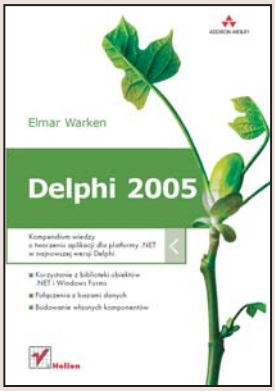
Delphi 2005

Autor: Elmar Warken
Tłumaczenie: Wojciech Moch
ISBN: 83-7361-993-3
Tytuł oryginału: Delphi 2005
Format: B5, stron: 810



Środowisko programistyczne Delphi jest od dawna jednym z najpopularniejszych narzędzi stosowanych przez twórców aplikacji. Każda z jego wersji wnosiła wiele nowości, jednak wersja oznaczona symbolem 2005 to prawdziwy przełom. Umożliwia ona bowiem projektowanie aplikacji przeznaczonych dla platformy .NET, co otwiera przez programistami tysiące nowych możliwości. Mogą wykorzystywać bibliotekę klas FCL, tworzyć aplikacje nie tylko w znanym z poprzednich wersji Delphi języku Object Pascal, ale również w zyskującym coraz większą popularność języku C#, a także stosować w swoich programach klasy i obiekty napisane w dowolnym języku zgodnym z platformą .NET. Delphi 2005 to prawdziwa rewolucja.

Książka „Delphi 2005" wyczerpująco omawia najnowszą wersję tego środowiska programistycznego. Przedstawia jego możliwości i ich praktyczne zastosowanie praktyczne. Szczegółowo opisuje zagadnienia podstawowe, takie jak praca z interfejsem użytkownika i stosowanie komponentów oraz tematy zaawansowane związane z tworzeniem aplikacji bazodanowych, korzystaniem z klas i obiektów specyficznych dla platformy .NET oraz pisaniem własnych komponentów.

- Korzystanie z elementów interfejsu użytkownika
- Zarządzanie plikami projektu
- Biblioteka klas .NET
- Przetwarzanie plików XML
- Zasady programowania obiektowego w Object Pascal
- Tworzenie aplikacji z wykorzystaniem biblioteki VCL.NET
- Połączenia z bazą danych za pomocą ADO.NET
- Zasady tworzenia własnych komponentów

Dzięki tej książce poznasz wszystkie możliwości najnowszej wersji Delphi

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

Spis treści

Rozdział 3.

Język Delphi w środowisku .NET

W tym rozdziale zajmować się będziemy językiem programowania Delphi — znanym też pod nazwą Object Pascal — i jego przekształcaniem w klasy i kompilaty środowiska .NET. Rozdział ten nie ma być dokumentacją tego języka (taka dokumentacja stanowi część systemu aktywnej pomocy Delphi; znaleźć ją można w gałęzi *Borland Help/Delphi 2005 (Common)/Reference/Delphi Language Guide*) i w związku z tym nie będę tu opisywał wszystkich jego szczegółów. W rozdziale tym będę się starał przedstawić jak najpełniejsze wprowadzenie do języka, skierowane do osób „przesiadających" się z innych narzędzi programistycznych lub z wcześniejszych wersji Delphi. Poza tym omawiał będę powiązania istniejące pomiędzy Delphi i CLR (ang. *Common Language Runtime* — wspólne środowisko uruchomieniowe dla wszystkich aplikacji .NET) oraz wyjaśniał wszystkie właściwości języka, jakie będą wykorzystywane w książce (rozdział ten opisywać będzie też ograniczenia języka w tym zakresie).

Na początku rozdziału nie będziemy zajmować się drobnymi elementami języka, takimi jak typy danych, zmienne i instrukcje, ale od razu przejdziemy do większych zagadnień, takich jak kompilaty (podrozdział 3.1) i model obiektów (podrozdziały 3.2 do 3.4). Od podrozdziału 3.5 przejdziemy do szczegółów języka, czyli zmiennych, stałych, typów, instrukcji, deklaracji metod i wyjątków.

Wprowadzenie

Język Object Pascal lub Delphi, jak ostatnio nazywa go firma Borland, jest bezpośrednim następcą języka Borland Pascal with Objects, który został wprowadzony w roku 1989 w pakiecie Turbo Pascal 5.5 (środowisko programistyczne dla systemu DOS), a krótko potem w pakiecie Turbo Pascal dla Windows, działającym również w systemie Windows. Od czasu powstania pierwszej wersji Delphi w roku 1995 do języka tego dodawano wiele poprawek i rozszerzeń, ale z każdą następną wersją języka liczba dodatków cały czas się zmniejszała, co oznacza, że język ustabilizował się na względnie wysokim poziomie.

Wraz z przeniesieniem Delphi do środowiska .NET, do języka Object Pascal znów wprowadzono kilka ważnych rozszerzeń. Wygląda jednak na to, że osoby przesiadające się z języka C++ na język C# muszą przyswajać sobie dużo więcej zmian w języku, niż osoby zmieniające Delphi 7 na Delphi dla .NET. Częściowo można to wytłumaczyć wspomnianym wyżej wysokim poziomem języka Object Pascal w Delphi 7, do którego języki przygotowywane przez Microsoft musiały dopiero dotrzeć, ale częściowo wynika też z tego, że firmie Borland dość dobrze udało się zamaskować przed programistami różnice pomiędzy językiem stosowanym w Delphi a wymaganiami środowiska .NET. Dzięki temu zachowany został wysoki zakres wstecznej zgodności języka z poprzednimi wersjami i pozwoliło to na znacznie łatwiejsze przenoszenie programów na inne platformy, w których działa Delphi, takie jak Win32 lub Linux.

Nowości w stosunku do Delphi 7

W tym rozdziale opisywać będę następujące unowocześnienia języka wprowadzone do niego od czasu Delphi 7:

- Działanie mechanizmu oczyszczania pamięci (ang. *Garbage Collector*); przenoszenie starych mechanizmów zwalniania pamięci z języka Object Pascal, takich jak destruktory i ręczne zwalnianie obiektów metodą `Free` (punkt 3.3.2).
- Zagnieżdżone deklaracje typów (punkt 3.2.8).
- Koncepcja typów wartości i typów wskaźników (punkt 3.6.6).
- Mechanizm Boxingu (punkt 3.6.6).
- Rekordy jako klasy typów wartości z metodami (3.6.5).
- Zmienne klas, właściwości klas, konstruktory klas (punkt 3.2.5).
- Drobne rozszerzenia języka: `strict private`, `strict protected` (punkt 3.2.2) i `sealed` (punkt 3.2.6).
- Operatory przeciążone (tylko zastosowanie operatorów przeciążonych — punkt 3.6.2).

Delphi a języki C++ i C#

Język Object Pascal już we wcześniejszych wersjach Delphi charakteryzował się rozwiązaniami, których nie można się było doszukać w języku C++, a które w tej lub innej formie obecne są dzisiaj w języku C#:

- Wirtualne konstruktory, rozszerzające koncepcję polimorfizmu również na mechanizmy konstrukcji obiektów (będzie o tym mowa w punkcie 3.3.5). W języku C# konstruktory wirtualne nie są co prawda dostępne, ale w językach środowiska .NET podobny efekt uzyskać można za pomocą mechanizmu Reflection, „wirtualnie" wywołując konstruktor poprzez metodę `InvokeMember` danego obiektu `Type`.

- Wskaźniki metod, które są znacznie wydajniejsze i praktyczniejsze od podobnych wskaźników z języka C++ (punkt 3.8.3). W języku C# wskaźniki te nazywane są delegacjami. Mówiąc dokładniej, typ wskaźnika metody obecny w Delphi od pierwszej wersji w języku C# odpowiada egzemplarzowi (instancji) typu `Delegat`.
- Wyjątki odpowiadające wyjątkom obsługiwanym w stylu języka C. Mają one tę przewagę nad wyjątkami języka C++, że pozwalają na stosowanie sekcji `finally` (dostępna jest ona również w języku C# — podrozdział 3.9).
- Informacje o typach wykraczające poza możliwości oferowane przez mechanizm RTTI z języka C++ (punkt 3.3.4). W środowisku .NET informacje te dostępne są w jeszcze szerszym zakresie niż Delphi 7 (mechanizm Reflection).
- Konstruktory tablic otwartych pozwalające na uzyskanie praktycznie dowolnej listy parametrów (punkt 3.8.1).
- Interfejsy, bardzo podobne do interfejsów znanych z języka Java, pozwalające na uzyskanie wielu operacji, które w języku C++ możliwe są tylko z wykorzystaniem dziedziczenia wielobazowego, a dodatkowo wolne są od zagrożeń, jakie stwarza ta właściwość języka C++ (podrozdział 3.4).

Oprócz przedstawionych na początku jasnych stron języka Object Pascal, wymienić można tu jeszcze inne zalety, które znane były już we wcześniejszych wersjach Delphi: zbiory (punkt 3.6.5), otwarte tablice (punkt 3.8.1) i chyba najczęściej wykorzystywana w tej książce przewaga języka Pascal nad językiem C++ — instrukcja `with` (podrozdział 3.7).

3.1. Przestrzenie nazw i kompilaty

Jak już mówiłem, forma Borland tworząc Delphi dla .NET chciała uzyskać jak największy stopień zgodności z poprzednimi wersjami Delphi i przenośności oprogramowania do systemów Windows i Linux. W wyniku tych dążeń do środowiska .NET przeniesione zostały koncepcje znane z poprzednich wersji Delphi, takie jak moduły i pakiety, które połączone zostały z koncepcjami funkcjonującymi w środowisku .NET, takimi jak przestrzenie nazw i kompilaty. Zanim zajmiemy się samym Delphi, w tym podrozdziale postaram się dokładniej opisać te podstawowe pojęcia środowiska .NET.

3.1.1. Podstawowe pojęcia środowiska .NET

W tym punkcie zajmiemy się najpierw pojęciami *kompilatów* (ang. *Assembly*) i *przestrzeni nazw* (ang. *Namespace*) oraz przyjrzymy środowisku zarządzanemu przez CLR, w którym wykonywane są wszystkie kompilaty. Opisywać będę również takie mechanizmy jak oczyszczanie pamięci (ang. *Garbage Collector*) i system wspólnych typów (ang. *Common Type System*), które w kolejnych rozdziałach grać będą znaczącą rolę.

Kompilaty

Kompilaty (ang. *Assemblies*) są podstawowym budulcem aplikacji środowiska .NET. W wielu przypadkach kompilat jest po prostu biblioteką dynamiczną *.dll* albo plikiem wykonywalnym *.exe*, ale koncepcja kompilatów jest o wiele szersza niż koncepcja plików. Jeden kompilat może rozciągać się na wiele plików, które traktowane są jako jedna całość. Pliki te, traktowane jako jeden kompilat, mają dokładnie te same numery wersji oraz wspólną „strefę prywatną" — w środowisku .NET istnieje szczególny atrybut widoczności o nazwie `assembly`, umożliwiający dostęp do identyfikatora tylko z wnętrza danego kompilatu.

Co prawda podziały istniejące w kompilatach wpływają przede wszystkim na sposób wykonywania programu, jednak część z nich dotyczy również jego fizycznej struktury (kompilaty ładowane są jako całkowicie niezależne od siebie składniki programu, w związku z czym w czasie aktualizacji oprogramowania mogą być aktualizowane niezależnie od siebie). Istnieje też podział wpływający na logiczną strukturę programu, który największe znaczenie ma w czasie projektowania aplikacji.

Przestrzenie nazw

Przestrzenie nazw (ang. *Namespaces*) pozwalają na dokonanie podziałów w cały czas rosnących i przez to coraz trudniejszych do ogarnięcia bibliotekach klas. Na przykład klasy przeznaczone do generowania grafiki umieszczone zostały w przestrzeni nazw `System.Drawing`, natomiast wszystkie kontrolki znane z aplikacji systemu Windows, takie jak kontrolki *ListBox* lub *Button*, zapisane są w przestrzeni nazw `System.Windows.Forms`. Dostępne są też inne kontrolki, przygotowane specjalnie do wykorzystania w aplikacjach WWW, które również nazywają się ListBox i Button, chociaż są to klasy całkowicie niezależne od klas standardowych kontrolek. Rozróżnienie pomiędzy tymi pierwszymi a tymi drugimi kontrolkami umożliwiają właśnie przestrzenie nazw: `System.Windows.Forms.ListBox` to kontrolka listy współdziałająca ze standardowymi aplikacjami działającymi bezpośrednio na komputerze użytkownika i na ekranie wyświetlana jest z wykorzystaniem interfejsu GDI+. Z kolei `System.Web.UI.WebControls.ListBox` to kontrolka listy współpracująca z aplikacjami WWW działającymi na serwerach WWW, a na komputer użytkownika przenoszona w postaci kodu HTML.

Różnorodność koncepcji przestrzeni nazw i kompilatów polega między innymi na tym, że jedna przestrzeń nazw może w sobie zawierać wiele kompilatów, a każdy z kompilatów może przechowywać w sobie wiele przestrzeni nazw. Mówiąc dokładniej, kompilat nie może w sobie tak naprawdę „zawierać" przestrzeni nazw, ponieważ może być ona rozbudowywana przez inne kompilaty. W związku z tym należałoby powiedzieć, że kompilat może przechowywać nazwy pochodzące z wielu przestrzeni nazw, które z kolei mogą być rozsiane wśród wielu kompilatów.

Hierarchia przestrzeni nazw

Hierarchia przestrzeni nazw budowana jest za pomocą kropek umieszczanych pomiędzy kolejnymi nazwami. Na przykład przestrzeń nazw `System.Drawing.Printing` podporządkowana jest hierarchicznie przestrzeni nazw `System.Drawing`, która z kolei podporządkowana jest przestrzeni nazw `System`. Można też powiedzieć, że nadrzędne przestrzenie nazw zawierają w sobie wszystkie przestrzenie podrzędne.

Takie hierarchiczne związki zawierania mają jednak naturę wyłącznie koncepcyjną i mają służyć przede wszystkim ludziom korzystającym ze środowiska .NET, umożliwiając lepszą orientację w jego zawartości. Istnieje na przykład konwencja mówiąca, że podrzędna przestrzeń nazw powinna być zależna od przestrzeni nadrzędnej, ale nie odwrotnie[1].

Dla środowiska CLR związki zawierania przestrzeni nazw nie mają żadnego znaczenia. Zawierane przestrzenie nazw mogą być zapisywane w osobnych kompilatach, niezależnych od nadrzędnej przestrzeni nazw, a takie kompilaty mogą być instalowane, ładowane i usuwane całkowicie niezależnie od kompilatów nadrzędnej przestrzeni nazw. Najczęściej jednak podrzędna przestrzeń nazw wykorzystuje elementy z przestrzeni nadrzędnej, a odpowiednie kompilaty mogą być załadowane tylko wtedy, gdy dostępne są też kompilaty nadrzędnej przestrzeni nazw. Ta zależność nie jest jednak definiowana przez nazwy przestrzeni nazw, ale przez bezpośrednie instrukcje nakazujące włączenie danej przestrzeni nazw (w języku C# instrukcja ta nazywa się `using`, a w Delphi — `uses`).

Kod zarządzany i CLR

Kod tworzony przez Delphi dla .NET jest kodem pośrednim zapisanym w języku MSIL (ang. *Intermediate Language*) przygotowanym przez Microsoft, który dopiero w czasie pracy programu w CLR przetłumaczony zostaje na kod maszynowy, dokładnie dopasowany do aktualnie używanego w systemie procesora. Z tego powodu aplikacje .NET mogą być uruchamiane tylko tym systemie, w którym zainstalowany jest pakiet środowiska .NET.

Kod, który wykonywany jest przez CLR, nazywany jest też *kodem zarządzanym* (ang. *Managed Code*). Na podobnej zasadzie, kod tworzony przez Delphi 7 określany jest mianem *kodu niezarządzanego* (ang. *Non-managed Code*), który tak naprawdę jest do pewnego stopnia zarządzany, choć nie przez system operacyjny, ale przez procesor. W przypadku kodu niezarządzanego system operacyjny ogranicza się do załadowania kodu w określone miejsce w pamięci i obsługi zgłoszonych przez procesor naruszeń zabezpieczeń występujących na przykład w sytuacji, gdy z powodu błędnie ustawionego wskaźnika program próbuje uzyskać dostęp do pamięci zarezerwowanej dla systemu operacyjnego.

Zupełnie inaczej wygląda natomiast zarządzanie kodem wykonywanym w ramach CLR: Środowisko CLR może odczytać wszystkie metadane zapisane w kompilatach i na ich podstawie poznać typ każdej zmiennej stosowanej w programie oraz przeanalizować kod programu w najdrobniejszych szczegółach. Nie ma tu możliwości uzyskania dostępu do obcych obszarów pamięci, ponieważ w zarządzanym kodzie nie istnieją wskaźniki, a CLR cały czas monitoruje wszystkie operacje na tablicach, nie pozwalając na dostępy wykraczające poza zakres tablicy. Innymi słowy, kod zarządzany tylko wtedy może spowodować naruszenie zabezpieczeń, kiedy nieprawidłowo działać będzie CLR.

Pozostałe cechy środowiska CLR, które można zaliczyć do kategorii „zarządzanie", to automatyczne zwalnianie pamięci (ang. *Garbage Collector*) i monitorowanie reguł bezpieczeństwa zabraniających na przykład zapisywania plików klasom pochodzącym z internetu i wykonywanym w przeglądarce internetowej.

[1] Więcej na ten temat znaleźć można w dokumentacji środowiska .NET, w dokumencie „Namespace Naming Guidelines", *ms-help://borland.bds3/cpgenref/html/cpconnamespacenamingguidelines.htm*.

System wspólnych typów

Ze względu na bardzo ścisłe reguły stosowane w czasie automatycznego monitorowania aplikacji przez CLR, nie ma już potrzeby przechowywania poszczególnych aplikacji w całkowicie odizolowanych od siebie przestrzeniach pamięci, tak jak dzieje się to w nowoczesnych systemach operacyjnych. Wszystkie aplikacje mogą być ładowane do tego samego obszaru pamięci i działać w ramach jednego egzemplarza środowiska CLR. Bardzo ważną zaletą takiego rozwiązania jest możliwość realizowania łatwej komunikacji pomiędzy aplikacjami. Aplikacje odizolowane od siebie nawzajem mogą wymieniać się danymi tylko poprzez specjalne mechanizmy, podczas gdy w środowisku CLR aplikacje mogą przekazywać sobie i współużytkować całe obiekty.

W tym wszystkim najbardziej rewolucyjne jest to, że nie ma tu znaczenia język, w jakim przygotowany został dany obiekt. Aplikacja napisana w języku C# może korzystać z obiektów przygotowanych w Delphi, a Delphi może z kolei korzystać z obiektów tworzonych w VB.NET. Co więcej, można napisać klasę w Delphi, będącą rozbudową klasy przygotowanej oryginalnie w języku C#, a klasa ta może być tworem zbudowanym na podstawie klasy środowiska .NET.

Kluczem do możliwości wspólnego wykorzystywania obiektów jest jeden z elementów środowiska CLR: wspólny system typów (ang. *Common Type System — CTS*). Definiuje on model obiektów każdej aplikacji środowiska .NET, a także typy podstawowe, z którymi pracować muszą wszystkie aplikacje i z których można budować bardziej złożone typy i klasy. Znany z Delphi typ `SmallInt` jest tylko inną nazwą dla funkcjonującego w środowisku .NET typu `Int16`. Każda klasa w Delphi wśród swoich przodków ma przynajmniej klasę `Object` pochodzącą ze środowiska .NET. Podobnie, wszystkie pozostałe języki programowania .NET w ten czy inny sposób wykorzystują typ `Int16` i klasę `Object`.

Jak widać, CTS jest wspólnym systemem typów obowiązujących wszystkie języki dostępne w środowisku .NET i w czasie działania tworzy wspólne implementacje wszystkich podstawowych typów, wykorzystywanych we wszystkich aplikacjach .NET. Jak wiemy, wszystkie aplikacje działające w środowisku .NET składają się z klas rozwijających klasy CTS, wobec tego każdą aplikację można traktować jak część CTS lub jego rozszerzenie.

Więcej szczegółów na temat związków pomiędzy typami CTS a typami Delphi podawać będę w podrozdziale 3.6.

3.1.2. Przestrzenie nazw w Delphi

Jako programiści tworzący w Delphi, związkami łączącymi przestrzenie nazw i kompilaty martwić się musimy dopiero wtedy, gdy sytuacja zmusza nas do wykorzystania „obcych" przestrzeni nazw i kompilatów. W programowaniu w Delphi przestrzenie nazw i kompilaty wynikają ze starszych pojęć funkcjonujących w języku Object Pascal — modułu (ang. *unit*), programu (ang. *program*) i pakietu (ang. *package*):

- Każdy projekt w Delphi kompilowany jest do jednego kompilatu i w związku z tym zawiera w sobie te przestrzenie nazw, które tworzone są w modułach tego projektu.
- Każdy moduł w Delphi automatycznie tworzy nową przestrzeń nazw, o nazwie zgodnej z nazwą modułu. Moduł *MyUnit.pas* powiązany jest z przestrzenią nazw `MyUnit`, a jeżeli moduł ten znajdować się będzie w projekcie *MyProject.dpr*, to z plikiem projektu powiązana zostanie kolejna przestrzeń nazw, o nazwie `MyProject`. Jedyną metodą pozwalającą na ręczną zmianę nazewnictwa przestrzeni nazw jest zmiana nazewnictwa modułów.

Tworzenie przestrzeni nazw

W czasie nadawania nazw nowym modułom i programom tworzymy jednocześnie nazwy przestrzeni nazw, dlatego twórcy języka Delphi w firmie Borland pozwolili na stosowanie kropek w nazwach programów i modułów. Dzięki temu możemy teraz nazwać plik projektu na przykład *MojaFirma.MojProgram.dpr*, a jednemu z modułów nadać nazwę *MojaFirma.MojProgram.UI.Dialogi.Login.pas*. Na podstawie nazwy pliku zawierającej kropki kompilator przygotuje nazwę przestrzeń nazw, usuwając z nazwy pliku ostatni człon wraz z kropką, tak jak pokazano to na listingu 3.1.

Listing 3.1. *Sposób tworzenia przestrzeni nazw na podstawie nazw programów i modułów*

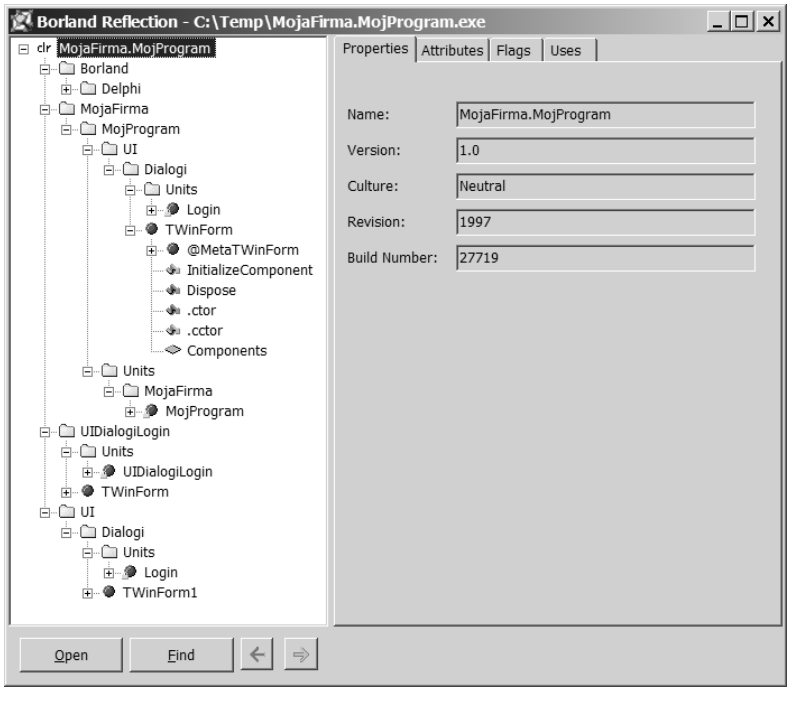
```
// Pierwszy wiersz w pliku .dpr:
program MojaFirma.MojProgram;
  // -> Przestrzeń nazw nazywa się MojaFirma

// Pierwszy wiersz w pliku .pas:
unit MojaFirma.MojProgram.UI.Dialogi.Login;
  // -> Przestrzeń nazw nazywa się MojaFirma.MojProgram.UI.Dialogi
```

Mimo kropek znajdujących się w nazwach, kompilator Delphi rozpoznaje je tylko jako całość. Jeżeli w programie użyjemy osobnego identyfikatora, takiego jak `MojaFirma` lub `Login`, to kompilator zgłosi błąd, informując nas o znalezieniu nieznanego identyfikatora.

Informacja dla osób, które pracowały już w Delphi 8: w Delphi 2005 zmieniona została stosowana w kompilatorze konwencja nazewnictwa przestrzeni nazw. W Delphi 8 nie był usuwany ostatni człon nazwy, tak więc w podawanym wyżej przykładzie nazwa modułu w całości wchodziłaby do nazwy przestrzeni nazw, włącznie z ostatnim członem `Login`. Jeżeli weźmiemy pod uwagę fakt, że przestrzenie nazw są kontenerami dla typów, a w module `Login` z całą pewnością zadeklarowany będzie typ formularza o nazwie `LoginForm`, to dojdziemy do wniosku, że mechanizm nazywania stosowany w Delphi 2005 jest lepiej zorganizowany. W Delphi 2005 pełny identyfikator typu tego formularza otrzymałby nazwę `MojaFirma.MojProgram.UI.Dialogi.LoginForm`, podczas gdy w Delphi 8 ten sam identyfikator otrzymałby nieco dłuższą i powtarzającą się na końcu nazwę `MojaFirma.MojProgram.UI.Dialogi.Login.LoginForm`.

Korzystając z narzędzia Reflection, można kontrolować przestrzenie nazw, jakie zapisywane są w kompilatach tworzonych przez Delphi. Każda z nazw tych przestrzeni nazw rozkładana jest zgodnie z umieszczonymi w niej kropkami na części określające hierarchiczne poziomy, a każdemu poziomowi przyporządkowany jest osobny symbol folderu. W kompilacie naprawdę znajdują się tylko te przestrzenie nazw, przy których symbolach folderów znajdują się kolejne węzły podrzędne, a przynajmniej jeden symbol o nazwie `Unit`.

Na rysunku 3.1 przedstawiono projekt, do którego w ramach porównania dołączono trzy podobne nazwy modułów (proszę przyjrzeć się zawartości okna menedżera projektu znajdującego się w prawej dolnej części rysunku. Przedstawiony projekt został skompilowany do pliku *.exe* i załadowany do programu Reflection.



Rysunek 3.1. *Moduły Delphi wyświetlane w menedżerze projektu i powiązane z nimi przestrzenie nazw wyświetlane w programie Reflection*

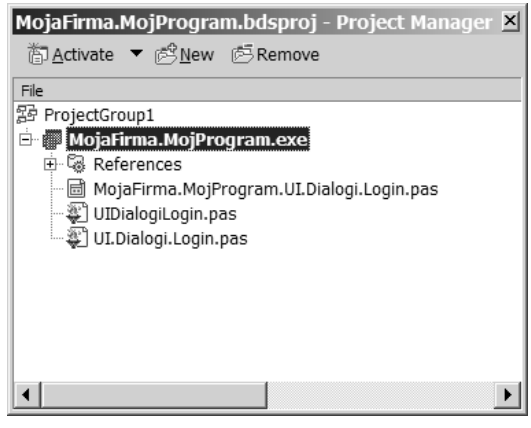
Wykorzystywanie przestrzeni nazw i kompilatów

Chcąc w swoim programie wykorzystywać symbole pochodzące z innych przestrzeni nazw, musimy wymienić je po słowie kluczowym `uses` we wszystkich modułach programu, w których używane są symbole danej przestrzeni nazw. W module `MonitorForm` z przykładowego programu *SystemLoadMonitor* (punkt 1.5.4) wykorzystywany jest cały szereg klas środowiska .NET, w związku z czym do modułu musiało zostać dołączonych kilka przestrzeni nazw, o czym można przekonać się, przeglądając listing 3.2.

Listing 3.2. *Przestrzenie nazw używane w module MonitorForm programu SystemLoadMonitor*

```
unit MonitorForm;

interface

uses
  System.Drawing, System.Collections,
  System.ComponentModel, System.Windows.Forms, System.Data,
  System.Diagnostics, Microsoft.Win32,
  System.Runtime.InteropServices, System.Globalization;
```

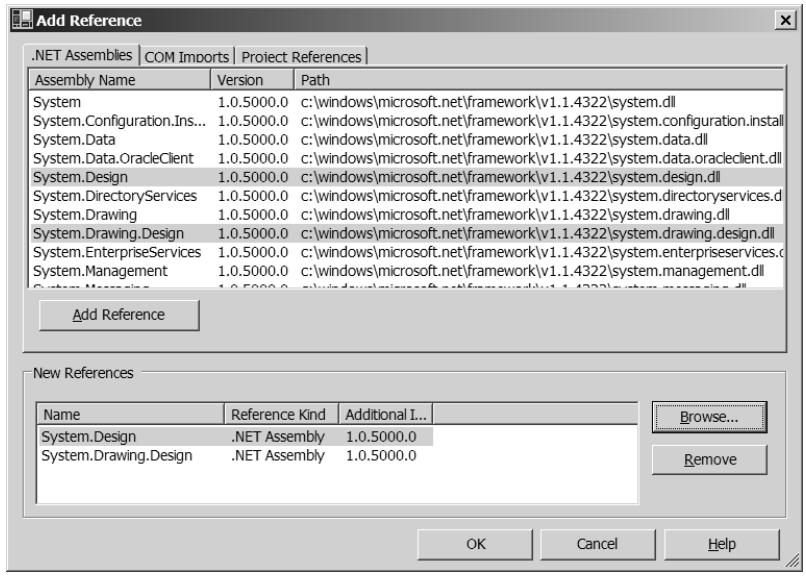
Oprócz tego, kompilator musi jeszcze wiedzieć, w których kompilatach będzie mógł znaleźć podane przestrzenie nazw (a dokładniej: symbole zapisane w podanych przestrzeniach nazw). W tym celu do projektu należy dodać odpowiednie referencje, umieszczając je w menedżerze projektu, w gałęzi `References` (proszę zobaczyć rysunek 3.2).



Rysunek 3.2. *Kompilaty, do których odwołuje się program, muszą zostać wymienione w menedżerze projektu*

Kompilaty przedstawione na rysunku dodawane są do projektu automatycznie, zaraz po utworzeniu nowego projektu. Przechowują one w sobie bardzo wiele ważnych klas środowiska .NET. Przy okazji dodawania do formularza nowego komponentu Delphi rozbudowuje listę kompilatów, uzupełniając ją w razie potrzeby o wszystkie te kompilaty, których wymaga nowo dodany komponent.

Jeżeli w programie korzystać chcemy z klas, które nie są umieszczone w standardowych kompilatach, to musimy własnoręcznie rozbudować listę kompilatów, dodając do niej nowe pozycje. W tym celu należy wywołać z menu kontekstowego menedżera projektu pozycję *Add Reference* i wybrać jeden lub kilka kompilatów z przedstawionej listy (okno z tą listą zobaczyć można na rysunku 3.3). Po naciśnięciu przycisku *Add Reference* wybrane kompilaty przenoszone są do dolnej listy *New References*, a po naciśnięciu przycisku *OK* kompilaty znajdujące się na dolnej liście dołączane są do projektu. Jeżeli potrzebny nam kompilat nie jest obecny na przedstawionej w oknie liście, to korzystając z przycisku *Browse* możemy wyszukać na dysku dowolny kompilat.



Rysunek 3.3. *Wybrane zostały dwa kompilaty środowiska .NET, które zajmują się funkcjonowaniem komponentów w czasie projektowania aplikacji*

Każda referencja wymieniona w menedżerze projektu wpisywana jest też do pliku projektu (zawartość tego pliku zobaczyć można, wybierając z menu pozycję *Project\View Source*), którego przykład można zobaczyć na listingu 3.3.

