежить від того, у якому порядку зазначені оголошення всіх подій і інших підпрограм. Подивіться на рис. 9.4 - там MyPrivat оголошена останньою, тому й згенерувався опис унизу модуля.

Ніяких параметрів нам у цьому випадку не потрібно, ми будемо звертатися до властивості Text компонента Edit1 прямо з нашої підпрограми. І зверніть увагу, в описі перед ім'ям процедури зазначене ім'я форми. Відредагуйте підпрограму в такий спосіб:

```
procedure TfMain.MyPrivat;
var
  r: real;
begin
  //перетворимо в число те, що ввів користувач:
  r:= StrToFloat(Edit1.Text);
  //тепер подвоїмо його:
  r:= r * 2;
  //тепер виведемо результат у повідомленні:
  ShowMessage(FloatToStr(r));
end;
```

Коментарі досить докладні, щоб ви змогли розібратися з кодом. Подібний спосіб застосування підпрограм з попереднім оголошенням є найбільш зручним, раджу використовувати саме його. Підпрограми без попереднього оголошення варто використовувати тільки в найпростіших випадках, коли, приміром, потрібно розрахувати якісь дані, ну наприклад, перетворити значення температури зі шкали у Фаренгейтах у шкалу за Цельсієм. Тоді можна описати функцію подібного перетворення вище, і передавати в неї різні величини. Але навіть і таку підпрограму можна попередньо оголосити!

7. Область видимості змінних

Дотепер ми не розглядали змінні в цьому аспекті. Просто повідомляли їх у підпрограмах, і використовували тільки всередині них. Такі змінні називаються **локальними**, оскільки мають локальну область видимості. Поясню. Змінна, оголошена всередині процедури або функції, фізично створюється тільки тоді, коли компілятор звертається до даної підпрограми. До цього змінної не існує. Тільки коли відбувається виклик процедури або функції, в оперативній

пам’яті фізично виділяється місце для оголошених у підпрограмі змінних, констант і інших об’єктів. Коли ж підпрограма завершує свою роботу, то всі ці змінні (константи й т.п.) автоматично знищуються. До них не можна звернутися з інших підпрограм, їх там просто не видно. От чому ми можемо повідомляти змінні з однаковим ім’ям у різних підпрограмах - це різні змінні, і вони не заважають одна одній. Подивіться на код нашого проекту - ми тричі повідомляли змінну r дійсного типу, і щораз це була інша змінна.

Однак бувають моменти, коли потрібно використати **глобальні змінні** - змінні, які видні по всьому модулі, і в інших модулях, якщо до них підключається поточний. Такі змінні ми можемо оголосити або в розділах private і public, **до** оголошення підпрограм, або в розділі interface, після оголошень змінної з ім’ям форми, до ключового слова implementation. Давайте оголосимо змінну MyNum:

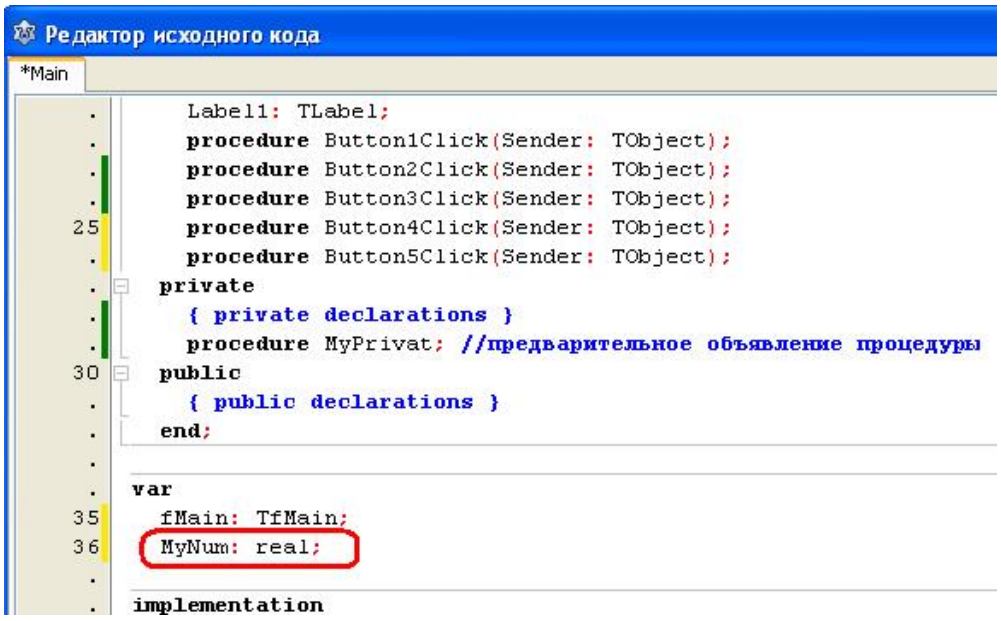

Рис. 9.5. Оголошення глобальної змінної

Тепер додамо на форму п’яту кнопку, текст в Caption буде відповідний. Згенеруйте для неї подія OnClick. Зверніть увагу, вона створилася вище попередньої процедури MyPrivat, оскільки оголошення різних подій розташовуються вище розділів private і public. Опишемо подію в такий спосіб:

```
procedure TfMain.Button5Click(Sender: TObject);
begin
 MyNum:= StrToFloat(Edit1.Text);
 //тепер подвоїмо його:
```

```
  MyDouble;
  //виводимо результат на екран:
  ShowMessage(FloatToStr(MyNum));
end;
```

Зверніть увагу: змінну MyNum ми в події не повідомляємо - вона вже оголошена глобально, і нею можна користуватися в будь-якому місці модуля. Процедури MyDouble ще ні, оголосимо їх в розділі private нижче MyPrivat:

```
procedure MyDouble; //подвоєння глобальної змінної
```

Далі згенеруємо опис цієї процедури (<Ctrl + Shift + C>). Сам опис буде дуже простим:

```
procedure TfMain.MyDouble;
begin
  //подвоїмо глобальну змінну:
  MyNum:= MyNum * 2;
end;
```

Як бачите, тут ми працюємо з тієї ж глобальною змінною, не повідомляючи її всередині процедури, оскільки вона вже оголошена.

Глобальні змінні дуже зручні, однак використовуйте їх тільки там, де це дійсно необхідно. Адже пам'ять для глобальної змінної виділяється, коли ви завантажуєте програму, і не звільняється, поки ви програму не закриєте.

8. Достроковий вихід з підпрограм і програми

Іноді буває необхідно терміново завершити процедуру або функцію. Приміром, залежно від яких умов, вам потрібно або продовжувати обробку даних, або закінчити підпрограму й вивести готовий результат. Для цього існує ключове слово Exit. Запам'ятаєте: якщо в підпрограмі зустрілося exit, підпрограма достроково завершує свою роботу й компілятор передає керування наступному за викликом підпрограми операторові.

Якщо ж зустрінеться ключове слово Halt, то вже вся програма достроково завершує свою роботу. Вона закривається, пам'ять, займана програмою, звільняється, а керування передається

операційній системі. Найчастіше цю команду застосовують для аварійного завершення роботи.

Тема 9. Підпрограми

1. Підпрограми

Спочатку мови програмування були простіші, вони виконувалися чітко вниз, один оператор за іншим. Такі мови ще називали **лінійними**. Типовий приклад лінійних мов - Бейсик. Єдину можливість організувати хоч якусь логіку в таких мовах надавав оператор безумовного переходу GOTO, що залежно від умови, «перестрибував» на заздалегідь розставлені мітки. У сучасних мовах програмування GOTO теж залишився, напевно, для аматорів антикваріату. Але його застосування може привести до складних логічних часових помилок, які важко відслідкувати (run-time errors). Використання GOTO у сучасному програмуванні вважається дурним тоном. Ми не будемо вивчати цю можливість, оскільки для організації логіки є куди більш «просунуті» засоби! Одним з таких засобів є **підпрограми**.

