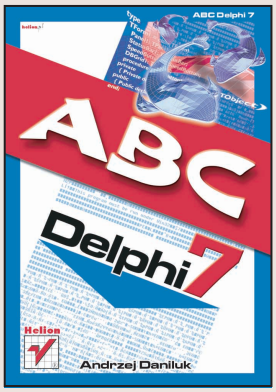
ABC Delphi 7

Autor: Andrzej Daniluk
ISBN: 83-7361-269-6
Format: B5, stron: około 168



Delphi 7 jest kolejną wersją najpopularniejszego zintegrowanego środowiska programowania typu RAD dla platformy Windows. Delphi 7, współpracując z Kyliksem firmy Borland - pierwszym środowiskiem programistycznym RAD dla Linuksa - sprawia, że możliwości wykorzystania Delphi przez osoby znające język Object Pascal znacznie wzrastają. Dzięki prostocie obsługi i zaletom wzorowanego na Pascalu języka Object Pascal, Delphi jest doskonałym narzędziem dla początkujących programistów, także dla tych, którzy nie mieli wcześniej wiele wspólnego z programowaniem obiektowym.

Książka omawia:

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

W porównaniu z poprzednim wydaniem tej książki rozbudowano rozdziały traktujące o podstawach programowania w języku Object Pascal. Znacznie poszerzono też rozdział poświęcony programowaniu obiektowemu.

Pomocą w zgłębianiu tajników Delphi 7 będzie 28 kompletnych, przykładowych projektów dołączonych do książki, ilustrujących najważniejsze poruszane zagadnienia.

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

Spis treści

Rozdział 3.

Projektowanie obiektowe OOD

Programowanie oparte na zasadach projektowania obiektowego *OOD* (ang. *Object Oriented Design*) stanowi zespół metod i sposobów, pozwalających elementom składowym aplikacji stać się odpowiednikiem obiektu lub klasy obiektów rzeczywiście istniejących w otaczającym nas świecie. Wszystkie aplikacje tworzone są po to, aby w sposób mniej lub bardziej prawdziwy odzwierciedlały lub modelowały otaczającą nas rzeczywistość. Każda aplikacja jest zbiorem współdziałających ze sobą różnych elementów. Przed rozpoczęciem projektowania aplikacji należy:

1. Zdefiniować nowy (lub zaimplementować istniejący) typ danych — klasę.
2. Zdefiniować obiekty oraz ich atrybuty.
3. Zaprojektować operacje, jakie każdy z obiektów ma wykonywać.
4. Ustalić zasady widoczności obiektów.
5. Ustalić zasady współdziałania obiektów.
6. Zaimplementować każdy z obiektów na potrzeby działania aplikacji.
7. Ustalić mechanizmy dziedziczenia obiektów.

Klasy

Klasa, definiując nowy typ (wzorzec) danych, może być źródłem definicji innych klas pochodnych. Klasa jest jednym z podstawowych pojęć języka Object Pascal. Za pomocą słowa kluczowego `class` definiujemy nowy typ danych, będący w istocie połączeniem danych i instrukcji, które wykonują na nich działania, umożliwiających tworzenie nowych (lub wykorzystanie istniejących) elementów, będących reprezentantami klasy. W największym przybliżeniu konstrukcja klasy umożliwia deklarowanie elementów *prywatnych* (ang. *private*), *publicznych* (ang. *public*), *chronionych* (ang. *protected*)

oraz publikowanych (ang. *published*). Domyślnie w standardowym języku Object Pascal wszystkie elementy klasy traktowane są jako prywatne, co oznacza, że dostęp do nich jest ściśle kontrolowany i żadna funkcja, nienależąca do klasy, nie może z nich korzystać. Jeżeli w definicji klasy pojawią się elementy publiczne, oznacza to, że mogą one uzyskiwać dostęp do innych części programu. Chronione elementy klasy dostępne są jedynie w danej klasie lub w klasach potomnych. Ogólną postać definicji klasy można przedstawić w sposób następujący:

```
type
NazwaKlasy = class
  private
    //prywatne dane i funkcje
  protected
    //chronione dane i funkcje
  public
    //publiczne dane i funkcje
  published
    //publikowane dane i funkcje
end;
...
var
  EgzemplarzKlasy: NazwaKlasy;
Begin
// program główny
End.
```