Listing 3.3. *Referencje kompilatów zapisane w pliku projektu*

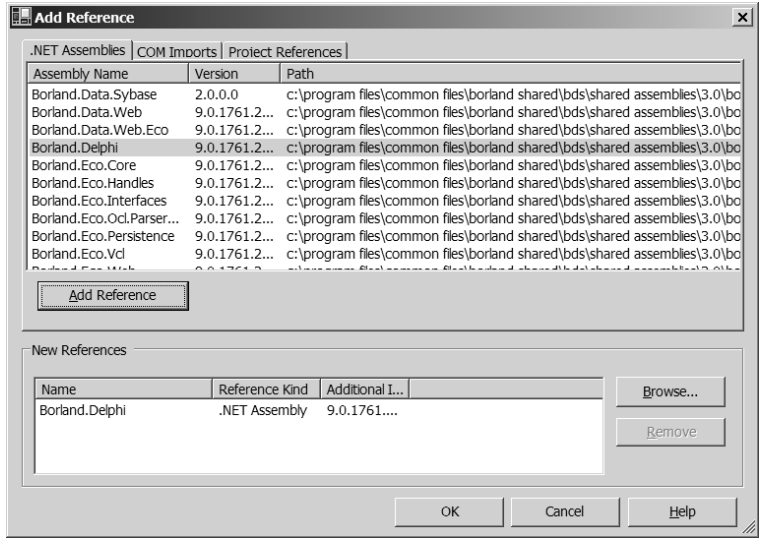
```
program SystemLoadMonitor;

{%DelphiDotNetAssemblyCompiler
'$(SystemRoot)\microsoft.net\framework\v1.1.4322\System.dll'}
{%DelphiDotNetAssemblyCompiler
'$(SystemRoot)\microsoft.net\framework\v1.1.4322\System.Data.dll'}
...
```

Konsolidacja statyczna i dynamiczna

Do programów przygotowanych w Delphi kompilaty środowiska .NET dołączane są dopiero w czasie ich działania, co oznacza, że dołączone są one poprzez konsolidację dynamiczną. Części samego programu, czyli jego moduły, włączane są do pliku *.exe* już w czasie kompilacji programu (konsolidacja statyczna). W przypadku modułów dostarczanych razem z Delphi nie jesteśmy ograniczeni do stosowania konsolidacji statycznej, ale wszystkie biblioteki czasu wykonania obecne w Delphi można też dołączać do programu dynamicznie.

W tym miejscu ponownie trzeba wykorzystać polecenie menedżera projektu *Add Reference*. Biblioteki dostarczane razem z Delphi zapisane są w szeregu plików *.dll*, do których referencje możemy umieścić w naszym programie. Moduły znajdujące się w tak dołączonych do programu bibliotekach nie muszą być już fizycznie integrowane z plikiem *.exe* aplikacji w procesie kompilacji.

Mały przykład: W każdej aplikacji tworzonej w Delphi nieodzowna jest jedna z bibliotek czasu wykonania dostarczanych razem z Delphi (moduł `System`), która znajduje się w pliku *borland.delphi.dll*. Jeżeli bibliotekę tę dodamy do naszej aplikacji tak jak pokazano na rysunku 3.4, to w efekcie uzyskamy znacznie mniejszy plik wykonywalny aplikacji. W minimalnej aplikacji Windows-Forms, składającej się z jednego pustego formularza, wielkość pliku *.exe* spada z 21 do 7 kB, a najprostsza aplikacja konsolowa kompilowana z wykorzystaniem konsolidacji dynamicznej zamyka się w pliku o wielkości 5 kB. W przypadku aplikacji konsolowych korzystających z modułu `SysUtils` konieczne jest dołączenie do programu referencji biblioteki *borland.vcl*, co ma zablokować włączenie tej biblioteki do pliku wykonywalnego.



Rysunek 3.4. *Dynamicznie konsolidowane mogą być też biblioteki czasu wykonania dostarczane razem z Delphi*

Tworząc aplikacje konsolidowane dynamicznie musimy upewnić się, że kompilaty, do których tworzymy referencje, rzeczywiście znajdować się będą na komputerze, na którym pracować ma nasza aplikacja.

Jeżeli chcielibyśmy się dowiedzieć, czy w pliku *.exe* naszej aplikacji nadal znajdują się niepotrzebnie obciążające go biblioteki, to wystarczy otworzyć ten plik w IDE Delphi, a wszystkie dane pliku wyświetlone zostaną w narzędziu Reflection. Jeżeli w wyświetlanym drzewie nie ma gałęzi o nazwie `Borland`, to znaczy, że w pliku nie zostały zapisane żadne moduły przygotowane przez firmę Borland.

Wymaganie powiązania każdej aplikacji Delphi (w sposób statyczny lub dynamiczny) z modułem `System` nie jest niczym wyjątkowym, ponieważ takie samo wymaganie istnieje we wszystkich innych językach środowiska .NET z wyjątkiem języka C#. Dołączenie biblioteki dopasowującej język do podstawy tworzonej przez CLR wymagane jest także w językach VB.NET, i C++ dla .NET. Jedynie język C# nie potrzebuje stosowania takiego dopasowania, ponieważ został on od podstaw zaprojektowany do współpracy ze środowiskiem .NET.

Wszystkie trzy przykłady tworzenia aplikacji („aplikacja WinForms konsolidowana statycznie", „aplikacja WinForms konsolidowana dynamicznie" i „aplikacja konsolowa konsolidowana dynamicznie") znaleźć można na płycie CD dołączonej do książki w katalogu *Rozdzial3\LinkSorts*.

3.1.3. Kompilaty w Delphi

Jak już mówiłem w punkcie 3.1.2, kompilator Delphi przekształca projekt aplikacji w kompilat środowiska .NET. O typie tworzonego kompilatu (jak wiemy, istnieją dwa rodzaje plików kompilatów) decyduje typ projektu przygotowywanego w Delphi.

Kompilaty EXE

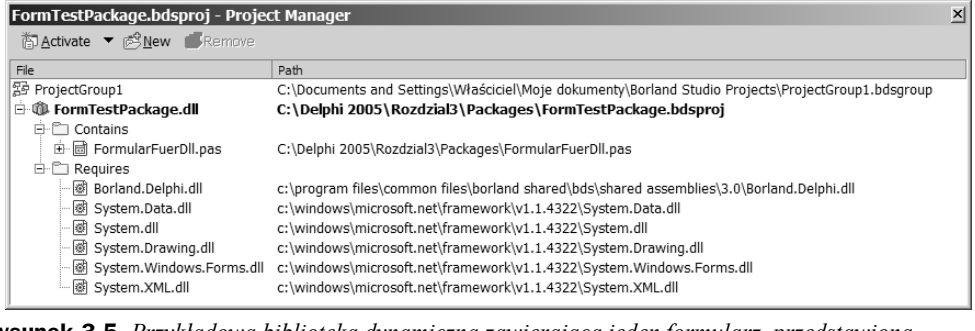
Jeżeli projekt Delphi zapisywany jest w pliku o rozszerzeniu *.dpr*, a tekst źródłowy projektu rozpoczyna się od słowa `program`, to taki projekt może być nazywany również aplikacją, a Delphi na jego podstawie przygotowywać będzie plik (kompilat) wykonywalny, którego rozszerzenie będzie identyczne z rozszerzeniem aplikacji stosowanych w środowisku Win32 — *.exe*. To rozszerzenie informuje nas też o tym, że plik ten może być uruchamiany dokładnie tak samo jak każdy inny plik o rozszerzeniu *.exe*, na przykład dwukrotnym kliknięciem w Eksploratorze Windows albo wywołaniem z wiersza poleceń. Takie działanie jest możliwe dlatego, że każdy z takich plików zawiera w sobie kod maszynowy, który może być uruchomiony w systemach Win32. Jedynym zadaniem tego kodu jest sprawdzenie obecności w systemie środowiska .NET i CLR, i w przypadku ich znalezienia — wywołanie tego środowiska w celu uruchomienia właściwej aplikacji.

W IDE Delphi dostępne są wstępnie przygotowane szablony projektów przeznaczone do budowania różnych aplikacji, takich jak *aplikacje konsolowe* (ang. *Console Application*), *aplikacje Windows-Forms* (ang. *Windows Forms Application*) lub *aplikacje VCL* (ang. *VCL Forms Application*; są to pozycje menu *File\New\Other*).

Kompilaty DLL

Kompilaty DLL tworzone są wtedy, gdy kompilowany jest *pakiet* (ang. *Package*) Delphi. Projekt pakietu zapisywany jest w pliku źródłowym o rozszerzeniu *.dpk*, w którym najważniejszym słowem jest słowo kluczowe `package`. Nowy projekt pakietu środowiska .NET tworzony jest w IDE Delphi wywołaniem pozycji menu *File\New\Other\Delphi for .NET Projects\Package*.

Pakiety również mogą mieć własne referencje na inne kompilaty, ale w menedżerze projektów nie są one nazywane referencjami (ang. *References*), ale wypisywane są w węźle `Required` (wymagane), co dobrze widać na rysunku 3.5.



Rysunek 3.5. *Przykładowa biblioteka dynamiczna zawierająca jeden formularz, przedstawiona w menedżerze projektu*

Przykład przedstawiony na rysunku (na płycie CD znaleźć można go w katalogu *Rozdzial3\Packages\FormTestpackage.dpr*) jest pakietem, który ma za zadanie udostępniać poprzez bibliotekę dynamiczną prosty formularz testowy. Pakiet ten przygotowany został w trzech bardzo prostych krokach:

- Utworzenie nowego pakietu wywołaniem z menu pozycji *File\New\Other\Delphi for .NET Projects\Package* (na tym etapie w węźle `Requires` znajduje się tylko pozycja `Borland.Delphi.dll`).
- Przygotowanie nowego formularza wywołaniem z menu pozycji *File\New\Other\Delphi for .NET Projects\New Files\Windows Form* (w tym momencie do menedżera projektu dodawane są pozostałe wpisy).
- Ułożenie na formularzu etykiety pozwalającej na rozpoznanie go w czasie testowych wywołań.

Jawne użycie modułów

Najważniejsza reguła tworzenia pakietów mówi, że wszystkie moduły muszą być jawnie dowiązywane do pakietu, co oznacza, że:

- Albo moduł musi znajdować się w bibliotece dynamicznej, która dowiązywana jest do pakietu poprzez referencję (w menedżerze projektu wypisana musi być w gałęzi `Requires` — jeżeli do tego węzła dodamy nowe pozycje korzystając z polecenia *Add Reference*, to operację tę przeprowadzać będziemy w oknie przedstawionym na rysunku 3.3).
- Albo moduł musi znajdować się w węźle o nazwie `Contains` (zawiera). W menu kontekstowym tego węzła znajdziemy pozycję *Add...*, która otwiera okno dialogowe umożliwiające wybranie potrzebnego nam modułu.

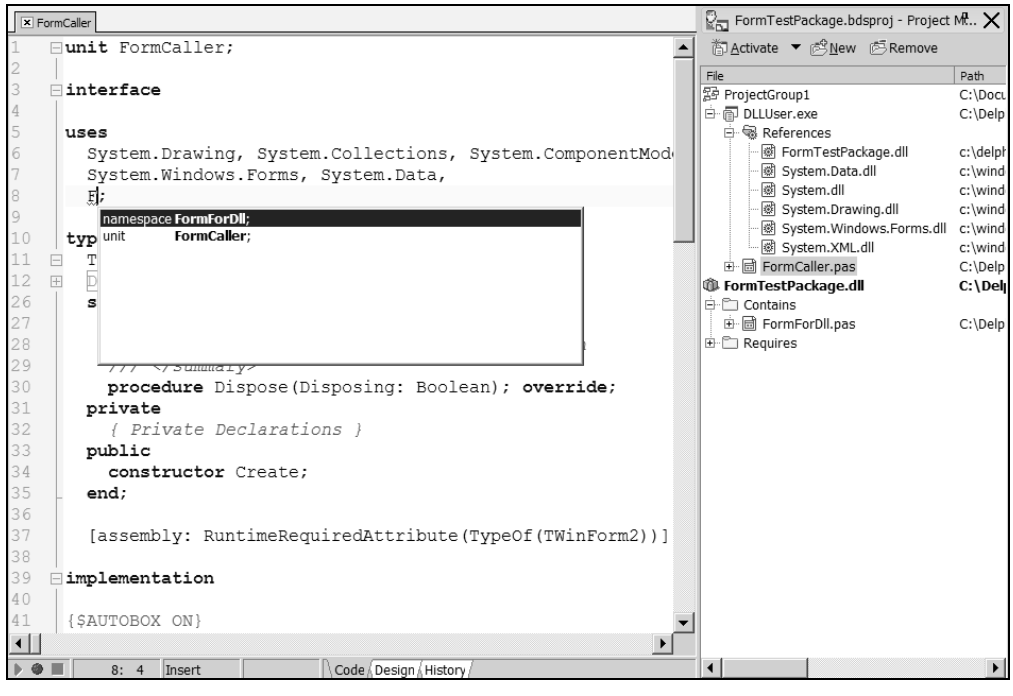
Wszystkie biblioteki DLL pochodzące z Delphi powinny być dynamicznie dołączane do tworzonych pakietów, ponieważ tylko taka konfiguracja pozwoli na stosowanie pakietu wewnątrz aplikacji przygotowanej w Delphi, która również wymaga zastosowania tych samych bibliotek dynamicznych. Jeżeli w tworzonym pakiecie usuniemy z węzła `Requires` zapisaną w nim bibliotekę *Borland.Delphi.dll*, to moduł `Borland.Delphi.System` zostanie dołączony do pakietu statycznie. W takiej sytuacji użycie tego pakietu w aplikacji przygotowanej w Delphi wiązałoby się z dwukrotnym uruchomieniem biblioteki DLL — jednym z aplikacji, a drugim z używanego przez nią pakietu. Po wykryciu takiego przypadku kompilator przerwie kompilowanie aplikacji i wypisze następujący komunikat:

```
[Fatal Error] Package 'FormTestPackage' already contains unit
'Borland.Delphi.System'.
```

Używanie samodzielnie przygotowanych pakietów

W celu dowiązania pakietu do własnego projektu należy skorzystać z procedury przedstawionej w punkcie 3.1.2 i dodać do projektu referencję pliku *.dll* pakietu. Jeżeli tego nie zrobimy, a jedynie umieścimy nazwy modułów zawartych w pakiecie w klauzuli `uses`, to kompilator dołączy te moduły bezpośrednio do pliku wykonywalnego naszego projektu.

Na rysunku 3.6 zobaczyć można, w jaki sposób przygotowana przed chwilą biblioteka dynamiczna o nazwie *FormTestPackage.dll* wykorzystywana jest w aplikacji tworzonej w Delphi. Przygotowany został nowy projekt aplikacji Windows-Forms, a biblioteka DLL dołączona została do niego w menedżerze projektu za pomocą pozycji menu *Required/Add reference/Browse*. Następnie moduł z biblioteki został wprowadzony do modułu formularza aplikacji poprzez dopisanie jego nazwy do klauzuli uses (w czasie uzupełniania listy uses można skorzystać z pomocy w programowaniu).



Rysunek 3.6. *Przykładowa biblioteka dynamiczna z formularzem przedstawiana w menedżerze projektu*

Kolejnym krokiem było umieszczenie na formularzu aplikacji przycisku opisanego *Wywołaj formularz biblioteki*, który opatrzymy został bardzo prostą procedurą obsługi zdarzenia `Click`, przedstawioną na listingu 3.4.

Listing 3.4. *Procedura wywołująca formularz pobrany z biblioteki dynamicznej*

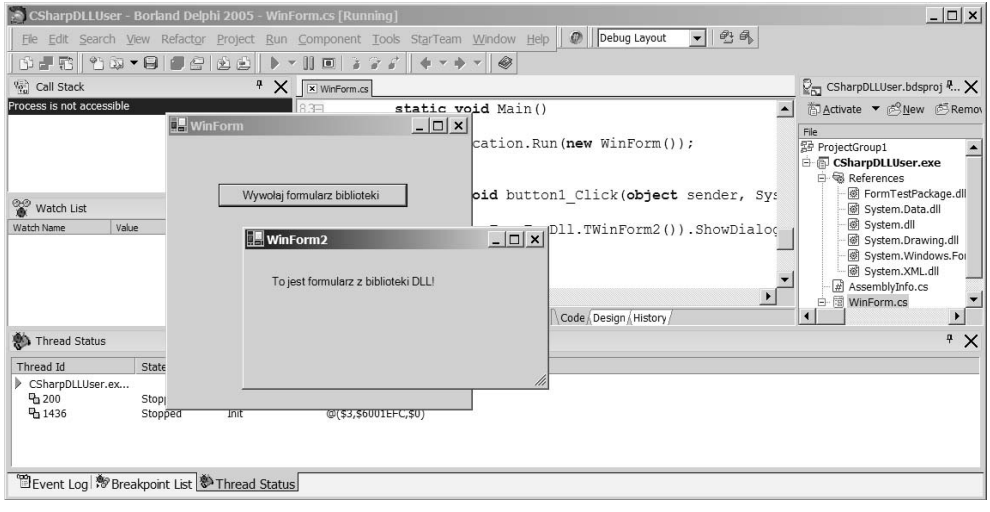
```
// Na płycie CD: Rozdział3\Packages\DLLUser.bdsproj (projekt Delphi)
procedure TWinForm2.Button1_Click(sender: System.Object;
  e: System.EventArgs);
begin
  FormForDll.TWinForm2.Create.ShowDialog;
end;
```

Nic więcej nie trzeba — program i biblioteka DLL są od razu gotowe do uruchomienia. Należy tu zauważyć, że równie łatwo biblioteki napisane w Delphi można wykorzystywać w programach tworzonych w języku C#. Kroki, jakie należy w tym celu wykonać w pakiecie C#-Builder, są niemal identyczne z tymi znanymi z Delphi: utworzyć

nowy projekt, przygotowaną w Delphi bibliotekę dopisać do referencji projektu, dowiązać przestrzeń nazw (w C# zamiast słowa `uses` stosowane jest słowo `using`) i wywołać formularz. Ten ostatni krok wymaga użycia nieco zmienionej składni, ponieważ z punktu widzenia języka C# konstruktory nie tylko nie nazywają się `Create`, ale w ogóle nie mają nazwy i wywoływane są przez operator `new`. Kod takiego wywołania zapisanego w języku C# podaję na listingu 3.5, a wynik działania tego kodu zobaczyć można na rysunku 3.7.

Listing 3.5. *Procedura wywołująca formularz pobrany z biblioteki dynamicznej zapisana w języku C#*

```
// Na płycie CD: Rozdzial3\Packages\CSharpDLLUser.bdsproj (projekt C#-Builder)
using FormForDll;
...
private void button1_Click(object sender, System.EventArgs e)
{
  (new FormForDll.TWinForm2()).ShowDialog();
}
```



Rysunek 3.7. *Formularz skompilowany w Delphi wyświetlany jest przez aplikację napisaną w języku C#. Tutaj jeszcze w starym, oddzielnym pakiecie C#-Builder, który teraz zintegrowany jest z Delphi 2005*

W punkcie 6.1.4 dokładniej przyjrzymy się pakietowi przykładowych komponentów przygotowanych na potrzeby tej książki, który oprócz tego, że zawiera kilka ciekawych komponentów, jest przykładem całkowicie normalnego pakietu.

Biblioteki z punktami wejścia Win32

Specjalną alternatywą tworzenia bibliotek dynamicznych środowiska .NET jest projekt typu `library`, który w IDE Delphi znaleźć można w pozycji menu *File\New\Other\Library*. Po wybraniu tej pozycji menu otrzymamy nowy plik *.dpr*, który nie będzie rozpoczynał się słowem kluczowym `program`, ale słowem `library`. W czasie kompilowania takiego projektu Delphi również przygotuje kompilat dynamicznie ładowanej biblioteki, która właściwie niczym nie będzie różniła się od skompilowanego pakietu.

Różnica między pakietami a bibliotekami polega na tym, że pliki *.dll* typu `library` mogą być też wywoływane przez aplikacje Win32 pracujące w kodzie niezarządzanym. W tym celu kompilator musi generować, „niebezpieczne" z punktu widzenia środowiska .NET, punkty wejścia używane w środowisku Win32, a zatem musimy jawnie zezwolić na stosowanie „niebezpiecznego" kodu. Procedury, które mogą być wywoływane przez aplikacje Win32, muszą być dodatkowo wypisane w klauzuli `exports`, tak jak pokazano na listingu 3.6.

Listing 3.6. *Szkielet kodu biblioteki zawierającej procedurę dostępną też w środowisku Win32*

```
{$UNSAFECODE ON}
library Library1;
...
procedure F1; begin end;

exports
  F1;   // procedura F1 może być wywoływana przez aplikacje Win32

end.
```

Bez bardzo ważnych powodów nie należy stosować dyrektywy kompilatora `$UnsafeCode`, ponieważ przygotowany z jej użyciem kompilat nie może na przykład zostać zweryfikowany jako kompilat typowy przez narzędzie wiersza poleceń *PEVerify* dostępne w pakiecie .NET SDK. Wynika z tego, że normalne biblioteki dynamiczne środowiska .NET powinny być generowane w Delphi jako zwyczajne pakiety.

Manifest kompilatów

Każdy kompilat środowiska .NET zawiera w sobie tak zwany *manifest*, w którym zapisane są między innymi następujące dane:

- kompilaty, jakich oczekuje dany kompilat (są to kompilaty, które w menedżerze projektu w Delphi wypisywane są w węźle `Requires` lub `References`). Każdy z tych kompilatów opatrywany jest numerem wersji, a czasami również specjalnym kluczem uniemożliwiającym zastosowanie przy kolejnych uruchomieniach programu zmodyfikowanych lub fałszywych kompilatów.
- atrybuty samego kompilatu, takie jak nazwa produktu, nazwa producenta, prawa własności i numer wersji.

Pierwsza część manifestu tworzona jest automatycznie przez Delphi na podstawie danych zapisanych w menedżerze projektu, natomiast na kształt drugiej części manifestu wpływać można poprzez ręczne modyfikacje tekstu źródłowego projektu, czyli edytowanie standardowo wstawianych do niego atrybutów. W tym celu należy wyświetlić zawartość kodu źródłowego projektu (pozycja menu *Project\View source*), rozwinąć ukryty region tekstu o nazwie *Program/Assembly information* i zamiast pustych ciągów znaków wstawić odpowiednie dla danego projektu dane. Przykład takiego uzupełniania danych manifestu przedstawiam na listingu 3.7.

Listing 3.7. *Część danych manifestu kompilatu modyfikować można ręcznie*

```
{$REGION 'Program/Assembly Informations'}
...
[assembly: AssemblyDescription('Komponenty do książki o Delphi 8')]
[assembly: AssemblyConfiguration('')]
[assembly: AssemblyCompany('EWLAB')]
[assembly: AssemblyProduct('')]
[assembly: AssemblyCopyright('Copyright 2004 Elmar Warken')]
[assembly: AssemblyTrademark('')]
[assembly: AssemblyCulture('')]

[assembly: AssemblyTitle('Tytuł mojego kompilatu')] // nadpisuje ustawienia
                          // pobrane z opcji Project\Options\Linker\Exe description
[assembly: AssemblyVersion('1.0.0.0')]
...
{$ENDREGION}
```

Jeżeli chodzi o numer wersji kompilatu, to trzeba powiedzieć, że w środowisku .NET składa się on z czterech części. Pierwsze dwie części określają numer główny i poboczny (w podanym wyżej listingu był to numer `1.0`), a za nimi znajdują się numer kompilacji (ang. *Build number*) i numer poprawki (ang. *Revision*). Jak się przekonamy za chwilę w podpunkcie Instalacja kompilatów, środowisko .NET może przechowywać wiele wersji tej samej biblioteki, zapisując je w globalnej składnicy kompilatów (ang. *Global Assembly Cache — GAC*). W pliku konfiguracyjnym aplikacji zapisać można też, która wersja kompilatu zapisana w GAC ma być wykorzystywana w powiązaniu z tą aplikacją.

Standardowo nowy projekt w Delphi otrzymuje numer `1.0.*`, gdzie znak gwiazdki (*) zastępowany jest automatycznie przez kompilator odpowiednimi numerami kompilacji i poprawek. Numer kompilacji powiększany jest o jeden każdego dnia, natomiast numer poprawki zmienia się na bieżąco w trakcie prac nad aplikacją. Wynika z tego, że dwie następujące po sobie kompilacje rozróżnić można już po ich numerze wersji. Jeżeli gwiazdka zostanie zastąpiona konkretnymi numerami wersji (tak jak w powyższym listingu), to opisane przed chwilą automatyczne mechanizmy generowania kolejnych numerów wersji zostają wyłączone.

Podpisane kompilaty

Kompilat może zostać zabezpieczony przed próbami dokonywania w nim późniejszych zmian i innych manipulacji. Zabezpieczenie takie polega na nadaniu mu mocnej nazwy (ang. *Strong name*), składającej się z pary specjalnie w tym celu przygotowanych kluczy. Jeden z tych kluczy jest kluczem tajnym, przechowywanym na komputerze wytwórcy kompilatu, który w czasie kompilowania pośrednio wpływa na kształt danych tekstowych zapisywanych do kompilatu. Drugi klucz — publiczny — jest o wiele krótszy i może być jawnie zapisywany wewnątrz kompilatu.

Za pomocą klucza publicznego CLR może w czasie ładowania kompilatu stwierdzić, czy znajduje się on w oryginalnym stanie. Manipulacja dokonana w kompilacie będzie mogła być zatajona przed CLR tylko wtedy, gdy podobnej manipulacji poddany zostanie klucz tajny. Próby wprowadzenia takiej modyfikacji klucza są jednak z góry skazane na niepowodzenie, ponieważ każda aplikacja, która dowiązuje do siebie kompilaty

za pomocą nazwy mocnej, zapisuje w sobie (a dokładniej wewnątrz swojego manifestu) znany klucz publiczny tych kompilatów (względnie jego skróconą formę, czyli tak zwany ekstrakt klucza), przez co wszystkie manipulacje na tych kompilatach wykrywane są przez CLR w czasie ich ładowania, co powoduje wygenerowanie wyjątku.

Dla przykładu, do przedstawionej wyżej biblioteki dynamicznej `FormTestPackage` dodamy mocną nazwę. Na początku musimy przygotować klucz pakietu i zapisać go w pliku *FormTestPackage.snk*. W tym celu skorzystać musimy z programu *sn.exe* będącego częścią pakietu .NET SDK, a dostępnego w katalogu *Program Files\Microsoft. NET\SDK\[NumerWersji]\Bin*:

```
sn -k FormTestPackage.snk
```

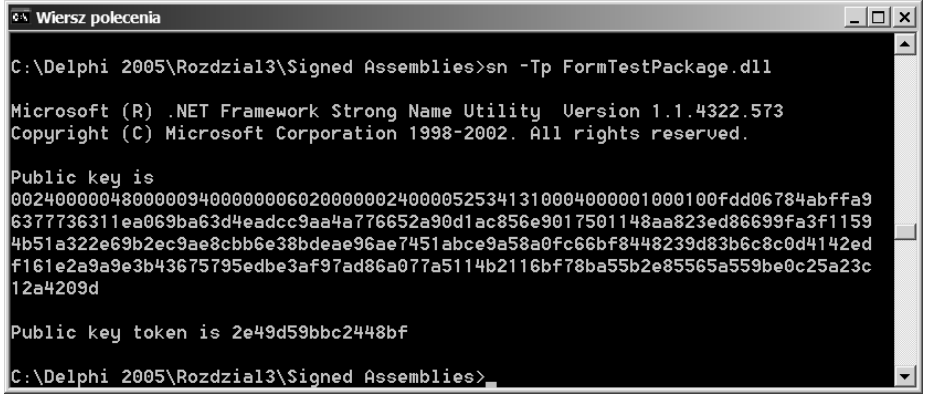
Następnie plik z kluczem musi zostać dodany do kompilatu. W pliku źródłowym projektu `FormTestPackage` zmienić należy tylko jeden z trzech atrybutów powiązanych z podpisami kompilatów, tak jak pokazano to na listingu 3.8.

Listing 3.8. *Wpisy w atrybutach opisujących podpisy kompilatów*

```
[assembly: AssemblyDelaySign(false)] // bez zmian
[assembly: AssemblyKeyFile('FormTestPackage.snk')]
[assembly: AssemblyKeyName('')] // bez zmian
```

Teraz biblioteka może zostać ponownie skompilowana, a powstały plik *.dll* będzie chroniony przed modyfikacjami przez wprowadzoną do biblioteki mocną nazwę. Ochronę tę można jeszcze skontrolować, ponownie wywołując program *sn.exe* (wyświetlane przez niego informacje przedstawiam na rysunku 3.8):

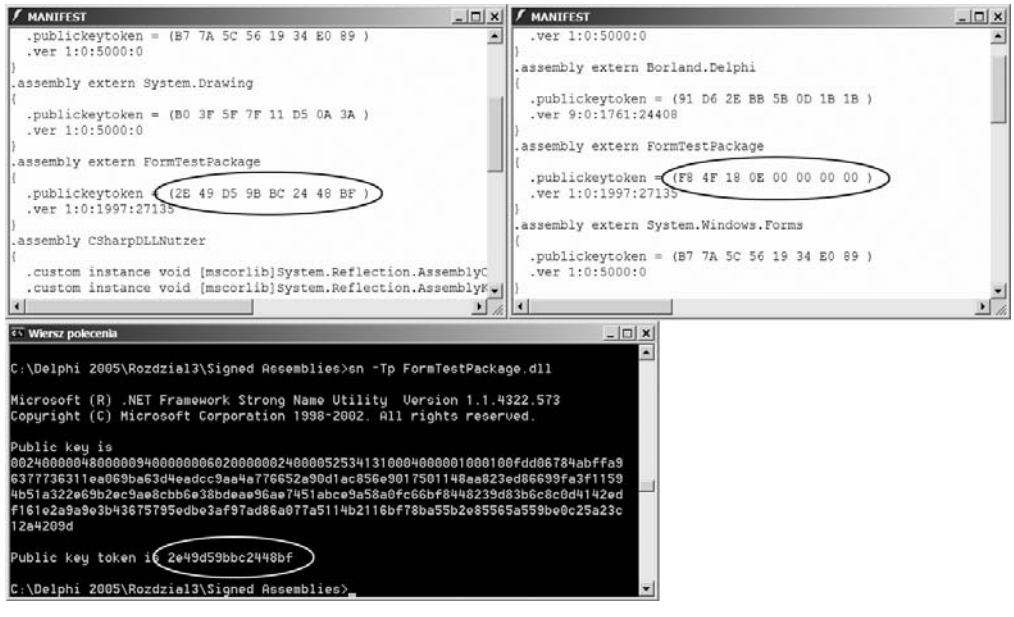
```
sn -Tp FormTestPackage.dll
```



Rysunek 3.8. *Program sn.exe potwierdza, że klucz publiczny został przez Delphi dopisany do biblioteki DLL. Wypisywany jest również skrót klucza*

Biblioteki chronione nazwami mocnymi są w Delphi traktowane dokładnie tak samo jak normalne pliki *.dll* i można je dodawać do projektów wywołując w menedżerze projektu polecenie *Add reference* oraz dopisując nazwy modułów i przestrzeni nazw z tej biblioteki do listy `uses`. Kompilator automatycznie zapisze skrót klucza takich bibliotek do manifestu tworzonego kompilatu.

W czasie przeprowadzanych przez mnie testów, z tak przygotowanej biblioteki *Form-➥TestPackage.dll* skorzystać można było wyłącznie w programie tworzonym w C#-Builder. Z niewiadomych powodów Delphi zapisywało do manifestu programu nieprawidłowy skrót klucza, w związku z czym CLR odmawiało załadowania biblioteki. Przygotowywany przez Delphi manifest można przejrzeć za pomocą programu *ILDasm.exe*, a wypisany przez niego skrót klucza musi być całkowicie zgodny ze skrótem podawanym przez program *sn.exe* (odpowiednie dane w tych programach zaznaczone zostały na rysunku 3.9).