Підпрограма - це частина коду, яку можна викликати з будь-якого місця програми невизначену кількість разів.

Інакше кажучи, підпрограми подібні до будівельних цеглинок, з яких, зрештою, виходить будинок - програма. Без підпрограм можна обійтися, якщо ви пишете невелику навчальну програму на парі десятків рядків коду. А якщо це серйозний додаток, з парою сотень модулів, у кожному з яких можуть бути тисячі рядків коду? Як таку програму написати, не розбиваючи завдання на окремі частини? Підпрограми допомагають поліпшити код, структурувати його. Тому мови високого рівня, які дозволяють використовувати підпрограми, називають ще **процедурно-орієнтованими** мовами. І наш компілятор FPC теж відноситься до таких мов.

Підпрограми бувають двох типів: **процедури** й **функції** - перші просто виконують свою роботу, другі ще й повертають результат цієї роботи.

2. Процедури

Насправді, ми вже неодноразово використовували процедури. Наприклад, коли генерували подію натискання на кнопку. Ця подія - процедура. Процедура починається із ключового слова procedure і має наступний синтаксис:

```
procedure <ім'я процедури>(<список параметрів>);
const
```

```
  <оголошення констант>;
type
  <оголошення нових типів>;
var
  <оголошення змінних>;
<опис вкладених процедур і функцій>;
begin
  <тіло процедури>;
end;
```

Більша частина зазначеного синтаксису є необов'язковою - процедура може не мати параметрів, констант, користувальницьких типів даних, змінних і т.п. - вона може бути дуже простою, наприклад:

```
procedure ErrorMessage;
begin
  ShowMessage('Помилка!' +#13 + 'На нуль ділити не можна!');
end;
```

Таку процедуру можна викликати з будь-якого місця програми, але процедура обов'язково повинна бути описана вище - інакше компілятор не буде знати про неї. Є ще можливість попередньо оголосити процедуру, але про це трохи пізніше. Отже, якщо ця процедура описана вище, то ми можемо викликати її, просто вказавши її ім'я:

```
ErrorMessage;
```

Компілятор перейде до процедури й виконає її код (у цьому випадку - виведе повідомлення про помилку). Після цього компілятор повернеться назад, і виконає наступний за викликом процедури оператор.

3. Параметри

Параметри підпрограм - досить важлива тема, поговоримо про це детальніше. У процедуру (або у функцію) можна передати будь-які вихідні дані, щоб процедура їх обробила. Такі дані називаються **параметрами**, або **формальними параметрами**. Приклад - користувач ввів якесь число (а насправді, рядок із цифрових символів),

нам потрібно подвоїти його, а результат повідомити користувачеві. Описати подібну процедуру можна в такий спосіб:

```
procedure Udvoenie(st: string);
var
  r: real;
begin
  //отриманий рядок перетворимо в число:
  r:= StrToFloat(st);
  //тепер подвоїмо його:
  r:= r * 2;
  //тепер виведемо результат у повідомленні:
  ShowMessage(FloatToStr(r));
end;
```

Цей приклад уже складніше, правда? Насправді, все просто. Давайте розберемо. Отже, рядок оголошення процедури:

```
procedure Udvoenie(st: string);
```

повідомляє процедуру Udvoenie з параметром рядкового типу st. Це означає, що тепер ми можемо викликати процедуру, передавши їй як параметр якийсь рядок. Параметр st умовно можна вважати внутрішньою змінною процедури, у яку компілятор скопіює переданий процедурі рядок. Цей спосіб передачі даних у підпрограму називається **параметром за значенням**. Допустимо, надалі ми викликали процедуру в такий спосіб:

```
Udvoenie('123.4');
```

Компілятор зробить виклик процедури, передавши в параметр st зазначене значення '123.4'. Або ж ми можемо викликати процедуру інакше, передавши в неї значення, що зберігається в ту чи іншу рядкову змінну:

```
myst:= '123.4';
Udvoenie(myst);
```

Результат буде таким же. Отут важливо пам’ятати, що тип переданого значення обов’язково повинен збігатися з типом параметра. Якщо параметр у нас string, то й передавати йому потрібно значення типу string. Компілятор копіює це значення в параметр. Інакше кажучи, якщо всередині процедури ми змінимо значення параметра st, це ніяк не відіб’ється на змінній myst, оскільки ми змінимо копію даних, а не самі дані.

Підемо далі. А далі ми повідомляємо дійсну змінну r:

```
var
  r: real;
```

Тут вона необхідна, адже нам потрібно помножити значення параметра на два, тому ми змушені будемо перетворити рядкове подання числа в дійсне число - адже рядок на два не помножиш! Результат помістимо в r:

```
begin
  //отриманий рядок перетворимо в число:
  r:= StrToFloat(st);
```

Службовим словом begin ми починаємо тіло процедури. Стандартною функцією StrToFloat(st) ми перетворимо рядкове значення параметра st у число, і привласнимо це число змінній r. Далі все просто:

```
  //тепер подвоїмо його:
  r:= r * 2;
  //тепер виведемо результат у повідомленні:
  ShowMessage(FloatToStr(r));
end;
```

Ми подвоюємо значення r, результат цього поміщаємо знову в r, потім стандартною функцією FloatToStr(r) перетворимо отримане число в рядок, і виводимо цей рядок у повідомленні ShowMessage(). От, власне, і все.

Тепер ми зможемо викликати цю процедуру, коли необхідно, і передавати їй різні числа у вигляді рядка. А вже функція сама подбає про всі необхідні перетворення, про подвоєння числа й виведення результатів на екран.

До речі, самі дані, які ми передаємо в підпрограму, називаються **аргументами** або **фактичними параметрами**. У прикладі виклику процедури

```
myst:= '123.4';
Udvoenie(myst);
```

змінна myst - аргумент.

Як параметри в процедурі можна використати не одну, а безліч змінних. Якщо вони мають однаковий тип, то їхні імена розділяють комами, а тип вказується наприкінці відразу для всіх параметрів. Наприклад:

```
procedure MyStrings(st1, st2, st3: string);
```

Якщо параметри мають різні типи, їх розділяють крапкою з комою:

```
procedure MyProc1(st: string; r1:real);
procedure MyProc2(st1, st2, st3:string; r1:real);
```

Однак, розбавимо теорію практикою, і попрацюємо із процедурами на реальному прикладі. Відкрийте **Lazarus** з новим проектом. Як завжди, назвемо головну форму (властивість Name) fMain, збережемо проект у папку **09-01**, при цьому назвемо проект, наприклад, **MyPodprog**, а модулю дамо ім'я **Main**.

У властивості Caption форми напишемо

Приклади роботи з підпрограмами

Наше завдання: одержати від користувача дійсне число, подвоїти його, і результат вивести на екран. Користувач може ввести й ціле число, але процедура обробить його як дійсне наприклад, якщо користувач введе 3, то процедура одержить 3.0. У результаті обчислення вийде 6.0, але FloatToStr() кінцеві нулі не виводить, так що користувач побачить на екрані просто 6.

Добре, зараз потрібно вирішити, як одержати в користувача число. Для цього використаємо компонент TEdit, що нам уже знайомий по минулих лекціях. Для початку встановимо мітку TLabel з пояснювальним текстом

Введіть будь-яке число:

а поруч встановимо TEdit. Імена в TLabel і TEdit залишимо за замовчуванням, TEdit буде називатися Edit1. Не забудьте очистити в ньому властивість Text.