Jeszcze kilkanaście lat temu, przed pojawieniem się wizualnych środowisk programistycznych, słowo *obiekt* (ang. *object*) było jednoznacznie utożsamiane z klasą. Obecnie sytuacja nieco się skomplikowała, gdyż słowo *obiekt* ma o wiele szersze znaczenie. Z tego względu wygodniej jest posługiwać się szeroko stosowanym w anglojęzycznej literaturze sformułowaniem *egzemplarz klasy* (ang. *class instance*). Otóż po zdefiniowaniu klasy, tworzymy egzemplarz jej typu o nazwie takiej samej, jak nazwa klasy występująca po słowie `class`. Jednak klasy tworzymy po to, by stały się specyfikatorami typów danych. Jeżeli po instrukcji definicji klasy (kończącej się słowem kluczowym `end`), po słowie kluczowym `var` podamy pewną nazwę, utworzymy tym samym określony egzemplarz klasy, który od tej chwili traktowany jest jako nowy, pełnoprawny typ danych.

Jako przykład zaprojektujemy bardzo prostą w budowie klasę `TStudent`, za pomocą której będziemy mogli odczytać wybrane informacje o pewnej osobie.

Deklaracja klasy `TStudent`, składającej się z części publicznej i prywatnej, może wyglądać następująco:

```
type
  TStudent = class
    public
      procedure JakiStudent(js: PChar);
    private
      Nazwisko: PChar;
  end;
```

W sekcji publicznej (rozpoczynającej się od słowa kluczowego `public`) deklaracji klasy `TStudent` umieściliśmy prototyp jedynej procedury składowej klasy, natomiast w prywatnej (rozpoczynającej się od słowa kluczowego `private`) umieściliśmy deklarację

jednej zmiennej (dokładniej wskaźnika) składowej klasy. Mimo że istnieje możliwość deklarowania zmiennych publicznych w klasie, to jednak zalecane jest, aby dążyć do tego, by ich deklaracje były umieszczane w sekcji prywatnej, zaś dostęp do nich był możliwy poprzez funkcje z sekcji publicznej. Jak już się zapewne domyślamy, dostęp do wskaźnika `Nazwisko` z sekcji prywatnej będzie możliwy właśnie poprzez procedurę `JakiStudent()`, której jedynym parametrem jest właśnie wskaźnik. Tego typu technika programowania nosi nazwę *enkapsulacji danych*, co oznacza, że dostęp do prywatnych danych jest zawsze ściśle kontrolowany.

Enkapsulacja (ang. *encapsulation*) jest mechanizmem wiążącym instrukcje z danymi i zabezpieczającym je przed ingerencją z zewnątrz i błędnym użyciem.

W ciele procedury `JakiStudent()` dokonujemy porównania dwóch ciągów znaków, czyli łańcucha wskazywanego przez `Nazwisko` oraz drugiego, odpowiadającego już konkretnemu nazwisku danej osoby. Jeżeli oba łańcuchy są identyczne, procedura wyświetli odpowiedni komunikat. Ponieważ omawiana procedura jest częścią klasy `TStudent`, to przy jej zapisie w programie musimy poinformować kompilator, iż właśnie do niej należy. W Object Pascalu służy temu celowi operator w postaci kropki (`.`), zwany też niekiedy *operatorem rozróżniania zakresu*.

```
procedure TStudent.JakiStudent(js: PChar);
  begin
    nazwisko:=js;
    if (Nazwisko = 'Wackowski') then
      Writeln('Bardzo dobry student')
      else
        if (Nazwisko = 'Jankowski') then
          Writeln('Kiepski student')
          else
            Writeln('Brak takiego studenta');
  end;
```

Kompletny kod modułu projektu *Projekt_11.dpr*, korzystającego z omawianej klasy, przedstawiono na listingu 3.1.