Rysunek 3.9. *W lewym górnym rogu widoczny jest skrót klucza publicznego, który według kompilatora C# powinien znajdować się w bibliotece. W prawym górnym rogu widoczny jest skrót klucza publicznego, przy którym upiera się Delphi. Poniżej widać skrót klucza publicznego rzeczywiście wpisany do biblioteki*

Podpisaną wersję biblioteki znaleźć można na płycie CD dołączonej do książki, w katalogu *Rozdzial3\Signed Assemblies*, obok projektów języka C# i Delphi korzystających z tej biblioteki. Do „zainstalowania" tego przykładu konieczne jest ręczne wywołanie programu `sn -i` przed kompilowaniem projektów.

Instalacja kompilatów

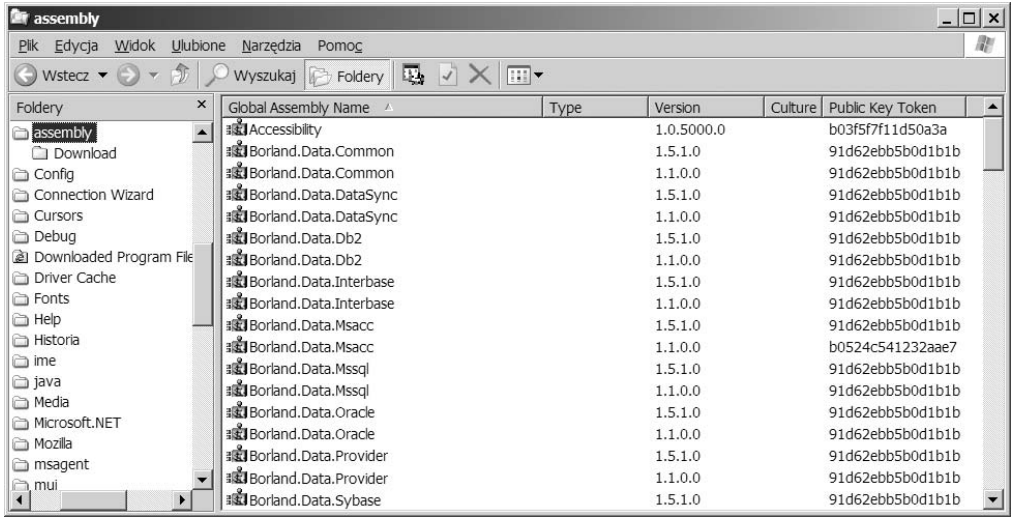
Do tej pory korzystaliśmy tylko z kompilatów dostarczanych przez firmy Microsoft i Borland albo z własnych kompilatów, które zapisane były w tym samym katalogu co plik wykonywalny naszej aplikacji. Oba te warianty przedstawiają dwa podstawowe sposoby instalowania bibliotek dynamicznych środowiska .NET.

W przypadku kompilatów wykorzystywanych przez wiele różnych aplikacji właściwym miejscem instalowania jest globalna składnica kompilatów GAC. Tutaj znajdziemy kompilaty dostarczane przez firmy Microsoft i Borland, ale także wszystkie te

kompilaty, które samodzielnie w niej zainstalujemy (w GAC instalowane mogą być tylko te kompilaty, które zabezpieczone zostały mocną nazwą). Instalacja przedstawionej wyżej, podpisanej biblioteki realizowana jest poniższym poleceniem:

```
gacutil /i FormTestPackage.dll
```

W każdej chwili możemy też sprawdzić zawartość składnicy, przeglądając w tym celu katalog *[KatalogSystemuWindows]\assembly*. Dzięki rozszerzeniom powłoki instalowanym razem ze środowiskiem .NET, Eksplorator Windows przedstawia ten katalog nie w swojej fizycznej, zagnieżdżonej strukturze, ale wyświetla wszystkie kompilaty w postaci przejrzystej listy, którą zobaczyć można na rysunku 3.10.



Rysunek 3.10. *Po zainstalowaniu Delphi 8 i C#-Builder wszystkie kompilaty przygotowane przez firmę Borland dają doskonały przykład tego, że w środowisku .NET można przechowywać obok siebie kilka wersji tego samego kompilatu. W Delphi 2005 przedstawione na rysunku kompilaty mają numery wersji zmienione na 2.0.0.0*

Automatyczne kopiowanie dołączanych kompilatów można oczywiście wyłączyć. Wystarczy w tym celu odnaleźć referencję w menedżerze projektu i nadać jego właściwości `Copy Local` wartość `False`.

Jeżeli chodzi o umiejscowienie kompilatu, to środowisko CLR jest bardzo elastyczne. Można na przykład zdefiniować inny standardowy katalog z dowiązywanymi bibliotekami i zapisać go w pliku konfiguracyjnym projektu aplikacji (plik o rozszerzeniu *.config* znajdujący się w katalogu aplikacji), wykorzystując przy tym atrybut `codebase`. Jeżeli kompilat zostanie zainstalowany w GAC, to najprawdopodobniej będzie trzeba jeszcze dokonywać rozróżnienia pomiędzy jego różnymi wersjami. Zagadnienie konfigurowania aplikacji wykracza niestety poza ramy tego rozdziału, a na dodatek nie ma właściwie nic wspólnego z Delphi. O złożoności dostępnych w tym zakresie możliwości przekonać się można przeglądając dokument z systemu aktywnej pomocy dostępny w gałęzi: *.NET Framework SDK\Configuring Applications*.

Instalowanie kompilatów w GAC nie jest jednak wymagane, ponieważ CLR poszukuje najpierw kompilatów wykorzystywanych przez aplikację w katalogu, w którym była ona uruchamiana. Jeżeli w projekcie Delphi dowiążemy pewien kompilat poleceniem *Add Reference*, a kompilat ten nie jest zapisany w składnicy GAC i nie znajduje się też w katalogu projektu, to Delphi automatycznie skopiuje go do tego katalogu. Proces kopiowania powtarzany będzie przy każdej kompilacji projektu, chyba że oryginalny kompilat nie zmienił się od czasu ostatniej kompilacji.

3.1.4. Moduły Delphi

W punkcie 3.1.2 wskazywałem na podobieństwo przestrzeni nazw środowiska .NET i modułów Delphi. W tym miejscu zajmiemy się zawartością i wewnętrzną budową modułów Delphi, a przy okazji jeszcze raz sprawdzimy, jak bardzo moduły Delphi pokrywają się z funkcjonującym w środowisku .NET pojęciem przestrzeni nazw.

Zawartość przestrzeni nazw

Na poziomie środowiska .NET przestrzeń nazw może zawierać tylko jeden rodzaj elementów — typy. Typ w środowisku .NET zawsze jest typem obiektowym, czyli klasą. Poza tym definiowane mogą być jeszcze następujące rodzaje typów:

- typy wartości (C#: `struct`, Delphi: `record`),
- typy wyliczeniowe (C#: `enum`, Delphi: typ wyliczeniowy),
- interfejsy (C# i Delphi: `interface`),
- delegacje (C#: `delegate`, Delphi: typ metody, `procedure … of object`),
- klasy (C# i Delphi: `class`).

Wszystkie pięć rodzajów typów można definiować również w programach tworzonych w Delphi, wykorzystując przy tym słowa kluczowe specyficzne dla Delphi (zostały one wymienione na powyższej liście). Każda z takich definicji typów musi być w Delphi zapisana w sekcji pliku źródłowego opisanej słowem `type`.

Zawartość modułów

Moduł w Delphi składa się z następujących elementów, które mogą w nim występować w dowolnej kolejności, pod warunkiem, że nie przeczą temu żadne powiązania łączące poszczególne zadeklarowane elementy:

- typy (sekcje pliku źródłowego opisane słowem kluczowym `type`),
- stałe (sekcje `const`),
- zmienne (sekcje `var`),
- procedury i funkcje (nie ma opisu sekcji, ale każda procedura musi rozpoczynać się od słowa kluczowego `procedure`, a każda funkcja od słowa kluczowego `function`).

Wszystkie typy z języka Object Pascal można łatwo przekształcić w odpowiadające im typy środowiska .NET, ale nadal problemem pozostają inne elementy języka, takie jak stałe, zmienne i metody, które w środowisku .NET występują wyłącznie jako elementy klasy, a w języku Object Pascal mogą występować całkowicie niezależnie od deklaracji klas, jako zmienne, stałe i metody *globalne*.

Te reguły funkcjonowania języka Object Pascal nie powinny być zmieniane, dlatego kompilator Delphi stosuje pewien wybieg pozwalający na używanie globalnych stałych, zmiennych i metod, które opakowywane są w klasę środowiska .NET o nazwie `Unit`, całkowicie niewidoczną dla programisty. Podczas gdy te globalne symbole mogą być normalnie zmieniane wewnątrz programów tworzonych w Delphi, to zewnętrzne kompilaty elementy te widzieć będą jako część klasy `Unit`.

Budowa modułu

Moduł Delphi zbudowany jest tak, jak pokazano na listingu 3.9.

Listing 3.9. *Ogólna zawartość jednego modułu Delphi*

```
unit ModuleName; {Nazwa modułu nie może zostać pominięta}

interface
[Klauzula uses]
[Deklaracje]

implementation
[Klauzula uses]
[Deklaracje i definicje]

Główny program modułu zakończony słowem "end."
```

Moduł może zawierać w sobie wszystkie elementy języka Object Pascal: procedury, funkcje, zmienne, stałe, typy i w szczególności klasy, a zadaniem modułu jest częściowe lub całościowe udostępnianie tych elementów innym modułom.

Wszystkie obiekty, które mogą być używane także przez inne moduły, muszą być zadeklarowane w części `interface`. W ten sposób wszystkie te identyfikatory uznawane są za publiczne, czyli widoczne również na zewnątrz. W przypadku zmiennych, stałych i typów prostych wystarczająca jest tylko deklaracja, ale w przypadku funkcji i procedur, a także metod różnych klas konieczne jest też dopisanie ich definicji w sekcji `implementation`. Wszystkie pozostałe deklaracje umieszczone w sekcji implementacji traktowane są jako prywatne.

Cykliczne łączenie modułów

Powodem stosowania w modułach dwóch osobnych instrukcji `uses` umieszczonych w sekcjach `interface` i `implementation` jest umożliwienie cyklicznych związków użytkowania, na przykład w sytuacji, gdy dwa moduły muszą wykorzystywać się wzajemnie.

W przykładzie przedstawionym na listingu 3.10, kompilator mógłby wpaść w nieskończoną pętlę analizowania modułów, jeżeli nie rozpoznałby prawidłowo błędu w modułach (instrukcja `uses Unit2` nakazałaby mu wczytać dane z modułu `Unit2`, a znajdująca się w nim instrukcja `uses Unit1` ponownie nakazałaby wczytanie danych z modułu `Unit1`).

Listing 3.10. *Nieprawidłowy sposób tworzenia cyklicznych dowiązań modułów*

```
{początek pliku pierwszego modułu}
unit Unit1; interface uses Unit2;
{początek pliku drugiego modułu}
unit Unit2; interface uses Unit1;
```

W takim wypadku jedna z klauzul `uses` musi zostać przeniesiona do części implementacji modułu, na przykład w sposób przedstawiony na listingu 3.11.

Listing 3.11. *Właściwy sposób tworzenia cyklicznych dowiązań modułów*

```
{początek pliku pierwszego modułu}
unit Unit1; uses Unit2;
...
{początek pliku drugiego modułu}
unit Unit2;
interface
implementation
uses Unit1;
```

Jeżeli kompilator zajmować się będzie analizowaniem modułu `Unit1`, to może w ramach tego procesu wczytać interfejs modułu `Unit2` i nie natrafi tam na instrukcję odsyłającą go z powrotem do modułu `Unit1`. W trakcie analizowania modułu `Unit1` kompilator zupełnie nie interesuje się częścią implementacji modułu `Unit2`. Co prawda interfejs modułu `Unit1` jest uzależniony od modułu `Unit2`, ale interfejs modułu `Unit2` jest całkowicie niezależny od modułu `Unit1`. Przedstawione wyżej rozwiązanie powoduje oczywiście pewne komplikacje, ponieważ uniemożliwia stosowanie identyfikatorów z modułu `Unit1` w interfejsie modułu `Unit2`. Oznacza to, że dwa moduły wykorzystujące się wzajemnie muszą zostać zaprojektowane tak, żeby w interfejsie jednego z nich nie były używane klasy ani typy deklarowane w drugim module.

Inicjalizacja i zamykanie modułów

W przedstawionej wyżej konstrukcji modułów część oznaczona jako „główny program modułu" wywoływana jest w momencie uruchamiania programu, co pozwala modułowi prawidłowo się zainicjować. Sekcję tę można rozpocząć za pomocą tradycyjnego słowa kluczowego `begin` lub wprowadzonego niedawno słowa `initialization`.

W każdym module zastosować można jeszcze słowo kluczowe `finalization` rozpoczynające tę część kodu modułu, która zawsze wywoływana jest w momencie kończenia programu. Jak widać, zakończenie modułu może wyglądać tak jak na listingu 3.12.

Listing 3.12. *Końcowa część modułu*

```
initialization
  {kod wykonywany przy inicjalizacji modułu}
finalization
  {kod czyszczący, wykonywany przy zamykaniu programu}
end.
```

3.1.5. Moduły Delphi dla nowicjuszy

Moduły języka Object Pascal wyróżniają się spośród modułów innych języków programowania, takich jak C++, poprzez swoją wyjątkową niezależność i zamkniętą strukturę. W języku C++ pliki mogą być ze sobą wiązane na wiele różnych sposobów, a jeden moduł najczęściej zapisywany jest w pliku implementacji i pliku nagłówkowym (*.cpp* i *.h*), natomiast w języku Object Pascal moduł jest pojedynczym zamkniętym plikiem tekstowym, zawierającym w sobie zarówno część interfejsu modułu (co odpowiada plikowi nagłówkowemu języka C++), jak i część implementacji

Programiści korzystający z języków Java lub C# przyzwyczajeni są do pracy z takimi całkowicie kompletnymi plikami modułów. W Delphi będą musieli się dodatkowo przyzwyczaić do tego, że klasy rozbijane są na część deklaracji i część implementacji, przy czym w części deklaracji znajdować się mają wyłącznie deklaracje klas publicznych, a w części implementacji zapisywane są implementacje wszystkich rodzajów klas.

Rozdzielenie na część interfejsu i implementacji nie generuje aż tak wielkiej ilości dodatkowej pracy, jak mogłoby się początkowo wydawać. W Delphi dostępna jest funkcja automatycznego uzupełniania klas, która między innymi pozwala na automatyzację synchronizacji części interfejsu i implementacji modułu. Poza tym bardzo wygodne jest to, że deklaracja klasy przechowuje wyłącznie nagłówki metod, a nie ich pełny kod. Dzięki temu można łatwo przejrzeć interfejs klasy nawet w najprostszym edytorze nierealizującym funkcji pomocniczych, takich jak zwijanie kodu lub okna struktury.

3.2. Obiekty i klasy

Główną koncepcją w programowaniu zorientowanym obiektowo jest połączenie danych i kodu, które w programowaniu proceduralnym są od siebie całkowicie oddzielne, wewnątrz zamkniętych struktur nazywanych *obiektami*. Obiekty, z którymi stykamy się w projektancie formularzy, to przede wszystkim komponenty, kontrolki i formularze, ale wśród właściwości wyświetlanych w oknie inspektora obiektów równie często spotyka się obiekty, takie jak obiekty `Font` definiujące czcionkę stosowaną w kontrolce lub obiekty kolekcji przechowujące wpisy kontrolki *ListBox*, kolumny kontrolki *ListView* lub zakładki kontrolki *TabControl*.

W środowisku .NET nawet najprostsze wartości, takie jak liczby lub ciągi znaków, mogą być traktowane jako obiekty, dzięki czemu liczbę całkowitą można zamieniać w ciąg znaków stosując stare proceduralne wywołanie:

```
str := IntToStr(liczba);
// Funkcja IntToStr może być też wywoływana jako część modułu SysUtils,
// w takim wypadku wywołanie należy zapisać tak:
str := SysUtils.IntToStr(liczba);
```

… ale równie dobrze można potraktować liczbę jak obiekt i skorzystać z jednej z jego metod:

```
str := liczba.ToString;
```

Pojęcie klasy

Język Pascal zawsze był językiem bardzo restrykcyjnie pilnującym typów, w którym każda zmienna musiała mieć określony typ, dlatego obiekty w języku Object Pascal (a także w środowisku .NET) również mają swoje określone typy. W przypadku obiektów typ nazywany jest klasą obiektu, a każdy obiekt, którego typ określa pewna klasa, jest *egzemplarzem* lub *instancją* tej klasy.

Przykładem ułatwiającym rozróżnianie klas i ich egzemplarzy (instancji) są formularze przygotowywane w projektancie formularzy. Tworzone w nich klasy okien w czasie działania programu zamieniane są w rzeczywiste okna wyświetlane na ekranie.

W tej książce stosował będę wyrażenie *egzemplarz*. Równie często występujące określenie *instancja* powstało najprawdopodobniej w wyniku pierwszych i nie do końca przemyślanych tłumaczeń angielskiego słowa *instance*. Słowo „instancja" w języku polskim nie jest dokładnym tłumaczeniem słowa angielskiego, ale oznacza różne stopnie w hierarchii instytucji.

3.2.1. Deklaracja klasy

Jako przykład deklaracji klasy przedstawiam na listingu 3.13 uproszczoną wersję klasy `TimerEvent` znanej nam już z punktu 2.2.3.

Listing 3.13. *Uproszczona deklaracja klasy TimerEvent*

```
type
  TimerEvent = class
  public
    // Zmienne:
    ActivationTime: DateTime;
    // Metody:
    function ToString: String; override; // Zmienia dane zdarzenia w tekst
    procedure Trigger; virtual;          // Wywołuje zdarzenie
  end;
```

Deklaracje klas zawsze znajdują się w części pliku źródłowego rozpoczynającej się od słowa kluczowego `type`. Najważniejszym słowem kluczowym w takiej deklaracji jest słowo `class`, odróżniające zapisaną dalej strukturę od rekordów, które deklarowane są za pomocą słowa kluczowego `record`.

Za słowem kluczowym wypisywane są elementy klasy pogrupowane w sekcje opisane słowami kluczowymi `private` i `public`, przy czym na podanym wyżej przykładowym listingu z deklaracją klasy znajdziemy wyłącznie sekcję `public`.

Wewnątrz jednej z sekcji trzeba deklarować najpierw zmienne, a dopiero za nimi metody i właściwości. Po pojawieniu się pierwszej deklaracji metody następne zmienne deklarowane mogą być dopiero po pojawieniu się kolejnego opisu sekcji, na przykład `public`. W poszczególnych deklaracjach obowiązują następujące reguły:

- Zmienne deklarowane są tak samo jak zwyczajne zmienne języka Object Pascal. Szczegóły na ten temat podaję w punkcie 3.5.3. Na początek wystarczy, że będziemy znać podstawową składnię *nazwa*: *typ*, za pomocą której deklarowana jest zmienna *nazwa* o typie *typ*.
- Deklaracje metod również stosują się do składni języka Object Pascal obowiązującej w deklaracjach wszystkich funkcji i procedur. Jeżeli metoda zwraca w wyniku pewną wartość, to deklarowana jest słowem kluczowym `function`, a typ zwracanej wartości zapisywany jest na końcu deklaracji po dwukropku. Pozostałe rodzaje deklaracji wykorzystują słowo kluczowe `procedure`. Wszystkie parametry przyjmowane przez metodę zapisywane są w nawiasach zaraz za jej nazwą. Więcej informacji na temat ogólnej składni stosowanej w deklaracjach znaleźć można w podrozdziale 3.8. W deklaracjach metod stosowane są jeszcze pewne specjalne dyrektywy, takie jak widoczne w powyższym listingu dyrektywy `override` i `virtual`, które opisywał będę za chwilę.
- Właściwości to specjalne elementy występujące **wyłącznie** wewnątrz klas. Właściwościom poświęcony został punkt 3.2.4.

Uzupełnianie klas

Po zapisaniu w deklaracji klasy samych nazw, parametrów i typów wartości zwracanych przez metody, w dalszej części pliku źródłowego przygotowane muszą być implementacje tych metod (czyli ciała metod z zapisanym wykonywanym w nich kodem), całkowicie niezależnie od ich deklaracji. W języku Object Pascal nie jest możliwe implementowanie metod bezpośrednio w deklaracjach klas, co umożliwiają języki C# i Java. Wszystkie implementacje metod muszą być zapisane w części implementacyjnej modułu.

Najprostszą metodą na uzyskanie szkieletu implementacji wszystkich metod jest wywołanie dostępnej w Delphi funkcji automatycznego uzupełniania klasy. Po umieszczeniu kursora edytora wewnątrz przedstawionej wyżej deklaracji klasy i naciśnięciu kombinacji klawiszy *Ctrl+Shift+C*, Delphi przygotuje w części implementacji modułu szkielety metod przedstawione na listingu 3.14.

Listing 3.14. *Implementacje metod przygotowane przez funkcję automatycznego uzupełniania klas*

```
implementation

{ TimerEvent }

constructor TimerEvent.Create;
begin
```

```
end;

function TimerEvent.ToString: String;
begin

end;

procedure TimerEvent.Trigger;
begin

end;
```

Konwencje nazewnicze

Delphi stanowić ma pomost pomiędzy różnymi światami programowania, dlatego nie znajdziemy w nim żadnych z góry narzuconych konwencji nazewniczych, które miałyby obowiązywać we wszystkich klasach zaprogramowanych w Delphi lub wykorzystywanych w tworzonych programach. W poprzednich wersjach Delphi sytuacja była jeszcze całkiem przejrzysta, ponieważ programiści stosowali ogólną zasadę przygotowaną przez firmę Borland, mówiącą, że nazwy klas zaczynać się mają od wielkiej litery `T` (podobnie jak i wszystkie inne nazwy typów w języku Object Pascal), z której wynikają wszystkie nazwy klas obecnych w bibliotece VCL: na przykład `TColor`, `TForm` i `TButton`.

W środowisku .NET nie ma takiej tradycji, a klasy nazywają się tutaj po prostu `Color`, `Font` lub `Button`. Ze względu na ogromne ilości klas dostępnych w środowisku .NET, klasy Delphi z przedrostkiem `T` znajdują się w mniejszości. Pozwala to jednak na dość łatwe określenie pochodzenia danej klasy. Jeżeli nie wykorzystujemy żadnych dodatkowych klas pochodzących od firm trzecich, to można łatwo stwierdzić, że klasy z przedrostkiem `T` przygotowane zostały przez firmę Borland, a wszystkie pozostałe pochodzą z firmy Microsoft.

W przykładowych programach prezentowanych w tej książce te dwa schematy nazywania klas stosowane są zamiennie, w zależności od tego, czy programowi bliżej jest do biblioteki VCL, czy też do biblioteki klas środowiska .NET. W ten sposób klasy prezentowane w rozdziale 1. nie miały w nazwach dodatkowej litery `T`, ale w rozdziale 4. nazwy klas aplikacji tworzonych w bibliotece VCL.NET będą zaczynały się od tej litery.

3.2.2. Atrybuty widoczności

Dzięki stosowaniu w deklaracjach klas opisów poszczególnych sekcji, podobnych do opisu `public`, jaki mieliśmy okazję zobaczyć w deklaracji klasy `TimerEvent`, możemy chronić pewne elementy klasy przed dostępem z zewnątrz. Do wyboru mamy tutaj następujące stopnie ochrony:

- `public` — Oznacza elementy publiczne, czyli niepodlegające żadnej ochronie. Do elementów tych można dowolnie odwoływać się z zewnątrz klasy. W idealnie przygotowanym programie obiektowym w klasach publiczne są wyłącznie metody i właściwości, ale nie zwyczajne dane.

- `strict protected` — Oznacza elementy chronione. Zabezpiecza elementy przed dostępem z zewnątrz, co oznacza na przykład, że zmienna zadeklarowana w chronionej sekcji kontrolki *Button* nie będzie dostępna wewnątrz procedury obsługi zdarzenia formularza, na którym ułożona jest ta kontrolka. Do elementów deklarowanych w tej sekcji można się jednak odwoływać bez żadnych ograniczeń w klasach wywiedzionych.
- `strict private` — Oznacza elementy prywatne. Nie pozwala na dostęp do tak zabezpieczonych elementów nawet z klas wywiedzionych, przez co klasa ta ma niepodzielną kontrolę nad tymi elementami.
- `protected` i `private` — Oba powyższe atrybuty mogą też występować bez przedrostka `strict`, co umożliwia dostęp do tak zabezpieczanych elementów z dowolnych metod zadeklarowanych w tym samym module, w którym znajduje się deklaracja klasy. Dzięki temu do tych elementów dostęp będą miały też **inne** klasy zadeklarowane w tym module. To rozszerzenie pozwolenia dostępu odpowiada poziomowi widoczności `assembly`, jaki istnieje w środowisku CLR, z kolei atrybutowi `strict private` w środowisku CLR odpowiada poziom widoczności `private`, a atrybutowi `strict protected` — poziom widoczności `family`.
- `published` — Oznacza elementy opublikowane. Ten poziom zabezpieczeń ma znaczenie tylko dla komponentów, które mają być używane w ramach projektanta formularzy. W takich komponentach właściwości, które mają być wyświetlane w inspektorze obiektów, muszą być deklarowane jako opublikowane.

Standardowym ustawieniem zabezpieczeń elementów klasy, czyli obowiązującym do czasu pojawienia się pierwszego atrybutu zabezpieczeń, jest brak jakichkolwiek zabezpieczeń, czyli `public`, a w kontekście biblioteki VCL.NET — `published`.

3.2.3. Samoświadomość metody

Połączenie danych i kodu w ramach programowania zorientowanego obiektowo szczególnie dobrze widać w sposobie, w jakim obiekt odwołuje się do swoich własnych danych. Na listingu 3.15 przedstawiam przykład metody formularza, w której zmieniany jest kolor tła okna.

Listing 3.15. *Procedura formularza zmieniająca kolor okna*

```
procedure TwinForm.ChangeBackgroundColor;
begin
  BackColor := Color.Red;
end;
```

Formularz odwołuje się do swojego elementu `BackColor` po prostu wymieniając jego nazwę; na pierwszy rzut oka jest to jak najbardziej naturalne działanie. Formularz jest jednak tylko klasą, na podstawie której tworzony jest faktyczny obiekt okna, a co więcej — na podstawie klasy formularza przygotować można kilka takich samych okien. A gdy istnieć już będzie kilka okien typu `TWinForm`, wtedy powstanie pytanie — kolor tła którego okna zmienia instrukcja przedstawiona na powyższym listingu?

Niejawny parametr self

Odpowiedź na to pytanie brzmi: do każdej metody automatycznie przekazywany jest niejawny parametr `self`. Parametr ten jest obiektem, na rzecz którego wywołana została dana metoda. Na potrzeby każdego obiektu, który stosowany jest wewnątrz metody klasy, kompilator automatycznie tworzy ten specjalny parametr `self`. Wynika z tego, że ciało metody można przedstawić za pomocą kodu z listingu 3.16.

Listing 3.16. *Rzeczywisty zapis wnętrza jednej z metod formularza*

```
procedure TWinForm.Methode(self: TForm1; ... tutaj zapisane są jawnie zadeklarowane
parametry);
begin
  with self do begin
    ... tutaj pojawiają się jawnie zapisane instrukcje
  end;
end;
```

Dzięki zastosowaniu instrukcji `with self do` wszystkie nazwy zapisane w tym bloku, które są nazwami elementów obiektu `self`, będą dotyczyły wyłącznie tego obiektu. To wszystko oznacza, że przedstawiona wyżej instrukcja zmieniająca kolor tła formularza wewnątrz tej procedury wygląda tak:

```
self.BackColor := Color.Red;
```

Obiekt self w czasie działania programu

Dla lepszego zobrazowania działania tego mechanizmu na listingu 3.17 przedstawię wycinek kodu, w którym stosowane są dwa obiekty tego samego formularza.

Listing 3.17. *Kod, w którym wykorzystywane są dwa obiekty utworzone na podstawie tego samego formularza*

```
var
  Form1, Form2:TWinForm;
begin
  ...
  Form2.ChangeBackgroundColor;
```

W tym przypadku parametr `self` wywoływanej metody wskazywać będzie na obiekt `Form2`, a w związku z tym w kodzie metody `ChangeBackgroundColor` (jego aktualną wersję przedstawiam na listingu 3.18) zmieniany będzie kolor tła okna `Form2`.

Listing 3.18. *Kod metody wykonywany w wyniku wywołania przedstawionego na listingu 3.17*

```
procedure TwinForm.ChangeBackgroundColor(self = Form2);
begin
  Form2.BackColor := Color.Red;
end;
```

Jak widać, rozróżnianie wielu obiektów w czasie działania programu obywa się właściwie całkowicie automatycznie. W czasie programowania metod jednej klasy sytuację tę można sobie wyobrazić tak, że w systemie nie istnieje wiele obiektów tej klasy,

ale tylko jeden, a my przygotowujemy nie metody klasy, ale metody tego jednego obiektu. Obiekt, o którym tu mówię, nazywa się właśnie `self`. Nie można jednak nigdy zapomnieć o tym, że w czasie działania programu tworzone przez nas metody mogą być wywoływane na rzecz różnych wartości obiektów zapisanych w parametrze `self` (chyba że nasza klasa została celowo tak ograniczona, że na jej podstawie może być utworzony tylko jeden obiekt).

Dopóki sami nie zadeklarujemy osobnej zmiennej, która również będzie nazywała się `self`, możemy jawnie stosować identyfikator `self`, aby w ten sposób pominąć na przykład działanie innej instrukcji **`with`**. W kodzie przedstawionym na listingu 3.19, ze względu na taką właśnie instrukcję **`with`**, kompilator uznaje, że identyfikator `BackColor` nie jest powiązany z obiektem `self`, ale z obiektem `Button1`. W takich warunkach, chcąc odwołać się do właściwości `BackColor` formularza, musimy jawnie odwołać się do obiektu `self`.

Listing 3.19. *Instrukcja with może wymusić jawne odwołanie się do obiektu self*

```
with Button1 do
  BackColor := self.BackColor;
  {bez instrukcji with można stosować taki zapis}
Button1.Color := Color;
```

3.2.4. Właściwości

Właściwości obiektów znoszą sprzeczność powstającą pomiędzy filozofią programowania zorientowanego obiektowo a chęcią jak najłatwiejszego tworzenia programów. Jeżeli bez stosowania właściwości chcielibyśmy jak najprościej zmieniać wartości elementów danych obiektów, to zmuszeni bylibyśmy do stosowania takiego zapisu:

```
obiekt.ElementDanych := NowaWartosc;
```

Taki zapis zadziałałby tylko wtedy, gdyby `ElementDanych` był zadeklarowany jako publiczny element klasy, co byłoby jednak zaprzeczeniem filozofii programowania obiektowego, która wymaga, aby takie przypisanie wartości odbywało się poprzez następujące wywołanie metody:

```
obiekt.SetElementDanych(NowaWartosc);
```

Takie rozwiązanie ma tą zaletę, że kod wywołujący nie jest zależny od wewnętrznej struktury danych klasy (jeżeli `ElementDanych` zostałby później inaczej nazwany, usunięty albo przeniesiony do zupełnie innej struktury danych, to konieczna byłaby tylko przebudowa metody `SetElementDanych`, ale wszystkie jej wywołania mogłyby pozostać bez zmian). Poza tym, metoda `SetElementDanych` mogłaby przy okazji wykonywać jeszcze inne operacje, które wiązałyby się ze zmianami wartości zmiennej `ElementDanych` (na przykład aktualizacja informacji na ekranie).

