helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

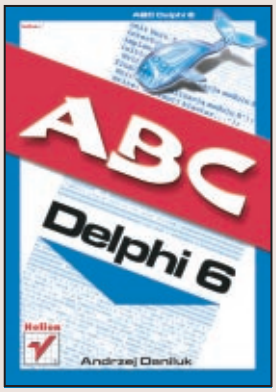
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

ABC Delphi 6



Autor: Andrzej Daniluk
ISBN: 83-7197-504-X
Format: B5, stron: 136
Przykłady na ftp: 1002 kB

Delphi 6 to kolejna wersja popularnego środowiska programistycznego firmy Borland, służącego do szybkiego tworzenia aplikacji za pomocą języka ObjectPascal. W Delphi napisano już wiele profesjonalnych aplikacji, co nie oznacza, iż jest ono środowiskiem wyłącznie dla zawodowców. Wręcz przeciwnie, dzięki prostocie obsługi i zaletom wzorowanego na Pascalu języka ObjectPascal, jest ono doskonałym narzędziem dla początkujących programistów, także dla tych, którzy nie mieli wcześniej wiele wspólnego z programowaniem obiektowym.

Dla nich właśnie przeznaczona jest ta książka omawiająca:

- Podstawy programowania w języku ObjectPascal
- Projektowanie zorientowane obiektowo (OOD)
- Zintegrowane środowisko programistyczne
- ObjectPascal w wydaniu Delphi 6
- Biblioteki VCL i CLX
- Tworzenie własnych komponentów
- Biblioteki DLL

Pomocą w zgłębianiu tajników Delphi 6 jest 19 kompletnych przykładowych projektów, ilustrujących najważniejsze ćwiczenia. Po przeczytaniu „ABC Delphi 6”, będziesz mógł samodzielnie pisać aplikacje działające w środowisku Windows. Książka stanowi także doskonały wstęp do innych, bardziej zaawansowanych pozycji, omawiających Delphi.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Wstęp 5

Rozdział 1. Elementarz Object Pascala 7

Moduły 7
Program główny 8
Stałe 10
Zmienne 11
Typy całkowite 12
Typy rzeczywiste 12
Typ Currency 13
Typy logiczne 13
Typy znakowe 13
Typy łańcuchowe 14
Literały łańcuchowe 14
Tablice 15
Rekordy 16
Typ okrojony 18
Typ mnogościowy 18
Typ wyliczeniowy 19
Typ Variant 19
Operatory 20
Wskazania i adresy 21
Instrukcje sterujące przebiegiem programu 22
- Instrukcja warunkowa If...Then 22
- Instrukcja warunkowa Case...Of 23
- Instrukcja iteracyjna Repeat...Until 24
- Instrukcja iteracyjna While...Do 25
- Instrukcja iteracyjna For...To...Do 26
- Procedura przerwania programu Break 26
- Procedura przerwania programu Exit 27
- Procedura wyjścia z programu Halt 27
- Procedura zatrzymania programu RunError 27
- Procedura kontynuacji programu Continue 28

Procedury 28
- Parametry formalne 29

Funkcje 31
Moduły na poważnie 32
Podsumowanie 34

Rozdział 2. Projektowanie obiektowe OOD 35
Klasa 35
Obiekt 35
Metody 36
Widoczność obiektów 36
Współdziałanie obiektów 36
Implementacja obiektu 36
Dziedziczenie 36
Podsumowanie 36

Rozdział 3. Środowisko programisty — IDE 37
Biblioteka VCL 39
Karta Standard 40
Karta Additional 41
Karta Win32 43
Karta System 45
Karta Dialogs 46
Biblioteka CLX 47
Karta Additional 48
Karta Dialogs 48
Podsumowanie 48

Rozdział 4. Object Pascal w wydaniu Delphi 49
Formularz 49
Zdarzenia 51
Wykorzystujemy własne funkcje i procedury 56
Metody przeciążane 58
Wyjątki 60
Operacje na plikach 65
Strukturalna obsługa wyjątków 71
Tablice otwarte 72
Tablice dynamiczne 73
Typ OleVariant 74
Rekordy w Delphi 76
Podsumowanie 83

Rozdział 5. Biblioteka VCL 85
Komponenty TActionList, TImageList, TOpenDialog, TSaveDialog i TMainMenu 85
Komponenty TActionManager i TActionMainMenuBar 91
Komponenty TFrame, TSpinEdit i TStaticText 96
Hierarchia własności obiektów. Właściciele i rodzice 100
Konstruktor i Destruktor 102
Podsumowanie 103

Rozdział 6. Biblioteka CLX 105
Komponenty TTimer i TLCDNumber 105
Podsumowanie 109

Rozdział 7. Tworzymy własne komponenty 111
Podsumowanie 117

Rozdział 8. Biblioteki DLL 119
Podsumowanie 126

Skorowidz 127

Rozdział 4.

Object Pascal w wydaniu Delphi

Rozdział ten poświęcony jest omówieniu praktycznych sposobów wykorzystania poznanych wcześniej elementów języka Object Pascal w graficznym środowisku Delphi 6. Zapoznamy się tutaj m. in. z pojęciem formularza, wyjątku czy procedury obsługi zdarzenia. Poznamy również metody wykorzystania w aplikacji własnych funkcji i procedur. Zastosowanie omówionych elementów Delphi zostanie zilustrowane odpowiednimi ćwiczeniami.

Formularz

Formularz jest pierwszym obiektem, z którym spotykamy się, rozpoczynając pisanie aplikacji. Po dwukrotnym kliknięciu w obszarze formularza dostajemy się do okna edycji kodu modułu *Unit1.pas*, który pokazany jest na rysunku 4.1.

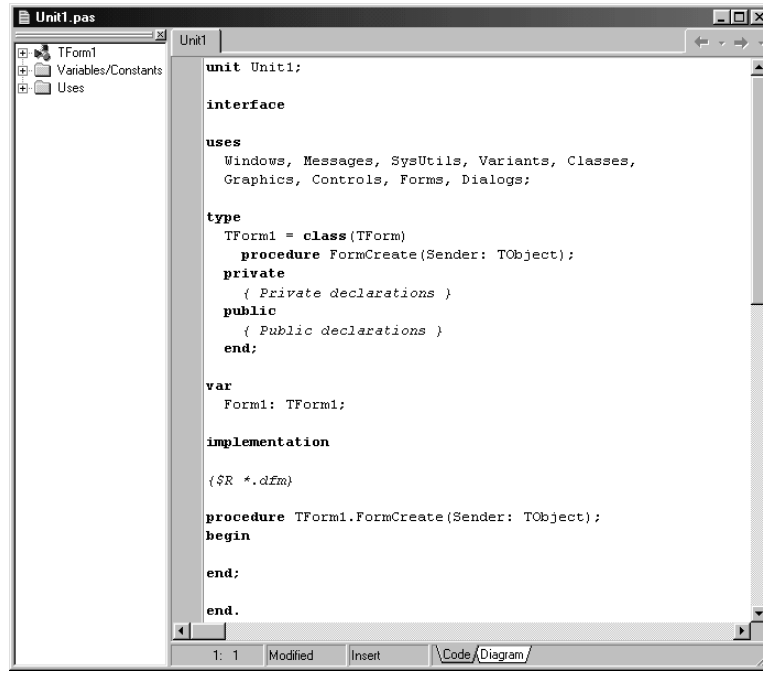
Object Pascal oferuje nam słowo kluczowe `class`, pozwalające na tworzenie obiektów. Przykładowa definicja klasy formularza wygląda następująco:

```
type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Zdefiniowana klasa dziedziczy własności bazowej klasy formularza *TForm*, natomiast sam formularz, traktowany jako zmienna obiektowa, deklarowany jest jako:

```
var
  Form1: TForm1;
```

Rysunek 4.1. *Okno edycji kodu głównego modułu aplikacji*



Z zapisu tego odczytamy, iż formularz jest zmienną obiektową, natomiast nazwa klasy stała się nowym specyfikatorem typu danych.

W definicji klasy możemy zauważyć procedurę:

```
procedure FormCreate(Sender: TObject);
```

Delphi odpowiednio inicjuje formularz (tylko jeden raz), kiedy jest on tworzony po raz pierwszy. *Sender* jest pewną zmienną typu *TObject*, wołaną przez wartość. W rzeczywistości *Sender* reprezentuje pewną właściwość, polegającą na tym, iż każdy obiekt łącznie z formularzem (oraz każdy obiekt VCL i CLX) musi być w pewien sposób poinformowany o przyszłym przypisaniu mu pewnego zdarzenia (w przypadku formularza zdarzenie to polega na jego inicjalizacji).



TObject jest bezwzględnym przodkiem wszystkich komponentów i obiektów VCL oraz CLX i umieszczony jest na samym szczycie hierarchii obiektów.

Z rysunku 4.1 możemy odczytać, iż standardowa definicja klasy składa się z kilku części. Sekcja *public* służy do deklarowania funkcji i procedur (czyli metod) oraz zmiennych (zwanych polami), które w przyszłości mogą być udostępniane innym. Zasadniczą różnicą pomiędzy metodami a zwykłymi funkcjami czy procedurami jest to, że każda metoda posiada niejawny parametr *Self*, wskazujący na obiekt będący przedmiotem wywołania tej metody. Sekcję *public* często nazywamy **interfejsem obiektu**. Sekcja *private* przeznaczona jest dla pól i metod widzianych jedynie wewnątrz klasy.

Oprócz elementów wymienionych, definicja klasy może posiadać jeszcze sekcje *protected* oraz *published*. W części *protected* można definiować pola i metody widoczne dla macierzystej klasy i klas po niej dziedziczących. Deklaracje zawarte w sekcji *published* (publikowanej) pełnią taką samą rolę, jak deklaracje umieszczone w sekcji *public* (publicznej). Różnica pomiędzy nimi polega na tym, iż te pierwsze nie tworzą tzw. informacji czasu wykonania.

Zdarzenia

Zdarzenie (ang. *event*) określane jest jako zmiana, która występuje w aktualnym stanie obiektu i jest źródłem odpowiednich komunikatów, przekazywanych do aplikacji lub bezpośrednio do systemu. Reakcja obiektu na wystąpienie zdarzenia udostępniana jest aplikacji poprzez procedurę obsługi zdarzeń (ang. *event procedure*) będącą wydzieloną częścią kodu. Rolę zdarzeń w aplikacji najlepiej jest prześledzić, wykonując praktyczne ćwiczenie.

Tradycyjnie już założymy na dysku nowy katalog. Po uruchomieniu Delphi 6 znanymi nam już poleceniami menu zapiszmy w nim główny moduł aplikacji, który nazwiemy *Unit_08.pas*. Zapiszmy również projekt aplikacji pod nazwą *Projekt_08.dpr*.

Rozmiary formularza ustalimy, korzystając z cech *Height* (wysokość) i *Width* (szerokość), znajdujących się w karcie właściwości (*Properties*) *Inspektora Obiektów*. Jeżeli chcemy, aby po uruchomieniu formularz nie „rozpływał" się po ekranie w odpowiedzi na kliknięcie pola maksymalizacji, w *Inspektorze Obiektów* rozwińmy cechę *Constraints* (ograniczenie) i we właściwe miejsca wpiszmy żądane rozmiary formularza (w pikselach), tak jak pokazano na rysunku 4.2.



Rysunek 4.2. *Ograniczenie rozmiarów formularza*

Przejdźmy następnie do cechy *Position* i wybierzmy *poScreenCenter*. Wybrane przypisanie spowoduje, że w momencie uruchomienia aplikacji formularz pozostanie w centrum ekranu (ale nie pulpitu *poDesktopCenter*); jeżeli oczywiście w Inspektorze Obiektów cechy *Align* (zakotwiczenie) nie ustawiliśmy inaczej niż w pozycji *alNone*.

Na tak przygotowanym formularzu umieśćmy teraz dwa komponenty reprezentujące klasę *TButton* z karty *Standard*. Korzystając z Inspektora Obiektów oraz z karty właściwości, cechy *Caption* przycisków *Button1* oraz *Button2* zmieńmy odpowiednio na *&Zamknij* i *&Tekst*. Znak *&*, który występuje w nazwach przycisków, spowoduje, że litera, występująca bezpośrednio za nim, stanowić będzie klawisz szybkiego dostępu do procedury obsługi wybranego zdarzenia. W podobny sposób możemy zmienić cechy *Font* wybranych przycisków. Dla każdego z przycisków stworzymy procedurę obsługi odpowiedniego zdarzenia. Klikając dwukrotnie przycisk *Zamknij* lub w widoku drzewa obiektów (*Object TreeView*) odpowiednio oznaczony komponent, dostaniemy się do wnętrza właściwej procedury obsługi zdarzenia:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
end;
```