Listing 3.1. *Kod projektu Projekt_11.dpr*

```
program Projekt_11;
{$APPTYPE CONSOLE}
uses
  SysUtils;

type
  TStudent = class
    public
      procedure JakiStudent(js: PChar);
    private
      Nazwisko: PChar;
  end;
//---------------------------------------------------------
procedure TStudent.JakiStudent(js: PChar);
  begin
    nazwisko:=js;
    if (Nazwisko = 'Wackowski') then
```

```
      Writeln('Bardzo dobry student')
      else
        if (Nazwisko = 'Jankowski') then
          Writeln('Kiepski student')
          else
            Writeln('Brak takiego studenta');
  end;
//---------------------------------------------------------
var
  // egzemplarz Student klasy TStudent
  Student: TStudent;
  KtoryStudent: PChar;
begin
  KtoryStudent:='Jankowski';
  Student.JakiStudent(KtoryStudent);
  ReadLN;
end.
```

Metody

Każdy wykorzystywany w programie egzemplarz klasy (lub obiekt) wykonuje (lub my wykonujemy na nim) pewne czynności — operacje, zwane potocznie metodami. Metodami nazywamy funkcje lub procedury, będące elementami klasy i obsługujące obiekt przynależny do danej klasy. W przykładzie pokazanym na listingu 3.1 procedura `JakiStudent()` jest właśnie taką metodą.

Obiekty

Obiekt jest dynamicznym egzemplarzem (reprezentantem) macierzystej klasy. Stanowi też pewien element rzeczywistości, którą charakteryzuje określony stan, tzn. obiekt jest aktywny lub nie. W odróżnieniu od zwykłych egzemplarzy klas, obiekty są zawsze alokowane dynamicznie w pewnym obszarze pamięci, zwanej *stertą*. Jedną z najważniejszych cech odróżniających obiekty od normalnych egzemplarzy klas jest to, że za tworzenie i zwalnianie obiektów w trakcie działania programu odpowiedzialny jest programista. Każdy obiekt tworzony jest za pomocą funkcji składowej klasy, zwanej *konstruktorem*. Definicja konstruktora rozpoczyna się od słowa kluczowego `constructor`. W Delphi domyślną nazwą konstruktora pozostaje słowo `Create`. W celu zwolnienia (zniszczenia) obiektu w momencie, gdy nie jest on już potrzebny, używamy jego destruktora. Definicja destruktora rozpoczyna się od słowa kluczowego `destuctor`. Powszechnie stosowaną nazwą destruktora jest słowo `Destroy`. Należy jednak pamiętać, iż jawne wywoływanie w programie destruktora klasy wiąże się z dość poważnymi konsekwencjami i wymaga pewnej wprawy, dlatego dużo bezpieczniejsze jest używanie prostej funkcji (metody) `Free` w celu zniszczenia danego obiektu.

Na listingu 3.2 zaprezentowano zmodyfikowaną wersję poprzednio omawianego programu. W obecnej wersji tworzony jest dynamicznie egzemplarz (obiekt) klasy `TStudent`.

Listing 3.2. *Idea tworzenia i zwalniania dynamicznego egzemplarza klasy*

```
program Projekt_11;
{$APPTYPE CONSOLE}
uses
  SysUtils;

type
  TStudent = class
    public
      // deklaracja bezparametrowego konstruktora
      constructor Create;
      procedure JakiStudent(js: PChar);
    private
      Nazwisko: PChar;
  end;
//---------------------------------------------------------
procedure TStudent.JakiStudent(js: PChar);
  begin
    nazwisko:=js;
    if (Nazwisko = 'Wackowski') then
      Writeln('Bardzo dobry student')
      else
        if (Nazwisko = 'Jankowski') then
          Writeln('Kiepski student')
          else
            Writeln('Brak takiego studenta');
  end;
//---------------------------------------------------------
constructor TStudent.Create;
begin
  // opcjonalnie ciało konstruktora
end;
//---------------------------------------------------------
var
  Student: TStudent;
  KtoryStudent: PChar;
begin
  // tworzenie dynamicznego egzemplarza (obiektu)
  // Student klasy TStudent
  // obiekt Student tworzony jest zawsze z poziomu odwołania
  // do nazwy klasy TStudent
  Student:=TStudent.Create;
  KtoryStudent:='Jankowski';
  Student.JakiStudent(KtoryStudent);
  ReadLN;
  // zwalnianie (niszczenie) obiektu Student
  // poprzez bezpośrednie wywołanie metody Free
  Student.Free;
end.
```

Często początkujący programiści Delphi popełniają dość poważny błąd, polegający na próbie utworzenia obiektu poprzez bezpośrednie odwołanie się do konstruktora `Create`:

```
Student.Create;
```

Zawsze należy pamiętać, iż obiekty tworzymy wyłącznie z poziomu odwołania się do nazwy macierzystej klasy, a dopiero potem wywołujemy jej konstruktor:

```
Student:=TStudent.Create;
```

Zwalnianie tak utworzonego obiektu odbywa się poprzez bezpośrednie wywołanie metody `Free`:

```
Student.Free;
```

Podczas niszczenia obiektu nie korzystamy już z odwołania do nazwy macierzystej klasy.