Jeżeli teraz `ElementDanych` nie będzie zmienną, ale właściwością, to można wygodnie zapisać:

```
obiekt.ElementDanych := NowaWartosc;
```

…a mimo to wywołana zostanie metoda `Set`….

Deklarowanie własnych właściwości

Właściwości umieszczane są wewnątrz deklaracji klasy dokładnie w tym samym miejscu co metody, czyli wewnątrz jednej sekcji zabezpieczeń (`public`, `private`, …) nie można ich deklarować przed znajdującymi się w niej zmiennymi. Deklaracja przykładowej właściwości `Width` wygląda następująco:

```
property Width: Integer read GetWidth write SetWidth;
```

Dyrektywy `read` i `write` znajdujące się za deklaracją typu właściwości (`Integer`) wskazują metody stosowane w czasie odczytywania i zapisywania wartości właściwości. Opuszczenie jednej z tych dyrektyw spowoduje, że deklarowana właściwość będzie mogła być tylko odczytywana lub tylko zapisywana.

Metoda odczytująca właściwość musi być funkcją bezparametrową, której typ zwracanej wartości zgodny jest z typem właściwości. Z drugiej strony, metoda zapisująca właściwość musi być procedurą pobierającą jeden parametr typu zgodnego z typem właściwości. Kody metod powiązanych z deklarowaną wyżej właściwością `Width` podaję na listingu 3.20.

Listing 3.20. *Kody metod odczytujących i zapisujących wartości właściwości Width*

```
funktion TPewnaKlasa.GetWidth: Integer;
begin
  Result := FWidth;
end;

procedure TPewnaKlasa.SetWidth(NewWidth: Integer);
begin
  FWidth := NewWidth;
  Invalidate;
end;
```

Tak jak w przedstawionych wyżej metodach, z właściwościami powiązane są też zmienne o takim samym typie, a zadaniem metod właściwości jest wykonywanie dodatkowych operacji powiązanych z operacjami odczytu i zapisu danych do właściwości. Najczęściej zmienna ta otrzymuje nazwę składającą się z początkowej litery *F* i nazwy samej właściwości (w naszym przykładzie zmienna nazywa się `FWidth`).

Zmienne „przebrane" za właściwości

Pewną alternatywą w stosunku do stosowania metod podawanych w dyrektywach `read` i `write` jest możliwość podawania w nich nazwy zmiennych, które mogą być bezpośrednio zapisywane lub odczytywane. W podanym wyżej przykładzie metoda `GetWidth` zajmuje się wyłącznie odczytaniem wartości zmiennej `FWidth`, wobec czego ten sam efekt można uzyskać deklarując właściwość tak jak na listingu 3.21.

Listing 3.21. *Inny sposób zadeklarowania właściwości Width*

```
{ Wszystkie deklaracje wymagane do zadeklarowania właściwości }
Fwidth: Integer;
procedure SetWidth(NewWidth: Integer);
property Width: Integer; read FWidth write SetWidth;
```

Przykłady właściwości

W tej chwili wskażę tylko praktyczne przykłady właściwości, jakie można znaleźć w innych miejscach tej książki:

- W klasie `RTFAttributes` prezentowanej na stronie 171 zadeklarowanych jest wiele właściwości opisujących atrybuty stosowane w kontrolce *RichTextBox*.
- W klasie `DesktopChangeEvent` ze strony 216 stosowana jest właściwość `ColorARGB`, umożliwiająca dopasowanie klasy do wymogów serializacji XML.

Ponadto w rozdziale 6. definiowane są komponenty, których nieodłączną częścią są właściwości pozwalające na edytowanie pewnych ustawień komponentu w inspektorze obiektów.

Właściwości tablicowe

Właściwości tablicowe na pierwszy rzut oka działają dokładnie tak samo jak zwyczajne tablice, a dostęp do ich elementów można uzyskać na przykład stosując poniższy zapis:

```
Colors[0] := System.Drawing.Color.White;
```

Instrukcja ta przypisuje wartość zerowemu elementowi właściwości `Colors`. Deklarowanie takiej właściwości wymaga zadeklarowania nazwy właściwości i zamknięcia danych o wymiarach tablicy wewnątrz nawiasów prostokątnych (na przykład `[I: byte]`). Typ poszczególnych elementów tablicy podawany jest po prostokątnym nawiasie zamykającym, tak jak na listingu 3.22.

Listing 3.22. *Deklarowanie właściwości tablicowych*

```
// Tablica obiektów typu TColor indeksowana wartościami typu integer
property Colors[i: Integer]: TColor read GetColor write SetColor;
// Trójwymiarowa tablica obiektów
// indeksowana trzema wartościami typu integer
property TrzyWymiary[x, y, z: Integer]: TObject; read Read3D write Set3D;
```

W deklaracjach metod odczytujących i zapisujących właściwości tablicowe muszą pojawić się zmienne indeksowe wymienione w deklaracji wymiarów samej właściwości. Dla przedstawionych wyżej właściwości deklaracje tych metod muszą wyglądać tak jak na listingu 3.23.

Listing 3.23. *Metody zapisujące i odczytujące wartości właściwości tablicowych*

```
function GetColor(i: Integer): TColor;
procedure SetColor(i: Integer; val: TColor);
function Get3D(x, y, z: Integer): TObject;
procedure Set3D(x, y, z: Integer; val: TObject);
```

Właściwość tablicy standardowej

Jeżeli obok deklaracji właściwości tablicowej zapisane zostanie słowo kluczowe `default` (taką deklarację przedstawiam na listingu 3.24), to właściwość ta stanie się *standardową właściwością* tej klasy.

Listing 3.24. *Deklarowanie standardowej właściwości klasy*

```
type
  TList = class
    property Items[Index: Integer]: TObject;
             read GetItem write SetItem; default;
```

Dla obiektu klasy `TList` taka deklaracja oznacza na przykład, że cały obiekt tej klasy można obsługiwać dokładnie tak samo jak tablicę, bez konieczności wymieniania konkretnej właściwości, tak jak w kodzie przedstawionym na listingu 3.25.

Listing 3.25. *Traktowanie obiektu listy jako tablicy*

```
var
  List: TList;
  PierwszyObiekt: TObject;
begin
  List := TList.Create(...);  // Inicjowanie listy
  PierwszyObiekt := List[0];
```

Zapisane w powyższym listingu wyrażenie `List[0]` ma dokładnie takie samo znaczenie jak zapis `List.Items[0]`. Jak można się domyślać, każda klasa może mieć najwyżej jedną właściwość standardową tego rodzaju.

Jedna metoda dla wielu właściwości

Właściwości indeksowane są bardzo podobne do właściwości tablicowych, ale w ich przypadku dostęp do poszczególnych elementów uzyskiwany jest nie za pomocą indeksów, ale poprzez specjalne nazwy. W klasie `TGraphicElement` programu *TreeDesigner* właściwości `Left`, `Right`, `Top` i `Bottom` zadeklarowane są za pomocą kodu przedstawionego na listingu 3.26.

Listing 3.26. *Deklarowanie właściwości w klasie TGraphicElement*

```
property Left:   Integer index 1 read  GetCoord;
property Right:  Integer index 2 read  GetCoord;
property Top:    Integer index 3 read  GetCoord;
property Bottom: Integer index 4 read  GetCoord;
```

Wszystkie cztery właściwości odczytywane są za pomocą tej samej metody, która rozróżnia te właściwości korzystając z indeksu zapisanego w ich deklaracji. Kompilator automatycznie dodaje właściwą wartość indeksu do wszystkich wywołań metody odczytującej (kod tej metody podaję na listingu 3.27).

Listing 3.27. *Odczytywanie indeksowanych właściwości w klasie TGraphicElement*

```
function TGraphicElement.GetCoord(Index : Integer) : Integer;
begin
  with Points do
    case Index of
      1 : Result:=Left;
      2 : Result:=Right;
      3 : Result:=Top;
      4 : Result:=Bottom;
    end;
end;
```

Dokładnie tak samo działają procedury zapisujące wartości do takich właściwości. One również wymagają podania dwóch parametrów: indeksu właściwości i jej nowej wartości. W podanym wyżej przykładzie nie były definiowane żadne metody zapisujące, ponieważ wartości czterech przedstawionych właściwości można wyłącznie odczytywać.

3.2.5. Metody klas i zmienne klas

Delphi dla .NET pozwala na stosowanie w klasach metod statycznych, a także statycznych zmiennych i właściwości. W Delphi 7 dozwolone było stosowanie wyłącznie metod statycznych, które od tego czasu nazywane były też „metodami klas", co wynikało ze specyficznej składni stosowanej w czasie ich deklarowania (`class function`/`procedure`). Takie metody są odpowiednikami metod statycznych funkcjonujących na poziomie CLR (mają w stosunku do tych metod statycznych pewne ograniczenie, ale na ten temat mówił będę za chwilę).

Określenie „metoda klasy" oznacza, że jest ona wywoływana **dla** klasy, a więc nie ma związku z którymkolwiek obiektem utworzonym na podstawie tej klasy. Na listingu 3.28 przedstawiam przykład takiej metody.

Listing 3.28. *Przykład fikcyjnej metody statycznej*

```
type
  TDemoClass = class
    class function GetSpeedEstimate:  Integer;
    { Ta funkcja określa szacunkową prędkość działania klasy.
      Jeżeli zwracana wartość jest większa niż 100, to znaczy,
      że klasa działa bardzo szybko}

// Funkcja klasy wywoływana jest następująco
if TDemoClass.GetSpeedEstimate < 100
  then ShowMessage('Za wolno!')
```

Metody statyczne w stylu CLR

W przedstawionym wyżej przykładzie metoda statyczna `GetSpeedEstimate` jest odpowiednikiem metod statycznych funkcjonujących w środowisku .NET i w związku z tym jest w kompilatorze Delphi przekształcana w taką właśnie metodę.

W bibliotece klas środowiska .NET metody statyczne stosowane są przede wszystkim do tworzenia funkcji narzędziowych, które mają działać całkowicie niezależnie od jakichkolwiek obiektów. Wcześniej, w języku C++ (a także w Delphi) takie funkcje narzędziowe deklarowane były jako funkcje globalne, jednak środowisko CLR nie pozwala na stosowanie elementów globalnych. Statyczne metody traktowane są tutaj jako wybieg, ponieważ pozwalają na definiowanie metod, które można wykorzystywać dokładnie tak samo jak wszechobecne wcześniej metody globalne. Do wywołania metody statycznej nie jest potrzebne tworzenie żadnego obiektu, na rzecz którego można by ją wywołać. W tym wypadku takim obiektem jest sama klasa, i to na jej potrzeby wywoływana jest metoda.

Wiele klas środowiska .NET udostępnia też metody statyczne, ale przykładem szczególnie typowym dla przedstawionej wyżej koncepcji zamienników dla funkcji globalnych są klasy udostępniające wyłącznie metody statyczne, takie jak `File`, `Directory` lub `Path` (mówiliśmy o nich w punkcie 2.6.2).

W konsekwencji takiego funkcjonowania metod statycznych w środowisku .NET, nie otrzymują one niejawnego parametru `self`. I to właśnie jest podstawowa różnica pomiędzy metodami statycznymi środowiska .NET a metodami statycznymi Delphi.

Metody statyczne w stylu Delphi

Metody statyczne w Delphi w parametrze `self` otrzymują wskazania na klasę, określające klasę, na rzecz której wywołana została metoda (w przypadku wywołania `TDemoClass.GetSpeedEstimate` w parametrze tym znajdowałoby się wskazanie na klasę `TDemoClass`). W tekście źródłowym w Delphi parametr `self` jest parametrem niejawnym, ale wywołując tę metodę z innego języka programowania albo przyglądając się jej w narzędziu Reflection zauważymy, że parametr ten jest pierwszym parametrem metody statycznej. Parametr ten stosowany jest przede wszystkim w ramach zachowania zgodności z wcześniejszymi wersjami Delphi, ponieważ w metodach statycznych, o których mówiliśmy do tej pory, był on całkowicie zbędny. W każdej metodzie statycznej doskonale wiadomo, z którą klasą jest ona związana.

W zakresie metod statycznych koncepcja języka stosowana w Delphi ma większe możliwości od tej związanej z językiem C#, w którym nie można definiować wirtualnych metod statycznych. Realizacja tej koncepcji w kompilatorze Delphi wskazuje, że można by pokusić się o „zasymulowanie" jej również w języku C#. Delphi nie przekształca wirtualnych metod statycznych w statyczne metody funkcjonujące w CLR, ale w wirtualne metody metaklasy, będącej częścią środowiska CLR. Taka metaklasa tworzona jest w Delphi dla każdej klasy zdefiniowanej w tekście programu, co pozwala na implementowanie również innych specjalnych rozwiązań działających w świecie Delphi. Dla przykładowej klasy `TDemoClass` tworzona przez Delphi metaklasa nazywałaby się `@MetaTDemoClass`, a w programie Reflection pojawiłaby się jako podrzędny węzeł klasy `TDemoClass` (na rysunku 3.1 zobaczyć można podobną metaklasę `@MetaTWinForm`).

To wszystko zmienia się jednak diametralnie, gdy przyjrzymy się jeszcze jednej właściwości Delphi. W języku Object Pascal metody statyczne mogą być deklarowane jako wirtualne (dyrektywa `virtual`), w związku z czym mogą być pokrywane w klasach

wywiedzionych (na temat metod wirtualnych mówić będziemy w punkcie 3.3.3). Wynika z tego, że parametr `self` przekazywany metodzie statycznej może wskazywać inną klasę niż ta, w której zadeklarowana jest ta metoda.

Statyczne metody klas

Jak już mówiłem, jedynym technicznym szczegółem różniącym statyczne metody Delphi od podobnych metod środowiska .NET jest dodatkowy parametr `self` przekazywany do statycznych metod w Delphi. Jeżeli chcielibyśmy być w pełni zgodni z zasadami obowiązującymi w środowisku .NET, to statyczne metody możemy deklarować z wykorzystaniem dyrektywy `static`, która powoduje, że Delphi tworzy całkowicie normalną metodę statyczną środowiska .NET, do której nie przekazuje parametru `self`:

```
class function MyClassFunction: Integer; static;
```

Taka deklaracja nabiera większego znaczenia, gdy deklarujemy właściwości klas, do których dostęp powinny mieć też inne języki programowania funkcjonujące w środowisku .NET. Języki te oczekują, że metody `Get` i `Set` obsługujące właściwość **nie** będą pobierały parametru `self`.

Zmienne klas

Zmienne i właściwości również można deklarować tak jak metody statyczne, w wyniku czego pojawiają się one tylko raz w ramach klasy, ale nie stają się częścią każdego obiektu tworzonego na podstawie klasy. W przypadku zmiennych zadeklarowanie takiego rodzaju zmiennej wymaga tylko dodania przed jej deklaracją słów kluczowych `class var`.

Jako przykład przedstawię tutaj klasę `Color` (jej deklarację przedstawiam na listingu 3.29), wzorowaną na klasie `Color` pochodzącej ze środowiska .NET, w której udostępniane są przygotowane wcześniej klasy kolorów zdefiniowane jako zmienne klasy.

Listing 3.29. *Deklaracja klasy udostępniającej statyczne zmienne obiektów kolorów*

```
type
  Color = class
    class var White: Color;
    class var Red: Color;
    ...

// Zastosowanie zmiennych klas:
Color.Red;   // To wyrażenie odczytuje zmienną Red tej klasy
```

Inicjowanie zmiennych klasy

Jeżeli zmienne klasy nie mają początkowo otrzymywać standardowych wartości stosowanych w środowisku CLR (`NULL` lub `nil`), to muszą zostać odpowiednio zainicjowane. Do statycznych zmiennych klasy może mieć dostęp dowolny zewnętrzny kod i to jeszcze przed utworzeniem jakiegokolwiek obiektu tej klasy, dlatego powstaje pytanie, kiedy najwcześniej można próbować inicjować wartości tych zmiennych.

Odpowiedź na to pytanie znaleźć można w konstruktorach klas, które również są nowością w stosunku do Delphi 7. Konstruktory klas to metody klas, deklarowane za pomocą słów kluczowych `class constructor`, a w środowisku CLR wywoływane są automatycznie jeszcze przed pierwszym wykorzystaniem danej klasy. W takim konstruktorze można na przykład zainicjować wartości wszystkich obiektów opisujących kolory, tak jak na listingu 3.30.

Listing 3.30. *Statyczny konstruktor klasy*

```
type
  Color = class
    class var White: Color;
    class var Red: Color;
    ...
     class constructor Create;
...

class constructor Color.Create;
begin
  White := Color.Create;
  Red := Color.Create;
  ...
```

Właściwości klas

W podobny sposób można deklarować też właściwości, tworząc w wyniku właściwości statyczne lub właściwości klas. W tym celu właściwości muszą być zapisane w części deklaracji klasy rozpoczynającej się od słów kluczowych `class var`. Tych słów użyliśmy już wcześniej do deklarowania zmiennych statycznych, i choć na listingu 3.30 zastosowane były one przy każdej deklaracji zmiennej, to w rzeczywistości wystarczyłoby tylko raz zapisać je przed pierwszą zmienną. W przykładowym kodzie z listingu 3.31 deklarowane są dwie zmienne i dwie właściwości tylko do odczytu, wszystkie będące statycznymi elementami klasy.

Listing 3.31. *Deklarowanie statycznych zmiennych i właściwości*

```
type
  Color = class
    class function GetWhite: Color; static;
    class var
       FRed: Color;
       FWhite: Color;
       property Red read FRed;         // Dostęp poprzez zmienną statyczną
       property White read GetWhite; // Dostęp poprzez metodę statyczną
    ...
    public  // Koniec części "class var" w deklaracji klasy
    ...
end;          // Koniec deklaracji klasy
```

W podanym kodzie część deklaracji klasy zapoczątkowana słowami kluczowymi `class var` kończy się na słowie `public`. W podobny sposób zakończyć można tę część deklaracji klasy stosując inne atrybuty widoczności, takie jak `private` lub `protected`; zakończeniem sekcji elementów statycznych jest też pierwsza deklaracja metody.

Zapisane we właściwości statycznej dyrektywy `read` i `write` muszą wskazywać albo na statyczne zmienne klasy, albo na statyczne metody klasy, czyli metody, które zadeklarowane zostały z wykorzystaniem słów kluczowych `class function/procedure` i słowa `static`.

3.2.6. Dziedziczenie

Jedną z najbardziej podstawowych cech programowania obiektowego jest dziedziczenie, w którym klasa bazowa przekazuje wszystkie swoje elementy do klasy wywiedzionej. Klasa wywiedziona staje się dzięki temu *rozszerzeniem* klasy bazowej. W środowisku .NET i w języku Object Pascal każda klasa może mieć tylko jedną klasę bazową.

Obiekty klasy wywiedzionej zachowują się początkowo dokładnie tak samo jak obiekty klasy bazowej, ale dzięki nowym deklaracjom metod i pokrywaniu implementacji istniejących metod można dodawać do wywiedzionej klasy nowe funkcje lub zmieniać w niej działanie funkcji odziedziczonych, przez co zmienia się też zachowanie obiektów tej klasy.

W języku Object Pascal i w środowisku .NET mechanizm dziedziczenia zaszyty jest tak głęboko, że nie da się w nich przygotować nowej klasy **bez** dziedziczenia, nawet jeżeli w definicji nowej klasy nie podamy klasy bazowej. Taka klasa automatycznie zostanie wywiedziona z klasy `System.Object`, która to w języku Object Pascal stosowana jest po nazwą aliasu `TObject`.

Przykładem stosowania dziedziczenia jest każda klasa formularza przygotowywana w projektancie formularzy:

```
type
  TWinForm = class(System.Windows.Forms.Form)
```

Dzięki zapisowi umieszczonemu w nawiasie za słowem kluczowym `class`, tworzona w projektancie formularzy klasa `TWinForm` dziedziczy z klasy `System.Windows.Forms.Form`. To właśnie dzięki temu dziedziczeniu w klasie formularza dostępne są wszystkie zdefiniowane w nim właściwości, takie jak wykorzystywana już w niejednym przykładzie właściwość `BackColor`, której można używać w metodach klasy `TWinForm`, tak jakby została ona zdefiniowana bezpośrednio w tej klasie.

Proces wywodzenia klas można powtarzać przez wiele poziomów dziedziczenia, dzięki czemu można tworzyć całe hierarchie klas, w których istnieją klasy ogólne będące klasami bazowymi dla bardziej specjalizowanych klas wywiedzionych. Wszystkie klasy wywiedzione muszą realizować tylko swój własny zestaw funkcji, a w zakresie, w którym istnieje zgodność z działaniami wykonywanymi w klasach bazowych, może wykorzystać funkcje odziedziczone z tych klas. Pierwszym przykładem takiego funkcjonowania hierarchii klas są klasy opisujące działanie kontrolek, zdefiniowane w bibliotece FCL i VCL.NET (proszę przyjrzeć się rysunkowi 2.1).

Dziedziczenie i zgodność przypisywania

Jedną z konsekwencji mechanizmu dziedziczenia jest fakt, że dany obiekt jest zgodny nie tylko z obiektami tej samej klasy, ale może być też wykorzystywany wszędzie tam, gdzie oczekiwane jest podanie obiektu jednej z jego klas-przodków, dokładnie tak, jak to zaprezentowano na listingu 3.32.

Listing 3.32. *Używanie obiektów klasy wywiedzionej w miejsce obiektów klasy bazowej*

```
var
  Object: System.Object; { Object to klasa bazowa dla wszystkich innych klas }
  aComponent: Component; { Component jest klasą wywiedzioną z klasy Object }
  aForm: Form;           { Form jest klasą pośrednio wywiedzioną z klasy Component }
  aButton: Button;       { Klasa Button również wywodzi się z klasy Component }
...
aComponent := aForm;
aComponent := aButton;
Object := aComponent;
```

Formularze i przyciski dziedziczą wszystkie elementy klasy `Component`, dlatego mogą być używane w miejscu obiektów typu `Component` (w powyższym kodzie zapisywane są w zmiennej `aComponent`). Oprócz tego klasa `Component` jest klasą wywiedzioną z klasy `Object`, co oznacza, że może ona wykonać wszystkie operacje, jakie normalnie wykonuje klasa `Object`, w związku z czym kompilator pozwoli również na wykonanie ostatniego przypisania z powyższego listingu. Trzeba przy tym pamiętać, że działania w przeciwnym kierunku nie są dopuszczalne:

```
Form := Object;  { błąd niezgodności typów }
```

Od formularza oczekuje się, że będzie wykonywał wiele szczegółowych zadań, z których jednym jest na przykład wyświetlanie formularza na ekranie. Z klasą `Object` może być powiązany jednak dowolny obiekt, wobec czego nie można zakładać, że będzie on spełniał wszystkie wymagania klasy `Form`. To wszystko oznacza, że do zmiennej typu `Form` przypisać można wyłącznie obiekty klasy `Form` lub klas z niej wywiedzionych.

Wynika z tego, że wywiedzenie jednej klasy z drugiej nie prowadzi wyłącznie do dziedziczenia wszystkich elementów klasy bazowej, ale tworzy też związki pokrewieństwa odgrywające niezwykle istotną rolę w mechanizmie polimorfizmu, o czym za chwilę się przekonamy.

W związku z powyższymi wyjaśnieniami trzeba jeszcze powiedzieć, że zmienne obiektowe przechowują wyłącznie wskazania na rzeczywiste obiekty. W przedstawianej wyżej operacji przypisania kopiowany jest tylko wskaźnik, a sam obiekt nie jest w żaden sposób zmieniany.

Klasy zamknięte

Delphi dla .NET obsługuje też koncepcję klas zamkniętych funkcjonującą w środowisku .NET i w związku z tym wprowadzone zostało specjalne słowo kluczowe `sealed`. Klasa zadeklarowana z tym słowem kluczowym nie może stać się już klasą bazową

dla innej klasy. Przykładem takiej zamkniętej klasy może być klasa `Thread` będąca częścią biblioteki klas środowiska .NET. W języku Object Pascal deklaracja tej klasy wyglądać mogłaby tak jak na listingu 3.33.

Listing 3.33. *Hipotetyczna deklaracja klasy Thread w języku Object Pascal*

```
type
  Thread = class sealed
    procedure Start;
    ...
  end;
  MyThread = class(Thread)  // Błąd: "Zamknięta klasa 'Thread' nie może być już
rozbudowywana"
  end;
```

Deklarowanie klasy jako zamkniętej przydaje się wtedy, gdy dana klasa wewnątrz pewnej biblioteki uznawana jest za niezmienną, co oznacza, że nie powinno się zmieniać jej zachowania poprzez pokrywanie metod w klasach potomnych. Oczywiście podobny efekt uzyskać można, nie deklarując w klasie metod wirtualnych. Deklarowanie klasy jako zamkniętej daje jednak użytkownikowi dodatkową informację. Oznacza to, że rozbudowywanie tej klasy w mechanizmie dziedziczenia nie ma już sensu, wobec czego należy w tym zakresie korzystać z innych metod, na przykład z agregacji, takiej jak pokazana na listingu 3.34.

Listing 3.34. *Rozbudowywanie klasy Therad poprzez agregację*

```
type
  MyThread = class    // Klasa MyThread jest rozszerzeniem klasy Thread; przechowuje ona
obiekt typu Thread
    T: Thread;        // i ewentualnie przekazuje wywołania różnych metod do przechowywanego
obiektu
    procedure Start; // wywołuje metodę T.Start
  end;
```

Wyczerpujący przykład tej metody rozbudowywania klasy `Thread` znaleźć można w punkcie 2.8.2.

3.2.7. Uprzedzające deklaracje klas

W języku Object Pascal identyfikatory mogą być używane dopiero wtedy, gdy zostaną one przedstawione kompilatorowi. Należy przy tym zakładać, że kompilator przeglądał będzie tekst źródłowy tylko raz od początku do końca, w związku z czym na danym etapie „zna" tylko te identyfikatory, które do tej pory znalazł w kodzie programu.

W wielu programach bardzo często zdarza się, że dwie klasy korzystają z siebie nawzajem. Jak w takim razie pierwsza klasa ma być zadeklarowana przed drugą, skoro druga klasa powinna być też zadeklarowana przed pierwszą?

W takich sytuacjach skorzystać należy z uprzedzających deklaracji klas. Jeżeli przykładowo chcielibyśmy poinformować kompilator o istnieniu identyfikatora `TDocument`, to wystarczy użyć następującej deklaracji:

```
type
  TDocument = class;
```

Właściwa deklaracja klasy może zostać zapisana w dalszej części pliku źródłowego. Dzięki takim deklaracjom wzajemne korzystanie z siebie dwóch klas może wyglądać tak jak na listingu 3.35.

Listing 3.35. *Deklaracje dwóch klas korzystających z siebie nawzajem*

```
type
  TDocument = class; { Klasa dokumentu }
  TItem = class      { Klasa jednego z elementów dokumentu }
  { Element musi wiedzieć, w jakim dokumencie się znajduje }
ParentDocument: TDocument;
end;
TDocument = class;
  { Dokument musi znać przynajmniej swój pierwszy element: }
  FirstItem: TItem;
end;
```

3.2.8. Zagnieżdżone deklaracje typów

W bibliotece FCL bardzo łatwo można natknąć się na typy zagnieżdżone, na przykład w opisywanych w punkcie 2.2.5 klasach kolekcji. Kompilator Delphi dla .NET rozpoznaje oczywiście typy zagnieżdżone z biblioteki FCL, a oprócz tego pozwala też na deklarowanie typów zagnieżdżonych wewnątrz programów tworzonych w języku Object Pascal. Klasa `ListViewItemCollection` mogłaby zostać w Delphi zapisana tak jak na listingu 3.36.

Listing 3.36. *Przykład klasy zagnieżdżonej*

```
type
  ListView = class(System.Windows.Forms.Control)
  // klasa zagnieżdżona:
  type ListViewItemCollection = class(IList, ICollection, IEnumerable)
  ... Elementy klasy ListViewItemCollection ...
  end;
  ... Pozostałe elementy klasy ListView ...
end;
```

Po takiej deklaracji klasy, każde użycie typu `ListViewItemCollection` w programie wymaga jawnego wypisania też nazwy klasy zewnętrznej:

```
var
  WskaznikNaKolekcje: ListView.ListViewItemCollection;
```

3.3. Obiekty w czasie działania programu

Zajmiemy się teraz procesami, które w czasie pracy programu mogą zachodzić wewnątrz obiektów. Zaliczyć do nich można inicjalizację wykonywaną przez konstruktory i usuwanie obiektów wykonywane przez destruktory, a także najważniejsze zagadnienie programowania obiektowego: polimorfizm, dzięki któremu dopiero w czasie wykonywania programu podejmowana jest decyzja, która z metod zostanie wywołana (tak zwane późne wiązanie).

3.3.1. Inicjalizacja obiektów: konstruktory

Deklarując zmienną typu jednej z klas uzyskujemy zmienną, która w czasie wykonywania programu może wskazywać na pewien obiekt, ale początkowo inicjowana jest przez CLR wartością `nil` (co oznacza mniej więcej tyle co „ta zmienna nie wskazuje na żaden obiekt"):

```
var
  MojPrzycisk: Button;  // CLR automatycznie inicjuje zmienną wartością nil
```

Próbując skorzystać z takiej zainicjowanej zmiennej spowodujemy wywołanie błędu czasu wykonania. Stan ten można oczywiście zmienić, przypisując do zmiennej istniejący już obiekt typu `Button`:

```
MojPrzycisk := PewienFormularz.Button2;
```

Po wykonaniu tej operacji przypisania zmienne `MojPrzycisk` i `PewienFormularz.Button2` będą wskazywały na ten sam obiekt typu `Button`.

Często konieczne jest jednak samodzielne przygotowanie nowego obiektu i do tego właśnie potrzebne są nam konstruktory. W czasie wywoływania konstruktora wykonywane jest przynajmniej rezerwowanie pamięci dla obiektu, ponieważ wszystkie obiekty o typie deklarowanym za pomocą słowa kluczowego `class` są w środowisku CLR automatycznie tworzone na stercie. Poza tym, w konstruktorze klasa ma okazję przypisać właściwe wartości początkowe zmiennym tworzonego obiektu oraz zarezerwować zasoby potrzebne w czasie późniejszej pracy obiektu. Typowo konstruktory obiektów otrzymują nazwę `Create`.

Wywoływanie konstruktorów

Zadaniem konstruktorów jest nie tylko inicjowanie obiektu, ale i samo jego utworzenie (czyli zarezerwowanie dla niego pamięci). Na przykład, chcąc utworzyć obiekt typu `DynamicForm` nie można zastosować wywołania przedstawionego na listingu 3.37.

Listing 3.37. *Niewłaściwa próba utworzenia nowego obiektu*

```
var
  DynamicForm: Form;
begin
   DynamicForm.Create; { obiekt nie zostanie utworzony! }
```

Przedstawiona w powyższym listingu instrukcja wywołuje konstruktor `Create` w połączeniu z obiektem `DynamicForm`, który jeszcze nie istnieje. Chcąc utworzyć obiekt, musimy **przypisać** do zmiennej obiektu (tutaj `DynamicForm`) całkiem nowy obiekt. Nowy obiekt tworzony jest przez wywołania konstruktora na rzecz klasy, a nie na rzecz zmiennej obiektu. Wywołanie metody `Form.Create` zarezerwuje pamięć na nowy obiekt dynamiczny, zainicjuje go i zwróci wskazanie na niego, dzięki czemu będzie można je przypisać do zmiennej obiektu:

```
DynamicForm := Form.Create;
```

Kontrolki i formularze definiowane w bibliotece Windows-Forms najczęściej mają konstruktory bezparametrowe. W bibliotece VCL.NET formularze i komponenty oczekują natomiast podania w parametrze konstruktora komponentu-właściciela.

Tworzenie własnych konstruktorów

Wewnątrz własnych klas konstruktory deklarować można tak jak i inne metody, ale wyróżniać je należy słowem kluczowym `constructor`. Przykład deklaracji konstruktora w klasie przedstawiam na listingu 3.38.