Нижче встановіть просту кнопку **TButton**, в Caption якій напишіть текст:

Приклад подвоєння №1

Зрозуміло, будуть і інші приклади. Підрівняйте компоненти, при необхідності змініть їхні розміри. Наша форма повинна виглядати пр

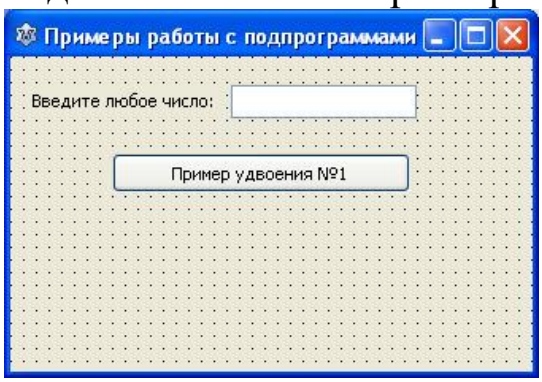

Рис. 9.1. Вікно програми MyPodprog

Поки що ми будемо змушені довіряти користувачеві, що він введе в поле **Edit1** число, і нічого більше. Але на минулій лекції вам було обіцяно показати реалізацію «захисту від дурнів», так що трохи пізніше ми й це зробимо.

Згенеруйте подію натискання на кнопку, вона буде такою:

```
procedure TfMain.Button1Click(Sender: TObject);
begin
  Udvoenie(Edit1.Text);
end;
```

Тепер нам потрібно створити процедуру Udvoenie **вище** нашої події, відразу після коментаря

```
{ TfMain }
```

Текст процедури наведений вище.

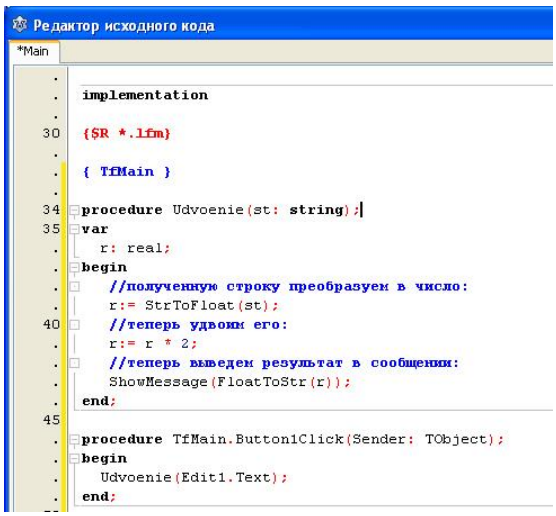

Рис. 9.2. Реалізація підпрограми Udvoenie

Зверніть увагу, ми передаємо в підпрограму значення, що ввів користувач, і яке зберігається у властивості Text компонента Edit1:

```
Udvoenie(Edit1.Text);
```

Ніяких додаткових змінних у цьому випадку створювати не потрібно. Збережіть проект і запустіть його на виконання. Спробуйте ввести ціле число. Потім дійсне. Зверніть увагу: якщо у вас установлена російська версія Windows, то як роздільник дійсного числа нам потрібно вводити кому, а не крапку! Пам'ятаєте про глобальну змінну DecimalSeparator?

Якщо ж ви випадково або навмисно ввели крапку, то вийде повідомлення про помилку, подібне цьому:

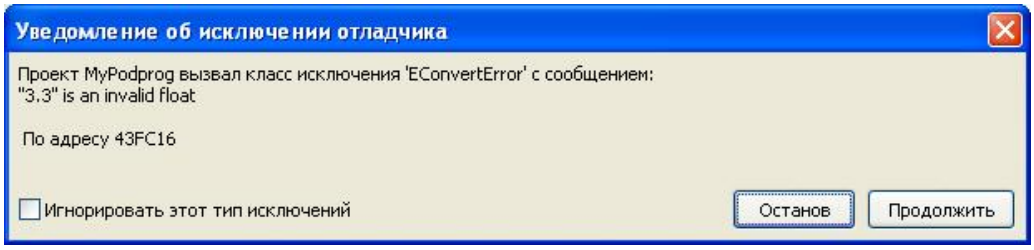

Рис. 9.3. Повідомлення про помилку

Нічого страшного, натисніть кнопку «**Останов**», потім виберіть команду головного меню «**Запуск -> Скинути отладчик**». Lazarus закриє завислий проект, і ви зможете запустити його знову. Схожа помилка виникне, якщо ви спробуєте подвоїти порожній рядок. Якщо ж ви ввели числа правильно, то програма буде працювати як потрібно в незалежності, ціле це було число, або дійсне. Не закривайте поки проект, він нам ще знадобиться.

4. Функції

Функції є такими ж підпрограмами, як і процедури, з однією різницею: вони повертають значення зазначеного типу. Функція починається із ключового слова function і має наступний синтаксис:

```
function <ім'я функції>(<список параметрів>): <тип поверненого значення,>;
const
  <оголошення констант>;
type
  <оголошення нових типів>;
var
  <оголошення змінних>;
<опис вкладених процедур і функцій>;
begin
 <тіло процедури>;
end;
```

У тілі функції повинен бути оператор, що привласнює повернене значення, стандартної змінної Result, що й приводить до повернення функцією значення. Result повідомляти не потрібно, він уже є в кожній функції. Зрозуміло, в Result потрібно привласнювати значення тільки зазначеного в заголовку типу. Приклад:

```
function MyFunc(i: integer): integer;
begin
  Result:= i * 2;
end;
```

Тут функція MyFunc приймає як параметр якесь ціле число, подвоює його й привласнює результат змінної Result. Це приводить до того, що функція повертає цей результат. Тепер ми можемо подвоїти ціле значення, викликавши цю функцію, наприклад:

```
myperem:= 5;
myperem:= MyFunc(myperem);
```

Що буде в результаті в змінної myperem? Якщо ви відповіли 10, то ви праві. Тут ми в першому кроці привласнюємо зміній цілого типу myperem значення 5. У другому кроці ми викликаємо функцію MyFunc, передаючи їй як параметр значення myperem. У третьому кроці ми привласнюємо отриманий від функції результат подвоєння знову в змінну myperem.

До речі, замість системної змінної Result можна використати ім'я функції:

```
function MyFunc(i: integer): integer;
begin
  MyFunc:= i * 2;
end;
```

Даний приклад дасть точно такий же результат. Який спосіб використати - справа вибору. Особисто я волію використовувати Result, це виглядає якось більш стандартно. У кожному разі, ви повинні знати про обидва способи.

На відміну від багатьох інших мов, в Lazarus змінній Result (або імені функції) значення можна привласнювати неодноразово, якщо цього вимагає логіка підпрограми. Наприклад, функція одержує два дійсні числа числа. Перше потрібно розділити на друге, і результат повернути. Але на нуль ділити не можна, тому щоб уникнути зависання програми, якщо друге число - нуль, то й повернути потрібно нуль. Таку функцію можна реалізувати в такий спосіб:

```
function Delenie(r1, r2: real): real;
begin
  if r2 = 0 then Result:= 0
  else Result:= r1 / r2;
end;
```

Добре, випробуємо функції на практиці. Нижче **Button1** додайте **Button2** з текстом в Caption:

Приклад подвоєння №2

Згенеруйте для другої кнопки події OnClick, трішки вище цієї події опишіть функцію подвоєння FuncUdvoenie. У результаті у вас вийде наступне:

```
function FuncUdvoenie(st: string): string;
var
  r: real;
begin
  //отриманий рядок спочатку перетворимо в число:
  r:= StrToFloat(st);
  //тепер подвоїмо його:
  r:= r * 2;
  //тепер повернемо результат у вигляді рядка:
  Result:= FloatToStr(r);
end;

procedure TfMain.Button2Click(Sender: TObject);
begin
  ShowMessage(FuncUdvoenie(Edit1.Text));
end;
```

Зверніть увагу, тут ми пішли трішки іншим шляхом. Ми створили функцію, що лише повертає результуюче число у вигляді рядка, а висновок цього рядка на екран організували при виклику функції, у рядку

```
ShowMessage(FuncUdvoenie(Edit1.Text));
```

Отут у першому кроці ми викликаємо функцію FuncUdvoenie, передаючи їй як параметр текст із Edit1. У другому кроці відпрацьовує функція - перетворить цей текст у число, подвоює його, знову перетворить у рядок, і результат повертає компіляторові. А в третьому кроці за допомогою функції ShowMessage() ми виводимо цей результат на екран. Реалізовано інакше, але працювати буде точно також. Спробуйте. І поки не закривайте проект.