Już w tym miejscu możemy zauważyć, iż w definicji klasy program Delphi wygenerował automatycznie deklarację przycisku oraz deklarację procedury obsługującego go zdarzenia. Korzystając z notacji kropkowej, kompilator automatycznie został poinformowany, do której klasy należy wywoływana procedura.

Użycie notacji kropkowej stanowi informację dla kompilatora, że przykładowa procedura *Button1Click()* należy do przykładowej klasy *TForm1* (jest metodą zdefiniowaną w klasie *TForm1*).

Należy zawsze pamiętać, iż szkielety procedur obsługi odpowiednich zdarzeń, np. takich jak `Button1Click()`, zostaną automatycznie wygenerowane przez Delphi w odpowiedzi na dwukrotne kliknięcie danego przycisku. W żadnym wypadku procedur tych nie należy wpisywać samodzielnie.

Omawianą procedurę obsługi zdarzenia wypełnimy przedstawionym poniżej kodem, co spowoduje, że po naciśnięciu wybranego przycisku aplikacja zostanie zamknięta.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Application.Terminate();
end;
```

Każdy standardowy program Delphi (oparty na formularzu) zawiera pewną zmienną globalną o nazwie *Application* typu *TApplication*. W czasie tworzenia nowego projektu Delphi konstruuje obiekt aplikacji i przypisuje mu właśnie zmienną *Application*. Obiekty klasy *TApplication* przechowują informacje odnośnie zasad współpracy aplikacji z systemem operacyjnym. Informacje te dotyczą np. rozpoczynania i kończenia działania aplikacji czy tworzenia okna głównego programu. Istnieje wiele metod klasy *TApplication*. Jedną z nich jest `Terminate()`, umożliwiająca zamknięcie aplikacji.

Zajmiemy się teraz zaprojektowaniem procedury obsługi zdarzenia przycisku *Button2*, który nazwaliśmy *Tekst*. Po kliknięciu tego przycisku, bezpośrednio na formularzu zostanie wyświetlony prosty tekst. Skorzystamy z tego, iż formularz, będący w istocie też komponentem, posiada swoje własne płótno (ang. *canvas*), reprezentowane przez klasę *TCanvas*, posiadającą właściwość *Canvas*. Procedura obsługi naszego zdarzenia będzie wyglądać następująco:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Canvas.Font.Color:=clBlue;
  Canvas.Font.Height:=30;
  Canvas.TextOut(ClientHeight div 4,ClientWidth div 4,
                 ' Pierwsza aplikacja w Delphi 6');
end;
```

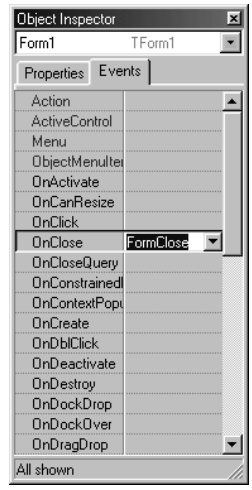
Powyższy przykład ilustruje sposób odwoływania się do obszaru płótna poprzez właściwość *Canvas* klasy *TCanvas* z wykorzystaniem właściwości czcionka (*Font*). Zastosowana metoda `TextOut()`:

```
procedure TextOut(X, Y: Integer; const Text: string);
```

pozwala na umieszczenie dowolnego tekstu identyfikowanego przez stałą *Text* w miejscu formularza o współrzędnych *X*, *Y* (odległość liczona jest w pikselach, natomiast lewy górny róg formularza posiada współrzędne 0, 0).

Może również zdarzyć się i taka sytuacja, w której zechcemy zamknąć formularz, korzystając bezpośrednio z jego pola zamknięcia (por. rysunek 3.2). Zamykając w ten sposób działającą aplikację, należy mieć na uwadze fakt, iż w Windows 9x bezpośredni sposób zamknięcia działającego programu może nie działać w pełni poprawnie (nie dotyczy to Win 2000, NT i XP). Aby mieć pewność, że w momencie zamknięcia aplikacji wszystkie jej zasoby zostaną prawidłowo zwolnione, skorzystamy ze zdarzenia `OnClose`. Klikając (tylko raz) w obszar formularza w inspektorze obiektów, przejdźmy do zakładki *Events* (zdarzenia). Zdarzenie `OnClose` określmy jako `FormClose` (rysunek 4.3) i potwierdźmy klawiszem *Enter*.

Rysunek 4.3.
Zdarzenie OnClose



W ten sposób Delphi automatycznie wygeneruje procedurę obsługi zdarzenia

```
FormClose(Sender: TObject;  var Action: TCloseAction);
```

Zmiennej `Action` (akcja) można tutaj przypisać jeden z elementów typu wyliczeniowego

```
type
TCloseAction = (caNone, caHide, caFree, caMinimize);
```

gdzie:

- `caNone` oznacza, że formularz nie zostanie zamknięty;
- `caHide` oznacza, że formularz nie zostanie zamknięty, lecz ukryty;
- `caFree` oznacza, że formularz zostanie zamknięty z jednoczesnym zwolnieniem wszystkich zasobów pamięci, z których aktualnie korzysta;
- `caMinimize` oznacza, że formularz zostanie zminimalizowany.

Procedurę obsługi zdarzenia `FormClose()` wypełnimy następującym kodem:

```
procedure TForm1.FormClose(Sender: TObject;
var Action: TCloseAction);
begin
  case (MessageBox(0, 'Zamknięcie aplikacji ?','Uwaga',
        MB_YESNO or MB_ICONQUESTION)) of
     ID_YES:
        Action := caFree;
     ID_NO:
        Action := caNone;
  end;
end;
```

Skorzystaliśmy tutaj z omawianej wcześniej instrukcji `Case...Of` oraz z funkcji `MessageBox()`, wyświetlającej odpowiedni komunikat w okienku dialogowym.

Projekt aplikacji *Projekt_08.dpr*, wykorzystującej omawiane zdarzenia, znajduje się w katalogu *Kody\08*, natomiast na wydruku 4.1 zamieszczono kod źródłowy modułu *Unit_08.pas*.

Wydruk 4.1. *Moduł Unit_08.pas projektu Projekt_08.dpr*

```
unit Unit_08;

interface

uses
Windows, Messages, SysUtils, Classes,
Graphics, Controls, Forms, Buttons,
StdCtrls;

type
TForm1 = class(TForm)
Button1: TButton;
Button2: TButton;
procedure Button1Click(Sender: TObject);
procedure Button2Click(Sender: TObject);

procedure FormClose(Sender: TObject; var Action: TCloseAction);

private
{ Private declarations }
public
{ Public declarations }
end;

var
Form1: TForm1;

implementation

{$R *.DFM}

//-----------------------------------------------------------
procedure TForm1.Button2Click(Sender: TObject);
begin
```

```
  Canvas.Font.Color:=clBlue;
  Canvas.Font.Height:=30;
  Canvas.TextOut(ClientHeight div 4,ClientWidth div 4,
                 ' Pierwsza aplikacja w Delphi 6');
end;
//---------------------------------------------------------
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  case (MessageBox(0, 'Zamknięcie aplikacji ?','Uwaga',
        MB_YESNO or MB_ICONQUESTION)) of
     ID_YES:
       Action := caFree;
     ID_NO:
       Action := caNone;
  end;
end;
//---------------------------------------------------------
procedure TForm1.Button1Click(Sender: TObject);
begin
   Application.Terminate();
end;
//---------------------------------------------------------
end.
```

Zajrzyjmy teraz do katalogu, w którym został zapisany nasz projekt. Oprócz plików, wymienionych w rozdziale 1, pojawiły się tam 3 dodatkowe:

- *.dpp* — plik *Diagram Delphi Page*. Można w nim zapisać diagramy będące wizualizacją logicznych relacji pomiędzy widocznymi i niewidocznymi komponentami, składającymi się na całość projektu. Relacje takie można uzyskać, korzystając z zakładki Diagram.
- *.dfm* — plik zawierający opis formularza. Dyrektywa kompilatora `{$R}` lub `{$RESOURCE}` dołącza do projektu zewnętrzny plik zasobów, zawierający opis formularza. Plik zasobów jest włączany do końcowego pliku wynikowego *.exe* w czasie łączenia projektu.
- *.res* — plik zasobów Windows.