Listing 3.38. *Przykładowa deklaracja konstruktora wewnątrz deklaracji klasy*

```
type
  TGraphicElement = class(TPersistent)
    constructor Create(InitRect: TRect);
      { Nazwa "Create" nie jest obowiązkowa }
```

W powyższym przykładzie konstruktor klasy oczekiwał będzie podania w parametrze prostokąta, którego współrzędne wykorzystane zostaną do inicjalizacji (nieprzedstawionych na listingu) danych obiektu.

W celu inicjalizowania formularzy w bibliotece Windows-Forms wystarczy dopisywać instrukcje do ciała konstruktora klasy formularza przygotowanego przez Delphi. W przypadku biblioteki VCL.NET najczęściej wystarcza obsłużenie zdarzenia `OnCreate`.

Wewnątrz konstruktora klasy bardzo często konieczne jest wywołanie konstruktora klasy bazowej, który musi wykonać swoją część inicjalizacji klasy. Właściwy sposób wywołania konstruktora klasy bazowej przedstawiam na listingu 3.39.

Listing 3.39. *Wywoływanie konstruktora klasy bazowej*

```
constructor  TGraphicElement.Create(InitRect: TRect);
begin
  inherited Create;   // Do odziedziczonego konstruktora nie trzeba przekazywać żadnych
parametrów
  ...
```

Jeżeli konstruktor klasy bazowej pobiera jakieś parametry, to dobrym rozwiązaniem jest przygotowanie w nowej klasie deklaracji konstruktora również pobierającego te same parametry. Dzięki temu można przekazać te parametry do konstruktora odziedziczonego.

Ostrzeżenie dla osób znających Borland Pascal

W starszych wersjach języka Pascal przygotowywanych przez formę Borland, do wywoływania odziedziczonego konstruktora zamiast instrukcji `inherited Create` można też stosować instrukcję `TPersistent.Create`, nie tworząc jednocześnie nowego obiektu. W Delphi wywołanie to spowodowałoby utworzenie nowego obiektu klasy `TPersistent`, który jednak nie mógłby być do niczego wykorzystany, ponieważ wskazanie na utworzony obiekt nie zostałoby przypisane do żadnej zmiennej. To wszystko oznacza, że w Delphi **musimy** korzystać w tym zakresie ze słowa kluczowego `inherited`.

3.3.2. Zwalnianie zasobów i czyszczenie pamięci

We wszystkich programach przeciwieństwem konstruktorów są destruktory. Każdy obiekt, który tworzony jest w czasie działania programu i któremu przydzielana jest pamięć, musi w pewnym momencie zostać z tej pamięci usunięty.

Zasada działania mechanizmu oczyszczania pamięci

W środowisku .NET w tle cały czas działa proces oczyszczania pamięci (ang. *Garbage Collector*), automatycznie usuwający z pamięci wszystkie obiekty, które nie są już używane przez żadną część programu. Dzięki temu w sytuacji idealnej moglibyśmy tworzyć dowolne obiekty i w ogóle nie przejmować się ich zwalnianiem. Tak właśnie w punkcie 2.2.4 przygotowany został dynamicznie formularz i wyświetlony wywołaniem metody `ShowDialog` (odpowiedni kod przedstawiam na listingu 3.40).

Listing 3.40. *Dynamiczne tworzenie i wyświetlanie okna dialogowego*

```
procedure TWinForm.Button3_Click(sender: System.Object; e: System.EventArgs);
var
  F: Form;
begin
  F := Form.Create;
  ... Przygotowywanie właściwości formularza ...
  ... Tworzenie i dodawanie elementów formularza ...
  F.ShowDialog;  // Wyświetlanie formularza jako okna dialogowego
end;
```

Środowisko CLR rozpoznaje teraz zakończenie metody i dzięki temu wie, że po słowie kluczowym `end` wszystkie **zmienne** lokalne przestaną istnieć. W środowisku CLR z każdym utworzonym **obiektem** powiązany jest licznik określający liczbę istniejących jeszcze wskazań na obiekt. W przedstawionym wyżej kodzie licznik ten inicjowany jest wartością `1`, zaraz po zapisywaniu wskazania na ten obiekt do zmiennej `F`. W momencie, gdy wykonywanie metody dojdzie do słowa kluczowego `end`, zmienna ta przestanie istnieć, a w związku z tym środowisko CLR zmniejsza wartość licznika obiektu na zero. Zerowa wartość licznika oznacza, że obiekt jest **oznaczony** do automatycznego usunięcia z pamięci.

Możemy też założyć, że w trakcie działania procedury zawartość zmiennej `F` przekazywana jest też do innego obiektu, tak jak to pokazano na listingu 3.41.

Listing 3.41. *Przekazanie wskazania na obiekt do innej zmiennej*

```
var
  ZF: ZbiorFormularzy;
begin
  ...
  ZbiorFormularzy.ZapiszFormularz(ZF);
```

Załóżmy, że metoda `ZapiszFormularz` zachowuje przekazany jej w parametrze formularz (na przykład wewnątrz pewnej kolekcji), w wyniku czego CLR automatycznie powiększa wartość licznika obiektu tego formularza do wartości `2`. Jeżeli teraz zmienna `F` przestanie istnieć, to wartość licznika spadnie znowu do jedynki, ale tym razem nie osiągając zera, przez co obiekt formularza nie zostanie **oznaczony** do usunięcia z pamięci.

W przypadku zwalniania obiektu (a mówiąc dokładniej, zmniejszania jego licznika wskazań), który przechowuje w sobie inne obiekty, procedura zwalniania pamięci jest bardzo podobna do tej stosowanej przy zakończeniu działania metody lokalnie rezerwującej pamięć dla obiektów. Oznacza to, że w przypadku zwalniania formularza tak jak w powyższym przykładzie, razem z formularzem usuwane są z pamięci wszystkie kontrolki tego formularza (oczywiście pod warunkiem, że nie są one wskazywane z innych miejsc w kodzie programu).

Środowisko CLR ma pełną kontrolę nad wszystkimi wskazaniami wewnątrz kodu zarządzanego, dlatego można z czystym sumieniem zakładać, że zawsze będzie wiedziało, które obiekty mogą być już bezpiecznie usunięte z pamięci, a które nie[2].

Po co w takim razie destruktory?

Co prawda mechanizm oczyszczania pamięci jest bardzo skutecznym remedium na problemy ze znikaniem wolnej pamięci spowodowane przez źle napisane programy, które nie zwalniają nieużywanych już obiektów (tak zwane wycieki pamięci — ang. *Memory leaks*), ale nie spełni wszystkich życzeń programisty dotyczących operacji jakie mają być wykonywane w momencie zwalniania obiektów:

- Po pierwsze, może się zdarzyć, że zasoby używane przez obiekt muszą być zwolnione wcześniej, niż zrobiłby to mechanizm oczyszczania pamięci. Takim przykładem mogą być pliki, które powinny być zamykane w momencie, gdy nie są już używane przez program. Co prawda zamknięcie nieużywanego pliku nastąpi automatycznie w momencie usuwania z pamięci obiektu `FileStream`, ale będzie to wymagało pewnego oczekiwania, zanim inna aplikacja będzie mogła uzyskać dostęp do tego pliku. W takim wypadku konieczne jest ręczne zwolnienie zasobów, które w przypadku obiektów `FileStream` realizowane jest przez wywołanie metody `Close`.
- Po drugie, bywa też tak, że przy zwalnianiu obiektu nie tylko nastąpić ma zwolnienie zajmowanej przez niego pamięci, ale wykonane mają być też inne operacje, na przykład zapisanie do pliku statystyk zbieranych w czasie całego

[2] Jeżeli jednak w grę wchodzi też kod nieobsługiwany, który otrzymuje wskazanie na obiekt zarządzany, to nie można zagwarantować prawidłowego działania mechanizmu oczyszczania pamięci.

czasu życia obiektu albo zachowanie w rejestrze lub pliku konfiguracyjnym zmienionych ustawień dotyczących konfiguracji programu. W takiej sytuacji konieczna jest możliwość wykonania dodatkowych operacji w momencie, gdy mechanizm oczyszczania pamięci przystępuje do zwolnienia pamięci zajmowanej przez dany obiekt.

W środowisku .NET można wykonywać obie te operacje. Pierwszy z wymienionych przypadków — ręczna możliwość zwolnienia zasobów — jest dość oczywisty, zważywszy fakt, że metodę `Close` możemy samodzielnie zdefiniować w każdej klasie. Możliwość ta jest w środowisku .NET uzupełniana przez interfejs `IDisposable`, za pomocą którego takie zwolnienia zajętych zasobów można przeprowadzać w sposób standaryzowany. Przy automatycznym zwalnianiu obiektu CLR wywołuje metodę `Finalize` tego obiektu tuż przed faktycznym zwolnieniem zajmowanej przez niego pamięci. Metoda `Finalize` pod wieloma względami odpowiada destruktorom z wielu języków programowania, choć na poziomie CLR pojęcie destruktora nie zostało zdefiniowane.

Obie te cechy — metoda `Finalize` i wzorzec `IDispose` — dostępne są również w Delphi, a na dodatek firma Borland przeniosła do świata .NET destruktory jako cechę języka programowania. Osoby znające język C# albo mające zamiar nauczyć się go w przyszłości muszę tutaj ostrzec przed niebezpieczeństwem popełnienia pomyłki: destruktory z języka C# są dokładnym odpowiednikiem metody `Finalize` funkcjonującej na poziomie CLR. Oznacza to, że destruktory języka C# w Delphi można realizować wyłącznie poprzez jawne przygotowanie metody `Finalize`. Z kolei działanie faktycznego destruktora Delphi w języku C# odtworzyć można wyłącznie poprzez ręczną implementację wzorca `IDispose`. W tabeli 3.1 przedstawiam podsumowanie wszystkich tych różnic i jednocześnie zaznaczam, że w Delphi dostępne są dwa warianty destruktorów, które opiszę dokładnie w dalszej części podrozdziału.

Tabela 3.1. *Rodzaje destruktorów*

	Poziom CLR	**... w języku C# odpowiada**	**... wariant destruktora w Delphi**	**... wariant „ręczny" w Delphi**
Wywoływanie automatyczne	`Finalize`	Destruktor	`Finalize`	`Finalize`
Wywoływanie ręczne	Wzorzec `IDispose`	Wzorzec `IDispose`	`Free`; `Dispose`; Destruktor	Wzorzec `IDispose`

Destruktory w Delphi

Tworząc samodzielnie destruktor, należy deklarację metodę rozpocząć od słowa kluczowego `destructor`. W Delphi dla .NET wszystkie tworzone destruktory muszą nazywać się `Destroy`, nie mogą przyjmować żadnych parametrów, a w deklaracji klasy muszą być oznaczane dyrektywą `override`. W każdym wypadku, ostatnią instrukcją w kodzie destruktora powinno być wywołanie destruktora klasy bazowej, tak jak na listingu 3.42.

Listing 3.42. *Kod przykładowego destruktora*

```
// Deklaracja destruktora w klasie
destructor TGraphicElement.Destroy; override;

// Implementacja destruktora:
destructor TGraphicElement.Destroy;
begin
  ...
  inherited Destroy;
end;
```

Firma Borland zezwoliła na stosowanie tej składni w Delphi dla .NET, ponieważ istniejący już kod przygotowany w Delphi bardzo często wykorzystuje takie właśnie destruktory. Kompilator Delphi automatycznie przekłada ten wzorzec destruktorów na wzorzec `IDisposable` stosowany w środowisku .NET:

- W skompilowanym kompilacie co prawda pojawia się metoda `Destroy`, która jednak stosowana jest do implementowania wymaganej przez interfejs `IDisposable` metody `Dispose`. Jeżeli nasz obiekt zostanie przekazany obcemu obiektowi, który właściwie posługuje się interfejsem `IDispose`, to obiekt ten może wywołać metodę `Dispose` naszego obiektu, co doprowadzi do wywołania przygotowanego przez nas destruktora.
- Chcąc spowodować zwolnienie zasobów, nie należy bezpośrednio wywoływać metody `Destroy`, ale skorzystać z automatycznie przygotowywanej przez kompilator metody `Free`. Metoda ta sprawdza wartość — również przygotowywanej automatycznie przez kompilator — zmiennej `Disposed`, przez co zapobiega wielokrotnemu wywoływaniu metody `Destroy`.

Wykorzystanie tak przygotowanych obiektów wyglądać powinno tak jak na listingu 3.43.

Listing 3.43. *Sposób wykorzystania obiektu wyposażonego w destruktor*

```
var
  Obiekt : TGraphicElement;
begin
  Obiekt := TGraphicElement.Create;  // Konstruowanie obiektu
  ... używanie obiektu ...
  Obiekt.Free;  // Zwalnianie zasobów obiektu
  // Zwolnienie pamięci realizowane przez mechanizm oczyszczania pamięci
  // następuje później automatycznie, ale może być też wymuszone ręcznie:
  // GC.Collect;
  // GC.WaitForPendingFinalizers;
```

Mimo że interfejs `IDispose` definiowany jest w samym środowisku .NET, to jednak metoda `Dispose` nie jest automatycznie wywoływana w momencie usuwania obiektu z pamięci.

Ręczna implementacja interfejsu IDispose

Interfejs `IDispose` można też implementować ręcznie, a kod, który normalnie umieszczony byłby w destruktorze, przenieść do metody `Dispose`. W takim rozwiązaniu kompilator nie pozwoli już na stosowanie w danym obiekcie destruktora.

W dokumentacji środowiska .NET opisany został pewien wzorzec zastosowania interfejsu `IDispose`, w którym ten sam kod wykonywany jest zarówno przy ręcznym zwalnianiu obiektu, jak również przy jego automatycznej finalizacji. W tym celu należy pokryć metodę `Finalize` — jedyną metodę, która wykonywana jest przy automatycznej finalizacji obiektu w CLR — i ręcznie przekazywać w niej kontrolę do metody `Dispose`.

Metoda `Dispose` musi być przygotowana na dwie ewentualności: wywołania ręcznego albo automatycznego wywołania w czasie finalizacji. W tym celu przygotowana musi być specjalna wersja metody `Dispose`, w której oba te przypadki rozróżniane są za pomocą parametru logicznego. Standardowa, wywoływana automatycznie metoda `Dispose` nie ma żadnego parametru, dlatego wywołuje swoją przeciążoną wersję przekazując jej w parametrze wartość `True`, natomiast metoda `Finalize` wywołuje tę samą metodę z parametrem `False`.

Listing 3.44. *Ujednolicenie kodu dla ręcznego i automatycznego zwalniania obiektu*

```
// Deklaracja w części interfejsu modułu:

TestClass = class (TInterfacedObject, IDisposable)
  private
    Disposed: Boolean;
  public
    procedure Dispose; overload;
    procedure Dispose(ExplicitCall: Boolean); overload;
  strict protected
    procedure Finalize; override;
  end;

// Implementacja w części implementacji modułu:

procedure TestClass.Finalize;
begin
  inherited;
  Dispose(False);
end;

procedure TestClass.Dispose;
begin
  inherited;
  Dispose(true);
  // Metoda Dispose została właśnie wywołana ręcznie,
  // dlatego metoda Finalize nie musi być już wywoływana
  // przy zwalnianiu obiektu
  GC.SuppressFinalize(self);
end;

procedure TestClass.Dispose(ExplicitCall: Boolean);
begin
```

```
  if not Disposed then begin
    if ExplicitCall then begin
      // Przy wywołaniu bezpośrednim wywołujący chciałby,
      // żeby wszystkie zasoby obiektu zwolnione zostały jeszcze
      // przed automatyczną finalizacją obiektu. Będzie to możliwe
      // tylko wtedy, gdy zwolnione zostaną też wszystkie zasoby zarządzane,
      // używane w innych obiektach powiązanych z naszym obiektem.
      // Na przykład:
      // MojKomponent.Dispose;
    end;
    // W każdym wypadku (również w czasie finalizacji) w tym miejscu
    // można zwalniać też wszystkie zasoby niezarządzane.
    // Przykład: PlikUżytkownika.Close;
  end;
  Disposed := True;  // zabezpieczenie przed podwójnym wywołaniem
  // W klasie wywiedzionej z klasy TestClass zamiast instrukcji
  // Disposed := True należy wywołać odziedziczoną wersję
  // metody, stosując przy tym słowo kluczowe inherited.
end;
```

Zastosowanie tej klasy musi w takim razie wyglądać tak jak na listingu 3.45.

Listing 3.45. *Zastosowanie klasy TestClass*

```
var
  obiekt: TestClass;
begin
  obiekt := TestClass.Create;
  ... Użytkowanie klasy ...
  // Zwolnienie zasobów klasy (bez zwalniania pamięci)
  obiekt.Dispose;  // Ręczne zwolnienie
```

Zwolnienie pamięci zajmowanej przez obiekt nastąpi automatycznie w czasie nieokreślonym po wywołaniu metody `Dispose`, kiedy mechanizm oczyszczania pamięci znajdzie czas na wykonanie tej operacji albo zostanie ona ręcznie wymuszona wywołaniem metody `GC.Collect`. Najważniejsze jest tutaj to, że dzięki ręcznemu wywołaniu metody `Dispose` nie ma już potrzeby wykonywania **automatycznej finalizacji** obiektu (dzięki wywołaniu metody `SuppressFinalize` z wnętrza metody `TestClass.Dispose`), co oszczędza mechanizmowi oczyszczania pamięci części prac związanych z zarządzaniem obiektem.

Przedstawiony wyżej wycinek kodu, połączony z inną klasą bazową realizującą tę samą koncepcję za pomocą stosowanych w Delphi destruktorów, można znaleźć na płycie CD dołączonej do książki, w projekcie *GCDisposeVariants*.

Jeżeli w klasie przygotowujemy destruktora i jednocześnie pozwalamy kompilatorowi automatycznie wygenerować interfejs `IDispose`, to tracimy możliwość ręcznego wywołania metody `Dispose` tak jak pokazano na powyższym listingu. Konieczne jest tutaj wykonanie dodatkowej konwersji typów: `(obiekt as IDisposable).Dispose`. (Mała uwaga teoretyczna: Jeżeli przygotowywać będziemy destruktor zgodny ze starą tradycją Delphi, to najprawdopodobniej będziemy wywoływać go zgodnie z tą tradycją, czyli za pomocą metody `Free`).

Metoda Dispose dla formularzy

Przedstawiony wyżej wzorzec częściowo implementowany jest również w kodzie formularzy Windows-Forms automatycznie generowanym przez Delphi. Każdy formularz otrzymuje specjalną metodę `Dispose` pobierającą parametr `Disposing`, będący odpowiednikiem przedstawionego wyżej parametru `ExplicitCall`. Kod takiej metody przedstawiam na listingu 3.46.

Listing 3.46. *Kod procedury Dispose formularza*

```
procedure TWinForm1.Dispose(Disposing: Boolean);
begin
  if Disposing then
  begin
    if Components <> nil then
      Components.Dispose();
  end;
  inherited Dispose(Disposing);
end;
```

W przypadku, gdy parametr `Disposing` ma wartość `True`, metoda `Dispose` wywołuje metody `Dispose` wszystkich komponentów znajdujących się na formularzu. Takie działanie ma znaczenie tylko w przypadku, gdy formularz ma być aktywny w czasie wyświetlania go w projektancie formularzy, ponieważ w czasie działania programu lista komponentów zapisana we właściwości `Components` nie jest używana i w związku z tym jest całkiem pusta[3].

Metoda Finalize w Delphi

Jak już wspominałem, metoda `Finalize` jest jedyną metodą, która jest standardowo wywoływana przez mechanizm oczyszczania pamięci. Przedstawiony na listingu 3.44 kod jest przykładem pokrywania w klasie metody `Finalize` i przedstawia wzorzec pozwalający na wykonanie tego samego kodu niezależnie do tego, czy automatycznie została wywołania metoda `Finalize`, czy też nastąpiło ręczne wywołanie metody `Free`.

Jeżeli nasza klasa nie potrzebuje stosowania metod ręcznego zwalniania zasobów i chcemy oprogramować wyłącznie operacje zwalniania automatycznego wykonywane w czasie finalizowania obiektu, to metody `Finalize` można też używać całkowicie niezależnie od istniejącej implementacji interfejsu `IDisposable` lub przygotowanego destruktora. Trzeba przy tym przestrzegać tylko jednej zasady mówiącej, że metoda `Finalize` nie może tworzyć żadnych nowych obiektów.

Wewnętrzne działanie finalizacji obiektów

Metody `Finalize` powodują opóźnienia w działaniu mechanizmu oczyszczania pamięci. Po pierwsze, zmuszają go do dodatkowego przejrzenia wszystkich swoich wewnętrznych list obiektów: jeżeli obiekt posiadający metodę `Finalize` nie jest już nigdzie

[3] Kiedy kod formularza jest aktywny już w czasie projektowania? Tylko wtedy, gdy inny formularz zostaje z niego wywiedziony, co mieliśmy okazję obserwować w punkcie 2.1.4.

używany (czyli jego licznik wskazań ma wartość zero), to początkowo dopisywany jest do listy obiektów do finalizacji. W systemie od czasu do czasu wykonywane jest oczyszczanie pamięci i wtedy dla każdego obiektu z tej listy wywoływana jest metoda `Finalize`, a obiekty przekazywane są do listy obiektów przeznaczonych do zwolnienia. Wszystkie obiekty usuwane są z pamięci dopiero po przejrzeniu przez mechanizm oczyszczania pamięci tej drugiej listy.

Do tego wszystkiego dochodzi jeszcze taki problem, że mechanizm oczyszczania pamięci nie może zwolnić pamięci tych obiektów, które wskazywane są przez obiekt wykonujący metodę `Finalize`, ponieważ przez cały czas wykonywania tej metoda może się ona odwoływać do tych obiektów.

W metodzie finalizującej jeden z obiektów można odwoływać się też do innych obiektów wykorzystywanych przez ten obiekt, ponieważ ich zwolnienie nastąpi najwcześniej **po** zakończeniu metody `Finalize` danego obiektu. Trzeba się jednak liczyć z tym, że dla tych istniejących jeszcze obiektów wywołana mogła być już metoda `Finalize`, przez co nie będziemy mieli pełnej możliwości korzystania ze wszystkich funkcji tych obiektów. Kolejność wywoływania metod `Finalize` w grupie obiektów nie została nigdzie zdefiniowana i może zmieniać się przy poszczególnych uruchomieniach programu, a także w zależności od wersji stosowanego środowiska CLR. Oznacza to, że w czasie tworzenia metod `Finalize` nie możemy zakładać konkretnej kolejności wywołań tych metod.

Zwalnianie zmiennych obiektu

Przedstawiane w dotychczasowych przykładach wywołania metod `Free` i `Dispose` zakładały, że zwalniane obiekty przechowywane są w lokalnych zmiennych metody. Zmienne lokalne przestają istnieć wraz z zakończeniem pracy metody, dlatego nie trzeba było się w tym przypadku martwić o ewentualne próby wielokrotnego zwolnienia tego samego obiektu.

W przypadku obiektów zapisanych w zmiennej danej klasy może się w pewnych sytuacjach zdarzyć, że chcielibyśmy zwolnić zapisany w nich obiekt i jednocześnie zaznaczyć, że w zmiennej nie ma już żadnego obiektu. W takim wypadku, po zwolnieniu obiektu należy do przechowującej go zmiennej przypisać wartość `nil` (tak jak na listingu 3.47), która oznaczać będzie, że „tu nie ma żadnego obiektu".

Listing 3.47. *Zwalnianie obiektu i oznaczanie zmiennej jako „pustej"*

```
// Jeżeli stosowana jest składania destruktora:
DynamicObject.Free;
// A jeżeli stosowany jest interfejs IDisposable:
// DynamicObject.Dispose;
DynamicObject := nil;
```

Po takim oznaczeniu obiektu możemy w każdej chwili sprawdzić, czy do zmiennej nadal przypisany jest jakiś obiekt, wywołując w tym celu funkcję `Assigned`. Funkcja ta zwraca wartość `False`, jeżeli podana zmienna przechowuje wartość `nil`. Dzięki temu można uniknąć pomyłkowego wywoływania metod zwolnionego już obiektu:

```
if Assigned(DynamicObject)
  then DynamicObject.ZrobCos;
```

Zwalnianie obiektów częściowo zainicjowanych

Jeżeli w trakcie wykonywania konstruktora wystąpi jakiś wyjątek, to wykonanie tego konstruktora jest automatycznie przerywane. Jeżeli przechwycimy ten wyjątek i pozwolimy dalej pracować programowi, to w którymś momencie trzeba będzie zwolnić pamięć zajmowaną przez tak częściowo zainicjowany obiekt. Oznacza to, że metody `Finalize` i `Dispose`, a także destruktory powinny być przygotowane na to, że konstruktor może nie wykonać do końca swojej pracy, w wyniku czego obiekt będzie tylko częściowo zainicjowany. W kodzie przedstawionym na listingu 3.48 zasada ta nie jest przestrzegana, więc w pewnych sytuacjach w programie będą pojawiać się błędy.

Listing 3.48. *Metoda Finalize nieuwzględniająca możliwości wystąpienia błędów w czasie działania konstruktora*

```
constructor DemoObject.Create;
begin
  inherited Create;
  File1 := OpenFile1;  // Tutaj może wystąpić wyjątek,
  File2 := OpenFile2;  //  a wtedy zmienna File2 nie zostanie zainicjowana
end;

procedure DemoObject.Finalize;
begin
  File1.Close;
  File2.Close;  // Tutaj zmienna File2 może mieć wartość nil!
  inherited Destroy;
end;
```

Jeżeli w metodzie `Create` w czasie działania metody `OpenFile1` (albo wcześniej) wystąpi jakikolwiek wyjątek, to w czasie działania destruktora zmienne `File1` i `File2` nie będą zainicjowane. Wynika z tego, że kod finalizacji tego obiektu będzie bezpieczny tylko wtedy, gdy będzie wyglądał tak jak na listingu 3.49.

Listing 3.49. *Prawidłowa postać kodu destruktora obiektu DemoObject*

```
if Assigned(File2) then
  File2.Close;
if Assigned(File1) then
  File1.Close;
```

3.3.3. Metody wirtualne

W czasie prostego programowania formularzy nie trzeba sobie zawracać głowy metodami wirtualnymi, ale pamiętać należy o tym, że metody wirtualne są jednym z najważniejszych elementów programowania zorientowanego obiektowo, którego najczęściej nie da się pominąć tam, gdzie w grę wchodzi też mechanizm dziedziczenia klas. Solidna wiedza o dziedziczeniu i metodach wirtualnych jest nieodzowna na przykład w czasie przygotowywania kodu nowych kontrolek i komponentów.

Przykład motywacyjny

Standardowo wszystkie metody są niewirtualne, a metodami wirtualnymi stają się dopiero po zastosowaniu wobec nich dyrektywy `virtual` lub `override`. W poniższym przykładzie chcę odpowiedzieć na rodzące się tu pytanie: do czego w ogóle potrzebne są metody wirtualne? Załóżmy, że w programie definiujemy kilka klas, które wszystkie mają w sobie metodę `Pracuj`, ale w każdej klasie metoda ta wykonywać będzie całkowicie inne operacje. Przykładowy kod takich klas przedstawiam na listingu 3.50.

Listing 3.50. *Przykładowa deklaracja klas zawierających metodę o takiej samej nazwie*

```
type
  KlasaAbstrakcyjna = class
    // Konstruktor Create dziedziczony jest z klasy System.Object
    procedure Pracuj;
  end;
  Klasa1 = class (KlasaAbstrakcyjna) procedure Pracuj; end;
  Klasa2 = class (KlasaAbstrakcyjna) procedure Pracuj; end;
  ...
  Klasa10 = class (KlasaAbstrakcyjna) procedure Pracuj; end;
```

Teraz możemy pozwolić użytkownikowi aplikacji zadecydować, której klasy będzie chciał użyć. W zależności od wyboru dokonanego przez użytkownika, dynamicznie tworzymy obiekt odpowiedniej klasy i przypisujemy go do zmiennej `ObiektRoboczy`, tak jak na listingu 3.51.

Listing 3.51. *Tworzenie obiektu na podstawie wyboru dokonanego przez użytkownika*

```
var
  ObiektRoboczy: KlasaAbstrakcyjna;
begin
  case WyborUzytkownika of
    { "WyboremUzytkownika" może być na przykład wartość (Sender as TButton).Tag }
    Button1: ObiektRoboczy := Klasa1.Create;
    Button2: ObiektRoboczy := Klasa2.Create;
...
    Button10: ObiektRoboczy := Klasa10.Create;
```

Zakładamy teraz, że operację związaną z wybraną przez użytkownika klasą wykonać chcemy w zupełnie innym miejscu w programie:

```
ObiektRoboczy.Pracuj;
```

I jak teraz kompilator na podstawie takiego wywołania metody ma się dowiedzieć, którą z metod `Pracuj` ma w danym momencie wywołać? Nie może przecież przewidywać, jakiej klasy będzie obiekt przypisany do zmiennej `ObiektRoboczy`. W kodzie poinformowaliśmy go tylko o tym, że zmienna `ObiektRoboczy` zadeklarowana jest z jako zmienna klasy `KlasaAbstrakcyjna`. W związku z powyższym kompilator na stałe powiąże to wywołanie — z braku metod wirtualnych — z metodą `KlasaAbstrakcyjna.Pracuj`. Oznacza to, że metody `Pracuj` zdefiniowane w klasach wywiedzionych nie będą **nigdy** wywoływane.

Metody wirtualne

Przedstawione w powyższym przykładzie metody niewirtualne niszczą cały sens stosowania dziedziczenia klas, który to mechanizm ma przede wszystkim umożliwiać zmianę w klasach wywiedzionych zachowania metod zdefiniowanych w klasach bazowych. Rozwiązaniem tego problemu jest zrezygnowanie z trwałego zapisywania w kodzie programu metody `KlasaAbstrakcyjna.Pracuj` (takie rozwiązanie nazywa się *wczesnym dowiązaniem*), na rzecz określania wywoływanej implementacji metody dopiero w czasie pracy programu (*późne dowiązanie*).

Dokładnie tak zachowują się metody *wirtualne*. W kodzie przedstawionego wyżej przykładu przełączenie stosowanej metody dowiązywania z wczesnej na późną wymaga tylko dopisania za nagłówkiem metody `Pracuj` w klasie bazowej słowa kluczowego `virtual`, a za nagłówkami tej samej metody we wszystkich klasach wywiedzionych słowa kluczowego `override`, tak jak to pokazano na listingu 3.52.

Listing 3.52. *Poprawiona deklaracja klas włączająca metody wirtualne*

```
type
  KlasaAbstrakcyjna = class
  { Konstruktor Create dziedziczony jest z klasy System.Object }
    procedure Pracuj; virtual;
  end;
  Klasa1 = class (KlasaAbstrakcyjna)
    procedure Pracuj; override;
  end;
  ...
  ... pozostały kod - jak wyżej ...
  ...
  ObiektRoboczy.Pracuj;  { zawsze wywoływana jest właściwa metoda }
```

Teraz, w zależności od tego, który obiekt zostanie przypisany do zmiennej `ObiektRoboczy` przez instrukcję `case`, wywołanie `ObiektRoboczy.Pracuj` spowoduje uruchomienie implementacji metody `Klasa1.Pracuj`, `Klasa2.Pracuj` itd. W podanym kodzie w ogóle nie jest natomiast wykorzystywana implementacja `KlasaAbstrakcyjna. Pracuj`, dlatego mogłaby ona być zadeklarowana jako abstrakcyjna, tak jak zrobimy to z metodą w jednym z dalszych przykładów.