5. Параметри за посиланням

Параметри за посиланням дозволяють змінювати самі аргументи, оскільки в процедуру або функцію передається **не копія** аргументу, а посилання на самі аргументи. Щоб у функцію або процедуру передати параметр за посиланням, потрібно перед параметром вказати ключове слово var, наприклад:

```
procedure MyProc(var myparam: integer);
```

Якщо ви передаєте в підпрограму параметри як за значенням, так і за посиланням то параметри за посиланням повинні йти в описі останніми. От приклад оголошення процедури з безліччю параметрів:

```
procedure MyProc2(a,b: integer; c: real; var s1, s2: string);
```

Тут ми оголосили три параметри за значенням: a і b - цілі числа, c - дійсне; і два параметри за посиланням - s1 і s2, обидва рядкові типи. Якщо всередині процедури ми змінимо всі ці параметри, то вихідні цілі числа й дійсне число не зміняться, але зміни в обох рядках будуть збережені й після виходу із процедури. Давайте спробуємо попрацювати з параметрами за посиланням на практиці. Зробіть у проекті третю кнопку, аналогічно першим двом, і розташовану нижче. У властивості Caption цієї кнопки напишемо:

Приклад подвоєння №3

Згенеруйте подію OnClick для неї, і трішки вище опишіть процедуру, що буде приймати й змінювати параметр за посиланням, у такий спосіб:

```
procedure UdvoeniePoSsilke(var r: real);
begin
  r:= r * 2;
end;

procedure TfMain.Button3Click(Sender: TObject);
var
  myReal: real;
begin
  myReal:= StrToFloat(Edit1.Text);
  UdvoeniePoSsilke(myReal);
  ShowMessage(FloatToStr(myReal));
end;
```

Цей приклад відрізняється від попередніх, але виконує ту ж роботу, і з тим же результатом. У процедурі UdvoeniePoSsilke усього один рядок

```
r:= r * 2;
```

Цей рядок подвоює не просто параметр-копію, вона подвоює сам аргумент! Оскільки перед параметром r зазначене ключове слово var, то цей параметр - не копія аргументу, а посилання на нього. І подвоюючи r, ми подвоюємо аргумент. Що й було продемонстровано наступною подією натискання на третю кнопку. Тут, у першому кроці ми перетворимо рядкове подання числа з Edit1 у число, і результат привласнюємо дійсній змінній myReal. Потім, у другому кроці, ми передаємо цю змінну в процедуру UdvoeniePoSsilke, що змінює значення myReal. Ну й, нарешті, у третьому кроці, ми виводимо результат на екран, попередньо перетворивши його в рядкове подання.

6. Опис підпрограм з їхнім попереднім оголошенням

Застосування процедур і функцій вищеописаними способами має деякі недоліки. По-перше, ми змушені описувати процедури й функції вище того місця, де будемо їх застосовувати. Це пов’язано з тим, що компілятор не буде знати про ці підпрограми, якщо описати їх нижче, і не зможе їх виконати. Важливо, що ви не зможете навіть скомпілювати такий проект. По-друге, у таких підпрограмах ми не зможемо звертатися до властивостей компонентів. Якби ми спробували з підпрограми звернутися до тієї ж Edit1.Text, то не змогли б це зробити, хоча в події натискання на кнопку ми робимо це без зусиль. Справа в тому, що події належать до самої нашої форми, це видно за назвою події:

```
procedure TfMain.Button3Click(Sender: TObject);
```

Бачите, компілятор звертається спочатку до форми, і тільки потім до самої події? А от у випадку підпрограм ми звертаємося прямо до процедури або функції. Але є спосіб оголошення підпрограми, як частини форми. Такий спосіб надає нам наступні переваги:

4. Попередньо оголошену підпрограму можна описувати в будь-якому місці програми, не обов’язково вище місця її застосування.
5. У такій підпрограмі можна звертатися до властивостей і подій компонентів.

6. Підпрограму можна описати як приватну, або як публічну. Приватну можна використати тільки в поточному модулі (pas-файлі). Публічну можна використати в будь-якому іншому модулі, до якого підключається поточний.

Давайте подивимося, як це робиться на практиці. Додайте четверту кнопку нижче попередніх трьох, з написом в Caption

Приклад подвоєння №4

Згенеруйте для неї подію натискання на кнопку, у якій просто вкажіть виклик процедури MyPrivat:

```
procedure TfMain.Button4Click(Sender: TObject);
begin
  MyPrivat;
end;
```

Це нічого, що самої процедури ще немає, зараз ми її оголосимо, а потім створимо. Підніміть курсор на початок модуля. Ви побачите розділ type, у якому є оголошення класу TfMain - нашої форми. А там оголошені всі компоненти й згенеровані нами раніше події OnClick. Нижче розташовуються підрозділи private і public, де ми можемо повідомляти підпрограми відповідно, приватні й публічні. Давайте оголосимо процедуру MyPrivat у підрозділі private:

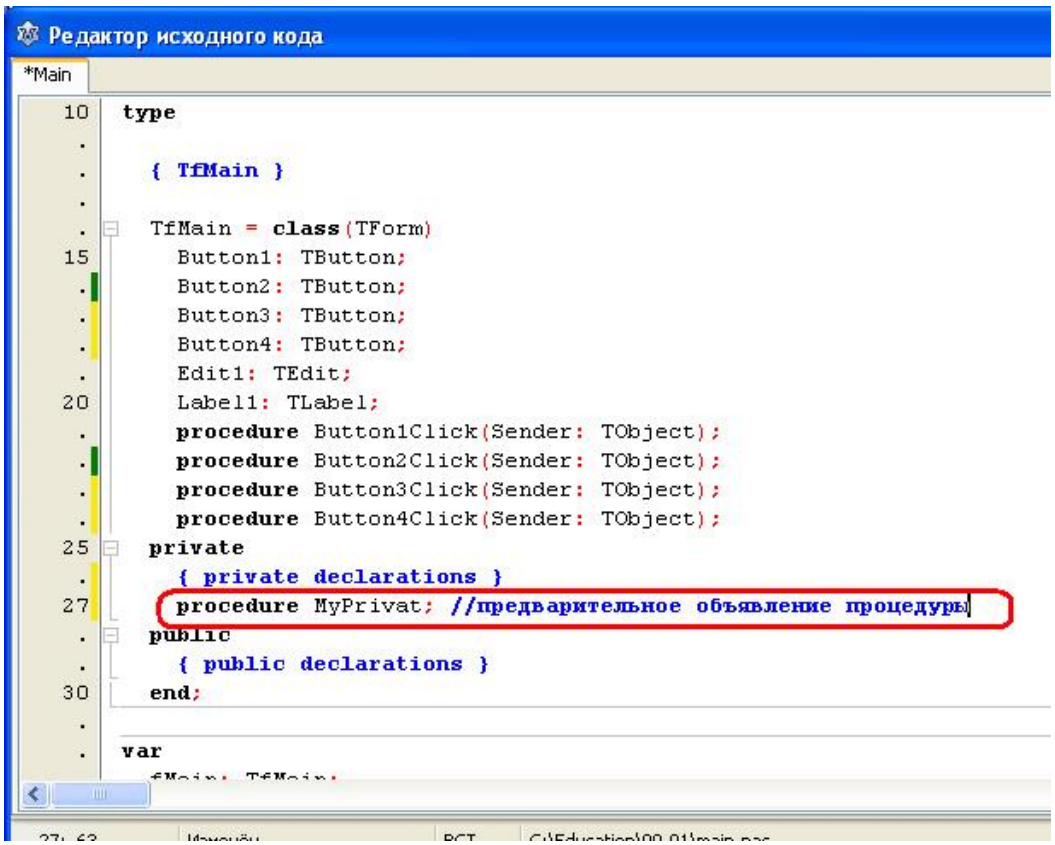

Рис. 9.4. Попереднє оголошення процедури в підрозділі private

Тепер, не забираючи курсору із цього рядка, натисніть клавіші **<Ctrl + Shift + C>**. Це приведе до того, що аж унизу модуля (але перед завершальним «end.») згенерується опис процедури MyPrivat. Насправді, вона могла б бути згенерована десь і в іншому місці, все залежить від того, у якому порядку зазначені оголошення всіх подій і інших підпрограм. Подивіться на рис. 9.4 - там MyPrivat оголошена останньою, тому й згенерувався опис унизу модуля.

Ніяких параметрів нам у цьому випадку не потрібно, ми будемо звертатися до властивості Text компонента Edit1 прямо з нашої підпрограми. І зверніть увагу, в описі перед ім'ям процедури зазначене ім'я форми. Відредагуйте підпрограму в такий спосіб:

```
procedure TfMain.MyPrivat;
var
  r: real;
begin
  //перетворимо в число те, що ввів користувач:
  r:= StrToFloat(Edit1.Text);
  //тепер подвоїмо його:
  r:= r * 2;
```

```
  //тепер виведемо результат у повідомленні:
  ShowMessage(FloatToStr(r));
end;
```

Коментарі досить докладні, щоб ви змогли розібратися з кодом. Подібний спосіб застосування підпрограм з попереднім оголошенням є найбільш зручним, раджу використовувати саме його. Підпрограми без попереднього оголошення варто використовувати тільки в найпростіших випадках, коли, приміром, потрібно розрахувати якісь дані, ну наприклад, перетворити значення температури зі шкали у Фаренгейтах у шкалу за Цельсієм. Тоді можна описати функцію подібного перетворення вище, і передавати в неї різні величини. Але навіть і таку підпрограму можна попередньо оголосити!

7. Область видимості змінних

Дотепер ми не розглядали змінні в цьому аспекті. Просто повідомляли їх у підпрограмах, і використовували тільки всередині них. Такі змінні називаються **локальними**, оскільки мають локальну область видимості. Поясню. Змінна, оголошена всередині процедури або функції, фізично створюється тільки тоді, коли компілятор звертається до даної підпрограми. До цього змінної не існує. Тільки коли відбувається виклик процедури або функції, в оперативній пам'яті фізично виділяється місце для оголошених у підпрограмі змінних, констант і інших об'єктів. Коли ж підпрограма завершує свою роботу, то всі ці змінні (константи й т.п.) автоматично знищуються. До них не можна звернутися з інших підпрограм, їх там просто не видно. От чому ми можемо повідомляти змінні з однаковим ім'ям у різних підпрограмах - це різні змінні, і вони не заважають одна одній. Подивіться на код нашого проекту - ми тричі повідомляли змінну r дійсного типу, і щораз це була інша змінна.

Однак бувають моменти, коли потрібно використати **глобальні змінні** - змінні, які видні по всьому модулі, і в інших модулях, якщо до них підключається поточний. Такі змінні ми можемо оголосити або в розділах private і public, **до** оголошення підпрограм, або в розділі interface, після оголошень змінної з ім'ям форми, до ключового слова implementation. Давайте оголосимо змінну MyNum:

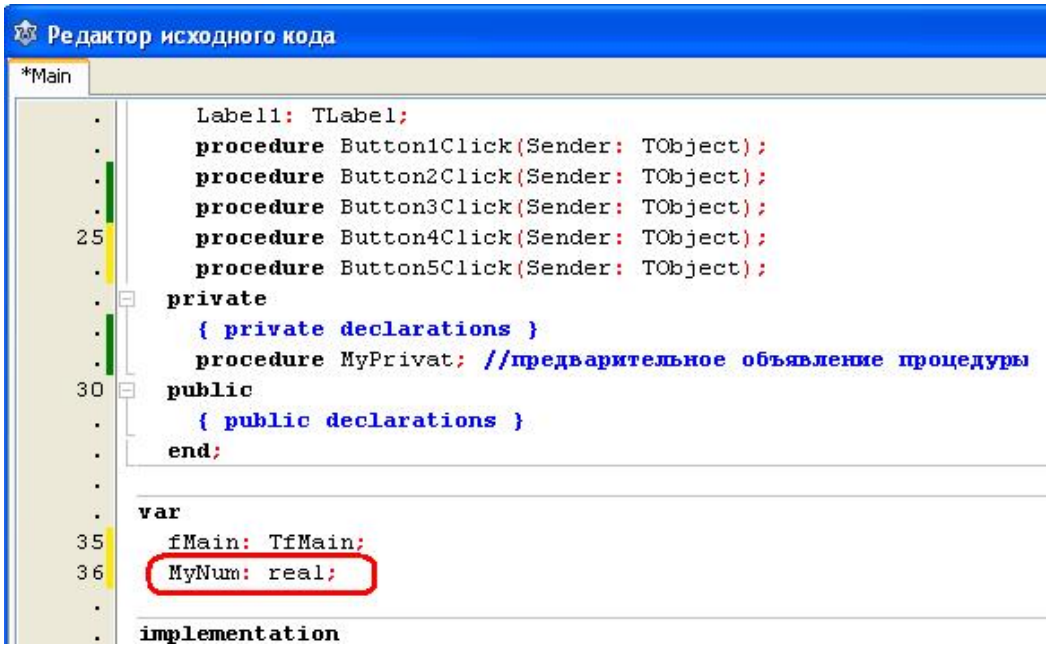

Рис. 9.5. Оголошення глобальної змінної

Тепер додамо на форму п'яту кнопку, текст в Caption буде відповідний. Згенеруйте для неї подія OnClick. Зверніть увагу, вона створилася вище попередньої процедури MyPrivat, оскільки оголошення різних подій розташовуються вище розділів private і public. Опишемо подію в такий спосіб:

```
procedure TfMain.Button5Click(Sender: TObject);
begin
  MyNum:= StrToFloat(Edit1.Text);
  //тепер подвоїмо його:
  MyDouble;
  //виводимо результат на екран:
  ShowMessage(FloatToStr(MyNum));
end;
```

Зверніть увагу: змінну MyNum ми в події не повідомляємо - вона вже оголошена глобально, і нею можна користуватися в будь-якому місці модуля. Процедури MyDouble ще ні, оголосимо їх в розділі private нижче MyPrivat:

```
procedure MyDouble; //подвоєння глобальної змінної
```

Далі згенеруємо опис цієї процедури (<Ctrl + Shift + C>). Сам опис буде дуже простим:

```
procedure TfMain.MyDouble;
begin
  //подвоїмо глобальну змінну:
  MyNum:= MyNum * 2;
end;
```

Як бачите, тут ми працюємо з тієї ж глобальною змінною, не повідомляючи її всередині процедури, оскільки вона вже оголошена.

Глобальні змінні дуже зручні, однак використовуйте їх тільки там, де це дійсно необхідно. Адже пам'ять для глобальної змінної виділяється, коли ви завантажуєте програму, і не звільняється, поки ви програму не закриєте.