Żadnego z nich nie powinnyśmy utracić.

Plik projektu Delphi *.dpr* środowiska graficznego różni się nieco od swojego odpowiednika aplikacji konsolowych. Programy Delphi są bardzo krótkie. Wynika to z faktu, iż aplikacje z interfejsem graficznym zazwyczaj wywołują procedurę inicjalizacyjną `Application.Initialize()`, następnie tworzony jest formularz (lub formularze) *Application.CreateForm(TForm1,Form1)*, w którym uruchamiane są procedury obsługi zdarzeń `Application.Run()`. Aplikacja zamykana jest poprzez główny formularz w odpowiedzi na wywołanie procedury obsługi odpowiedniego zdarzenia `Application.Terminate()` (patrz wydruk 4.1). Typowy plik kodu źródłowego projektu Delphi przedstawiony jest na wydruku 4.2.

Wydruk 4.2. *Typowy plik programu Delphi*

```
program Projekt_08;

uses
Forms,
Unit_08 in 'Unit_08.pas' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Deklaracja `uses` tworzy listę modułów wchodzących w skład programu. Nazwa modułu może być poprzedzona dyrektywą `in`, specyfikującą nazwę pliku. Moduły bez dyrektywy `in` są modułami bibliotecznymi i nie stanowią części kodu źródłowego projektu.

Wykorzystujemy własne funkcje i procedury

Własne procedury i funkcje możemy umieszczać w programie Delphi na parę sposobów. Zapoznamy się teraz z dwoma sposobami — najprostszymi i najczęściej stosowanymi.

Definicję funkcji lub procedury umieszczamy bezpośrednio w kodzie programu w sekcji implementacji modułu:

```
var
  Form1: TForm1;
implementation
{$R *.dfm}
procedure Power2toN(var x: Integer; y: Integer);
var
  i,z: Integer;
begin
  z:=1;
  for i:=1 to y do
   begin
     z:=z*x;
     Form1.Memo1.Lines.Add(IntToStr(x)+' do potęgi '+
                     IntToStr(i)+'='+IntToStr(z));
   end;
end;
```

Wyświetlanie kolejnych liczb całkowitych, będących kolejnymi potęgami całkowitego parametru *x*, dokonywane jest w komponencie edycyjnym *Memo1*, będącym reprezentantem klasy *TMemo* z karty *Standard*. Ponieważ nasza procedura nie została zadeklaro-

wana w definicji klasy formularza, więc odwołanie się w jej wnętrzu do odpowiedniego komponentu edycyjnego wymaga, aby jawnie wskazać, iż komponent ten należy do formularza *Form1*. Wyświetlanie kolejnych liczb całkowitych dokonywane jest za pośrednictwem funkcji `IntToStr()` formatującej wartość liczbową (wartość w postaci liczby całkowitej) na odpowiedni łańcuch znaków.

Dużo subtelniejszym (ale nie zawsze opłacalnym sposobem) jest uczynienie funkcji lub procedury jawnym obiektem klasy formularza *TForm1*. Należy wówczas definicję nagłówkową funkcji lub procedury uzupełnić o nazwę klasy, do której ma przynależeć. Musimy ponadto omawiane funkcje lub procedury zadeklarować w definicji klasy w jednej z sekcji, np.

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    Memo1: TMemo;
    Button2: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
    procedure Power2toN(x, y: Cardinal);
public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.dfm}
//-----------------------------------------------------------
procedure TForm1.Power2toN(x, y: Cardinal);
var
  i, z: Cardinal;
begin
  z:=1;
  for i:=1 to y do
   begin
     z:=z*x;
     Memo1.Lines.Add(IntToStr(x)+' do potęgi '+
                     IntToStr(i)+'='+IntToStr(z));
   end;
end;
```

Korzystając z tego sposobu definiowania własnej funkcji lub procedury, możemy wewnątrz nich bez problemów odwoływać się do innych obiektów formularza. Wszystkie własności, cechy, zdarzenia i metody właściwe tym obiektom będą widoczne w naszej funkcji lub procedurze.

Metody przeciążane

Chociaż można przypuszczać, iż pojęcia procedur lub funkcji przeciążanych (przeładowywanych) należy wprowadzać na bardziej zaawansowanym kursie programowania, to jednak wydaje się, iż w sytuacji kiedy środowiska programistyczne stają się coraz bardziej rozbudowane, powinniśmy posiadać pewne wiadomości na ten temat. Jest to szczególnie ważne w momencie, kiedy zechcemy samodzielnie posługiwać się plikami pomocy.

Pisząc programy w Delphi, możemy zadeklarować wiele funkcji czy procedur o tej samej nazwie, ale o różnej postaci argumentów. W momencie wywołania danej procedury czy funkcji kompilator powinien je rozróżniać. Do deklaracji procedur (funkcji) przeładowywanych służy dyrektywa `overload` (przeładowanie). Najlepszym (i jedynym) sposobem zapoznania się z ideą funkcji (procedur) przeładowywanych jest samodzielne napisanie prostej aplikacji, w której funkcje te zostaną wykorzystane. Stworzymy program wykorzystujący znane nam już procedury obliczające kolejne całkowite potęgi wybranej liczby, ale w ten sposób, by jednocześnie można było korzystać z procedur, których parametry wołane są przez zmienną i przez wartość. Przy okazji zapoznamy się z podstawowymi własnościami komponentu edycyjnego *TMemo*.

W skład formularza projektu *Kody\09\Projekt_09.dpr* będą wchodzić trzy przyciski będące reprezentantami klasy *TButton* oraz jeden reprezentant klasy *TMemo*. Umieśćmy je na formularzu w sposób pokazany na rysunku 4.4. W inspektorze obiektów właściwość *Caption* formularza *Form1* zmieńmy na *Projekt_09*.



Rysunek 4.4. *Podstawowe elementy formularza projektu Projekt_09.dpr*

W inspektorze obiektów własności *Caption* komponentów *Button1*, *Button2* i *Button3* zmieńmy odpowiednio na *&Oblicz — wywołanie przez zmienną*, *&Zamknij* i *O&blicz — wywołanie przez wartość*. Można również skorzystać z ich własności *Font*, aby dobrać odpowiednią czcionkę. Należy oczywiście pamiętać o uprzednim zaznaczeniu wybranego komponentu, np. poprzez pojedyncze kliknięcie go myszą.

Ponieważ należy się spodziewać, iż w komponencie edycyjnym *Memo1* będą wyświetlane kolejne liczby, musimy zadbać o możliwość pionowego przewijania zawartości tego komponentu. W tym celu w inspektorze obiektów jego właściwość *ScrollBars* ustalmy jako *ssVertical*. Jeżeli zechcemy, aby zawartość komponentu mogła być przewijana zarówno w pionie, jak i w poziomie, wystarczy wybrać *ssBoth*.

W momencie wybrania reprezentanta klasy *TMemo*, w oknie edycji zobaczymy jego nazwę. Aby nie była ona wyświetlana, należy skorzystać z własności *Lines* i poprzez *TStrings* w oknie edytora skasować napis *Memo1*. Kiedy wszystkie potrzebne komponenty zostaną już rozmieszczone na formularzu, możemy jego rozmiary odpowiednio do nich dopasować. W tym celu w inspektorze obiektów jego własności *AutoSize* wystarczy przypisać wartość *True*.

Po tych wstępnych czynnościach nie pozostaje nam nic innego jak zaimplementować w programie dwie procedury `PowerXtoY()` obliczające kolejne całkowite potęgi *Y* wybranej liczby całkowitej *X*. Musimy również wypełnić treści procedur odpowiednich zdarzeń, podobnie jak zostało to pokazane na wydruku 4.3.

Wydruk 4.3. *Kod głównego modułu Unit_09.pas projektu Projekt_09.dpr, wykorzystującego przeciążane procedury*

```
unit Unit_09;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    Memo1: TMemo;
    Button2: TButton;
    Button3: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
    procedure PowerXtoY(x, y: Cardinal);overload;
    procedure PowerXtoY(var x, y: Integer);overload;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.PowerXtoY(var x, y: Integer);
var
  i,z: Integer;
begin
  z:=1;
  for i:=1 to y do
   begin
     z:=z*x;
```

```
      Memo1.Lines.Add(IntToStr(x)+' do potęgi '+
                      IntToStr(i)+'='+IntToStr(z));
    end;
end;
//---------------------------------------------------------
procedure TForm1.PowerXtoY(x, y: Cardinal);
var
  i,z: Cardinal;
begin
  z:=1;
  for i:=1 to y do
   begin
     z:=z*x;
     Memo1.Lines.Add(IntToStr(x)+' do potęgi '+
                     IntToStr(i)+'='+IntToStr(z));
   end;
end;
//---------------------------------------------------------
procedure TForm1.Button1Click(Sender: TObject);
var a, b: Integer;
begin
  Memo1.Clear();
  a:=3; b:=10;
  PowerXtoY(a, b);   //wywołanie przez zmienną
end;
//---------------------------------------------------------
procedure TForm1.Button3Click(Sender: TObject);
begin
  Memo1.Clear();
  PowerXtoY(2, 10); //wywołanie przez wartość
end;
//---------------------------------------------------------
procedure TForm1.Button2Click(Sender: TObject);
begin
   Application.Terminate();
end;
end.
```

Program skompilujemy, korzystając z klawisza *F9* lub opcji menu *Run*, *Run*. Śledząc kod programu, możemy też zauważyć, iż w celu skasowania zawartości komponentu *Memo1* posługujemy się metodą `Clear()`.

Wyjątki

Podobnie jak w przypadku metod przeciążanych, ktoś mógłby powątpiewać w celowość wprowadzania pojęcia wyjątku w kursie programowania, przeznaczonym dla osób mniej zaawansowanych. Należy jednak zdawać sobie sprawę z faktu, iż wyjątki jako obiekty pełnią bardzo ważną rolę we wszystkich współczesnych systemach operacyjnych oraz środowiskach programowania i pewne własności kompilatora, które jeszcze do niedawna uważano za bardzo zaawansowane, obecnie już takimi być przestają.