W tekście źródłowym biblioteki VCL od czasu do czasu można znaleźć deklaracje metod wykorzystujące słowo kluczowe `dynamic`. W tych miejscach chodzi o specjalny wariant metody wirtualnej, którą w kodzie źródłowym programu wykorzystuje się dokładnie tak samo jak wszystkie opisywane wyżej zwyczajne metody wirtualne. Jedyna różnica pomiędzy tymi dwoma rodzajami metod polega na ich nieco innym wewnętrznym działaniu w środowisku Win32. W kompilatorze dla Win32 metody dynamiczne są optymalizowane pod względem wielkości kodu, a nie tak jak zwyczajne metody wirtualne — pod względem prędkości działania. Kompilator dla .NET metody dynamiczne traktuje dokładnie tak samo jak zwyczajne metody wirtualne, ponieważ w środowisku .NET w ogóle nie funkcjonuje pojęcie metody „dynamicznej".

Override

Proces ponownego definiowania odziedziczonej metody wirtualnej nazywany jest *pokrywaniem* tej metody. W tym celu niezbędne jest zastosowanie w definicji metody słowa kluczowego `override`. Jeżeli dyrektywa ta zostałaby opuszczona, to kompilator utworzy samodzielną metodę, która nie ma nic wspólnego z metodą odziedziczoną z poprzedniego przykładu (w którym zmienna `ObiektRoboczy` zadeklarowana jest z typem klasy abstrakcyjnej) i w związku z tym nie zostanie wywołana. Z punktu widzenia nowej klasy nowa metoda o takiej samej nazwie, niedeklarowana ze słowem kluczowym `override` zasłania całkowicie metodę odziedziczoną.

Wskazówka dla użytkowników języka Borland Pascal

Trzeba tu też zaznaczyć, że w języku Object Pascal nie istniała jeszcze dyrektywa `override`, a metody wirtualne w klasach wywiedzionych **zawsze** pokrywane były metodami o takich samych nazwach. Z tego powodu Delphi standardowo generuje ostrzeżenie, jeżeli odziedziczona metoda w klasie definiowana jest bez dyrektywy `override`, czyli jest zasłaniana.

Nie należy też mylić dyrektywy `override` z dyrektywą `overload`. Ta druga stosowana jest przy definiowaniu kilku metod o takiej samej nazwie, które mają być używane alternatywnie, i nie ma nic wspólnego z dziedziczeniem klas.

Reintroduce

Oprócz tego w Delphi dostępna jest jeszcze dyrektywa `reintroduce`, umożliwiająca zasłonięcie odziedziczonej metody podobnie do sytuacji opisanej przed chwilą, ale blokująca wypisywanie ostrzeżenia przez kompilator. Wystarczy tylko za deklaracją metody w klasie umieścić słowo kluczowe `reintroduce`.

Wynika z tego, że jeżeli w klasie wywiedzionej definiujemy też metodę, której nazwa jest zgodna z nazwą metody odziedziczonej to koniecznie musimy w jej deklaracji zastosować jedno ze słów kluczowych `override` lub `reintroduce`, dzięki czemu kompilator nie będzie generował komunikatów ostrzeżeń, a i dla człowieka łatwiejsze będzie czytanie kodu takiej klasy. Trzeba jednak pamiętać, że dyrektywę `override` stosować można tylko wtedy, gdy lista parametrów i typ zwracanej wartości nowej metody są zgodne z tymi samymi danymi o metodzie odziedziczonej. Ograniczenie to nie dotyczy dyrektywy `reintroduce`.

Implementowanie pokrywanych metod

Implementując metodę wirtualną we własnej klasie i korzystając przy tym z dyrektywy `override` do pokrywania odziedziczonej metody, najczęściej nie można zapominać o wywołaniu wewnątrz niej metody odziedziczonej. Dzięki temu nowa klasa wykorzystuje też funkcjonalność odziedziczoną z klasy bazowej. Do takich wywołań stosowana jest instrukcja `inherited`.

W przykładowym programie *WallpaperChanger* z punktu 2.2.3 zdefiniowana została klasa `TimerEvent`, w której metoda `Trigger` ustala wartość zmiennej `Done` na `True`. Kod tej metody przypominam na listingu 3.53.

Listing 3.53. *Metoda Trigger klasy TimerEvent*

```
procedure TimerEvent.Trigger;  // W deklaracji klasy metoda ta deklarowana jest jako
wirtualna
begin
  Done := True;
end;
```

W przypadku klasy specjalnej `AlarmEvent` w ramach wykonywania metody `Trigger` (jej kod podaję na listingu 3.54) oprócz ustawienia wartości zmiennej `Done` wyświetlane jest też okno z komunikatem. Dzięki wywołaniu metody `Trigger` odziedziczonej z klasy `TimerEvent` zapewniamy, że logika odziedziczonej metody nie pójdzie w zapomnienie.

Listing 3.54. *Metoda Trigger klasy AlarmEvent*

```
procedure AlarmEvent.Trigger;  // W deklaracji klasy metoda ta deklarowana jest
begin                          // ze słowem kluczowym override
  inherited;
  MessageBox.Show(MessageText);
end;
```

Dyrektywy `override`, `virtual` i `reintroduce` można stosować wyłącznie wewnątrz deklaracji klas i w dalszej części kodu, w miejscu implementowania metod nie powinny być powtarzane. Jeżeli szkielet metody tworzony jest za pomocą funkcji automatycznego uzupełniania klas, to w przygotowanym szkielecie znajdować się już będzie wstępne wywołanie odziedziczonej metody wykorzystujące słowo kluczowe `inherited`.

Polimorfizm

Koncepcja programowania realizowana za pomocą dziedziczenia i późnych dowiązań nazywana jest *polimorfizmem*. Obiektem polimorficznym nazywana jest zmienna obiektowa, dla której kompilator nie jest w stanie określić klasy obiektu, z jakim będzie ona związana w czasie działania programu. Co więcej, w czasie działania programu klasa tej zmiennej może się wielokrotnie zmieniać, ponieważ do jednej zmiennej przypisane mogą być obiekty wielu różnych klas.

Obiekt dostępny poprzez zmienną polimorficzną w kodzie programu opisany jest pewną konkretną klasą, ale w czasie działania może przyjmować różne formy. Obiekty polimorficzne spotyka się w programach w wielu różnych miejscach, na przykład:

- W metodach, które w parametrach przyjmować mogą dowolne obiekty, jako typ tych parametrów podają klasę `Object`. Taki typ mają na przykład parametry `sender` przekazywane do metod obsługujących zdarzenia formularza, a także poszczególne wpisy w kontrolce *ListBox*. Jak przekonaliśmy się w punkcie 2.4.1, wpisy tej kontrolki rzeczywiście mogą być całkowicie dowolnego typu (mówiąc dokładniej, w programie dostępne są **dwa** rodzaje wpisów umieszczanych w kontrolce *ListBox* — `DesktopChangeEvent` i `AlarmEvent`).

- Wszystkie kontrolki umieszczane na formularzu również są polimorficzne. Wewnątrz formularza przechowywane są one po prostu jako kolekcja obiektów typu `Control` (na jej temat mówiłem w punkcie 2.1.3), choć rzeczywiste kontrolki zawsze są klasy wywiedzionej z klasy `Control`, a jak wiemy, w Delphi dostępnych jest wiele różnych kontrolek.

Klasy abstrakcyjne

W wielkich hierarchiach dziedziczenia, takich jak biblioteka FCL lub VCL.NET, bardzo często zdarza się, że klasy wykorzystywane są tylko do tego, żeby zdefiniować części wspólne pewnych innych klas. W klasach tych może się zdarzyć, że co prawda deklarowane są metody wirtualne, które pokrywane są we wszystkich klasach wywiedzionych, ale wewnątrz klasy bazowej w ogóle nie są implementowane.

Deklarowanie takiej metody, która stanowi tylko rezerwację miejsca dla metod w klasach wywiedzionych, wymaga zastosowania w deklaracji słowa kluczowego `abstract`, które określa taką metodę jako *abstrakcyjną*. Dyrektywa `abstract` musi znajdować się zaraz za dyrektywą `virtual`, ponieważ metoda abstrakcyjna, która nie jest jednocześnie metodą wirtualną, nie ma racji bytu. W przykładowej klasie `KlasaAbstrakcyjna` metoda `Pracuj` mogłaby być w takim razie zadeklarowana tak:

```
procedure Pracuj; virtual; abstract;
```

Taka deklaracja ma takie zalety, że metody tej nie trzeba definiować w części implementacji modułu, a jej przypadkowe wywołania są automatycznie blokowane, ponieważ kompilator nie pozwala na tworzenie obiektu na podstawie klasy z metodami abstrakcyjnymi. Każda próba utworzenia takiego obiektu spowoduje wyświetlenie błędu już w czasie kompilacji programu.

W bibliotece VCL.NET znaleźć można kilka przykładów klas abstrakcyjnych: `TStream`, `TStrings` i `TPersistent`. W środowisku .NET klasy abstrakcyjne zdarzają się dużo rzadziej, ponieważ abstrakcyjne elementy klas najczęściej definiowane są jako *interfejsy*.

Daną klasę można też jawnie zadeklarować jako abstrakcyjną:

```
type
  Class1 = class abstract
  end;
```

Kompilator nie pozwala na tworzenie obiektów na podstawie klas abstrakcyjnych, nawet jeżeli sama klasa nie ma żadnych abstrakcyjnych metod.

3.3.4. Konwersja typów i informacje o typach

Czasami już w czasie działania programu konieczne jest sprawdzenie, jakiego typu jest dany obiekt. W ramach własnych klas można by to zrealizować za pomocą specjalnej metody wirtualnej `JakiegoJestemTypu`, ale na szczęście w Delphi istnieją bardziej standardowe rozwiązania tego problemu.

Operator is

Oba operatory `is` i `as` przeznaczone są do wykonywania bardzo eleganckiej konwersji typów i sprawdzania związków dziedziczenia łączących klasy. Po prawej stronie operatora zawsze znajduje się zmienna obiektowa (lub wyrażenie, którego wynik jest wskazaniem na obiekt) albo referencja klasy, natomiast po lewej stronie operatora może znajdować się wyłącznie referencja klasy. Operator `is` sprawdza, czy klasa prawego operandu jest jednym z potomków klasy podanej w lewym operandzie. W punkcie 2.4.1 znaleźć można przykładowy kod (jego część podaję na listingu 3.55), w którym, w zależności od klasy aktualnie wybranego wpisu kontrolki *ListBox*, wyświetlane jest okno dialogowe przeznaczone do modyfikowania danych danej klasy wpisów.

Listing 3.55. *Sprawdzanie klasy podanej zmiennej obiektowej*

```
if ListBox1.SelectedItem is DesktopChangeEvent then
  // Wywołanie okna dialogowego dla obiektów DesktopChangeEvent
end else  // w przeciwnym wypadku prawdziwe jest: ListBox1.SelectedItem is AlarmEvent
  // Wywołanie okna dialogowego dla obiektów AlarmEvent
```

Konwersja typów

Po sprawdzeniu za pomocą operatora `is`, czy podana zmienna wskazuje na obiekt pewnej konkretnej klasy, chcielibyśmy skorzystać z pewnych specjalnych elementów tej klasy. W przypomnianym wyżej przykładzie chcieliśmy z kontrolki *ListBox* odczytać wartość zmiennej `SelectedItem` i przekazać ją do okna dialogowego, ponieważ to właśnie w tej zmiennej zapisane są wszystkie dane, które chcemy edytować w oknie dialogowym. Ze względu na uprzednie sprawdzenie typu obiektu przeprowadzone operatorem `is` wiemy, że zmienna `SelectedItem` ma nie tylko zadeklarowany typ `System.Object`, ale również specjalny typ `DesktopChangeEvent`.

Chcąc uzyskać dostęp do tych specjalnych elementów — na przykład zmiennej `ImageFileName` — możemy skorzystać z dwóch wariantów składniowych konwersji typów, przedstawionych na listingu 3.56.

Listing 3.56. *Dwa warianty zapisu konwersji typów*

```
DesktopChangeEvent(ListBox1.SelectedItem).ImageFileName := NowaNazwa;
// lub:
(ListBox1.SelectedItem as DesktopChangeEvent).ImageFileName := NowaNazwa;
```

W środowisku Win32 te dwa warianty zachowują się nieco odmiennie: Przedstawione wyżej operacje kontrolne wykonywane są tylko w wariancie z operatorem `as`. Oznacza to, że pierwszy wariant jest nieco wydajniejszy pod warunkiem, że już wcześniej sami skontrolowaliśmy typ obiektu i mamy całkowitą pewność, że będzie on zgodny z typem konwersji. Jeżeli jednak będziemy próbować uzyskać dostęp do elementów obiektu nieobsługiwanych przez jego typ i w związku z tym skonwertować go na typ niezgodny, to wywołany zostanie odpowiedni wyjątek.

Oba warianty powodują dokładnie takie samo działanie na platformie .NET: Przed konwersją sprawdzane jest, czy podany obiekt jest zgodny z podanym typem (jest to odpowiednik jawnego zastosowania operatora `is`). Jeżeli tak jest, to umożliwiany jest dostęp do wybranego elementu obiektu, a w przeciwnym wypadku wywoływany jest wyjątek.

Inne informacje o typie

Za pomocą operatora `is` można stwierdzić w czasie działania programu, czy dany obiekt jest zgodny z podanym typem. To tylko jedna z możliwych do uzyskania informacji o typach, które szeroko udostępniało już Delphi dla Win32, a w środowisku .NET liczba tych informacji wzrosła jeszcze bardziej, dzięki metadanym zapisywanym w każdym kompilacie.

Metadane, jakich wymaga CLR, dla każdej klasy zachowywane są w specjalnym obiekcie klasy `System.Type`. Taki obiekt typu `Type` można odczytać w czasie działania programu z dowolnego obiektu w systemie. W tym celu należy wywołać metodę `GetType` danego obiektu. Poprzez uzyskany w ten sposób obiekt `Type` można odczytać pozostałe dane o danym typie, takie jak jego klasa bazowa, nazwa samej klasy i kompilat, w którym zdefiniowany jest ten typ.

Można na przykład w jednym z formularzy powiązać zdarzenia `MouseMove` wszystkich kontrolek z metodą podaną na listingu 3.57, w której przy każdym poruszeniu myszy na pasku tytułu wyświetlane będą informacje o klasie kontrolki, nad którą znajduje się aktualnie kursor myszy.

Listing 3.57. *Metoda sprawdzająca klasę kontrolki wskazywanej przez kursor myszy*

```
procedure TWinForm.AnyControl_MouseMove(sender: System.Object; e:
System.Windows.Forms.MouseEventArgs);
begin
  Text := 'Myszka jest nad kontrolką ' + sender.GetType.Name
          + ' o klsie bazowej: ' + sender.GetType.BaseType.Name;
```

Alternatywą w stosunku do metody `GetType` jest wykorzystanie wbudowanej w kompilator funkcji `typeof`. Jej zaletą w stosunku do metody `GetType` jest to, że nie potrzebuje ona obiektu, żeby odczytać metadane jego typu, ale wystarczy podać jej w parametrze sam typ, tak jak na listingu 3.58.

Listing 3.58. *Wykorzystanie funkcji typeof wbudowanej w kompilator*

```
// TypeData: System.Type;
TypeData := typeof(sender); // Odpowiada podanemu wyżej wywołaniu funkcji GetType
TypeData := typeof(Integer); // Zwraca metadane typu Integer
```

Dobre przykłady zastosowania funkcji `typeof` znaleźć można w podpunktach „Typy wyliczeniowe w środowisku .NET” (strona 360) i „Serializacja XML” (strona 213).

Referencje klas

Przedstawiona wyżej klasa `System.Type` dostępna jest wyłącznie w środowisku .NET, ale samo Delphi dla wszystkich platform udostępnia też tak zwane *referencje klas*, które pozwalają w sposób przenośny uzyskiwać najważniejsze informacje o danym typie. Moduł `System` przechowuje deklarację typu opisującego najbardziej podstawową referencję klasy:

```
type
  TClass = class of TObject;
```

Nie należy mieszać identyfikatora `TClass` ze zwykłymi klasami, które również deklarowane są z wykorzystaniem słowa kluczowego `class`, ale bez słowa `of`. Typ referencji klasy nazywany jest też *metaklasą*, ponieważ referencje klas przechowują tylko informacje o klasach, takie jak ich nazwa i dane klasy bazowej. Jeżeli typ referencji klas zadeklarowany zostanie zapisem `of TMojaKlasa`, to w ten sposób zawężany jest zakres klas, których dotyczyć mogą takie referencje, do klasy `TMojaKlasa` i klas z niej wywiedzionych. Wynika z tego, że typ `TClass` jest najbardziej ogólnym typem referencji klas, ponieważ umożliwia on odczytywanie informacji o dowolnych klasach.

Referencję klasy otrzymać można przypisując do zmiennej typu referencji nazwę interesującej nas klasy lub wywołując metodę `ClassType` dowolnego obiektu. Tak otrzymaną referencję klasy można zapisać do zmiennej o typie referencji klasy, tak jak to pokazano na listingu 3.59.

Listing 3.59. *Sposób uzyskiwania referencji klasy*

```
var
  ClassRef: TClass;
begin
  ClassRef := Button;  // Możliwość 1.: "Przypisanie" nazwy klasy
  ClassRef := Button1.ClassType;  // Możliwość 2.: Wywołanie metody danego obiektu
```

Metody klasy TObject

W tabeli 3.2 podane zostały metody dostępne w standardowej klasie Delphi `TObject` (oznacza to, że można je wywołać w każdym obiekcie w Delphi, nawet w tych implementowanych w innych językach, w których nie istnieje klasa `TObject`[4]), bezpośrednio związane z referencjami klas.

Funkcje typu `class function` podawane w tabeli mogą być wykorzystywane niezależnie od samych obiektów albo być wywoływane na rzecz konkretnego obiektu, tak jak na listingu 3.60.

[4] Do rozbudowywania takich „obcych obiektów" firma Borland stosuje specjalną technikę tak zwanych klas pomocniczych (ang. *Class helpers*), których nie można mylić z mechanizmem dziedziczenia. Są one krótko opisywane w systemie aktywnej pomocy Delphi, ale dla normalnego tworzenia aplikacji w Delphi nie mają praktycznie żadnego znaczenia.

Tabela 3.2. *Metody związane z referencjami klas*

Początek funkcji	Funkcja	Typ zwracany	Wynik
`class function`	`ClassName`	`String`	Nazwa obiektu lub klasy
`class function`	`ClassNamels(Str)`	`Boolean`	Sprawdza, czy nazwa klasy jest zgodna z podanym ciągiem znaków
`class function`	`ClassParent`	`TClass`	Klasa bazowa obiektu lub klasy
`function`	`ClassType`	`TClass`	Klasa obiektu
`class function`	`ClassInfo`	`Type`	Zwraca, uzależniony od platformy, obiekt zawierający dokładne informacje o typie. Na platformie .NET metoda ta odpowiada metodzie `Object.GetType` i zwraca obiekt typu `System.Type`.
`Class function`	`InheritsForm(AClass)`	`Boolean`	Sprawdza, czy klasa wywiedziona jest z klasy *AClass*

Listing 3.60. *Sposoby wywoływania metody ClassName*

```
// Zmienna Nazwa może być zadeklarowana z typem String
Nazwa := TButton.ClassName;
Nazwa := Button1.ClassName;
```

Ten drugi wariant tak naprawdę nie powinien być dopuszczalny, jako że wywoływana jest w nim metoda statyczna `ClassName` na rzecz konkretnego obiektu. To rozwiązanie działa, ponieważ w czasie pracy programu sprawdza on, jakiej klasy jest zmienna `Button1`, i wywołuje metodę `ClassName` na rzecz tej właśnie klasy, a nie na rzecz obiektu. W rzeczywistości drugie wywołanie metody `ClassName` wygląda następująco:

```
Name := Button1.ClassType.ClassName;
```

3.3.5. Konstruktory wirtualne

Na koniec, w tym punkcie przedstawię jeszcze jedną specjalność Delphi: konstruktory wirtualne. Realizacja takich konstruktorów możliwa jest wyłącznie dzięki omawianym w poprzednim rozdziale referencjom klas i tworzonemu przez nie polimorfizmowi.

W punkcie 3.3.3 przedstawiałem kod, w którym na podstawie wyboru użytkownika tworzony był obiekt określonej klasy (kod ten powtarzam na listingu 3.61).

Listing 3.61. *Tworzenie obiektu na podstawie wyboru użytkownika*

```
case WyborUzytkownika of
  { "WyboremUzytkownika" może być na przykład wartość (Sender as TButton).Tag }
    Button1: ObiektRoboczy := Klasa1.Create;
    Button2: ObiektRoboczy := Klasa2.Create;
...
    Button10: ObiektRoboczy := Klasa10.Create;
```

Jak widać, możemy co prawda utworzyć obiekt odpowiedni dla wyboru dokonanego przez użytkownika, ale zdecydowanie niezadowalający jest brak możliwości zapisania w zmiennej wybranej klasy. Po utworzeniu zmiennej `ObiektRoboczy`, uzyskujemy dostęp do wszystkich wirtualnych funkcji tego obiektu, tak jak tego chcieliśmy. Jeżeli jednak w innym miejscu w programie konieczne byłoby utworzenie drugiego obiektu o tym samym typie (uzależnionym od wyboru użytkownika), to ponownie zostaniemy zmuszeni do wpisywania długiej instrukcji `case` (albo przygotowania specjalnej procedury pozwalającej wielokrotnie wykorzystać raz wpisaną instrukcję `case`).

Problem ten można rozwiązać o wiele bardziej elegancko, wykorzystując przy tym referencje klas, tak jak na listingu 3.62.

Listing 3.62. *Przechowywanie referencji klasy w zmiennej*

```
type
  ReferencjaKlasyAbstrakcyjnej = class of KlasaAbstrakcyjna;
var
  WybranaKlasa: ReferencjaKlasyAbstrakcyjnej;
  Objekt: KlasaAbstrakcyjna;
begin
   Objekt := WybranaKlasa.Create;
```

Dzięki przedstawionej wyżej instrukcji uzyskujemy dokładnie ten sam efekt, jaki tworzony był przez długi blok instrukcji `case` z poprzedniego listingu. Oczywiście zmienna referencji klasy musi już wcześniej otrzymać odpowiednią wartość. Jeżeli wartość ta uzależniona byłaby od wyboru dokonanego przez użytkownika, to nadal będziemy musieli skorzystać z odpowiednio zmodyfikowanej instrukcji `case`, na przykład takiej jak na listingu 3.63.

Listing 3.63. *Instrukcja case przygotowująca referencję klasy*

```
case WyborUzytkownika of
   Button1: WybranaKlasa := Klasa1;
   Button2: WybranaKlasa := Klasa2;
... itd.
```

Jest to jednak niezaprzeczalna zaleta, że instrukcja `case` musi tylko raz obsłużyć wybór dokonany przez użytkownika i zapisać go jako referencję klasy. Referencję klasy można sobie też wyobrazić jako zmienną podobną do zmiennej obiektowej, która zamiast obiektu przechowuje klasę.

Konstruktory wirtualne

Przedstawiony wyżej kod niesie ze sobą pewne zagrożenie, znajdujące się w wierszu:

```
Objekt := WybranaKlasa.Create;
```

Wywołanie to zawsze przygotuje obiekt właściwej klasy, ale za każdym razem wywoływany będzie tylko konstruktor klasy `KlasaAbstrakcyjna`, który dziedziczony jest przez wszystkie klasy wywiedzione. Jeżeli klasy te muszą wykonywać w swoich konstruktorach inne ważne operacje inicjalizacji obiektu, to w powyższym wierszu kodu

wywoływane muszą być właśnie konstruktory klas wywiedzionych. Takie funkcjonowanie tego zapisu można osiągnąć, deklarując konstruktor `Create` klasy `KlasaAbstrakcyjna` ze słowem kluczowym `virtual`, przez co konstruktor ten traktowany jest jako konstruktor wirtualny, a w klasach wywiedzionych (czyli klasach `Klasa1`, `Klasa2` itd.) można w deklaracjach konstruktorów zastosować słowo kluczowe `override`.

Konstruktory wirtualne można sensownie wykorzystać tylko wtedy, gdy podobnie jak w naszym przykładzie wywoływane są poprzez referencję klasy. We wszystkich pozostałych przypadkach już w czasie kompilacji ustalany jest konstruktor wywoływany w celu utworzenia obiektu, i w takim zakresie całkowicie wystarczające jest dowiązywanie statyczne.

Ten sam efekt uzyskać można stosując środki udostępniane przez środowisko .NET, choć nie jest to już tak proste. Zamiast referencji klasy stosowany jest tu obiekt typu `System.Type`, a wywołanie konstruktora trzeba w tym wypadku „spowodować zdalnie", wykorzystując do tego metodę `System.Type.Invoke`. Jeżeli tworzone przez nas klasy mają być też stosowane w innych językach programowania, to lepiej byłoby zamiast wirtualnego konstruktora zastosować konstruktor statyczny i uzupełnić go o wirtualną metodę inicjalizującą, która musiałaby być wywoływana oprócz samego konstruktora.

3.4. Typy interfejsów

W programowaniu zorientowanym obiektowo wszystkie elementy danej klasy, które używane są z zewnątrz tej klasy, tworzą tak zwany *publiczny interfejs* tej klasy. W procesie dziedziczenia klasa wywiedziona dziedziczy również publiczny interfejs klasy bazowej. Części programu pracujące z obiektem klasy bazowej mogą równie dobrze rozpocząć pracę ze wszystkimi klasami z niej wywiedzionymi, co umożliwia właśnie ta pełna zgodność interfejsów.

Konstrukcja językowa interfejsów powoduje oddzielenie interfejsu od samej klasy, dzięki czemu można mówić o samym interfejsie niezwiązanym z żadną konkretną klasą.

Przedstawianych tutaj interfejsów obiektów i klas nie należy mylić z interfejsami modułów, które w języku Object Pascal definiowane są tym samym słowem kluczowym `interface`. W tym podrozdziale słowo kluczowe `interface` dotyczyć będzie wyłącznie interfejsów klas i obiektów.

3.4.1. Czym jest interfejs?

Interfejs jest właściwie tylko listą właściwości i metod, przy czym metody mogą być w interfejsie wyłącznie deklarowane (czyli mogą być wymieniane ich nazwy, parametry i typy zwracane), ale nie mogą być wewnątrz interfejsu implementowane. Przykładową postać interfejsu przedstawiam na listingu 3.64.

Listing 3.64. *Przykładowy interfejs*

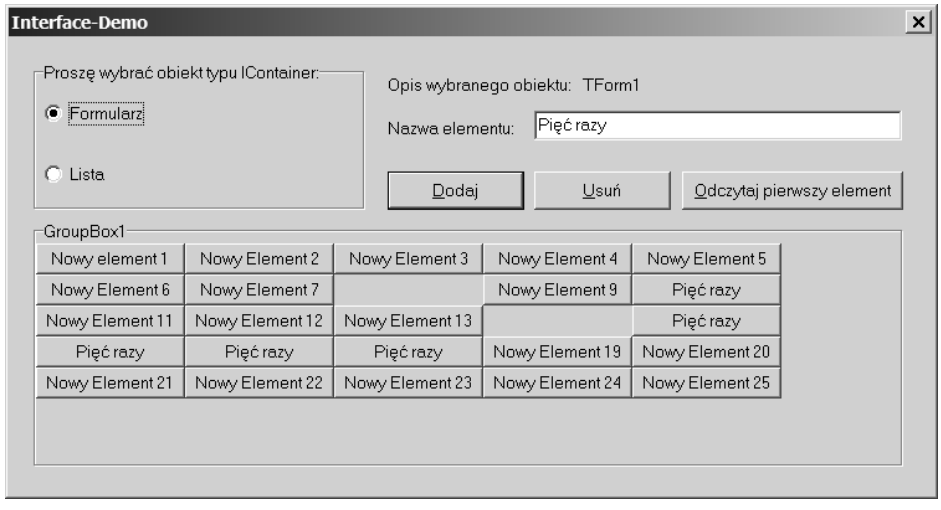
```
type
  IContainer = interface
    procedure AddElement(ElementName: string);
    procedure DeleteElement(ElementName: string);
    function GetElementCount: integer;
    function GetFirstElementName: string;
  end;
```

Powyższa deklaracja opisuje interfejs, za pomocą którego do obiektu kontenera możemy dodawać elementy opisywane ciągiem znaków (`AddElement`), a także usuwać je z tego kontenera podając nazwę elementu (`DeleteElement`). Pozostałe dwie metody pozwalają na odczytanie liczby elementów zapisanych w kontenerze, jak również na uzyskanie nazwy pierwszego elementu. Sposób definiowania pierwszego elementu, a także części składowych poszczególnych elementów kontenera, nie jest określany w samym interfejsie, ale musi być zdefiniowany w obiekcie kontenera. Przedstawiony interfejs nie mówi też nic o samym obiekcie kontenera, poza tym, że przechowuje on jakieś nieokreślone elementy.

Przykład interfejsu przedstawiony na listingu 3.64 znaleźć można też na płycie CD dołączonej do książki, w projekcie *InterfaceDemo*. Projekt ten jest aplikacją VCL.NET i ma demonstrować wyłącznie zasadę funkcjonowania interfejsów, jak należy ich używać i je implementować. Z całą pewnością nie jest to jednak przykład **dobrego** interfejsu. W środowisku .NET bardzo szeroko stosowany jest wzorowy interfejs `IList`, który również umożliwia dodawanie i usuwanie elementów ze zbioru i w związku z tym w przykładowym programie mógłby być wykorzystany w miejscu samodzielnie definiowanego interfejsu `IContainer`. Interfejs `IList` jest jednak o wiele bardziej rozbudowany, wobec czego w przykładowym programie łatwiejsze było zastosowanie znacznie mniejszego interfejsu `IContainer`.

Na rysunku 3.11 zostało przedstawione okno przykładowego programu. W czasie jego pracy za pomocą przełączników można wybrać jeden z dwóch kontenerów, do których dodawane i usuwane są ciągi znaków wprowadzane do pola edycyjnego:

- Elementami kontenera `Formularz` są przyciski. Za każdym razem, gdy wywoływana jest funkcja `AddElement`, w dolnej części formularza tworzony jest nowy przycisk. Podobnie, po wywołaniu funkcji `DeleteElement` odpowiedni przycisk jest usuwany z formularza. Ten wyjątkowo mało sensowny tryb działania wybrany został jako kontrast dla drugiego obiektu kontenera.
- Drugi z obiektów-kontenerów — `Lista` — po każdym wywołaniu metody `AddElement` zachowuje wybrany element, ale nigdzie go nie wyświetla. Chcąc przekonać się, że dodane do listy elementy rzeczywiście się w niej znajdują, trzeba odczytać z tej listy pierwszy element wywołaniem funkcji `GetFirstElementName`, a następnie go usunąć i powtarzać tę operację z kolejnymi elementami.



Rysunek 3.11. *Okno programu stosującego dwa całkowicie różne obiekty-kontenery obsługujące dokładnie ten sam interfejs (przykładowy program InterfaceDemo)*

Zmienne interfejsów

Programowanie z wykorzystaniem interfejsów polega na zadeklarowaniu zmiennej o typie interfejsu i wykorzystywaniu jej tak jak zwyczajnego obiektu. W przykładowym formularzu znajdziemy zmienną `CurrentContainer` wskazującą na kontener wybrany przez użytkownika za pomocą przełączników (jej deklarację przedstawiam na listingu 3.65). O tym, jak inicjowane są poszczególne kontenery i jak przypisywane są one do zmiennej `CurrentContainer`, będziemy mówić nieco później.

Listing 3.65. *Deklaracja zmiennej interfejsu kontenera*

```
var
  TForm1 = class ...
  CurrentContainer: IContainer;
```

Naciśnięcie przycisków *Dodaj*, *Usuń* i *Odczytaj pierwszy element* spowoduje w programie wywołanie odpowiednich metod interfejsu `IContainer`, które przedstawiam na listingu 3.66.