8. Достроковий вихід з підпрограм і програми

Іноді буває необхідно терміново завершити процедуру або функцію. Приміром, залежно від яких умов, вам потрібно або продовжувати обробку даних, або закінчити підпрограму й вивести готовий результат. Для цього існує ключове слово Exit. Запам'ятаєте: якщо в підпрограмі зустрілося exit, підпрограма достроково завершує свою роботу й компілятор передає керування наступному за викликом підпрограми операторові.

Якщо ж зустрінеться ключове слово Halt, то вже вся програма достроково завершує свою роботу. Вона закривається, пам'ять, займана програмою, звільняється, а керування передається операційній системі. Найчастіше цю команду застосовують для аварійного завершення роботи.

Тема 10. Цикли й перемикач case

1. Цикли

На цій лекції ми поговоримо ще про один спосіб організувати логіку програми - про **цикли**. Досить часто програмістові доводиться виконувати якусь ділянку коду задана кількість разів, або доти, поки не наступить якась умова. Такий повтор коду й називається циклом.

Цикл - послідовний повтор частини коду від нуля до нескінченно.

Дійсно, бувають умови, при яких цикл не виконується жодного разу. А буває й так, що цикл починає виконуватися нескінченно, при

цьому програма «зависає» - не реагує на клавіатуру й мишу, не дає продовжувати з нею роботу. Але це вже помилка програміста. Цикли бувають двох видів - за лічильником, і за умовою. Перші виконуються задану кількість разів, другі - поки не наступить певна логічна умова. Саме другий тип циклів починаючі програмісти можуть «зациклити», не передбачивши гарантованого настання умови їхнього завершення. Але давайте все по черзі.

2. Цикл for...to...do

Цикл for…to…do виконується певну кількість разів, за лічильником. Лічильник являє собою змінну цілочисельного типу - звичайно використовують integer, але якщо цикл повинен виконатися 10-20 разів, цілком можна обійтися типом byte, щоб не витрачати впусту зайві 3 байти оперативної пам'яті. Синтаксис циклу наступний:

```
for <лічильник>:= <початкове значення> to <кінцеве значення> do <оператор>;
```

Тут, лічильник, як говорилося вище - змінна цілого типу. Початкове й кінцеве значення - цілі числа, наприклад, від 1 до 10. Оператор - та частина коду, яку потрібно виконати потрібну кількість разів. Нерідко буває так, що потрібно виконати не один оператор, а декілька. У цьому випадку роблять складений оператор, помістивши весь потрібний код між дужками оператора begin...end:

```
for <лічильник>:= <початкове значення> to <кінцеве значення> do begin
  Оператор 1;
  Оператор 2;
  ...
  Оператор n;
end;
```

При кожному черговому проходженні циклу лічильник автоматично збільшується на одиницю, і як тільки він досягне кінцевого значення, цикл завершить свою роботу. Давайте, випробуємо цей інструмент на конкретному прикладі.

Завантажте **Lazarus** з новим проектом. Збережіть його в папку **10-01** там, де ви зберігаєте всі навчальні проекти. Проект назвіть, скажімо, **MyCycles**. Як назвати головну форму, вам уже, напевно, не потрібно

нагадувати? Посередині вікна встановіть просту кнопку **TButton**, перейменовувати її не потрібно, але в Caption напишіть

for...do

Згенеруйте для кнопки події OnClick, у якій напишіть наступний код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  b: byte;
begin
  for b:= 1 to 10 do ShowMessage('Прохід циклу №' + IntToStr(b));
end;
```

Що, по-вашому, відбудеться при натисканні на кнопку? Вийде віконце з повідомленням «**Прохід циклу №1**». Як тільки ви натиснете **<OK>**, вийде нове повідомлення «**Прохід циклу №2**». І так буде 10 разів, при цьому b буде автоматично збільшуватися на 1. Після того, як b стане дорівнює 10, і цикл виконається востаннє, процедура Button1Click завершить роботу й керування перейде назад головній формі.

У даному прикладі цикл for нарощував лічильник за зростанням, від 1 до 10. Але в цього циклу є ще одна форма, коли лічильник не збільшується, а зменшується на одиницю:

for <лічильник>:= <початкове значення> downto <кінцеве значення> do <оператор>;

Різниця тут тільки в тому, що замість ключового слова to ми вказуємо downto. І, зрозуміло, у цьому випадку початкове значення лічильника повинне бути більше кінцевого значення. Переробіть рядок із циклом так:

```
for b:= 10 downto 1 do ShowMessage('Прохід циклу №' + IntToStr(b));
```

Запустивши програму на виконання, ви переконаєтеся, що тепер лічильник зменшується від 10 до 1.

3. Інструкції break та continue

Цикли не завжди потрібно виконувати від початку, і до кінця. Іноді, залежно від умов, буває необхідно пропустити якісь кроки циклу, або зовсім достроково завершити цикл. Для цього й служать інструкції break та continue.

Break - інструкція дострокового завершення роботи циклу.

Змінимо приклад процедури Button1Click. Припустимо, нам потрібно вивести на екран результати ділення числа 100 на числа від -10 до 10. АЛЕ! На нуль ділити не можна, тому ми виконаємо перевірку: якщо друге число - нуль, ми завершимо цикл. А щоб нам не довелося багато разів натискати на **<OK>**, як у минулому прикладі, результати ми зберемо в один рядок і наприкінці циклу разом виведемо його на екран. Отже, код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  s: string; //для збору результатів ділення
  b: ShortInt; //лічильник
  r: real; //результат ділення
begin
  for b:= -10 to 10 do begin  //початок циклу
    //якщо нуль, не ділимо, а відразу виходимо із циклу:
    if b = 0 then break;
    // ділимо:
    r:= 100 / b;
    //тепер додаємо результат у рядок s:
    s:= s + '100 / ' + IntToStr(b) + ' = ' + FloatToStr(r) + #13;
  end;  //кінець циклу

  //тепер разом виводимо всі отримані результати:
  ShowMessage(s);
end;
```

Зупинимося на деяких моментах. Насамперед, впадає в око, що змінній b замість попереднього типу byte ми призначили тип ShortInt. Адже нам потрібно обрахувати від -10 до 10, а byte має діапазон від 0 до 255, від'ємні величини цим типом не підтримуються. Тип ShortInt також займає 1 байт, але він має діапазон від -128 до 127, і в цьому

випадку підходить нам найбільше. Якщо ж ми спробуємо залишити лічильнику тип byte, то компілятор просто виведе помилку порушення припустимого діапазону, і не збере виконувану програму та не запустить її.

Далі, для одержання результатів ділення ми використали змінну типу real. Адже ділення одного цілого числа на інше не завжди дасть ціле число! Наприклад, 100 / -8 = -12,5. Тому й змінна для результату в нас дійсного типу.

Потім у нас починається цикл. Спочатку ми перевіряємо, чи не дорівнює b нулю. Якщо дорівнює, то відразу ж завершуємо цикл, при цьому керування передається на наступний за циклом оператор

```
ShowMessage(s);
```

Якщо b не дорівнює нулю, то цикл продовжує свою роботу, виконуючи ділення 100 на b. Оскільки значення b буде змінюватися при кожному проході, то й результат увесь час буде інший. І так буде, поки b не поміняє значення від -10 до 0. Як тільки це трапиться, цикл закінчиться.