Wyjątki pozwalają osobie piszącej kod na uwzględnienie sytuacji, w której program w jakimś momencie wykonywania może ulec niekontrolowanemu zakończeniu. Wyjątek może być umiejscowiony w dowolnej metodzie. W Delphi podstawową klasą, zajmującą się obsługą wyjątków, jest *Exception* (ang. *wyjątek*). Klasa ta wywodzi się bezpośrednio z klasy *TObject*. Z kolei z *Exception* (chodź nie bezpośrednio) wywodzi się klasa *EMathError*, będąca z kolei nadklasą (klasą bazową) dla zbioru niezwykle użytecznych klas, obsługujących wyjątki, powstałe przy wykryciu przez kompilator błędów operacji matematycznych na liczbach zmiennopozycyjnych (liczbach dziesiętnych). Wyjątki dziedziczące z *EMathError* zostały przedstawione w tabeli 4.1.

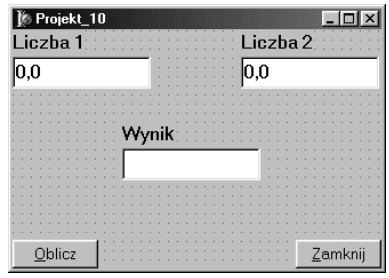
Tabela 4.1. *Klasy dziedziczące z EMathError*

Klasa wyjątku	Znaczenie
`EInvalidArgument`	Przekroczenie zakresu zmienności zadeklarowanego typu danych
`EInvalidOp`	Nieprawidłowa operacja zmiennoprzecinkowa
`EOverflow`	Przekroczenie zakresu typu zmiennoprzecinkowego
`EUnderflow`	Wartość typu zmiennoprzecinkowego jest zbyt mała
`EZeroDivide`	Zmiennoprzecinkowe dzielenie przez zero

Delphi posługuje się dwoma wyrażeniami przeznaczonymi do obsługi wyjątków. Z pierwszym, a zarazem podstawowym z nich, zapoznamy się obecnie. Wyrażenie `try...except...end` uaktywnia procedurę obsługi wyjątku przejmującą kontrolę nad dalszym wykonywaniem programu, w przypadku gdy zostanie wykonana jakaś niedozwolona operacja. Jako przykład niech nam posłuży prosty algorytm wykonujący operację zmiennopozycyjnego dzielenia dwóch liczb, które wprowadzimy do odpowiednich komponentów edycyjnych. Bez trudu możemy przewidzieć, iż wyjątek powstanie przy próbie wykonania dzielenia na znakach nie będących liczbami lub przy próbie dzielenia przez zero.

Zaprojektujmy bardzo prostą aplikację, wykonującą zmiennopozycyjne dzielenie dwóch liczb, wprowadzanych z klawiatury do dwóch nowoczesnych komponentów edycyjnych *LabeledEdit1* i *LabeledEdit2*. Wynik będzie przechowywany w komponencie *LabeledEdit3*.

Stwórzmy nowy formularz projektu *Projekt_10.dpr*, w skład którego wchodzić będą dwa przyciski (reprezentujące klasę *TButton*) oraz trzy komponenty edycyjne (reprezentujące klasę *TLabeledEdit* z karty *Additional*). W inspektorze obiektów właściwościom *Text* komponentów *LabeledEdit1* i *LabeledEdit2* przypiszmy wartości 0,0, aby ich cechy *Text* nie posiadały wartości nieokreślonej. Cechę *Text* komponentu *LabeledEdit3* wykasujmy. Komponenty reprezentujące klasę *TLabeledEdit* posiadają możliwość automatycznego ich opisu. Są one jakby połączeniem komponentów z klas *TLabel* i *TEdit*. Rozwijając w inspektorze obiektów opcje właściwości *EditLabel* komponentu *LabeledEdit1* (obok nazwy właściwości w inspektorze obiektów pojawi się znaczek „–”) opiszmy jego cechę *Caption* jako *Liczba 1*. Podobnie opiszemy wszystkie komponenty edycyjne, tak jak przedstawiono to na rysunku 4.5.



Rysunek 4.5. *Rozmieszczenie i opis komponentów reprezentujących klasę TLabeledEdit*

Bardzo często korzystanie z różnego rodzaju komponentów ułatwiają nam dymki podpowiedzi (ang. *hint*). Jeżeli zechcemy je zastosować na potrzeby naszych okienek edycji, musimy zamknąć opcje właściwości *EditLabel* (obok nazwy tej właściwości w inspektorze obiektów pojawi się znaczek „+"). Zaznaczmy myszką wybrany komponent i odszukajmy w inspektorze obiektów jego właściwość *Hint*, którą opiszemy jako *Podaj pierwszą liczbę*. Aby dymek podpowiedzi był rzeczywiście widoczny w trakcie działania programu, właściwości *ShowHint* należy nadać wartość `True`. Analogicznie postąpimy z następnym komponentem edycyjnym.

Przystąpmy obecnie do wypełniania głównego modułu *Unit_10.pas* naszego formularza odpowiednim kodem. W sekcji implementacji modułu zadeklarujmy własną klasę *EFloatingPointError* (wyjątek dzielenia zmiennopozycyjnego) przechwytującą wyjątki, która będzie dziedziczyć po klasie *EMathError*. Dodatkowo zastosujmy dwie deklaracje `resourcestring`, przechowujące odpowiednie komunikaty, informujące o powstaniu wyjątku podczas zmiennopozycyjnego dzielenia dwóch liczb:

```
implementation
{$R *.dfm}
type
   EFloatingPointError = class(EMathError);
resourcestring
   Msg1 = 'Błąd operacji zmiennopozycyjnych: %s';
   Msg2 = 'Próba dzielenia przez zero: %s';
```

W tym wypadku znaki formatowania `%s` pozwalają na zduplikowanie komunikatów w języku polskim i angielskim.

Procedurę obsługi zdarzenia `Button1Click()`, uruchamianego przyciskiem *&Oblicz*, wypełnimy następującym kodem:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   try
     LabeledEdit3.Text:=FloatToStr(StrToFloat(LabeledEdit1.Text)/
                                   StrToFloat(LabeledEdit2.Text));
   except
     on Ex: EInvalidOp do
       raise EFloatingPointError.CreateFmt(Msg1, [Ex.Message]);
     on Ex: EZeroDivide do
       raise EFloatingPointError.CreateFmt(Msg2, [Ex.Message]);
   end;
end;
```

W klauzuli `try...except`, przeznaczonej do obsługi błędów, umieściliśmy zapis właściwej operacji dzielenia dwóch liczb. Funkcje `StrToFloat()` dokonują konwersji ciągu znaków, przechowywanych w cechach *Text* komponentów *LabeledEdit1* oraz *LabeledEdit2*, na zmiennopozycyjną postać numeryczną, czyli po prostu na liczby z przecinkami. Używając operatora dzielenia zmiennopozycyjnego „/", te dwie liczby podzielimy przez siebie, natomiast wynik dzielenia zostanie z kolei przypisany cesze *Text* komponentu edycyjnego reprezentowanego przez *LabeledEdit3*. Aby postać numeryczna liczby mogła być wyświetlana w oknie edycyjnym, musi zostać zamieniona na łańcuch znaków (w tym wypadku typu *string*). Dokonujemy tego, stosując funkcję `FloatToStr()` konwertującą postać numeryczną liczby na odpowiedni łańcuch znaków.

Każde wyrażenie `try...except...end` może posiadać jedną lub więcej sekcji on*:* `Ex <klasa wyjątku> do`, z których każda deklaruje odrębną klasę wyjątku. Delphi przeszukuje sekcję `on` w zapisanym porządku (od góry do dołu), poszukując aktualnie pasującej klasy wyjątku odpowiadającego występującemu błędowi.

Ogólnie rzecz biorąc, każdy kod potencjalnie mogący wygenerować wyjątek (w naszym przypadku dzielenie zmiennopozycyjne) powinien mieć możliwość przekazania jego obiektu do wyrażenia *raise* (zakańczać, zbierać). W tym miejscu należy wykorzystać własną klasę wyjątku `EFloatingPointError. CreateFmt()` z konstruktorem konwertującym komunikat egzemplarza wyjątku na łańcuch znaków

```
constructor CreateFmt (const Msg: string; const Args: array of const);
```

gdzie argument *Args* jest tzw. wariantową tablicą otwartą (*array of const*), pozwalającą na przekazywanie niejednorodnych komunikatów (*Messages*).

Może zdarzyć się i taka sytuacja, w której kompilator w sekcji `try... except...end` nie znajdzie pasującego obiektu wyjątku. Wówczas należy użyć konstrukcji `try... except...else...raise...end`:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   try
     LabeledEdit3.Text:=FloatToStr(StrToFloat(LabeledEdit1.Text)/
                                   StrToFloat(LabeledEdit2.Text));
   except
     on Ex: EInvalidOp do
       raise EFloatingPointError.CreateFmt(Msg1, [Ex.Message]);
     on Ex: EZeroDivide do
       raise EFloatingPointError.CreateFmt(Msg2, [Ex.Message]);
     else
       raise;
   end;
end;
```

Mamy nadzieję, iż przedstawione w poprzednim punkcie rozważania nie są zbyt skomplikowane dla mniej zaawansowanych Czytelników. Jeżeli jednak ktoś czuje się nieco zagubiony pośród tych pojęć, zawsze można przedstawioną wyżej konstrukcję znacznie uprościć poprzez wyświetlanie komunikatu wyjątku poprzez okno dialogowe *MessageBox()*:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   try
     LabeledEdit3.Text:=FloatToStr(StrToFloat(LabeledEdit1.Text)/
                                   StrToFloat(LabeledEdit2.Text));
   except
     on Ex: EInvalidOp do
       MessageBox(0, 'Błąd operacji zmiennopozycyjnych','Uwaga',
                  MB_OK);
     on Ex: EZeroDivide do
       MessageBox(0, 'Próba dzielenia przez zero','Uwaga', MB_OK);
     else
       raise;
   end;
end;
```

Na wydruku 4.4 przedstawiono kompletny kod źródłowy głównego modułu aplikacji, wykorzystującej obiekty wyjątków powstałych podczas operacji zmiennopozycyjnego dzielenia dwóch liczb.

Wydruk 4.4. *Kod głównego modułu Unit_10.pas projektu KODY\10\Projekt_10.dpr, wykorzystującego przykładowe klasy wyjątków*