Listing 3.66. *Metody interfejsu IContainer wywoływane w przykładowym programie*

```
procedure TForm1.AddButtonClick(Sender: TObject);
begin
  CurrentContainer.AddElement(ElementName.Text);
end;

procedure TForm1.DeleteButtonClick(Sender: TObject);
begin
  CurrentContainer.DeleteElement(ElementName.Text);
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Pierwszy element to: '+
    CurrentContainer.GetFirstElementName);
end;
```

W czasie kompilacji programu nie jest jeszcze określone, który rodzaj kontenera będzie zapisany w zmiennej `CurrentContainer`. Wynika z tego, że zmienna ta jest *obiektem polimorficznym* (ang. *Polymorphes object*), który znamy już z opisywanego wcześniej podobnego traktowania klas. Jeżeli nie wykorzystywalibyśmy interfejsów, to zmienna `CurrentContainer` mogłaby być też zadeklarowana tak:

```
var
  CurrentContainer : TContainer
```

Przy czym `TContainer` musiałaby być wtedy klasą, której deklarację przedstawiam na listingu 3.67.

Listing 3.67. *Deklaracja klasy TContainer*

```
type
  TContainer = class
    procedure AddElement(ElementName: string); abstract;
    procedure DeleteElement(ElementName: string); abstract;
    function GetElementCount: integer; abstract;
    function GetFirstElementName: string; abstract;
  end;
```

Wielką przewagą interfejsów nad klasą `TContainer` jest w fakt, że interfejs może być implementowany w całkowicie **dowolnej** klasie. Jeżeli chcielibyśmy uzyskać konkretną implementację abstrakcyjnej klasy `TContainer`, to musielibyśmy przygotować nową klasę wywiedzioną z klasy `TContainer`. Nowa klasa mogłaby mieć tylko jedną klasę bazową, dlatego klasa ta nie mogłaby być już wywodzona z żadnej innej klasy. Stosując interfejs `IContainer` nie nakładamy na swoje poczynania takiego ograniczenia, wobec czego w przykładowym programie interfejs ten implementowany jest przez klasę formularza.

Dziedziczenie interfejsów

W przypadku interfejsów obowiązują zasady dziedziczenia podobne do tych, które znamy już z dziedziczenia klas. Oznacza to, że stosując zapis z listingu 3.68 możemy zdefiniować nowy interfejs, który będzie miał wszystkie metody interfejsu `IContainer` i dokładał do nich własną metodę `Mix`.

Listing 3.68. *Przykład dziedziczenia interfejsów*

```
type
  IMixableContainer = interface(IContainer)
    procedure Mix;
  end;
```

Najważniejsze jest w tym wszystkim jednak to, że w interfejsach możliwe jest też dziedziczenie wielobazowe, czyli interfejs może być wywiedziony z kilku interfejsów naraz, mniej więcej tak, jak pokazano to na listingu 3.69.

Listing 3.69. *Interfejsy mogą stosować dziedziczenie wielobazowe*

```
type
  IMixable = interface
    procedure Mix;
  end;
  IMixableContainer = interface(IContainer, IMixable)
  end;
```

3.4.2. Implementowanie interfejsu

Do implementowania interfejsu zawsze potrzebować będziemy jakiejś klasy. Implementowany interfejs podawany jest w nagłówku deklaracji klasy, zaraz za klasą bazową. Oczywiście, klasa może mieć tylko jedną klasę bazową, ale jednocześnie może implementować wiele interfejsów, które wymieniane są właśnie w nagłówku deklaracji klasy. Co więcej, klasa musi implementować wszystkie metody zapisanych w tym miejscu interfejsów, co oznacza, że nagłówki metod wymieniane w deklaracji interfejsu muszą zostać powtórzone w deklaracji samej klasy.

Z historycznych powodów związanych z konstrukcją Delphi, w języku Object Pascal każda klasa implementująca pewien interfejs musi być jawnie wywiedziona z innej klasy. Jeżeli nasza klasa nie wymaga żadnej klasy bazowej, to można ją zadeklarować zgodnie z wzorcem `TMojaImplementacja(TObject, IMojInterfejs)`. W Delphi dla Win32 interfejsy są dodatkowo powiązane z modelem COM (ang. *Component Object Model*), co oznacza, że w środowisku Win32 jako klasy bazowej dla klasy implementującej obiekty nie można wykorzystać klasy `TObject`, ale trzeba skorzystać z klasy `TInterfacedObject`. Jest to tylko jedna z wielu rzeczy, jaką należy uwzględnić w czasie implementowania interfejsów w środowisku Win32. Akurat w zakresie obsługi interfejsów w środowisku .NET nastąpiło znaczne uproszczenie obowiązujących procedur.

Ciąg dalszy przykładu z interfejsem IContainer

Jak już wspominałem, przykładowy program wykorzystuje dwie niezależne implementacje interfejsu `IContainer` i w związku z tym konieczne jest w nim przygotowanie dwóch klas, z których każda całkowicie odmiennie implementuje ten interfejs. Jedna z tych klas, implementująca interfejs w postaci listy, przedstawiona została na listingu 3.70.

Listing 3.70. *Implementacja interfejsu IContainer*

```
// uses Borland.Vcl.Classes
type
  TListContainer = class(TStringList, IContainer)
    procedure IContainer.AddElement = Append;
    procedure DeleteElement(const ElementName: string);
```

```
    function GetElementCount: integer;
    function GetFirstElementName: string;
  end;

implementation

procedure TListContainer.DeleteElement(const ElementName: string);
begin
  if IndexOf(ElementName)>-1 then
    Delete(IndexOf(ElementName));
end;

function TListContainer.GetElementCount: integer;
begin
  Result := Count;
end;

function TListContainer.GetFirstElementName: string;
begin
  if Count>0 then Result:=Strings[0]
    else Result := '(Lista jest pusta)';
  if Result='' then Result := '(Pierwszy element nie ma nazwy.)';
end;
```

Klasa `TListContainer` dziedziczy pełną funkcjonalność listy z klasy `TStringList` pochodzącej z biblioteki VCL.NET, i na przykład udostępnia wykorzystywane na listingu metody `Append`, `Delete`, `IndexOf`, `Count` i `Strings`. Jak widać, klasa `TListContainer` właściwie przenosi działanie funkcji klasy `TStringList` do metod wymienianych w interfejsie `IContainer`.

Zmiany nazw metod

W przypadku metody `IContainer.AddElement` operacje wykonywane w implementacji interfejsu są wyjątkowo proste, ponieważ działanie odziedziczonej metody `Append` jest całkowicie zgodne z tym, co powinna robić metoda `AddElement`, a na dodatek ma ona dokładnie taki sam format wywołania (jeden parametr typu `String` i brak wartości zwracanej). Właśnie dlatego w interfejsie `IContainer` nie ma potrzeby implementowania metody `AddElement`, ale wystarczy połączyć ją z istniejącą metodą `TStringList.Append`:

```
procedure IContainer.AddElement = Append;
```

Takie jawne powiązanie metody interfejsu z metodą implementacji możliwe jest również wtedy, gdy tworzymy własną metodę implementacji i nadajemy jest nazwę zupełnie inną od nazwy metody interfejsu. W takiej sytuacji trzeba tylko pamiętać o uzupełnieniu nazwy metody interfejsu o odpowiednią klauzulę zmiany nazwy.

Funkcje pomocy w programowaniu i uzupełnianie klas

Dostępne w edytorze Delphi funkcje pomocy w programowaniu, w zakresie implementowania interfejsów oferują jeszcze jedną ciekawą rzecz: Jeżeli kursor edytora znajduje się wewnątrz deklaracji klasy, która obsługuje interfejsy, to po naciśnięciu w pustym wierszu klawiszy *Ctrl+Spacja* wyświetlona zostanie lista brakujących jeszcze

implementacji metod interfejsów. Normalnie lista wyboru wyświetlana wewnątrz deklaracji klasy zawiera tylko metody odziedziczone z klasy bazowej, które można pokryć w klasie wywiedzionej. Jeżeli jednak w klasie tej brakuje jeszcze metod wymaganych przez implementowany interfejs, to Delphi wyświetla je na samym początku wyświetlanej listy, a dodatkowo wyróżnia kolorem czerwonym.

Oczywiście można też wywołać funkcję uzupełniania klasy (naciskając kombinację klawiszy *Shift+Ctrl+C*), żeby w ten sposób przygotować szkielety implementacji wszystkich metod wymienionych w deklaracji tej klasy.

Druga implementacja interfejsu IContainer

Druga klasa z przykładowego programu, implementująca interfejs `IContainer`, to klasa samego formularza aplikacji — jak już wspominałem, jest to aplikacja korzystająca z biblioteki VCL.NET, dlatego formularz ten nie jest wywiedziony z klasy `System.Windows.Forms.Form`, ale z klasy `TForm`:

```
type
  TForm1 = class(TForm, IContainer)
```

Możemy sobie tu podarować ponowne wypisywanie wszystkich metod interfejsu `IContainer`. Podobnie niewiele do omawianego tematu wnoszą nam implementacje metod tego interfejsu. W ramach przykładu przedstawię zatem (na listingu 3.71) wyłącznie implementację metody `AddElement`, która dynamicznie tworzy obiekty typu `TButton` pochodzącego z biblioteki VCL.NET i dodaje go do obszaru na formularzu (kontrolka typu *TScrollBox*) przeznaczonego na tworzone w ten sposób przyciski.

Listing 3.71. *Implementacja metody AddElement w drugiej klasie implementującej interfejs IContainer*

```
procedure TForm1.AddElement(const ElementName: string);
var
  B: TButton;
const
  AddCount: Integer = 0;
begin
  B := TButton.Create(self);
  B.Parent := ScrollBox1;
  B.Width := 100;
  B.Top := (AddCount div 5)*B.Height;
  B.Left := (AddCount mod 5)*B.Width;
  B.Caption := ElementName;
  inc(AddCount);
end;

procedure TForm1.DeleteElement(const ElementName: string);
var
  i: Integer;
begin
  for i := ScrollBox1.ControlCount-1 downto 0 do
    if (ScrollBox1.Controls[i] as TButton).Caption = ElementName
      then ScrollBox1.Controls[i].Free;
end;
```

Tworzenie obiektów z interfejsami

Teraz należałoby utworzyć obiekty na podstawie klas implementujących interfejs `IContainer`, co będzie wymagało przypomnienia sobie kilku rzeczy na temat klas w języku Object Pascal. Obiekt implementujący listę utrzymamy w sposób przedstawiany na listingu 3.72.

Listing 3.72. *Tworzenie obiektu na podstawie klasy implementującej interfejs*

```
var
  ListContainerObject: TListContainer;
begin
  ListContainerObject := TListContainer.Create;
end.
```

Obiekt implementacji formularza jest automatycznie tworzony w kodzie przygotowanym przez Delphi, pamiętamy wszak, że jest to formularz aplikacji.

Od tego momentu droga od obiektu do zmiennej interfejsu jest już bardzo prosta — wystarczy do tej zmiennej przypisać istniejący obiekt, tak jak na listingu 3.73.

Listing 3.73. *Sposoby wiązania zmiennej interfejsu z obiektem implementującym interfejs*

```
var
  ListContainer: IContainer;
begin
  ListContainer := ListContainerObject;
  (* Można też osiągnąć to samo nie wykorzystując pośredniczącej zmiennej ListContainerObject:
     ListContainer := TListContainer.Create; *)
```

Z poziomu zmiennej interfejsu nie będziemy mieli dostępu do wszystkich teoretycznie istniejących metod klasy `FormContainer`, ponieważ przypisanie to jest równoznaczne z konwersją typów z klasy wywiedzionej do jednej z klas wyższego poziomu.

Teraz możemy już przejść do zapisywania wartości do zmiennej `CurrentContainer`, o której mówiliśmy już w punkcie 3.4.1. W programie, po kliknięciu na przełączniku przypisywany jest do niej odpowiedni obiekt implementujący interfejs `IContainer`. Procedura obsługująca te przypisania przedstawiona została na listingu 3.74.

Listing 3.74. *Przypisanie wybranego przez użytkownika obiektu implementującego interfejs do zmiennej interfejsu*

```
procedure TForm1.ContainerTypClick(Sender: TObject);
begin
  case ContainerTyp.ItemIndex of
    0: CurrentContainer := self;
    1: CurrentContainer := ListContainer;
  end;
end;
```

Obiekty z wieloma interfejsami

Jak już mówiłem, jedna klasa może obsługiwać kilka interfejsów. Postaram się zademonstrować to za pomocą naszego przykładowego programu, ponieważ wiąże się z tym interesująca możliwość „zmiany" jednego interfejsu w inny.

Po pierwsze, w przykładowym programie istnieje też drugi interfejs deklarujący zaledwie jedną metodę, za pomocą której można uzyskać informację o tym, jaka konkretna klasa ukrywa się za polimorficznym interfejsem. Deklaracja tego interfejsu widoczna jest na listingu 3.75.

Listing 3.75. *Deklaracja drugiego interfejsu z przykładowego programu*

```
type
  IClassDescription = interface
    function GetClassDescription: string;
  end;
```

Ten nowy interfejs implementowany jest w obu przedstawianych do tej pory klasach. Na listingu 3.76 przedstawiam skrót tej implementacji.

Listing 3.76. *Implementacja drugiego interfejsu w programie przykładowym*

```
// TForm1 = class(TForm, IContainer, IClassDescription)
// TListContainer = class(TStringList, IItemContainer, IClassDescription)

function TListContainer.GetClassDescription: string;
begin
  Result:='Lista ciągów znaków';
end;

function TForm1.GetClassDescription: string;
begin
  Result := 'TForm1';
end;
```

Chcąc używać nowego interfejsu, nie musimy już tworzyć żadnych nowych zmiennych typu `IClassDescription`, bo w zupełności wystarczy nam istniejąca już zmienna `CurrentContainer`, choć jest ona typu `IContainer`, który nie ma żadnych związków z interfejsem `IClassDescription`.

Cały czas pracujemy tutaj na interfejsach, w związku z czym za pomocą operatora `as` możemy zmienić typ zmiennej `CurrentContainer` na `IClassDescription` i wtedy wywołać na jej rzecz metodę `GetClassDescription`. W przykładowym programie, po kliknięciu na przełączniku aktualizowana jest zawartość kontrolki typu *Label* wypisującej tekst, jakim przedstawia się aktualnie wybrana implementacja kontenera. Odpowiedni kod przedstawiam na listingu 3.77.

Listing 3.77. *Wywołanie metody pochodzącej z drugiego interfejsu implementowanego w obu obiektach*

```
procedure TForm1.ContainerTypClick(Sender: TObject);
begin
 ...
 Label2.Caption := (CurrentContainer as IClassDescription).GetClassDescription;
end;
```

Operator as powoduje, że w czasie działania programu środowisko CLR sprawdza, czy obiekt przypisany do zmiennej CurrentContainer obsługuje też wymieniony interfejs IClassDescription. Jeżeli tak nie będzie, to w programie wygenerowany zostanie wyjątek. W naszym przykładowym programie możemy jednak zakładać, że interfejs ten będzie obsługiwany i przekształcenie wykonywane przez operator as zakończy się sukcesem.

Dziedziczenie jedno- i wielobazowe

Podsumowując, można powiedzieć, że w języku Object Pascal, podobnie jak i w całym środowisku .NET, obowiązuje zasada mówiąca, że dziedziczenie wielobazowe możliwe jest wyłącznie w przypadku interfejsów. Jedna klasa może implementować kilka interfejsów, a nowy interfejs może rozbudowywać kilka innych interfejsów. Taki rodzaj dziedziczenia dotyczy jednak wyłącznie samych interfejsów, a nie ich implementacji. W przypadku implementacji język Object Pascal, a także środowisko .NET przewidują wyłącznie możliwość dziedziczenia prostego (jednobazowego). Każda klasa może mieć co najwyżej jedną klasę bazową, po której dziedziczy implementacje wszystkich jej metod. Nie ma przy tym żadnego znaczenia to, czy metody te obsługują jeden interfejs, kilka interfejsów, czy może nie ma w nich żadnego interfejsu. W zakresie stosowania dziedziczenia jedno- i wielobazowego język Object Pascal i środowisko .NET są całkowicie zgodne z językiem Java. Z kolei język C++ pozwala na stosowanie dziedziczenia wielobazowego również w implementacjach, ale jest to wyjątkowo problematyczna funkcja tego języka, przez którą bardzo łatwo można wprowadzić zamieszanie do programu i dlatego jest niezwykle rzadko stosowana.

3.5. Podstawy języka Object Pascal

W dotychczasowych podrozdziałach omawiałem strukturę modułów i model obiektów języka Object Pascal, a także wynikające z tych zagadnień części programów, takie jak moduły, klasy i obiekty. Teraz możemy przejść do bardziej szczegółowych elementów języka. W tym podrozdziale zajmiemy się najmniejszymi elementami programów, czyli poszczególnymi słowami, z których składa się kod źródłowy programu przygotowanego w języku Object Pascal, a następnie przedstawię kilka ogólnych uwag dotyczących samego języka. W poprzednich podrozdziałach operowaliśmy tylko najprostszymi typami danych, takimi jak integer lub string, natomiast w podrozdziale 3.6 postaram się przedstawić wszystkie inne typy jakie dostępne są w języku Object Pascal. W podrozdziale 3.7 analizować będziemy też poszczególne instrukcje, z których składa się kod implementacji metod obiektów.

3.5.1. Elementy leksykalne

Kod źródłowy programów w języku Object Pascal może składać się wyłącznie z następujących elementów:

- z góry ustalonych słów kluczowych języka, operatorów i znaków interpunkcyjnych,
- bezpośrednio wpisywanych wartości stałych różnych typów,
- identyfikatorów,
- komentarzy,
- atrybutów.

Wszystkie te punkty przeglądać będziemy w odwrotnej kolejności.

Atrybuty wprowadzone zostały do Delphi w celu dostosowania jego języka do wymogów środowiska .NET. Mogą się one znajdować przed określonymi deklaracjami, gdzie stanowią dodatkowe informacje opisujące deklarowany symbol. Można je rozpoznać po obejmujących je nawiasach prostokątnych. Więcej informacji na temat atrybutów podawał będę w punkcie 3.5.6.

Komentarze mogą być umieszczane w dowolnym miejscu w kodzie programu, pomiędzy innymi elementami składniowymi, i mogą zawierać w sobie całkowicie dowolne znaki, oczywiście z wyjątkiem znaków, które stosowane są jako oznaczenie końca komentarza. Język Object Pascal pozwala nam wybierać spośród trzech różnych oznaczeń komentarzy w kodzie programu:

```
{ Komentarze mogą być zamknięte w nawiasach klamrowych. }
(* Można też stosować takie oznaczenia, ale nie wolno mieszać ze sobą różnych oznaczeń. *)
 x := 0; // Ten komentarz zakończy się "automatycznie" wraz z końcem wiersza
```

Pierwsze dwa rodzaje komentarzy można zagnieżdżać w sobie na dwóch poziomach, pod warunkiem jednak, że dla każdego poziomu stosowane będą inne znaki ograniczające zakres komentarza.

Identyfikatory mogą składać się z liter, cyfr oraz znaku podkreślenia, ale ze względu na konieczność zachowania jednoznaczności zapisu nie mogą się rozpoczynać od cyfry. Bardzo interesującą nowinką wprowadzoną do Delphi 2005 jest to, że po 32 latach istnienia języka Pascal nie muszą składać się one wyłącznie z liter podstawowego zestawu znaków ASCII, ale dopuszczono w nich również znaki alfanumeryczne Unicode. Dzięki temu w identyfikatorach można stosować znaki narodowe, takie jak ą, ę, ó lub ś. Z tej możliwości wykluczone zostały jednak opublikowane (ang. *published*) elementy klas oraz ich typy. W podanym niżej wierszu kodu ukrywa się kolejna reguła języka:

```
Nie_ma_rozróżnienia_pomiędzy_Wielkimi_i_małymi_literami (* . *)
```

Stałe wartości w zależności od typu muszą stosować się do reguł zapisu przedstawionych w tabeli 3.3.

Tabela 3.3. *Reguły zapisu wartości stałych*

Typ	Budowa zapisanej wartości	Przykład
Liczby całkowite	Ciąg cyfr z zakresu od 0 do 9	`9876543210`
Liczby szesnastkowe	Liczba szesnastkowa z umieszczonym na początku znakiem dolara ($)	`$F0D9`
Liczby zmiennoprzecinkowe	Liczba z ułamkiem dziesiętnym + wykładnik	`3.4e5` lub `1.4e-100` lub `49e-4`
Znaki	Znaki zamknięte w apostrofach lub *#+kod znaku*	`'A'` lub (równoważne) `#65`
Ciągi znaków	Ciągi znaków zamknięte w apostrofach	`'XYZ'`
Zbiory	Listy elementów zamknięte w nawiasach prostokątnych	`[biSystemMenu, biMinimize, biMaximize]`

W przypadkach szczególnych, w których konieczne jest zastosowanie znaku ograniczającego wewnątrz ciągu znaków albo jako pojedynczego znaku, wystarczy zapisać dwa takie znaki jeden za drugim. Jeżeli chcielibyśmy zapisać apostrof jako pojedynczy znak, to należałoby zastosować zapis `''''`, natomiast cudzysłów wewnątrz ciągu znaków powinien wyglądać tak: `'"'`.

Słowa kluczowe języka Object Pascal są słowami zarezerwowanymi, których nie można stosować w programach jako identyfikatory. Oto lista słów kluczowych języka:

```
and             array           as              asm             begin           case
class           const           constructor     destructor      dispinterface   div
do              downto          else            end             except          exports
file            finalization    finally         for             function        goto
if              implementation  in              inherited       initialization  inline
interface       is              label           library         mod             nil
not             object          of              or              out             packed
procedure       program         property        raise           record          repeat
resourcestring  sealed          set             shl             shr             static
string          then            threadvar       to              try             type
unit            unsafe          until           uses            var             while
with            xor
```

Częścią języka Object Pascal są również słowa zapisane w następnej liście. Nazywane są one dyrektywami standardowymi, a od słów kluczowych różnią się tym, że nie są zarezerwowane wyłącznie dla kompilatora. Dyrektywy te mogą być używane też jako identyfikatory w tekście programu, pod warunkiem jednak, że nie będą tworzyć żadnych dwuznaczności w kodzie. Takie identyfikatory nie mogą znaleźć się tam, gdzie najczęściej używane są dyrektywy standardowe, czyli na końcach nagłówków funkcji i procedur. Z drugiej strony, dyrektywy standardowe nie mogą być zapisywane w ciele żadnej metody. Słowa `private`, `protected`, `public` i `published` zarezerwowane zostały w deklaracjach klas, dzięki czemu unika się nieporozumień w tym zakresie.

absolute	abstract	assembler	automated	cdecl	contains
default	deprecated	dispid	dynamic	export	external
forward	implements	index	library	local	message
name	nodefault	overload	override	package	pascal
platform	private	protected	public	published	read
readonly	register	reintroduce	requires	resident	safecall
stdcall	stored	varargs	virtual	write	writeonly

Niektóre z tych dyrektyw, takie jak `package` i `requires`, mają jakiekolwiek znaczenie wyłącznie w tekście źródłowym pakietów, dlatego w edytorze Delphi, w tekście zwykłego modułu, w ogóle nie są wyróżniane. Pozostałe wyróżniane w edytorze słowa stosowane w języku Object Pascal, niebędące słowami kluczowymi, to słowa `on` oraz `near` i `far`, które dzisiaj nie mają już żadnego praktycznego znaczenia, ale są obsługiwane w języku w ramach zgodności z poprzednimi wersjami języka.

Podobnie jak w przypadku identyfikatorów, język Object Pascal nie rozróżnia wielkich i małych liter również w słowach kluczowych i dyrektywach standardowych.

Operatory i znaki interpunkcyjne opisywane są w miarę potrzeb w dalszej części rozdziału.

3.5.2. Instrukcje kompilatora

Wewnątrz programu tworzonego w języku zapisywać można nie tylko instrukcje przeznaczone do wykonania w ramach samego programu, ale również instrukcje przeznaczone dla kompilatora. Te ostatnie nie są częścią faktycznego języka programowania, dlatego znaleziono dla nich miejsce, w którym nie można ich pomylić z elementami języka programowania — komentarze. Te specjalne komentarze różnią się jednak od zwyczajnych komentarzy tym, że zaczynają się od znaku dolara (`$`). Na przykład, do włączenia opcji kompilatora o nazwie `BoolEval` należy posłużyć się zapisem `{$BoolEval On}` lub jego formą skróconą `{$B+}`, umieszczając ją wewnątrz kodu źródłowego programu (znaczenie tego przełącznika objaśniam w podpunkcie „Wyliczanie wartości wyrażeń logicznych” z punktu 3.6.2). Wiele z opcji kompilatora można zmieniać nie tylko w podany wyżej sposób w kodzie programu, ale również w oknie dialogowym wywoływanym poprzez pozycję menu *Projekt/Options*. W tym drugim rozwiązaniu ustalone opcje zapisywane są w pliku opcji projektu. Pełną listę opcji kompilatora znaleźć można w systemie aktywnej pomocy Delphi pod indeksem *Compiler directives*.

Włączanie zasobów

Oprócz przedstawionych wcześniej przełączników dostępne są też inne specjalne instrukcje, spośród których najczęściej wykorzystywana jest instrukcja `$R`, zawsze generowana razem z kodem przygotowywanym przez Delphi. W każdym nowo utworzonym pliku projektu znaleźć można taki wiersz:

```
{$R 'WinForm.TWinForm.resources' 'WinForm.resx'}
```

Zapis ten powoduje, że automatycznie generowany plik zasobów *WinForm.resx* kompilowany jest do pliku *WinForm.TWinForm.resources* i włączany do aktualnego kompilatu.

Kompilacja warunkowa

Kompilacja warunkowa umożliwia wygenerowanie kilku wersji danego programu na podstawie jednego pliku z kodem źródłowym. Takie zabiegi opłacają się wtedy, gdy różnice pomiędzy tymi wersjami ograniczają się do zaledwie kilku wierszy. Na przykład, na podstawie aplikacji VCL.NET można przygotowywać wersję działającą w środowisku .NET, wersję dla systemów Windows, a nawet wersję dla Linuksa. Innym przykładem mogą być dodawane do programu dodatkowe instrukcje wspomagające wyszukiwanie błędów, które powinny być usunięte z ostatecznej wersji programu (takie dodatkowe instrukcje przedstawiam na listingu 3.78.

Listing 3.78. *Instrukcje kompilacji warunkowej*

```
{$ifdef WarunekKompilatora}
  ... Kod programu ...
{$else}
  ... Alternatywny kod programu ...
{$endif}
```

W podanym wyżej przykładzie kompilator tylko wtedy skompiluje pierwszą część programu, gdy spełniony będzie `WarunekKompilatora`, a w przeciwnym wypadku skompilowana zostanie druga część kodu (trzeba tu zaznaczyć, że część kodu za dyrektywą `$else` jest opcjonalna). Zamiast instrukcji `$ifdef` można też użyć instrukcji `$ifndef`, która stanowi odwrócenie logiki poprzedniej instrukcji (coś w rodzaju „jeżeli nie"). Symbol `WarunekKompilatora` nie ma nic wspólnego z jakimkolwiek identyfikatorem stosowanym w kodzie źródłowym programu, ale musi być definiowany w całkowicie inny sposób:

- za pomocą instrukcji kompilatora `{$define WarunekKompilatora}`,
- wpisując dany symbol do opcji projektu na zakładce *Directories/Contitionals*, w pole *Conditional defines*,
- wykorzystując do tego sam kompilator. W kompilatorze zdefiniowane są symbole, które umożliwiają rozróżnianie jego wersji i przygotowanie dla każdej z nich osobnego kodu. Wyliczanie kolejnych wersji kompilatorów rozpoczęło się już od Turbo Pascala 1.0, dlatego aktualne systemy rozwojowe już od dawna operują dwucyfrowymi numerami wersji. W tabeli 3.4 przedstawiam najważniejsze symbole zdefiniowane w dotychczasowych produktach firmy Borland, zgodnych w Delphi.

Tabela 3.4. *Symbole kompilatora definiowane w różnych wersjach Delphi*

Produkt	Zdefiniowane symbole
Delphi 1	VER80, Windows, CPU86
Delphi 2	VER90, Win32, CPU386
Delphi 7	VER150, Win32, MSWINDOWS, CPU386
Kylix	Linux, CPU386, wersje od VER140 (Kylix 1)
Delphi 8	VER160, CLR
Delphi 2005 dla .NET	VER170, CLR
Delphi 2005 dla Windows	VER170, CLR, MSWINDOWS, CPU386

Nowsze dyrektywy warunkowe

Od czasu wersji `VER140` kompilator obsługuje jeszcze alternatywny zestaw dyrektyw kompilacji warunkowej: `$if`, `$elseif` i `$ifend`. Za pomocą tych dyrektyw, oprócz sprawdzania prostych symboli, można też kontrolować wartości całych wyrażeń:

```
const MojaStala = 'LN';
{$if MojaStala = 'XA'}
```

W dyrektywach tych można stosować też zadeklarowane stałe języka Object Pascal, takie jak przedstawiona wyżej stała `MojaStala`. Szczególnie ciekawe możliwości daje nam tu możliwość sprawdzania wartości stałej `RTLVersion`, która w Delphi 2005 zadeklarowana została z wartością `17.0` i tym samym odpowiada numerowi wersji kompilatora. Podstawowa różnica w stosunku do prostego sprawdzenia symbolu kompilatora `VER170` polega na tym, że pozwala ona na uwzględnienie też wszystkich przyszłych wersji biblioteki:

```
{$ifdef VER170}         // dotyczy wyłącznie Delphi 2005
{$if RTLVersion >= 17} // warunek będzie spełniony również w przyszłych wersjach Delphi
```

W końcu, Delphi udostępnia nam w połączeniu z dyrektywą `$if` jeszcze dwie specjalne „funkcje": `Declared(MojSymbol)`, pozwalającą skontrolować, czy zadeklarowany został symbol języka Object Pascal, oraz `Defined(SymbolKompilatora)`, która sprawdza, czy zdefiniowany został podany symbol kompilatora. Ta druga funkcja jest praktycznie równoznaczna z dyrektywą `$ifdef`.

Ostrzeżenia ważne dla przenośności programu

Od 14. wersji kompilatora obsługiwane są też trzy nowe dyrektywy, za pomocą których można oznaczać dowolne deklaracje (zmiennych, typów, funkcji, klas lub modułów) jako „niemożliwe do stosowania bez ograniczeń":

- `deprecated` — oznacza deklarację jako przestarzałą, i jednocześnie zaleca stosowanie nowocześniejszej alternatywy.
- `platform` — opisuje deklarację jako zależną od platformy i w ten sposób ostrzega przed ewentualnymi problemami w czasie prób przeniesienia aplikacji na inną platformę.
- `library` — oznacza deklarację jako zależną od stosowanej aktualnie biblioteki.

Takimi dyrektywami oznaczać można również dowolne obiekty programu, stałe i funkcje. Przykład takiego oznaczenia stałej przedstawiam na listingu 3.79.

Listing 3.79. *Oznaczenie stałej jako zależnej od platformy*

```
const
  // Przykład pochodzi z pliku SysUtils.pas: Znacznik ReadOnly
  // jest atrybutem obecnym wyłącznie w systemach plików stosowanych
  // w systemach Windows (w Linuksie dostęp do plików regulowany jest
  // za pomocą uprawnień Użytkownika, Grupy i Wszystkich).
  faReadOnly = $00000001 platform;
```

Oczywiście, podobnie można oznaczać też całe rekordy lub klasy, a nawet całe moduły (odpowiednie deklaracje przedstawiam na listingu 3.80).

Listing 3.80. *Oznaczenie klasy i modułu jako zależnych od platformy*

```
unit ZbiorFunkcjiDlaWindows platform;

type
  // Przykład z pliku SysUtils.pas: Klasa języków obsługiwanych
  // przez system dostępna jest wyłącznie w systemie Windows
  TLanguages = class
  ...
  end platform;
```

Działanie tych dyrektyw polega na tym, że kompilator generuje ostrzeżenia, gdy nasz program próbuje wykorzystać obiekt lub inny element oznaczony taką dyrektywą. Sposób wypisywania tych ostrzeżeń może być sterowany za pomocą dyrektyw kompilatora `$Hints ON/OFF` i `$Warnings` (więcej na ten temat znaleźć można w pomocy Delphi).