Widoczność obiektów

Jeżeli uznamy to za konieczne, możemy ustalić zakres widoczności obiektów w odniesieniu do fragmentu programu. Obiekt taki będzie korzystał ze zmiennych dostępnych jedynie dla metod klasy, w której je zdefiniowano.

Współdziałanie obiektów

Jeżeli obiekt lub grupę obiektów uczynimy widocznymi w całej aplikacji, należy ustalić zasady porozumiewania się obiektów, czyli relacje pomiędzy nimi. Dla każdego z obiektów ustanawiamy ściśle określony zbiór reguł i funkcji, dzięki którym korzystać z niego mogą inne obiekty.

Implementacja obiektu

Implementacja, czyli oprogramowanie obiektu, oznacza stworzenie kodu źródłowego, obsługującego metody z nim związane. Korzystając z zasad programowania obiektowo-zdarzeniowego, z poszczególnymi obiektami kojarzymy odpowiadające im zdarzenia.

Własności

Nowoczesny Object Pascal wprowadza bardziej ogólne pojęcie klasy poprzez implementację *własności*. Własność podobna jest do pola w klasie, ale może zachowywać się jak metoda. Własności pełnią rolę metod dostępu i modyfikacji (nazywanych metodami `Get` i `Set`) i mają szczególne znaczenie w zintegrowanym środowisku programisty IDE, chociaż ich użycie nie jest ograniczone tylko do niego (co wykażemy za chwilę). Właściwość posiada mechanizmy odczytu (`read`) i zapisu (`write`), służące do pobierania i ustawiania wartości właściwości. Mechanizmem odczytu może być nazwa pola lub metoda zwracająca wartość własności, natomiast mechanizmem zapisu — nazwa pola lub metoda ustawiająca wartość pola. Zaniedbując mechanizm zapisywania, tworzy się własność tylko do odczytu. Dopuszczalne jest również stworzenie własności tylko do zapisu, jednak celowość takiego tworu jest bardzo wątpliwa. Większość mechanizmów zapisujących i odczytujących stanowią nazwy pól lub metod, chociaż mogą to być również części pól agregujących, takich jak struktury lub tablice. Jeżeli dyrektywa `read` lub `write` odwołuje się do elementu tablicy, indeks tablicy musi być stały, a typ pola nie może być tablicą dynamicznie alokowaną w pamięci.

Własności mogą być definiowane przy użyciu dyrektyw `default` lub `stored`. Informacja ta nie ma żadnego znaczenia dla Delphi, jednak używa się jej podczas wykonywania opisów formularzy w pliku *.dfm* lub *.xfm*. Wartością dyrektywy `default` jest stała o typie takim samym, jak typ właściwości. Wartość domyślną mogą posiadać jedynie właściwości typu wyliczeniowego, całkowitego lub zbiorowego. Dyrektywa `default` posiada znaczenie tylko w przypadku właściwości publikowanych. Brak dyrektywy `default` równoznaczny jest nadaniu własności dyrektywy `nodefault`. Wartością dyrektywy `stored` jest stała boolowska, pole typu boolowskiego lub bezargumentowa metoda, zwracająca wynik typu `boolean`. Warto pamiętać, iż nadanie własności za pomocą dyrektywy `stored` wartości `true` (wartości domyślnej) nie powoduje, iż taka właśnie wartość domyślna zostanie zapisana do pliku *.dfm* (ewentualnie *.xfm*). Można jedynie pominąć wartość własności w pliku *.dfm* (*.xfm*), nadając dyrektywie `stored` wartość `false`.

W celu zilustrowania metod posługiwania się własnościami zbudujemy prostą klasę, której wykorzystanie w programie umożliwi w bardzo elegancki i przejrzysty sposób odczytywane i zapisywanie danych w postaci nazwisk wybranych osób. W tym celu skorzystamy z definicji własności. Ponieważ ogólnie przyjętą konwencją jest, aby w tego typu programach posługiwać się pewnymi standardowymi przedrostkami dla zmiennych oraz funkcji, w dalszej części opisu będziemy wykorzystywać nazewnictwo angielskie po to, by nie tworzyć mieszanki nazw polskich i angielskich (np. *SetNazwisko*).