```
unit Unit_10;
interface
uses
  Windows, Messages, SysUtils, Variants,
  Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ExtCtrls;

type
  TForm1 = class(TForm)
    Button1: TButton;
    LabeledEdit1: TLabeledEdit;
    LabeledEdit2: TLabeledEdit;
    LabeledEdit3: TLabeledEdit;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

type
   EFloatingPointError = class(EMathError);

resourcestring
   Msg1 = 'Błąd operacji zmiennopozycyjnych: %s';
   Msg2 = 'Próba dzielenia przez zero: %s';
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   try
     LabeledEdit3.Text:=FloatToStr(StrToFloat(LabeledEdit1.Text)/
                                   StrToFloat(LabeledEdit2.Text));
   except
     on Ex: EInvalidOp do
       raise EFloatingPointError.CreateFmt(Msg1, [Ex.Message]);
     //MessageBox(0, 'Błąd operacji zmiennopozycyjnych','Uwaga',
                  MB_OK);
     on Ex: EZeroDivide do
       raise EFloatingPointError.CreateFmt(Msg2, [Ex.Message]);
     //MessageBox(0, 'Próba dzielenia przez zero','Uwaga', MB_OK);
     else
       raise;
   end;
end;
//---------------------------------------------------------
procedure TForm1.Button2Click(Sender: TObject);
begin
   Application.Terminate();
end;
end.
```

Powyższy algorytm najlepiej jest testować, uruchamiając program wynikowy *Projekt_10.exe*.

Operacje na plikach

Zarówno w Windows, jak i Linux wszelkie operacje wejścia-wyjścia realizowane są za pomocą czytania z plików lub pisania do plików. Wszystkie urządzenia zewnętrzne, łącznie z końcówką użytkownika, traktowane są jako pliki wchodzące w skład szerokiego systemu plików. Niniejszy podrozdział poświęcony jest pewnym aspektom realizacji przez Delphi różnego rodzaju operacji na plikach.

Większość operacji plikowych, pochodzących ze standardowego Pascala, działa również w graficznym środowisku Delphi 6. Przypomnijmy, iż standardowo nazwa pliku może być zmienną typu *File* lub *TextFile*. Wywołaniem procedury

```
procedure AssignFile(var F; FileName: string);
```

dokonujemy przypisania nazwy pliku do zmiennej plikowej. Procedura

```
procedure Reset(var F [: File; RecSize: Word ] );
```

otwiera istniejący plik. Tryb otwarcia pliku zależy od przypisania zmiennej *FileMode* odpowiedniej wartości. Domyślnie przyjmowana jest wartość *2* pozwalająca na odczyt i zapis do pliku. Przypisanie zmiennej *FileMode* wartości *0* przed wywołaniem procedury `Reset()` spowoduje otwarcie pliku w trybie tylko do odczytu danych, natomiast wartości *1* — w trybie tylko do zapisu. Z kolei procedura

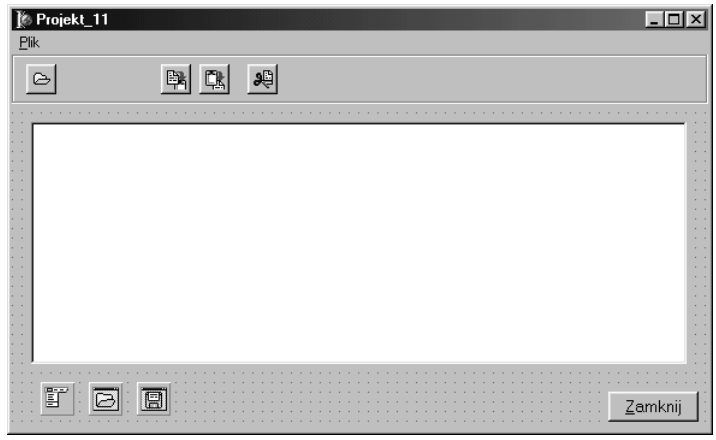
```
procedure Rewrite(var F: File [; Recsize: Word ] );
```

tworzy nowy plik z jednoczesnym jego otwarciem. Plik zamykamy, wywołując procedurę:

```
procedure CloseFile(var F);
```

Zaprojektujmy aplikację, której zadaniem będzie wczytanie pliku tekstowego z dysku oraz jego ponowne zapisanie po ewentualnej modyfikacji.

Umieśćmy na formularzu po jednym reprezentancie klas *TRichEdit* i *TCoolBar* z karty *Win32*. Dodatkowo uzupełnimy go trzema komponentami reprezentującymi klasę *TSpeedButton* z karty *Additional* oraz po jednym komponencie z klas *TMainMenu* i *TButton* z karty *Standard* oraz *TOpenDialog* i *TSaveDialog* z karty *Dialogs*. Sposób rozmieszczenia wymienionych komponentów na formularzu obrazuje rysunek 4.6.



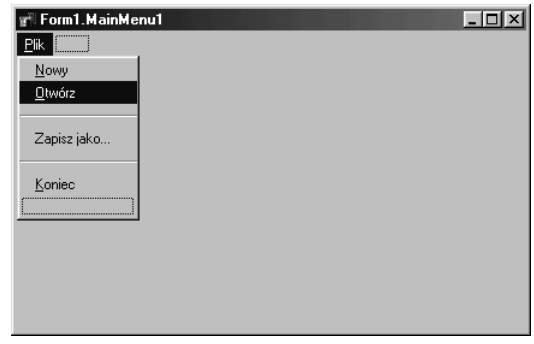
Rysunek 4.6. *Sposób rozmieszczenia komponentów na formularzu projektu Projekt_11.dpr*

Zaprojektujmy proste menu, dzięki któremu będziemy mogli utworzyć nowy plik, wczytać istniejący i ewentualnie powtórnie zapisać na dysku w wybranym katalogu. Aby dostać się do okna służącego do tworzenia menu głównego, należy zaznaczyć komponent *MainMenu1*, a następnie dwukrotnie kliknąć myszką pole *Items* karty zdarzeń inspektora obiektów (oczywiście, ten sam efekt otrzymamy, klikając dwukrotnie samą ikonę na formularzu). Zmieńmy cechę *Caption* (nagłówek) na *&Plik*, pojawi się wówczas nowe pole obok naszej opcji. W ten sposób możemy tworzyć nawet bardzo rozbudowane menu, ale o tym wszystkim jeszcze sobie powiemy w dalszej części książki. Teraz jednak wskażmy pole poniżej i cesze *Caption* przypiszmy *&Nowy*, następnie przejdźmy do karty *Events* inspektora obiektów i zdarzeniu *OnClick* przypiszmy *NewFileClick*. Klikając teraz dwa razy pole *&Nowy*, od razu znajdziemy się wewnątrz procedury obsługi zdarzenia `NewFileClick()`. Powróćmy do okna *Form1.MainMenu1* i przejdźmy niżej. Następną opcję zatytułujmy *&Otwórz*. W karcie zdarzeń inspektora obiektów jej cechę *Name* zmieńmy na *OpenFile*, natomiast w karcie zdarzeń zdarzeniu *OnClick* przypiszmy *FileOpenClick*. Dwukrotnie klikając, dostaniemy się do wnętrza procedury obsługi zdarzenia `FileOpenClick()`. Procedurę tę wypełnimy odpowiednim kodem:

```
procedure TForm1.FileOpenClick(Sender: TObject);
var
  InFile: TextFile;
  Data: string;
```

```
begin
  RichEdit1.Lines.Clear();
  if OpenDialog1.Execute then
    begin
      AssignFile(InFile, OpenDialog1.FileName);
      Reset(InFile);
      while not EOF(InFile) do
        begin
          ReadLn(InFile, Data);
          RichEdit1.Lines.Add(Data);
        end;
      CloseFile(InFile);
      Form1.Caption:='Edycja ['+ OpenDialog1.FileName +']';
    end;
end;
```

W bardzo podobny sposób zaprojektujmy pozostałe części składowe menu, tak jak pokazano na rysunku 4.7.

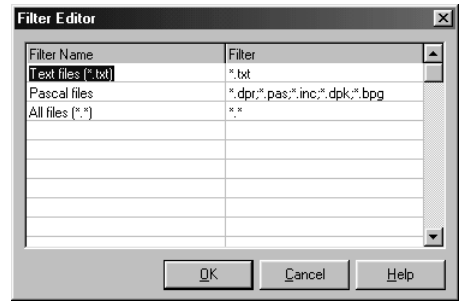


Rysunek 4.7. *Elementy składowe głównego menu projektu Projekt_11.dpr*

W omawianym programie menu *Plik, Otwórz* będzie zdublowane jednym z przycisków *TSpeedButton*. Najpierw na formularzu umieśćmy komponent *TCoolBar*, natomiast bezpośrednio na nim kolejno komponenty *TSpeedButton*. Ich cechy *Name* zmieńmy, posługując się inspektorem obiektów, odpowiednio na *FileOpen*, *CopyText*, *PasteText*, *CutText*. Korzystając z właściwości *Glyph*, rozwińmy opcję *TBitmap* i umieśćmy na każdym z przycisków *TSpeedButton* odpowiednią mapę bitową, tak jak przedstawiono na rysunku 4.6. Rysunek taki możemy wykonać samodzielnie, korzystając z Edytora Graficznego Delphi (menu *Tools, Image Editor*), którego obsługa nie różni się w istocie od zasad obsługi programu graficznego, jakim jest Paint.

Aby przycisk *FileOpen* obsługiwał znaną nam już procedurę obsługi zdarzenia `FileOpenClick()`, wystarczy w karcie zdarzeń inspektora obiektów jego zdarzeniu *OnClick* przypisać *FileOpenClick()*.

Na wydruku 4.5 zamieszczono kompletny kod aplikacji *Projekt_11.dpr*. W funkcji `FormCreate()` wykorzystaliśmy właściwość *InitialDir* obiektów *TOpenDialog* i *TSaveDialog*. Właściwość ta już w momencie uruchomienia aplikacji pozwala ustalić odpowiednią ścieżkę dostępu do aktualnego katalogu. Z kolei wykorzystując właściwość *Filter* (rysunek 4.8) tych obiektów, zapewnimy możliwość odczytania plików posiadających wymagane przez nas rozszerzenia.



Rysunek 4.8. *Właściwość Filter klas TOpenDialog i TSaveDialog*

Dymki podpowiedzi do poszczególnych przycisków uzyskamy, korzystając z właściwości *Hint* oraz *ShowHint*. Śledząc poniższy wydruk, zauważymy też, że aby komponenty *TOpenDialog* i *TSaveDialog*, niewidoczne przecież w trakcie uruchomienia programu, generowały zdarzenia, polegające na wyświetleniu odpowiednich okien dialogowych, należy w funkcjach odpowiednich zdarzeń skorzystać z metody `Execute()`. Plik zapisujemy na dysku, korzystając z procedury obsługi zdarzenia `SaveFileAsClick()`.

Procedury zdarzeniowe `CutTextClick()`, `PasteTextClick()`, `CopyTextClick()`, zaimplementowane w odpowiednich przyciskach, zgrupowanych w panelu *CoolBar1*, korzystają z metody `RichEdit1.CutToClipboard()`, `RichEdit1. PasteFromClipboard()`, `RichEdit1.CopyToClipboard()`, zapewniając możliwość usunięcia fragmentu tekstu, wstawienia fragmentu tekstu znajdującego się w schowku (ang. *clipboard*) oraz skopiowania fragmentu tekstu do schowka. Możliwe jest również zaznaczenie całości tekstu przy wykorzystaniu metody `RichEdit1.SelectAll()`. Aby powtórzyć ostatnio wykonaną (na tekście) operację, należy skorzystać z metody `RichEdit1.HandleAllocated()`, którą możemy już samodzielnie zastosować.

Wydruk 4.5. *Kod głównego modułu Unit_11.pas projektu Projekt_11.dpr*