Самим складним рядком циклу є оператор збору рядка s:

```
s:= s + '100 / ' + IntToStr(b) + ' = ' + FloatToStr(r) + #13;
```

Подивимось, як формується цей рядок. На початку виразу в нас стоїть «s + «, тобто, попереднє значення s не губиться, а додається до нового рядка, причому при першому проході циклу s ще порожня, зате потім вона вже буде містити текст! Потім ми збираємо рядок з різних частин. Спочатку йде частина «100 / », потім до неї додається значення лічильника b, перетворене в рядок, потім додається частини рядка « = ». Потім до рядка додається результат ділення, перетворений функцією FloatToStr() у рядковий вираз. І в самому кінці ми додаємо символ переходу на новий рядок #13. Таким чином, у нас кожн вираз буде на новому рядку, не змішуючись у кашу. В результаті натискання на кнопку ми одержимо наступне повідомлення:

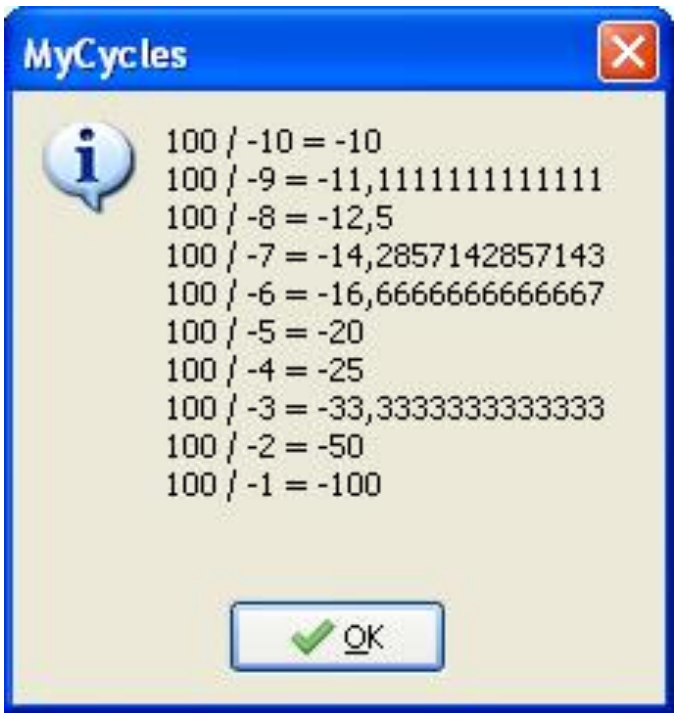

Рис. 10.1. Достроковий вихід із циклу

Як бачите, рядок циклу

```
if b = 0 then break;
```

не дозволив нам зробити ділення на нуль, і ми уникли помилки. Але отут відразу кидається незавершеність роботи циклу. А як же ціла частина, від 1 до 10?! Щоб виправити положення, замініть break на continue:

```
if b = 0 then continue;
```

Continue - інструкція пропуску частини, що залишилася, циклу й перехід до нового кроку циклу. Цикл при цьому не завершується.

Натиснувши на кнопку, ви переконаєтеся, що тепер цикл виконується до кінця, від -10 до 10, пропустивши лише нульове значення. Однак і отут допитливий користувач зможе знайти недолік. Куди дівся нуль? Ну так, на нуль ділити не можна, але щось адже потрібно вивести в цій позиції! Давайте спрацюємо, як стандартний калькулятор Windows: у випадку, якщо b дорівнює нулю, виведемо повідомлення, що ділення на нуль заборонено. Нам потрібно тільки змінити умову if, зробивши складений оператор:

```
if b = 0 then begin
```

```
  s:= s + 'Ділення на нуль заборонене';
  continue;
end;
```

Як бачите, у випадку, якщо b дорівнює нулю, ми в даній позиції виводимо відповідне повідомлення, після чого пропускаємо всю частину, що залишилася, даного кроку циклу (не робимо ділення).

Про всяк випадок, повний код процедури:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  s: string; //для збору результатів ділення
  b: ShortInt; //лічильник
  r: real; //результат ділення
begin
  for b:= -10 to 10 do begin  //початок циклу
    //якщо нуль, не ділимо, а пропускаємо крок циклу з відповідним повідомленням:
    if b = 0 then begin
      s:= s + 'Ділення на нуль заборонене' + #13;
      continue;
    end;
    //спочатку ділимо:
    r:= 100 / b;
    //тепер додаємо результат у рядок s:
    s:= s + '100 / ' + IntToStr(b) + ' = ' + FloatToStr(r) + #13;
  end;  //кінець циклу

  //тепер разом виводимо всі отримані результати:
  ShowMessage(s);
end;
```

В результаті ми отримаємо таке повідомлення:

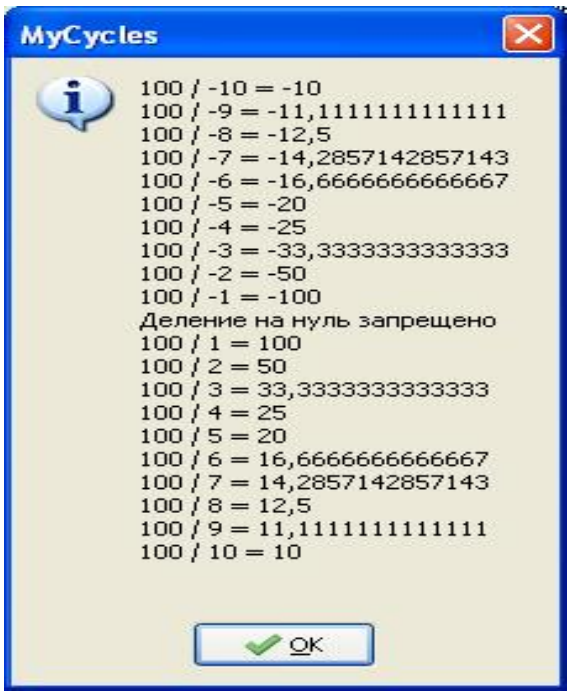

Рис. 10.2. Пропуск кроку циклу

4. Умовний цикл while...do

Слово while перекладається як поки, а do - як робити. Інакше кажучи, цикл while...do виконується доти, поки якась умова вірна. Синтаксис:

```
while <умова> do <оператор>;
```

Тут, умова – будь який умовний вираз або логічна змінна. Якщо змінна або вираз мають значення true, то цикл буде виконуватися доти, поки це значення не стане false. І тільки тоді цикл припинить свою роботу.

Якщо ж значення умови постійно буде false, то цикл не буде виконаний жодного разу.

Оператор також може бути складним, якщо перебуває в дужках begin...end.

У циклі while...do також можна використати оператори break та continue.

У цьому циклі дуже легко зробити помилку, не передбачивши всередині циклу зміни умови. От найпростіший приклад:

```
var b: boolean;
begin
  b:= true;
```

```
while b = true do
  ShowMessage('Істина');
```

Тут цикл while буде виконуватися доти, поки b буде залишатися true. А оскільки всередині циклу не передбачена зміна значення b, цикл вийшов «зациклением». Він буде виводити повідомлення «Істина» доти, поки у вас не згорить комп'ютер, або поки ви самі його зі злості не розіб'єте (жартую, у вас завжди їсти можливість перервати завислу програму).

Зверніть увагу на рядок

```
while b = true do
```

Тут відразу було зрозуміло, що до чого. Для циклу важливо, щоб умова була дійсною. Оскільки b дійсно дорівнює true, та умова істинно. Але в цьому випадку можна було написати простіше:

```
while b do
```

Адже b і так містить true! А зациклений цикл можна написати ще зрозуміліше, взагалі обійшовшись без змінної:

```
while true do
```

Робити, поки істина, тобто, завжди. Всі ці три приклади будуть рівноцінні. Однак повернемося до нашого проекту, і випробуємо цикл на практиці. Ми будемо 100 ділити на будь-яке введене користувачем число, і виводити результат. Причому будемо робити це, поки користувачеві не набридне, тобто, заздалегідь невідомо, скільки разів. Коли ж йому набридне вводити числа, він введе нуль. Ми зі своєї сторони виведемо повідомлення, що на нуль ділити не можна, і на цьому завершимо роботу циклу. Перевірку на коректність введеного числа робити не будемо. Користувач сам буде контролювати введене значення.

Щоб не займатися переробкою коду, додамо на форму ще одну кнопку, в Caption якій напишемо