W pierwszej kolejności określimy własność, za pomocą której będziemy w stanie odczytywać i przypisywać odpowiednie wartości (w tym wypadku łańcuchy znaków reprezentujące nazwiska i imiona osób). Każda własność służy do przechowywania wartości, zatem należy zadeklarować związaną z nią zmienną (tzw. pole w klasie). Ogólnie przyjętą konwencją jest to, że zmienne mają takie same nazwy, jak związane z nimi własności, ale poprzedzone są literą *F*. W naszym programie w sekcji prywatnej definicji klasy `TStudent` zadeklarujemy jedną taką zmienną typu `String`, reprezentującą tablicę indeksującą nazwiska studentów. Dodatkowo w tej samej sekcji zadeklarujemy

funkcję (metodę) `GetName()`, która w przyszłości będzie odczytywała za pomocą indeksu imię i nazwisko wybranej osoby, oraz procedurę `SetName()`, za pomocą której będzie można przypisać odpowiedni łańcuch znaków (imię i nazwisko) do odpowiedniego indeksu w tablicy:

```
private
      FName: array[1..4] of String;
      function GetName(i: Integer): String;
      procedure SetName(i: Integer; names: String);
```

Przechodzimy teraz do deklaracji samej własności. W tym celu należy użyć słowa kluczowego `property`. Dla potrzeb programu wystarczy, by własność służyła wyłącznie do przekazywania danych (imion i nazwisk osób) za pomocą indeksu. Własność zadeklarujemy w sekcji publicznej definicji klasy. Zdefiniowana własność `Name` będzie odczytywać aktualny stan tablicy `FName` za pomocą dyrektywy `read`, a następnie przekazywać (zapisywać) ją do procedury `SetName()`, korzystając z dyrektywy `write`:

```
public
      // deklaracja konstruktora
      constructor Create;
      // deklaracja własności tablicowej
      property Name[i: Integer]: String read GetName
                                         write SetName;
```

Jednoparametrowa funkcja składowa klasy (metoda) `GetName()` ma bardzo prostą budowę i służyć będzie do odpowiedniego indeksowania nazwisk:

```
function TStudent.GetName(i: Integer): String;
  begin
    Result:=FName[i];
  end;
```

Dwuparametrowa procedura `SetName()` również nie jest skomplikowana i przypisuje odpowiedniemu indeksowi tablicy ciąg znaków, określony przez zmienną `names`:

```
procedure TStudent.SetName(i: Integer; names: String);
  begin
    FName[i]:=names;
  end;
```

Kompletny kod projektu *Projekt_12.dpr*, wykorzystującego własność w klasie, pokazany jest na listingu 3.3.

Listing 3.3. *Praktyczny sposób posługiwania się własnościami w klasie*

```
program Projekt_12;
{$APPTYPE CONSOLE}
uses
  SysUtils;

type
  TStudent = class
    private
      FName: array[1..4] of String;
      function GetName(i: Integer): String;
      procedure SetName(i: Integer; names: String);
```

```
    public
      // deklaracja konstruktora
      constructor Create;
      // deklaracja własności tablicowej
      property Name[i: Integer]: String read GetName
                                          write SetName;
  end;
//-----------------------------------------------------------
function TStudent.GetName(i: Integer): String;
  begin
    Result:=FName[i];
  end;
//-----------------------------------------------------------
procedure TStudent.SetName(i: Integer; names: String);
  begin
    FName[i]:=names;
  end;
//-----------------------------------------------------------
constructor TStudent.Create;
begin
  // opcjonalnie ciało konstruktora
end;
//-----------------------------------------------------------
var
  Student: TStudent;
  i: Integer;
begin
  Student:=TStudent.Create;

  // wywołuje Student.SetName
  Student.Name[1]:='Wacek Jankowski';
  Student.Name[2]:='Janek Wackowski';
  Student.Name[3]:='Jola Łobuzińska';

  // wywołuje Student.GetName
  for i:=1 to 3 do
    Writeln(Student.Name[i]);

  ReadLN;
  Student.Free;
end.
```

Dziedziczenie

Dziedziczenie jest jednym z najważniejszych mechanizmów programowania zorientowanego obiektowo. Pozwala na przekazywanie właściwości *klasy bazowej* (ang. *base class*) *klasom pochodnym* (ang. *derived classes*). Oznacza to, że w prosty sposób można zbudować pewną hierarchię klas, uporządkowaną od najbardziej ogólnej do najbardziej szczegółowej. Na takiej właśnie hierarchii klas zbudowana jest w Delphi zarówno biblioteka komponentów wizualnych VCL, jak i biblioteka międzyplatformowa CLX.