```
unit Unit_11;
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ComCtrls,
  FileCtrl, ToolWin, Buttons, Menus;

type
  TForm1 = class(TForm)
    Button1: TButton;
    RichEdit1: TRichEdit;
    SaveDialog1: TSaveDialog;
    CoolBar1: TCoolBar;
    CopyText: TSpeedButton;
    PasteText: TSpeedButton;
    CutText: TSpeedButton;
    MainMenu1: TMainMenu;
    OpenDialog1: TOpenDialog;
    SaveFileAs: TMenuItem;
    OpenFile: TMenuItem;
    NewFile: TMenuItem;
```

```
    CloseApplication: TMenuItem;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure CopyTextClick(Sender: TObject);
    procedure PasteTextClick(Sender: TObject);
    procedure CutTextClick(Sender: TObject);
    procedure FileOpenClick(Sender: TObject);
    procedure NewFileClick(Sender: TObject);
    procedure SaveFileAsClick(Sender: TObject);
    procedure FormClose(Sender: TObject;
                        var Action: TCloseAction);
    procedure FormCloseQuery(Sender: TObject;
                             var CanClose: Boolean);

  private
    { Private declarations }
    sFile: String;
    procedure FormCaption(const sFile_s: String);
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

resourcestring
  s1 =  'Uwaga !';
  s2 = 'Zamknięcie aplikacji ';
//-----------------------------------------------------------
procedure TForm1.Button1Click(Sender: TObject);
begin
   case (MessageBox(0, PChar(s2), PChar(s1),
         MB_YESNO or MB_ICONQUESTION)) of
     ID_YES:
        Application.Terminate();
     ID_NO: Abort();
    end;
end;
//-----------------------------------------------------------
procedure TForm1.FormCreate(Sender: TObject);
begin
  OpenDialog1.InitialDir := ExtractFilePath(ParamStr(0));
  SaveDialog1.InitialDir := OpenDialog1.InitialDir;
end;
//-----------------------------------------------------------
procedure TForm1.CopyTextClick(Sender: TObject);
begin
  RichEdit1.CopyToClipboard();
end;
//-----------------------------------------------------------
procedure TForm1.PasteTextClick(Sender: TObject);
begin
  RichEdit1.PasteFromClipboard();
end;
```

```
//---------------------------------------------------------
procedure TForm1.CutTextClick(Sender: TObject);
begin
  RichEdit1.CutToClipboard();
end;
//---------------------------------------------------------
procedure TForm1.FormCaption(const sFile_s: String);
begin
  sFile := sFile_s;
  Caption := Format('%s - %s', [ExtractFileName(sFile_s),
                    Application.Title]);
end;
//---------------------------------------------------------
procedure TForm1.FileOpenClick(Sender: TObject);
var
  InFile: TextFile;
  Data: string;
begin
  RichEdit1.Lines.Clear();
  if OpenDialog1.Execute then
    begin
      AssignFile(InFile, OpenDialog1.FileName);
      Reset(InFile);
      while not EOF(InFile) do
        begin
          ReadLn(InFile, Data);
          RichEdit1.Lines.Add(Data);
        end;
      CloseFile(InFile);
      Form1.Caption:='Edycja ['+ OpenDialog1.FileName +']';
    end;
end;
//---------------------------------------------------------
procedure TForm1.NewFileClick(Sender: TObject);
begin
  FormCaption('Bez nazwy');
  RichEdit1.Lines.Clear();
  RichEdit1.Modified := FALSE;
end;
//---------------------------------------------------------
procedure TForm1.SaveFileAsClick(Sender: TObject);
var
  OutFile: TextFile;
begin
  if SaveDialog1.Execute then
    begin
      AssignFile(OutFile, SaveDialog1.FileName);
      Rewrite(OutFile);
      WriteLn(OutFile, RichEdit1.Text);
      CloseFile(OutFile);
      Form1.Caption:='Zapisany [ ' +SaveDialog1.FileName+ ']';
      RichEdit1.Modified := FALSE;
    end;
end;
//---------------------------------------------------------
procedure TForm1.FormClose(Sender: TObject;
                           var Action: TCloseAction);
```

```
begin
  Action:=caFree;
end;
//---------------------------------------------------------
procedure TForm1.FormCloseQuery(Sender: TObject;
                          var CanClose: Boolean);
begin
   case (MessageBox(0, PChar(s2), PChar(s1),
         MB_YESNO or MB_ICONQUESTION)) of
     mrYES:
       CanClose:=TRUE;
     mrNO:
       CanClose:=FALSE;
   end;
end;
//---------------------------------------------------------
end.
```

Strukturalna obsługa wyjątków

Być może temat niniejszego podrozdziału może się wydawać mało adekwatny w książce traktującej o podstawach programowania w Delphi, jednak przekonamy się, iż jest on naturalnym rozwinięciem poruszanych uprzednio zagadnień i wcale nie takim trudnym pojęciowo, jak ktoś mógłby sądzić.

Oprócz wyrażenia `try...except...end`, Delphi może posługiwać się również konstrukcją `try...finally...end`. Różnica pomiędzy nimi polega na tym, iż wyrażenie `try...finally` nie jest traktowane przez kompilator jako jawna konstrukcja obsługująca wyjątki pozwala natomiast na wykonanie części programu, występującego po słowie `finally`, nawet w przypadku wcześniejszego wykrycia jakiegoś wyjątku. W celu wyjaśnienia przedstawionego problemu posłużymy się przykładem procedury obsługi zdarzenia czytającego określony plik, który powinien znajdować się na dysku. Plik przeczytamy metodą „znak po znaku" za pomocą zmiennej `Character` typu `Char`.

```
procedure TForm1.Button2Click(Sender: TObject);
var
  InFile: TextFile;
  FName: string;
  Character: Char;
begin
  RichEdit1.Lines.Clear();
  FName:='Plik.txt';
  AssignFile(InFile, FName);
  try
    Reset(InFile);
    try
      while not EOF(InFile) do
        begin
          Read(InFile, Character);
          RichEdit1.Lines.Add(Character);
        end;
    finally
```

```
        CloseFile(InFile);
      end
    except
      on Ex: EInOutError do
        ShowMessage(' Błąd otwarcia pliku. Sprawdź, '+
                    'czy plik istnieje na dysku.');
    end;
    Form1.Caption:='Edycja ['+ FName +']';

  end;
```

Podczas testowania, a następnie analizowania powyższego algorytmu bez trudu zauważymy, iż w pierwszej kolejności zostaną wykonane instrukcje pomiędzy klauzulami `try` oraz `finally`. W następnej kolejności wykonywane będą instrukcje zawarte pomiędzy `finally` i `end` (polegające na zamknięciu pliku bez względu na to, czy został on otwarty prawidłowo, czy nie) niezależnie od rezultatu wykonania pierwszej grupy instrukcji czytających znaki z pliku. Najbardziej zewnętrzny blok `try...except...end` obrazuje znaną nam już ideę obsługi wyjątków. Pokazany sposób obsługi wyjątków nosi angielską nazwę Structural Exception Handling (w skrócie SEH). Dzięki zastosowaniu SEH dokonujemy rozdziału miejsca, w którym może wystąpić wyjątek (np. próba otwarcia nieistniejącego pliku) od miejsca, w którym będzie on obsługiwany. Zastosowany przez nas wyjątek *EInOutError* jest w rzeczywistości klasą wyjątków obsługujących operacje wejścia-wyjścia.

Tablice otwarte

W stosunku do standardowego języka Pascal Delphi znacznie rozszerza pojęcie tablicy. Jednym z takich rozszerzeń są tablice otwarte, które z reguły występują w roli parametrów procedur lub funkcji i mogą posiadać dowolne rozmiary. W przypadku, gdy procedura lub funkcja nie będzie modyfikować zawartości tablicy otwartej, deklarujemy ją za pomocą słowa kluczowego `const`. Słowa kluczowego `var` używamy w deklaracji funkcji lub procedury modyfikującej zawartość takiej tablicy. Ponieważ w trakcie działania program powinien w jakiś sposób kontrolować aktualny rozmiar takiej tablicy, musimy wskazać jej dolną i górną granicę. Do tego celu służą funkcje `Low()` i `High()`.

Jako przykład rozpatrzmy prostą funkcję `SumOfElement()`, obliczającą sumę elementów jednowymiarowej tablicy otwartej *Data*:

```
function SumOfElement(const Data: array of Double): Double;
var
  i: Integer;
begin
  Result:=0;
  for i:=Low(Data) to High(Data) do
    Result:=Result+Data[i];
end;
```

Widzimy, że deklaracja tablicy otwartej (z elementami np. typu *Double*) w charakterze argumentu funkcji przyjmuje bardzo prostą postać:

```
const Data: array of Double
```

Wywołanie w programie funkcji z argumentem w postaci tablicy otwartej nie powinno sprawić żadnego kłopotu nawet początkującemu programiście Delphi. Wyświetlenie wyniku (np. w komponencie edycyjnym *TEdit* z karty *Standard*), zwracanego przez funkcję `SumOfElement()`, może nastąpić w procedurze obsługi wybranego zdarzenia:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.Text:=FloatToStr(SumOfElement([-1,2,3,4,-5]));
end;
```

Widzimy, że w najprostszym wypadku wystarczy wywołać powyższą funkcję z argumentem w postaci kolejnych elementów tablicy, zapisanych w nawiasach kwadratowych.