```
while...do
```

щоб було зрозуміло, яка кнопка з яким циклом працює. Для кнопки згенеруємо таку подію OnClick:

```
procedure TfMain.Button2Click(Sender: TObject);
var
  s: string; //для одержання числа від користувача
  r: real; //результат ділення
  begin
   //починаємо цикл
   while true do begin
    //очистимо рядок:
    s:= '';
    //одержимо число від користувача:
    InputQuery('100 / на...', 'Введіть число', s);
    //якщо нуль, виводимо повідомлення й виходимо із циклу:
    if s = '0' then begin
     ShowMessage('На нуль ділити не можна!');
     break;
    end;
    //якщо ще не вийшли, значить не нуль. ділимо:
    r:= 100 / StrToFloat(s);
    //виводимо результат:
    ShowMessage('100 / ' + s + ' = ' + FloatToStr(r));
   end;  //кінець циклу
end;
```

Коментарі досить докладні, але все-таки прояснимо деякі деталі. Насамперед, ми зробили «зациклений» цикл:

```
   while true do begin
```

Адже заздалегідь ми не знаємо, коли користувачеві набридне вводити різні числа, і він введе нуль, щоб завершити цикл. Але усередині циклу в нас передбачений достроковий вихід із циклу:

```
    if s = '0' then begin
     ShowMessage('На нуль ділити не можна!');
     break;
```

```
end;
```

Тому комп'ютер не зависне - вартує користувачеві ввести нуль, як цикл припиниться. Як користуватися функцією InputQuery() ви, сподіваюся, не забули? Якщо забули, поверніться до лекции №6. Все інше повинно бути зрозуміло по попередніх прикладах.

Коли цикл почнеться, користувачеві вийде запит:

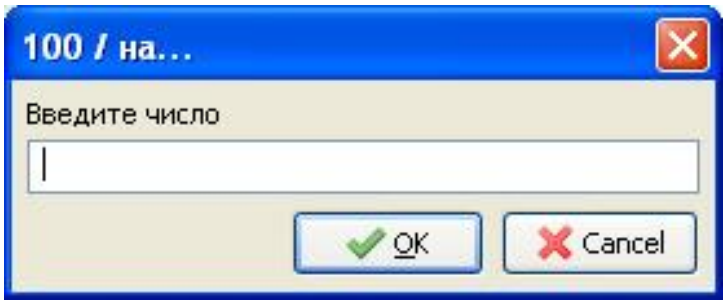

Рис. 10.3. Запит користувачеві

При введенні числа не забувайте про десятковий дільник, що залежить від мови операційної системи. Якщо введете неправильне (некоректне) число, програма зависне. Скинути налагодження просто: **<Запуск -> Скинути отладчик>**.

5. Умовний цикл repeat...until

Цикл repeat...until - ще один умовний цикл. Однак є й відмінності. Якщо while...do - цикл із передумовою, то repeat...until має післяумову. Тобто, якщо в циклі while...do спочатку перевіряється умова, і залежно від цієї умови приймається рішення - чи виконувати цикл, то в repeat...until спочатку виконується цикл, а потім тільки перевіряється умова. Інакше кажучи, цикл repeat...until обов'язково буде виконаний хоча б один раз. Причому, знову таки, на відміну від while...do, цикл repeat...until буде виконуватися доти, поки умова хибна, і припинить свою роботу, коли умова стане істиною.

Англ. repeat перекладається, як повторювати, until - доти, поки... Іншими словами - виконувати будь які дії доти, поки не виконається умова. Синтаксис простій:

```
repeat <оператор(и)> until <умова>;
```

Зверніть увагу: ключові службові слова repeat...until працюють, як дужки оператора begin...end, тому в тіло циклу можна включити скільки завгодно операторів. Перевіримо роботу циклу на прикладі. Додайте в проект третю кнопку, в Caption якій напишіть

repeat...until

Код натискання на кнопку буде наступний:

```
procedure TfMain.Button3Click(Sender: TObject);
var
  i: byte;
  s: string;
begin
  //задамо початкове значення лічильника:
  i:= 1;
  //тепер сам цикл:
  repeat
    s:= s + 'Квадрат від ' + IntToStr(i) + ' дорівнює ' + IntToStr(i * i) + #13;
    i:= i + 1;
  until i > 10;
  //звіт:
  ShowMessage(s);
end;
```

Як бачите, всередині циклу виконується два оператори, при цьому ніяких дужок begin...end не використовувалось. Всередині циклу ми забезпечуємо нарощування лічильника - якщо цього не зробити, програма зациклиться, тому що ніколи не стане більше 10. У нашому прикладі збільшується, і як тільки вона стане більше 10, цикл припиняє роботу, керування передається далі, на ShowMessage(). У результаті ми отримаємо наступний звіт:

Рис. 10.4. Робота циклу repeat…until

Оператори break та continue також можна використовувати в цьому циклі.

6. Перемикач case

Перемикач case не відноситься до циклів, це оператор множинного вибору. Синтаксис:

```
case <Селектор> of
  <Значення 1>: <Оператор 1>;
 <Значення 2>: <Оператор 2>;
 …
 <Значення n>: <Оператор n>
 else <Оператор Інакше>;
end;
```

Тут, Селектор - це змінна або вираз обчислювального типу. Тобто, як правило, це ціле число або символ (для ПК символ - число, номер у таблиці ANSI, тобто, обчислювальний тип), або вираз, що повертає в результаті ціле число або символ.

Далі, залежно від значення селектора, виконується відповідний оператор. Якщо потрібно виконати блок операторів, їх беруть в дужки оператора begin...end.

Блок else не є обов’язковим, але якщо він є, то цей блок виконується в тому випадку, якщо значення селектора не відповідає

жодному значенню, що перевіряється. Як звичайно, перед else крапка з комою не ставиться, а після else не потрібно ставити двокрапку.

Для прикладу встановимо на форму останню, четверту кнопку, в Caption яку напишемо:

```
case
```

Код події натискання на кнопку буде таким:

```
procedure TfMain.Button4Click(Sender: TObject);
var
  i: integer;
  s: string;
begin
  //очистимо рядок:
  s:= '';
  //одержимо число від користувача:
  InputQuery('Квадрат цілого числа', 'Введіть ціле число від 1 до 3:', s);
  //перетворимо рядкове подання числа в число:
  i:= StrToInt(s);
  //тепер вибір case
  case i of
    1: ShowMessage('Квадрат від 1 дорівнює 1');
    2: ShowMessage('Квадрат від 2 дорівнює 4');
    3: ShowMessage('Квадрат від 3 дорівнює 9')
    else ShowMessage('Невідоме число, немає рішення');
  end;
end;
```

Код досить зрозумілий, щоб його додатково коментувати. Коли користувач натисне на кнопку «**case**», вийде запит на введення цілого числа. Якщо ввести число від 1 до 3, то буде виконаний відповідний оператор. Якщо ціле число буде іншим, виконається оператор else. Якщо ввести не ціле число, а щось інше, то відбудеться помилка, яку можна буде зняти, скинувши налагоджувач.

Якщо вам буде потрібно виконати більше одного оператора в кожному випадку, то це можна оформити, наприклад, так (код для прикладу, виконувати його вам не потрібно):

```
case i of
 1 : begin
       ShowMessage('i = 1');
       f:= 10;
     end;
 2 : begin
       ShowMessage('i = 2');
       f:= 20;
     end;
 3 : begin
       ShowMessage('i = 3');
       f:= 30;
     end;
 else ShowMessage ('i не дорівнює 1, 2, або 3');
end; //кінець case
```

Важливо, цей же код можна було б реалізувати й за допомогою умовного оператора if:

```
if i = 1 then ...
else i = 2 then ...
```

і так далі. Однак даний приклад більше підходить для інструкції case - код виходить більше компактним.

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.