Dziedziczenie (ang. *inheritance*) jest procesem przejmowania przez jeden obiekt właściwości innego, umożliwiającym tym samym klasyfikowanie obiektów.

Ogólną postać definicji *klasy pochodnej* zapisujemy z reguły w sposób następujący:

```
NazwaNowejKlasy = class(NazwaKlasyDziedziczonej)
{
 // deklaracje sekcji w nowej klasie
end;
```

Jako przykład zdefiniujemy klasę o nazwie `TOcena`, dziedziczącą po pewnej klasie bazowej `TStudent`, która zawierać będzie zmienną prywatną `FNazwisko`, przechowującą nazwisko wybranej osoby. Ze względu na to, iż klasa `TOcena` dziedziczy po klasie `TStudent`, procedura `JakaOcena` z klasy `TOcena` będzie miała bezpośredni dostęp do pola `FNazwisko` klasy bazowej. Osoby będziemy rozróżniać poprzez ich imię i nazwisko oraz ocenę z wybranego przedmiotu. W ten oto sposób wszystkie interesujące nas informacje uzyskamy, wywołując w głównym programie jedynie procedury składowe obiektu `Student` klasy `TOcena`, tak jak pokazano to na listingu 3.4. Chociaż program nasz składa się z dwóch różnych klas, zawierających szereg odrębnych elementów składowych, to jedynymi obiektami, do których jawnie odwołujemy się w programie głównym, są procedury składowe `JakiStudent` oraz `JakaOcena` obiektu `Student` klasy `TOcena`.

Listing 3.4. *Przykład dziedziczenia klas*

```
program Projekt_13;
{$APPTYPE CONSOLE}
uses
  SysUtils;

type
  TStudent = class
    public
      procedure JakiStudent;
    private
      FNazwisko: String;
  end;
//-----------------------------------------------------------
  // Klasa TOcena dziedziczy po klasie TStudent
  TOcena = class(TStudent)
    public
      // deklaracja konstruktora
      constructor Create(Nazwisko: String);
      procedure JakaOcena;
  end;
//-----------------------------------------------------------
procedure TStudent.JakiStudent;
  begin
    if (FNazwisko = 'Janek Wackowski') then
      Writeln('Bardzo dobry student')
      else
        if (FNazwisko = 'Wacek Jankowski') then
          Writeln('Kiepski student')
          else
            Writeln('Brak takiego studenta');
  end;
```

```
//-----------------------------------------------------------
procedure TOcena.JakaOcena;
  begin
    if (FNazwisko = 'Janek Wackowski') then
      Writeln(FNazwisko,': Egzamin Informatyka: 5')
      else
        if (FNazwisko = 'Wacek Jankowski') then
          Writeln(FNazwisko,': Egzamin Informatyka: 2.5')
          else
            Writeln('Brak oceny');
  end;
//-----------------------------------------------------------
constructor TOcena.Create(Nazwisko: String);
  begin
    FNazwisko:= Nazwisko;
    // opcjonalnie ciało konstruktora klasy TOcena
    // z wartością domyślną pola FNazwisko klasy TStudent
    // przekazywaną jako argument konstruktora
  end;
//-----------------------------------------------------------
var
  Student: TOcena;
begin
  // tworzenie dynamicznego egzemplarza (obiektu)
  // Student klasy TOcena
  Student:=TOcena.Create('Wacek Jankowski');
  Student.JakaOcena;
  Student.JakiStudent;
  ReadLN;
  // zwalnianie (niszczenie) obiektu Student
  // poprzez bezpośrednie wywołanie metody Free
  Student.Free;
end.
```

Analizując przedstawione powyżej zapisy, warto też zwrócić uwagę na sposób definicji konstruktora klasy `TOcena`. Otóż konstruktor został definiowany z parametrem formalnym w postaci zmiennej `Nazwisko`. W ciele konstruktora wartość tego parametru została przypisana zamiennej `FNazwisko`, zdefiniowanej w części prywatnej klasy `TStudent`. Taka definicja konstruktora zapewnia możliwość wywołania go w programie głównym, z parametrem aktualnym w postaci nazwiska interesującej nas osoby. Wszystkie metody wykorzystywane przez powyższy program są metodami statycznymi. Oznacza to, iż kompilator Delphi łączy ich wywołanie z implementacją metody.