Tablice dynamiczne



Różnica pomiędzy tablicami otwartymi i dynamicznymi jest dosyć subtelna. Polega na tym, iż deklaracja tablicy użyta jako parametr bez typu indeksu jest tablicą otwartą, natomiast tablica bez indeksu, deklarowana jako zmienna lokalna, globalna, pole klasy lub nowy typ danych jest tablicą dynamiczną.

Do funkcji (lub procedury) deklarującej swój argument jako tablicę otwartą można przekazywać tablice dynamiczne. Funkcja ma wtedy dostęp do elementów tablicy dynamicznej, jednak nie ma możliwości zmienić jej rozmiaru. Ponieważ tablice otwarte i dynamiczne są deklarowane identycznie, jedynym sposobem zadeklarowania parametru jako tablicy dynamicznej jest zadeklarowanie nowego typu identyfikatora dla typu tablicy dynamicznej. Przykład takiego działania został pokazany poniżej, gdzie również wykorzystano funkcję `SumOfElement()` do obliczania sumy wszystkich elementów tablicy dynamicznej *Data*. Ciąg liczb w prosty sposób czytany jest z pliku:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Data : array of Double;  // tablica dynamiczna przechowująca
                           // wartości typu Double
  F: TextFile;
  Value: Double;
  I: Integer;
begin
  AssignFile(F, 'C:\dane.dat');
  Reset(F);
  I:=0;
  while not EOF(F) do
  begin
    ReadLn(F, Value);
    SetLength(Data, Length(Data)+1);
```

```
    Data[I] := Value;
    RichEdit1.Lines.Add(FloatToStr(Data[I]));
    I := I+1;
  end;
  CloseFile(F);
  Edit1.Text:=FloatToStr(SumOfElement(Data));
end;
```

Typ OleVariant

Jako przykład zastosowania w programie typu *OleVariant* pokażemy, w jaki sposób bardzo szybko można stworzyć klienta OLE, wyświetlającego aktualną wersję zainstalowanego w systemie PowerPointa, i ewentualnie z poziomu kodu Delphi uruchomić go.



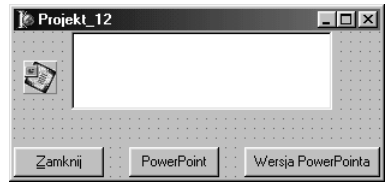
Technologia OLE (ang. *Object Linking and Embedding*) umożliwia osadzanie, łączenie i wzajemną wymianę różnych obiektów danych przy jednoczesnej pracy wielu aplikacji Windows (OLE 2).

Ole Automation jest częścią standardu OLE 2. Umożliwia zapisywanie w aplikacji sekwencji działań OLE w postaci ciągu poleceń, które dany program ma zinterpretować.

Component Object Model, w skrócie COM jest standardem, pozwalającym współdzielić obiekty pomiędzy wiele aplikacji. Określa też zasady komunikacji pomiędzy obiektami. Obiekty takie muszą być rozróżniane już na poziomie systemu operacyjnego.

Przejdźmy do karty *Servers*, zawierającej kilkadziesiąt elastycznych klas, służących do wizualizacji aktualnie dostępnych w systemie serwerów COM.

Zaprojektujmy naprawdę prostą aplikację, składającą się z jednego komponentu reprezentującego klasę *TPowerPointApplication*, jednego reprezentującego klasę *TRichEdit* oraz trzech *TButton*, tak jak przedstawiono to na rysunku 4.9.



Rysunek 4.9. *Aplikacja wykorzystująca przykładowy egzemplarz klasy TPowerPointApplication z karty Servers*

W najprostszym przypadku w celu ustanowienia połączenia z wybranym serwerem COM wykorzystamy metodę `Connect()`, w celu wizualizacji połączenia skorzystamy z metody `Visible`, natomiast aby rozłączyć się z wybranym uprzednio serwerem musimy skorzystać z metody `Disconnect()`.

Aby utworzyć klienta OLE, należy skorzystać z funkcji `CreateOleObject()` z modułu *ComObj*. Parametrem aktualnym tej funkcji jest nazwa odpowiedniej klasy, tak jak zostało to pokazane na wydruku 4.6.

Wydruk 4.6. *Kod głównego modułu Unit_12.pas projektu Projekt_12.dpr*

```
unit Unit_12;

interface

uses
  Windows, Messages, SysUtils,Variants,Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ComCtrls,
  ComObj, OleServer, MSPpt8;

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    RichEdit1: TRichEdit;
    Button3: TButton;
    PowerPointApplication1: TPowerPointApplication;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Button3Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}


//-----------------------------------------------------------
procedure TForm1.Button2Click(Sender: TObject);
var
  PowerPointApp: OleVariant;
begin
  try
    PowerPointApp := CreateOleObject('PowerPoint.Application');
    RichEdit1.Lines.Add('PowerPoint wersja '+PowerPointApp.Version);
  except
    Application.MessageBox('PowerPoint nie jest zainstalowany.',
                           'Błąd otwarcia aplikacji', MB_OK);

  end;
end;
//-----------------------------------------------------------
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  PowerPointApplication1.Disconnect();
  Application.Terminate();
end;
//---------------------------------------------------------
procedure TForm1.FormCreate(Sender: TObject);
begin
    PowerPointApplication1.Connect();
end;
//---------------------------------------------------------
procedure TForm1.Button3Click(Sender: TObject);
begin
  PowerPointApplication1.Visible:=1;
end;
//---------------------------------------------------------
end.
```

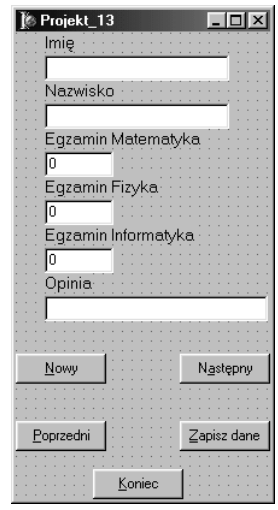
Widzimy, że sposób posługiwania się typami wariantowymi w kontekście tworzenia klienta OLE wybranego serwera COM jest bardzo prosty. Połączenie się z innymi dostępnymi serwerami pozostawimy Czytelnikom jako ćwiczenie do samodzielnego wykonania.

Rekordy w Delphi

Rekord w Delphi odgrywa bardzo podobne znaczenie jak w przypadku standardowego języka Pascal. Jeżeli zachodzi w programie konieczność przydzielenia wystarczającej ilości pamięci dla większej liczby zmiennych (elementów), stosowanie w programie rekordów daje dużo lepsze wyniki w porównaniu z posługiwaniem się klasami, pod jednym wszakże warunkiem, mianowicie zakładamy, że ilość operacji wykonywanych na elementach rekordu będzie stosunkowo niewielka. Jako przykład wykorzystania w środowisku graficznym Delphi prostego rekordu rozpatrzmy pewną modyfikację omawianego wcześniej przykładu rekordu, służącego do przechowywania informacji o studentach.

Zaprojektujmy formularz składający się z sześciu komponentów *TLabeledEdit* oraz pięciu *TButton*. W inspektorze obiektów we własności *EditLabel* cechy *Caption* egzemplarzy klasy *TlabeledEdit* zmieńmy odpowiednio na *Imię*, *Nazwisko*, *Egzamin Matematyka*, *Egzamin Fizyka*, *Egzamin Informatyka* oraz *Opinia*. Własności *Caption* komponentów *Button1*, *Button2*, *Button3*, *Button4* i *Button5* zmieńmy odpowiednio na *&Nowy*, *&Poprzedni*, *&Zapisz dane*, *N&astępny* i *&Koniec*. Rozmieszczenie i opis poszczególnych komponentów, wchodzących w skład formularza, zatytułowanego *Projekt_13*, pokazano na rysunku 4.10.

Bez trudu zauważamy, iż komponenty *LabeledEdit3*, *LabeledEdit4* i *LabeledEdit5* przechowywać będą liczby, dlatego aby w przyszłości uniknąć przykrych niespodzianek, w inspektorze obiektów ich cechom *Text* przypiszmy zerowe wartości. Cechy *Text* pozostałych egzemplarzy klasy *TLabeledEdit* wyczyśćmy.



Rysunek 4.10. *Formularz projektu Projekt_13.dpr*

Po to, aby nasz program był naprawdę funkcjonalny, przewidzimy możliwość zapisu danych na dysku. W tym celu w sekcji implementacji modułu zadeklarujemy rekord pod roboczą nazwą *TStudent* wraz ze zmienną *S* tego typu oraz zmienną plikową *F* typu *TStudentFile* (typ *file* przeznaczony jest dla plików binarnych):

```
implementation

{$R *.dfm}
type
  TStudent = record
    Imie: String[15];
    Nazwisko: String[20];
    EgzaminMat: Single;
    EgzaminFiz: Single;
    EgzaminInf: Single;
    JakiStudent: String[40];
  end;
TStudentFile = file of TStudent;
var
  F: TStudentFile;
  S: TStudent;
  CurRec: Integer;
```

Jeżeli programista nie poda innej deklaracji, plik w Delphi zawsze jest reprezentowany jako sekwencja rekordów o ustalonej długości. Dlatego aby uzyskać możliwość sekwencyjnego dostępu do rekordów pliku, musimy zadeklarować zmienną *CurRec* (*Current Record*).

W sekcji prywatnej klasy zadeklarujemy cztery procedury, których celem będzie wyświetlenie w komponentach edycyjnych aktualnych danych (procedura *ShowData*), wyczyszczenie zawartości komponentów edycyjnych (*ClearData*), zapisanie w pliku na dysku aktualnej zawartości elementów rekordu (*SaveData*) oraz wczytanie danych z dysku (*LoadData*). Ich zapis w sekcji implementacji modułu będzie bardzo prosty:

```
procedure TForm1.ShowData;
begin
   LabeledEdit1.Text:=S.Imie;
   LabeledEdit2.Text:=S.Nazwisko;
   LabeledEdit3.Text:=FloatToStr(S.EgzaminMat);
   LabeledEdit4.Text:=FloatToStr(S.EgzaminFiz);
   LabeledEdit5.Text:=FloatToStr(S.EgzaminInf);
   LabeledEdit6.Text:=S.JakiStudent;
end;
//---------------------------------------------------------
procedure TForm1.LoadData;
begin
  Read(F, S);
  ShowData;
end;
//---------------------------------------------------------
procedure TForm1.ClearData;
begin
   LabeledEdit1.Text:='';
   LabeledEdit2.Text:='';
   LabeledEdit3.Text:='0';
   LabeledEdit4.Text:='0';
   LabeledEdit5.Text:='0';
   LabeledEdit6.Text:='';
end;
//---------------------------------------------------------
procedure TForm1.SaveData;
begin
   S.Imie:=LabeledEdit1.Text;
   S.Nazwisko:=LabeledEdit2.Text;
   S.EgzaminMat:=StrToFloat(LabeledEdit3.Text);
   S.EgzaminFiz:=StrToFloat(LabeledEdit4.Text);
   S.EgzaminInf:=StrToFloat(LabeledEdit5.Text);
   S.JakiStudent:=LabeledEdit6.Text;
   Write(F, S);
end;
```

Zauważymy, iż dane czytamy za pomocą procedury `Read(F, S)`, natomiast zapisujemy za pomocą procedury `Write(F, S)`, gdzie `F` jest zmienną plikową, a `S` identyfikuje poszczególne pola rekordu.

W tego typu prostszych programach, jeżeli oczywiście nie mamy innych wymagań, postępujemy z reguły w ten sposób, aby już w momencie uruchomienia aplikacji ostatnio zapisane rekordy pliku były widoczne w poszczególnych komponentach edycyjnych. Efekt taki uzyskamy, odpowiednio wypełniając procedurę tworzącą formularz `FormCreate()`:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   ClearData;
   CurRec:=0;
   AssignFile(F, 'SdudentsData.dat');
   if FileExists('SdudentsData.dat') then
     begin
       Reset(F);
       while not EOF(F) do
```

```
        LoadData;
      end
        else
          begin
            ClearData;
            Rewrite(F);
          end;
end;
```

Najpierw czyścimy pola edycji, wywołując procedurę `ClearData`. Następnie zmiennej identyfikującej aktualny rekord pliku przypisujemy wartość zero (ustawiamy się na zerowym rekordzie pliku), gdyż początkiem każdego pliku jest rekord o numerze 0. Z kolei za pomocą procedury `AssignFile()` przypisujemy nazwę pliku o określonym typie do zmiennej plikowej. Dalej, sprawdzamy czy plik o podanej nazwie istnieje na dysku w aktualnym katalogu. Jeżeli plik takowy istnieje, otwieramy go procedurą `Reset()` z jednym parametrem w postaci zmiennej plikowej. Dane z pliku czytane są do momentu napotkania znaków końca pliku *EOF* (*End of File*). Może się zdarzyć, że plik o podanej nazwie nie będzie istnieć na dysku (np. pierwsze uruchomienie aplikacji). Wówczas należy go utworzyć, wywołując z jednym parametrem procedurę `Rewrite(F)`.

Po wypełnieniu odpowiednich rubryk zapisujemy je na dysku w sposób bardzo prosty, korzystając z przycisku Zapisz dane, wyzwalającego procedurę obsługi zdarzenia:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
   SaveData;
   ShowData;
end;
```

W celu rozpoczęcia wypełniania nowego rekordu uruchamiamy procedurę obsługi zdarzenia:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  repeat
    Inc(CurRec,1);
    Seek(F, CurRec);
  until EOF(F);
  ClearData;
  SaveData;
  Seek(F, CurRec);
end;
```

W instrukcji powtarzającej `repeat...until` za pomocą procedury `Inc()` zwiększamy aktualny numer rekordu w pliku o jeden. Używanie tej procedury w tzw. ciasnych pętlach daje dużo lepsze rezultaty w porównaniu z tradycyjnym przypisaniem:

```
CurRec:=CurRec+1;
```

Procedurą `Seek()` przesuwamy pozycję w pliku na miejsce wskazane przez numer aktualnego istniejącego rekordu *CurRec*. Czynność tę wykonujemy do momentu napotkania końca pliku. Następnie okna edycji są czyszczone i zapisywane, a numer pozycji w pliku przesuwany jest na aktualne miejsce *CurRec*.

W sposób bardzo podobny możemy np. poszukać następnego wypełnionego rekordu. Czynność tę wykonujemy za pomocą procedury obsługi zdarzenia:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  Inc(CurRec,1);
  Seek(F, CurRec);
  if not EOF(F) then
    begin
      Read(F, S);
      Seek(F, CurRec);
      ShowData;
    end
      else
        begin
          Inc(CurRec, -1); // to samo, co CurRec:=CurRec-1;
          Seek(F, CurRec);
          ShowMessage('Osiągnięto koniec pliku z danymi');
        end;
end;
```

Na wydruku 4.7 pokazano kompletny kod źródłowy głównego modułu *Unit_13.pas* aplikacji projektu *Kody\13\Projekt_13.dpr*.

Wydruk 4.7. *Kod źródłowy modułu Unit_13.pas*

```
unit Unit_13;

interface

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

type
  TForm1 = class(TForm)
    LabeledEdit1: TLabeledEdit;
    LabeledEdit2: TLabeledEdit;
    LabeledEdit3: TLabeledEdit;
    LabeledEdit4: TLabeledEdit;
    LabeledEdit5: TLabeledEdit;
    LabeledEdit6: TLabeledEdit;
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    procedure Button5Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
  private
    { Private declarations }
    procedure ShowData;
    procedure ClearData;
```

```
    procedure SaveData;
    procedure LoadData;
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}
type
  TStudent = record
    Imie: String[15];
    Nazwisko: String[20];
    EgzaminMat: Single;
    EgzaminFiz: Single;
    EgzaminInf: Single;
    JakiStudent: String[40];
  end;
TStudentFile = File of TStudent;
var
  F: TStudentFile;
  S: TStudent;
  CurRec: Integer;
//-----------------------------------------------------------
procedure TForm1.ShowData;
begin
   LabeledEdit1.Text:=S.Imie;
   LabeledEdit2.Text:=S.Nazwisko;
   LabeledEdit3.Text:=FloatToStr(S.EgzaminMat);
   LabeledEdit4.Text:=FloatToStr(S.EgzaminFiz);
   LabeledEdit5.Text:=FloatToStr(S.EgzaminInf);
   LabeledEdit6.Text:=S.JakiStudent;
end;
//-----------------------------------------------------------
procedure TForm1.LoadData;
begin
  Read(F, S);
  ShowData;
end;
//-----------------------------------------------------------
procedure TForm1.ClearData;
begin
   LabeledEdit1.Text:='';
   LabeledEdit2.Text:='';
   LabeledEdit3.Text:='0';
   LabeledEdit4.Text:='0';
   LabeledEdit5.Text:='0';
   LabeledEdit6.Text:='';
end;
//-----------------------------------------------------------
procedure TForm1.SaveData;
begin
   S.Imie:=LabeledEdit1.Text;
   S.Nazwisko:=LabeledEdit2.Text;
```

```
   S.EgzaminMat:=StrToFloat(LabeledEdit3.Text);
   S.EgzaminFiz:=StrToFloat(LabeledEdit4.Text);
   S.EgzaminInf:=StrToFloat(LabeledEdit5.Text);
   S.JakiStudent:=LabeledEdit6.Text;
   Write(F, S);
end;
//---------------------------------------------------------
procedure TForm1.Button5Click(Sender: TObject);
begin
   SaveData;
   CloseFile(F);
   Application.Terminate();
end;
//---------------------------------------------------------
procedure TForm1.Button3Click(Sender: TObject);
begin
   SaveData;
   ShowData;
end;
//---------------------------------------------------------
procedure TForm1.FormCreate(Sender: TObject);
begin
   ClearData;
   CurRec:=0;
   AssignFile(F, 'SdudentsData.dat');
   if FileExists('SdudentsData.dat') then
     begin
       Reset(F);
       while not EOF(F) do
         LoadData;
     end
       else
         begin
           ClearData;
           Rewrite(F);
         end;
end;
//---------------------------------------------------------
procedure TForm1.Button1Click(Sender: TObject);
begin
  repeat
    Inc(CurRec,1);
    Seek(F, CurRec);
  until EOF(F);
  ClearData;
  SaveData;
  Seek(F, CurRec);
end;
//---------------------------------------------------------
procedure TForm1.Button2Click(Sender: TObject);
begin
  if (CurRec-1) < 0 then
    begin
      CurRec:=0;
      Seek(F, CurRec);
      ShowMessage('Osiągnięto początek pliku z danymi');
    end
      else
```

```
        begin
          Inc(CurRec,-1);
          Seek(F, CurRec);
          Read(F, S);
          Seek(F, CurRec);
          ShowData;
        end;
end;
//---------------------------------------------------------
procedure TForm1.Button4Click(Sender: TObject);
begin
  Inc(CurRec,1);
  Seek(F, CurRec);
  if not EOF(F) then
    begin
      Read(F, S);
      Seek(F, CurRec);
      ShowData;
    end
      else
        begin
          Inc(CurRec, -1);   // to samo, co CurRec:=CurRec-1;
          Seek(F, CurRec);
          ShowMessage('Osiągnięto koniec pliku z danymi');
        end;
end;
//---------------------------------------------------------
end.
```

Testowanie przedstawionej aplikacji jest bardzo proste. Na pobranym z serwera FTP Wydawnictwa Helion pliku, w aktualnym katalogu znajduje się przykładowy plik *SdudentsData.dat* z informacjami o trzech wybranych studentach. Możemy samodzielnie go przeszukiwać i uzupełniać. Należy jednak pamiętać, aby każdy nowo wypełniony rekord od razu zapisać na dysku.



Warto pamiętać, iż procedury `Seek()` nie można stosować do wykonywania operacji na plikach tekstowych (*TextFile*). Jeżeli chcemy przesuwać pozycję w pliku tekstowym, należy skorzystać z funkcji bibliotecznej API `SetFilePointer()`.

Podsumowanie

Celem niniejszego rozdziału było zaprezentowanie Czytelnikom pewnych bardzo ważnych pojęć, z którymi nader często spotykamy się, tworząc aplikacje w nowoczesnym środowisku Delphi 6. Chociaż może się wydawać, iż omawianie na kursie programowania dla osób mniej zaawansowanych takich terminów, jak metody przeładowywane, wyjątki i klasy wyjątków, tablice otwarte i dynamiczne, serwery COM czy przeszukiwanie rekordów może być czynnością „na wyrost”, to jednak należy zdawać sobie sprawę, że obecnie terminy te posiadają już zupełnie fundamentalne znaczenie i ich znajomość (chociażby pobieżna) daje nam przepustkę do samodzielnego studiowania plików pomocy, które są nieocenionym i bogatym źródłem informacji o kompilatorze.