Programując hierarchiczną strukturę klas, zawsze należy pamiętać, że klasa bazowa tworzona jest tylko wówczas, gdy konstruktor klasy potomnej wywoła konstruktor klasy bazowej. Delphi zawsze w pierwszej kolejności wywołuje konstruktor klasy potomnej. Każda klasa potomna musi posiadać swojego przodka (swoją klasę bazową). Przodek może być dowolną inną klasą bazową, występującą w łańcuchu dziedziczenia, aż po klasę `TObject`. Jeżeli klasa bazowa nie zostanie jawnie określona, Delphi jako domyślną klasę bazową przyjmie `TObject`.

Klasy abstrakcyjne

W przeciwieństwie do omawianych wcześniej metod statycznych, metody deklarowane ze słowem kluczowym `virtual` (tzw. *metody wirtualne*) w klasie bazowej muszą być zastąpione nową definicją w klasie pochodnej poprzez wykorzystanie dyrektywy `override`. Cechą charakterystyczną metod wirtualnych jest to, iż są łączone przez Delphi w trakcie wykonywania programu. Metoda wirtualna może zostać w klasie bazowej zadeklarowana ze słowem `abstract`, co oznacza, iż nie może być definiowana przez tę klasę. Metoda abstrakcyjna zadeklarowana w klasie bazowej musi być przedefiniowana (przesłonięta) przez klasę potomną. Klasę deklarującą jedną lub więcej metod abstrakcyjnych nazywamy *klasą abstrakcyjną*. Na listingu 3.5 zamieszczono przykład projektu wykorzystującego klasę abstrakcyjną.

Listing 3.5. *Zmodyfikowany Projekt_13.dpr, posługujący się klasą abstrakcyjną*

```
program Projekt_13;
{$APPTYPE CONSOLE}
uses
  SysUtils;

type
  TStudent = class
    public
      procedure JakiStudent; virtual; abstract;
    private
      FNazwisko: String;
  end;
//---------------------------------------------------------
  // Klasa TOcena dziedziczy po klasie abstrakcyjnej TStudent
  TOcena = class(TStudent)
    public
      constructor Create(Nazwisko: String);
      procedure JakiStudent; override;
      procedure JakaOcena;
  end;
//---------------------------------------------------------
// Abstrakcyjna metoda JakiStudent z klasy TStudent
// przedefiniowana jest i zaimplementowana w
// klasie potomnej TOcena
procedure TOcena.JakiStudent;
  begin
    if (FNazwisko = 'Janek Wackowski') then
      Writeln('Bardzo dobry student')
      else
        if (FNazwisko = 'Wacek Jankowski') then
          Writeln('Kiepski student')
          else
            Writeln('Brak takiego studenta');
  end;
//---------------------------------------------------------
procedure TOcena.JakaOcena;
  begin
    if (FNazwisko = 'Janek Wackowski') then
      Writeln(FNazwisko,': Egzamin Informatyka: 5')
```

```
      else
        if (FNazwisko = 'Wacek Jankowski') then
          Writeln(FNazwisko,': Egzamin Informatyka: 2.5')
          else
            Writeln('Brak oceny');
  end;
//---------------------------------------------------------
constructor TOcena.Create(Nazwisko: String);
  begin
    FNazwisko:= Nazwisko;
  end;
//---------------------------------------------------------
var
  Student: TOcena;
begin
  Student:=TOcena.Create('Wacek Jankowski');
  Student.JakaOcena;
  Student.JakiStudent;
  ReadLN;
  Student.Free;
end.
```

Podsumowanie

W rozdziale tym przypomniano podstawowe terminy, z którymi spotykamy się w trakcie projektowania aplikacji. Przedstawione tu podstawowe wiadomości na temat budowy klas oraz praktycznego wykorzystania ich elementów okażą się bardzo pomocne w trakcie studiowania dalszej części książki.