3.5.3. Typy i zmienne

Klasy i obiekty są w Delphi wbudowane w funkcjonujące od zawsze w języku Pascal koncepcje zmiennych i typów. Koncepcje te przygotował w roku 1972 Niklaus Wirth w tworzonej przez siebie pierwszej wersji języka Pascal. I tak, klasy stanowią syntaktyczne rozwinięcie rekordów, a w związku z tym obiekty są odpowiednikiem zmiennych, których typowi przypisano klasę tego obiektu. W tym punkcie podsumowane zostaną ogólne reguły dotyczące typów i zmiennych, natomiast w podrozdziale 3.6 przygotujemy przegląd przez wszystkie typy dostępne w Delphi.

Bloki deklaracji

Dzięki strukturze programu w języku Object Pascal, będącej połączeniem poszczególnych sekcji struktury, znacznie łatwiejsze jest odróżnianie w programie zmiennych, typów i stałych niż na przykład w językach C++ i Java. Każdy z tych trzech rodzajów identyfikatorów musi być deklarowany w sekcji kodu źródłowego opatrzonej specjalnym podpisem. Sekcje te mogą pojawiać się w pliku wielokrotnie i w dowolnej kolejności, przy czym zawsze muszą znajdować się poza tymi częściami pliku, które zawierają w sobie kod programu (czyli poza wszystkimi blokami `begin … end`). Na przykład zmienne zapisywane są w sekcjach oznaczonych słowem `var`, tak jak na listingu 3.81.

Listing 3.81. *Deklarowanie zmiennych*

```
var
  PewnaLiczba: Integer;
  Przycisk1, Przycisk2: Char;
begin
...
```

Deklaracja zmiennej przypisuje jednemu lub kilku identyfikatorom zmiennych rozdzielanych przecinkami typ znajdujący się za dwukropkiem. Zastosowane w powyższym listingu typy `integer` i `char` są już zdefiniowane w Delphi.

Deklaracje typów

W każdym pliku formularza znajdziemy zapisaną deklarację typu formularza, czyli klasy formularza, która jest wyjątkowo obszernym przykładem deklaracji typu. Najprostszy sposób zadeklarowania własnego typu polega na przygotowaniu zastępczego identyfikatora dla jednego ze standardowo zdefiniowanych typów, tak jak na listingu 3.82.

Listing 3.82. *Deklarowanie typu jako innej nazwy typu standardowego*

```
TIndeksTabeli = Integer; { deklaruje nowy typ o nazwie TIndeksTabeli }
TIndeksStron = Integer;
TIndeksKsiazki = Byte;
```

Takie samodzielnie zdefiniowane typy stosować można dokładnie tak samo jak typy standardowe:

```
var
  NumerStrony : TIndeksStron;
```

Różnice w zapisach stosowanych w bloku deklaracji typów, w stosunku do bloku deklaracji zmiennych, to głównie słowo `type` rozpoczynające ten blok, możliwość tworzenia tylko jednego identyfikatora w ramach jednej deklaracji i stosowanie w miejscu dwukropka znaku równości. Dzięki stosowaniu przedstawionych wyżej deklaracji typów mamy później możliwość łatwego przestawienia typu `TIndeksKsiazki` z typu standardowego `Byte` na `Word`. W ten sposób uzyskamy większy zakres wartości dla indeksów książek, podczas gdy pozostałe zmienne stosujące typ `Byte` pozostaną niezmienione. Jest to rozwiązanie pozwalające na zaoszczędzenie pracy związanej z wyszukiwaniem i zastępowaniem odpowiednich zapisów w kodzie programu.

Pozostałe bloki deklaracji

Oprócz segmentów `var` i `type` w plikach można stosować jeszcze segmenty `const` (omawiane będą w punkcie 3.5.4), `exports` (stosowane w bibliotekach do eksportowania metod jako funkcji środowiska Win32) oraz `label` (do tworzenia etykiet, do których można w kodzie przeskakiwać za pomocą instrukcji `goto` — w tej książce nie będziemy rozpatrywać tej funkcji języka).

3.5.4. Stałe i zmienne inicjowane

W punkcie 3.5.1 mówiłem już o bezpośrednio podawanych wartościach stałych, jako o jednym z najbardziej podstawowych składników tekstu źródłowego. Takim wartościom stałym również można przypisać nazwy i w ten sposób wykorzystywać je wielokrotnie. Oczywiście, nawet bez takiej nazwy wpisaną jawnie wartość można traktować jak stałą, ale w języku Object Pascal terminem *stała* określane są wyłącznie *identyfikatory* opisujące wartość stałą.

W języku Object Pascal istnieją dwa różne rodzaje stałych. W rzeczywistych stałych danemu identyfikatorowi przypisywana jest wartość stała, której kompilator później używa wszędzie tam, gdzie w kodzie programu zastosowany zostanie ten identyfikator. W tym procesie deklaracja identyfikatora stałej jest całkowicie usuwana przez kompilator Delphi, w związku z czym w powstającym kompilacie środowiska .NET takiego identyfikatora już nie znajdziemy. Zasada tych podmian zobrazowana została na listingu 3.83.

Listing 3.83. *Zastosowanie identyfikatorów stałych w programie*

```
const
  SzerokoscStandardowa = 200;      // Stała SzerokoscStandardowa nie jest
                                   // częścią kompilatu środowiska .NET
begin
  Width := SzerokoscStandardowa; // zapis traktowany jest jak "Width := 200;"
```

Co ciekawe, kompilator pozwala też na stosowanie wyrażeń w deklaracjach stałych, pod warunkiem jednak, że w tych wyrażeniach stosowane będą wyłącznie wartości stałe.

Stałe z przypisanymi typami i zmienne końcowe

Deklaracje stałych nie muszą składać się wyłącznie z przypisywanych do nazw jawnych wartości stałych. Każdej deklarowanej stałej można też przypisać odpowiedni typ:

```
const
  SzerokoscStandardowa: Integer = 200;  // Stała SzerokoscStandardowa jest
                                        // częścią kompilatu środowiska .NET
```

Takie stałe w języku Object Pascal są traktowane podobnie do zmiennych, których wartość nie może być jednak zmieniana po pierwotnej inicjalizacji. Odpowiada to w pełni zmiennym końcowym (ang. *Final variable*) funkcjonującym w środowisku .NET CLR i, jak można się domyślać, kompilator Delphi dla .NET przekształca stałe z przypisanymi typami w zmienne końcowe.

Do czasu Delphi 5 takie deklaracje nie były traktowane przez kompilator jak prawdziwe (czyli niezmienne) stałe i można im było w tekście programu przypisywać nowe wartości. Jeżeli taki kod chcielibyśmy skompilować w podobny sposób w Delphi dla .NET, to musielibyśmy zastosować opcję kompilatora `{$J+}` lub `{$WRITEABLECONST}` albo w oknie dialogowym opcji kompilatora zmienić opcję *Assignable typed constants*.

Zmienne zainicjowane

W Delphi inicjowanie zmiennych już w momencie ich deklaracji dozwolone jest tylko w przypadku zmiennych globalnych. Wartości inicjujące przypisywane są takim zmiennym już w momencie uruchomienia programu, zastępując standardowe wartości zerowe. Składnia takich deklaracji jest identyczna ze składnią stosowaną przy deklarowaniu stałych z przypisanymi typami, a jedyna różnica polega na tym, że umieszczane są one w sekcji `var`:

```
var
  SampleRate: Word = 44100;
```

Lokalne stałe z przypisanymi typami

Zmienne lokalne nie mogą być automatycznie inicjowane. W języku Object Pascal można natomiast deklarować lokalnie, w zakresie danej funkcji, stałe z przypisanymi typami. Taka stała jest wtedy wewnątrz tej funkcji dostępna jako zmienna lokalna, chociaż tak naprawdę będzie ona zainicjowaną zmienną globalną. Obrazujący to wycinek kodu przedstawiam na listingu 3.84.

Listing 3.84. *Tworzenie zainicjowanych stałych lokalnych*

```
{$WRITEABLECONST ON} //<< Wymagane ze względu na instrukcję inc.
procedure AnyProcedure;
const
  CallCounter: Integer = 0;
begin
  inc(CallCounter);
```

W tym przykładzie procedura `AnyProcedure` zlicza swoje wywołania w prywatnej zmiennej `CallCounter`, która przy uruchomieniu inicjowana jest zerem.

Podobnie jak wszystkie inne zmienne globalne, Delphi musi jakoś przedstawić taką zmienną w środowisku .NET CLR. Przeglądając zawartość programu za pomocą narzędzia *Reflection* i rozwijając węzły podrzędne węzła `Unit`, znajdziemy w nim nazwę `@2$AnyProcedure$CallCounter`.

3.5.5. Obszary widoczności i zmienne lokalne

Każdy identyfikator, jaki wykorzystywany jest w języku Object Pascal poprzez przypisanie mu nazwy albo całego wyrażenia, musi zostać najpierw zadeklarowany. Taki identyfikator można stosować od momentu zadeklarowania do końca jego obszaru widoczności. Do obszarów widoczności zaliczane są moduły, funkcje i klasy.

Identyfikatory, które zostały zadeklarowane wewnątrz danej funkcji, procedury lub metody są widoczne wyłącznie w ich obrębie i określane są jako identyfikatory lokalne. Przykład takiego identyfikatora przedstawiam na listingu 3.85.

Listing 3.85. *Identyfikatory lokalne*

```
procedure TForm1.FormCreate(Sender: TObject);
var { Deklarowane będą zmienne lokalne }
  LicznikLokalny: Integer;
begin
  LicznikLokalny := 0; { zastosowanie zmiennej }
...
```

Jeżeli będziemy zagnieżdżać funkcje, to każda funkcja zagnieżdżona będzie miała dostęp do wszystkich wcześniej zadeklarowanych zmiennych lokalnych funkcji obejmującej.

Identyfikatory deklarowane wewnątrz deklaracji klasy dostępne są we wszystkich metodach tej klasy, a dodatkowo widoczne są też w klasach wywiedzionych, choć w tym zakresie trzeba jeszcze uwzględniać atrybuty widoczności (omawiane były w punkcie 3.2.2).

Identyfikatory zadeklarowane poza wymienionymi przed chwilą obszarami nazywane są identyfikatorami globalnymi, które widoczne są w ramach całego modułu, rozpoczynając od wiersza, w którym zostały zadeklarowane.

Zakrywanie

Identyfikatory mogą zostać czasowo zakryte przez identyfikatory zadeklarowane w zagnieżdżonych obszarach widoczności. W przykładzie przedstawionym na listingu 3.86 deklarowana jest zmienna globalna o nazwie `Start`, a zaraz obok niej deklarowana jest zmienna lokalna o tej samej nazwie, zakrywająca zmienną globalną. Mimo zakrycia zmiennej globalnej, nadal możemy uzyskać do niej dostęp bezpośrednio podając zakres jej widoczności. Zakres widoczności zmiennej globalnej ma zawsze nazwę modułu, w którym została ona zadeklarowana.

Listing 3.86. *Zakrywanie zmiennej globalnej przez zmienną lokalną*

```
unit Unit1;
interface
var
  Start: Integer;
implementation

procedure TForm1.FormCreate(Sender: TObject);
var
    Start: Integer;
begin
  Unit1.Start := 0; // zapisuje wartość do zmiennej globalnej
  Start := 0;       // zapisuje wartość do zmiennej lokalnej
...
```

Komentarz dla programujących w językach Java i C#

Kompilator języka Pascal zawsze działa na takiej zasadzie, że tylko jeden raz przegląda zawartość kodu źródłowego programu i w związku z tym nie może rozwiązywać referencji na zmienne, które deklarowane są później, tak jak robi to kompilator języka Java. W Delphi zastosowanie elementów klas możliwe jest dopiero po ich zadeklarowaniu (jedynym wyjątkiem są metody `Set…` i `Get…` stosowane w deklaracjach właściwości), wobec czego pierwsza implementacja metody może zostać zapisana dopiero po zakończeniu deklaracji klasy. Podobnie zmienne lokalne są zasadniczo deklarowane przed słowem kluczowym `begin` rozpoczynającym blok kodu, w którym widoczne mają być te zmienne. Jak widać, fakt, że kompilator tylko raz przegląda tekst źródłowy programu, nie jest tutaj prawie żadnym ograniczeniem — wyjątkiem są tu tylko procedury i funkcje, które nie są deklarowane jako część klasy, ani części interfejsu modułu (czyli stanowią prywatne procedury pomocnicze modułu). W ich wypadku możliwe jest, że zostaną przypadkowo wywołane jeszcze przed ich zdefiniowaniem, co spowoduje wygenerowanie błędu kompilatora. To samo dotyczy prywatnych zmiennych globalnych modułu lub prywatnych deklaracji typów (przy czym „prywatne” oznacza tutaj: niezadeklarowane w części interfejsu modułu).

3.5.6. Atrybuty

Atrybuty są najbardziej podstawową nowością wprowadzoną do Delphi dla .NET. Można je umieszczać przed każdą deklaracją identyfikatora, dodając w ten sposób uzupełniające informacje do deklarowanego identyfikatora, które co prawda nie grają żadnej roli dla kompilatora, ale znaczenia nabierają w czasie działania programu. Każdy atrybut staje się zatem w czasie działania programu obiektem. Na przykład deklaracja przedstawiona na listingu 3.87 łączy obiekt klasy `CategoryAttribute` z właściwością `NoSelection`.

Listing 3.87. *Atrybuty umieszczane przed deklaracjami*

```
[Category(ColorPalCategory)]
property NoSelection: boolean read FNoSelection write FNoSelection;
```

Definicja atrybutu umieszczona w nawiasach kwadratowych jest składniowo zgodna z wywołaniem funkcji, ale wewnętrznie powoduje wywołanie konstruktora nowego obiektu typu `CategoryAttribute`. Wszystkie parametry atrybutów znaleźć można w dokumentacji Delphi. Wystarczy odszukać w niej stronę opisującą klasy `…Attribute` i z niej przejść do stron opisujących konstruktory atrybutów.

Możliwe jest też powiązanie z jednym identyfikatorem wielu atrybutów. Wystarczy rozdzielać je przecinkami, umieszczając wewnątrz jednej pary nawiasów kwadratowych. Atrybuty mogą być też deklarowane tak, że dotyczyć będą tylko określonych typów identyfikatorów, a na dodatek w Delphi można definiować nowe klasy atrybutów i w czasie pracy programu samodzielnie odczytywać obiekty atrybutów poszczególnych identyfikatorów.

Wszystkie te zagadnienia zajęłyby tutaj zbyt dużo miejsca, dlatego w tej książce z atrybutów korzystać będziemy tylko tam, gdzie będą nam one niezbędne do osiągnięcia konkretnych celów. W związku z tym podrozdział ten zakończę kilkoma przykładami zastosowania atrybutów, o których dokładniej mówić będziemy w innych miejscach w książce. W punkcie 2.6.1 atrybuty stosowane były do przekazania serializerowi XML informacji o przekazywanych mu typach:

```
[Serializable(), XmlInclude(TypeOf(AlarmEvent)),
   XmlInclude(TypeOf(DesktopChangeEvent))]
EventList = class(ArrayList)
end;
```

Narzędzia projektowe stosowane w Delphi bardzo szeroko korzystają z atrybutów, na przykład podczas edytowania komponentów i kontrolek na formularzu oraz ustawiania ich właściwości w inspektorze obiektów. Oto bardzo prosty przykład: podany na listingu 3.88 atrybut dopisuje do właściwości `NoSelection` atrybut kategorii, przez co właściwość ta w inspektorze obiektów pojawia się w podanej w atrybucie kategorii właściwości.

Listing 3.88. *Atrybut kategorii inspektora obiektów*

```
[Category('Color palette')]
property NoSelection: boolean read FNoSelection write FNoSelection;
```

W punkcie 6.6.3 wykorzystywanych będzie jeszcze wiele innych atrybutów stosowanych w czasie projektowania.

W punkcie 3.1.3 wymienianych jest kilka atrybutów dotyczących samych kompilatów. W tym miejscu konieczne jest zastosowanie składni rozszerzonej, ponieważ atrybuty te nie dotyczą zapisanego za nimi elementu, ale wpływają na cały tworzony aktualnie kompilat i zapisywane są do jego manifestu:

```
[assembly: AssemblyTitle('Tytuł mojego kompilatu')]
[assembly: AssemblyVersion('1.0.0.0')]
```

3.6. Typy

Podstawowe typy języka Object Pascal mniej więcej odpowiadają różnym typom wartości właściwości wyświetlanych w inspektorze obiektów, a dodatkowo umożliwiają tworzenie kolejnych wariacji typów danych.

3.6.1. Typy proste

Do *typów prostych* należą na przykład typy liczb zmiennoprzecinkowych oraz typy umożliwiające prezentowanie wartości jako liczb całkowitych, czyli tak zwane *typy porządkowe* (ang. *Ordinal types*). Najważniejszą grupą typów porządkowych są typy z rodziny `integer`.

Typy liczb całkowitych

Wszystkie typy liczb całkowitych (`integer`) różnią się od siebie zapotrzebowaniem na pamięć oraz faktem, że jeden z bitów może przechowywać informację o znaku liczby (jeżeli dany typ tego bitu nie obsługuje, to może przechowywać wyłącznie liczby dodatnie). Wszystkie typy liczb całkowitych przedstawiam w tabeli 3.5.

Tabela 3.5. *Typy liczb całkowitych*

Typ	Zakres wartości	Wielkość	klasa .NET
`ShortInt`	-128 … 127	1 bajt	`SByte`
`Byte`	0 … 255	1 bajt	`Byte`
`SmallInt`	-32768 … 32767	2 bajty	`Int16`
`Word`	0 … 65535	2 bajty	`UInt16`
`LongWord`	0 … 42949672950	4 bajty	`UInt32`
`LongInt`	-2147483648 … 2147483647	4 bajty	`Int32`
`Int64`	$-2^{63} \cdots 2^{63}-1$	8 bajtów	`Int64`
Typy ogólne:			
`Integer`	-2147483648 … 2147483647	4 bajty	`Int32`
`Cardinal`	0 … 42949672950	4 bajty	`UInt32`

Typy `Cardinal` i `Integer` nie mają ustalonej z góry wielkości, ale dopasowują się do aktualnej architektury procesora (na przykład w Delphi 2 w momencie przejścia z 16-bitowej na 32-bitową architekturę systemów Windows wielkość tych dwóch typów podwoiła się z dwóch do czterech bajtów) i dlatego nazywane są one typami *ogólnymi* (ang. *Generic types*), w przeciwieństwie do typów *fundamentalnych* (ang. *Fundamental types*), które na pewno nie będą się zmieniać w różnych architekturach procesorów. Najprawdopodobniej w przyszłych architekturach 64-bitowych ogólne typy liczb całkowitych nie będą już rozbudowywane, dlatego każdy, kto chce korzystać z 64-bitowych liczb, powinien już teraz korzystać bezpośrednio z typu `Int64`.

Typy Int64 i LongWord w praktyce

Typy `LongWord` i `Int64` można w zasadzie stosować dokładnie tak samo jak pozostałe typy całkowite. Typy `LongWord` i `LongInt` zachowują się dokładnie tak samo jak typy `Word` i `SmallInt`, dlatego przyjrzymy się teraz szczególnemu zachowaniu typu `Int64`. Dobra wiadomość jest taka, że od czasu wprowadzenia typu `Int64` kompilator pozwala też na podawanie bezpośrednich stałych tego typu, czyli akceptuje liczby całkowite z nawet osiemnastoma zerami, takie jak poniższa:

```
ShowMessage('Liczba Int64: ' + IntToStr(1000000000000000000));
```

Kolejny przykład wskazuje na to, że stosując typ `Int64` trzeba zawsze pamiętać o tym, że standardowy typ liczby całkowitej w Delphi ma tylko szerokość 32-bitów. Poniższa instrukcja w programie konsolowym spowoduje błąd przepełnienia:

```
Writeln('Wynik testu Int64: ' + Int64(1000000000 * 1000000000).ToString);
```

Problem polega na tym, że Delphi każdą z podanych jawnie liczb interpretuje jako liczbę 32-bitową, ponieważ liczby te można przedstawić w takiej postaci. W związku z tym kompilator przygotuje na wynik mnożenia kolejną zmienną 32-bitową, chociaż tym przypadku potrzebna byłaby akurat liczba 64-bitowa. Problem ten rozwiązać można przekształcając obie liczby w wartości typu `Int64` jeszcze przed wykonaniem mnożenia:

```
Writeln(Int64(Int64(1000000000) * Int64(1000000000)).ToString);
```

Typ Currency

Typ `Currency` (waluta) może przechowywać wartości do 922 337 203 685 477,5807 i podobnie w zakresie liczb ujemnych, wobec czego nadaje się do wykonywania najdziwniejszych obliczeń finansowych. Mimo że pozwala on zapisywać wartości po przecinku dziesiętnym, to jednak nie jest to typ zmiennoprzecinkowy, ponieważ może obsłużyć najwyżej cztery pozycje po przecinku.

Jeżeli zapomnimy na chwilę o przecinku dziesiętnym, to zauważymy, że typ `Currency` jest wartością 64-bitową, która po prostu umożliwia zapisanie wartości do dziesięciotysięcznej części jednostki monetarnej. Obliczenia, w których stosowane są wyłącznie wartości typu `Currency`, mogą być wykonywane w procesorze tak jak zwyczajne operacje na liczbach całkowitych (choć na procesorach 32-bitowych będą one nieco wolniejsze niż „naturalne” dla nich operacje 32-bitowe).

Typ Currency może być wykorzystywany w dokładnie taki sam sposób jak wszystkie pozostałe typy, co oznacza, że można na nich wykonywać te same operacje, a w razie potrzeby — zamieniać go w wartości zmiennoprzecinkowe (co może spowodować pewną utratę dokładności). Oczywiście można posługiwać się tym typem tak, jakby był on typem wartości zmiennoprzecinkowych i w podobny sposób łączyć z typami liczb całkowitych (zmieniając liczbę całkowitą w liczbę zmiennoprzecinkową albo typ Currency przekształcając w liczbę całkowitą wywołaniami funkcji Trunc lub Round).

Wskazówka do środowiska .NET

Typ Currency występuje wyłącznie w Delphi i implementowany jest w systemowym module Delphi w specjalnej klasie. Typ ten nie jest identyczny z klasą Decimal pochodzącą ze środowiska .NET, która również stosowana jest w matematycznych operacjach finansowych, ale zajmuje w pamięci 96, bitów przez co udostępnia miejsce dla jeszcze dziewięciu dodatkowych cyfr znaczących. Zakres liczb obsługiwanych przez typ Currency w środowisku .NET dokładnie odwzorowuje typ System.Data.SqlTypes.SQLMoney.

Typy liczb zmiennoprzecinkowych

Typy liczb zmiennoprzecinkowych stosowane w Delphi są dokładnymi odpowiednikami typów zmiennoprzecinkowych stosowanych przez procesory firmy Intel. Standardowym typem zmiennoprzecinkowym jest typ Double, a typy dostępne oprócz niego wykorzystują 4 bajty pamięci (Single) lub 10 bajtów (Extended) i przez to uzyskują inną dokładność i zakres przechowywanych liczb. Specyfikacje tych typów przedstawiam w tabeli 3.6.

Tabela 3.6. *Typy zmiennoprzecinkowe w Delphi*

Typ	Zakres	Dokładność	Wielkość	Klasa .NET
Single	$1{,}5*10^{-45}$ … $3{,}4*10^{38}$	7.–8. pozycja	4 bajty	Single
Double	$5*10^{-324}$ … $1{,}7*10^{308}$	15.–16. pozycja	8 bajtów	Double
Extended	$3{,}4*10^{-4932}$ … $1{,}1*10^{4932}$	19.–20. pozycja	10 bajtów	Double
Real	Typ ogólny, w Delphi 2005 odpowiada typowi Double			

Znaki: AnsiChar, WideChar i Char

Ogólny typ opisujący pojedynczy znak w języku Pascal nosi nazwę Char, przy czym odpowiada on typowi WideChar (zapisuje on znak w dwóch bajtach) istniejącemu w środowisku .NET oraz typowi AnsiChar dostępnemu w środowisku Win32 (ten typ zapisuje znak w jednym bajcie).

W typie AnsiChar zapisane mogą być wszystkie znaki z zestawu ANSI (względnie ASCII, jeżeli odczytywane są stare pliki z systemu DOS), a w stałych typu char znaki zapisywane są pomiędzy dwoma apostrofami. Te znaki, których nie można znaleźć na klawiaturze, należy podawać z zastosowaniem znaku krzyżyka (#) i kodu liczbowego, na przykład #10.

Typ `WideChar` stosowany jest do zapisywania znaków z zestawu znaków Unicode. Pierwszych 256 znaków Unicode jest identycznych ze znakami zestawu ANSI, wobec czego do zmiennych typu `WideChar` można bez żadnych problemów przypisywać wartości typu `Char`. Podobnie, konwersja znaków z `WideChar` na `Char` również może przebiegać bez żadnych strat, pod warunkiem, że w konwertowanym ciągu nie ma znaków spoza pierwszych 256 znaków Unicode.

Typy wyliczeniowe

W języku Pascal funkcjonują dwa typy proste, które nie są definiowane żadnym słowem kluczowym, ale za pomocą specjalnie przygotowanej definicji: *typy zbiorów* i *typy wyliczeniowe*.

Typy wyliczeniowe stosowane są w wielu właściwościach standardowych komponentów biblioteki VCL.NET. Na przykład właściwość `WindowState` klasy `TForm` została zdefiniowana z wykorzystaniem następującego typu wyliczeniowego:

```
type
  TWindowState: (wsNormal, wsMinimized, wsMaximized);
```

Jak widać, typ wyliczeniowy składa się z listy nazwanych stanów, które można przypisywać zmiennej danego typu. Kompilator wewnętrznie obsługuje wszystkie te wartości w postaci stałych, którym wartości przypisywane są automatycznie. W powyższym przykładzie stała `wsNormal` otrzyma wartość `0`, a stała `wsMaximized` wartość `2`.

Najczęściej w typach wyliczeniowych w bibliotece VCL pierwsze znaki nazw poszczególnych stałych są skrótami od nazwy typu, na przykład skrót `ws` powstał z nazwy właściwości `WindowState`. W ten sposób unika się konfliktów z innymi typami wyliczeniowymi, w których również mógłby wystąpić stan o nazwie `Normal`.

W przeciwieństwie do praktyk stosowanych w językach C# i Java, w języku Object Pascal nie trzeba podawać nazwy typu wyliczeniowego przed wpisywaną nazwą stanu. Różnice pomiędzy praktykami stosowanymi w tych językach przedstawiam na listingu 3.89.

Listing 3.89. *Różnice w stosowaniu typów wyliczeniowych w różnych językach*

```
var
  ws: TWindowState;                 // Stan okna formularza VCL
begin
  ws := wsNormal;                   // Taki zapis jest prawidłowy w języku Object Pascal
  ws := TWindowState.wsNormal;      // W C# potrzebna jest też nazwa typu
```

Firma Borland ze względu na konieczność zapewnienia zgodności z poprzednimi wersjami Delphi zapewniła poprawność pierwszego z powyższych zapisów wyłącznie w programach tworzonych z języka Object Pascal.

Jeżeli jednak stosowalibyśmy typy wyliczeniowe pobrane z biblioteki klas środowiska .NET, to potrzebną nam wartość musimy zawsze poprzedzać nazwą typu, tak jak w przykładzie z listingu 3.90.

Listing 3.90. *Stosowanie typów wyliczeniowych z biblioteki klas środowiska .NET*

```
var
  // Stan okna formularza Windows-Forms
  ws : System.Windows.Forms.FormWindowState;
begin
  ws := FormWindowState.Maximized;
  // lub:
  ws := System.Windows.Forms.FormWindowState.Maximized;
```

Mimo tego wszystkiego nie możemy jednak zapominać o tym, że zarówno typ `Borland.Vcl.Forms.TWindowState`, jak i typ `System.Windows.Forms.FormWindowState` na poziomie środowiska CLR są równoprawnymi typami wyliczeniowymi wywodzącymi się z bazowej klasy `Enum`.

Typy wyliczeniowe w środowisku .NET

Za pomocą klasy `Enum` można zamienić poszczególne wartości, jakie przyjmować ma typ wyliczeniowy, w ciągi znaków, które nadają się do wyświetlania tych wartości w interfejsie użytkownika. Statyczna metoda `Enum.GetValues` przede wszystkim zwraca tablicę, w której zapisane są wszystkie wartości wyliczenia opakowane w osobne obiekty, natomiast metoda `Enum.GetName` zwraca ciąg znaków z nazwą danej wartości. Na początek potrzebny będzie nam obiekt typu `Type`, który przechowuje informacje o typie kontrolowanego typu wyliczeniowego. Najprostszą metodą uzyskania takiego obiektu jest wywołanie wbudowanej w kompilator funkcji `typeof`.

W wycinku programu podanym na listingu 3.91 każdej wartości typu wyliczeniowego `FormBorderStyle` przypisywany jest jeden przełącznik:

Listing 3.91. *Wypisanie wartości typu wyliczeniowego*

```
var
  EnumType: System.Type;
  values: System.Array;
  i: Integer;
  rb: RadioButton;
begin
  EnumType := typeof(System.Windows.Forms.FormBorderStyle);
  values :=  Enum.GetValues(EnumType);  // Uzyskanie tablicy wartości
  // przeglądanie tablicy:
  for i := 0 to values.GetLength(0)-1 do begin
    rb := RadioButton.Create;
    rb.Text := Enum.GetName(EnumType, values.GetLength(i));
    rb.Bounds := Rectangle.Create(8, i*16+8, 200, 12);
    GroupBox1.Controls.Add(rb);
  end;
end;
```

Równomierne rozłożenie tworzonych automatycznie przełączników, takie jak w powyższym przykładzie, uzyskać można stosując komponent `GroupBox` (opisywany będzie w punkcie 6.7.1). Komponent ten może też samodzielnie przygotowywać przełączniki dla wartości typu wyliczeniowego, jeżeli taki typ zostanie przypisany do właściwości `EnumType` komponentu.

Wartości porządkowe stałych wyliczenia

Od roku 2001 wszystkie kompilatory Delphi tworzone przez firmę Borland (Delphi 6, Kylix1) pozwalają też samodzielnie przypisywać wartości do poszczególnych stałych wyliczenia, tak jak na listingu 3.92.

Listing 3.92. *Samodzielnie wybierane wartości stałych wyliczenia*

```
type
  TBits = (bPierwszyBit = $01, bDrugiBit = $02,
           bTrzeciBit = $04, bCzwartyBit = $08);
```

Za pomocą typów wyliczeniowych można też przeprowadzać obliczenia albo przeglądać wszystkie wartości wyliczenia w pętli, wystarczy tylko wykorzystać porządkową naturę tych typów (o typach porządkowych mówić będę na stronie 362).

Typy zakresów częściowych

Definiując typ zakresu częściowego informujemy kompilator o tym, że w danym typie dopuszczalny jest ograniczony zakres wartości. Przykładową definicję typu zakresu częściowego przedstawiam na listingu 3.93.

Listing 3.93. *Definicje typu zakresu częściowego*

```
type
  ArrayIndex = 0..55;
  Temperature = -200..10000;
```

Kompilator będzie kontrolował, żeby zmiennym o tym typie przypisywane były wartości wyłącznie ze zdefiniowanego w nim zakresu. Jeżeli zmiennej przypisywany będzie wynik wyrażenia, który określany jest dopiero w czasie działania programu, to kompilator nie będzie mógł przewidywać, czy wynik tego wyrażenia będzie zawierał się w zakresie typu. W związku z tym możemy skorzystać z opcji kompilatora `$R+`, włączającej sprawdzanie zakresów, przez co zakresy sprawdzane będą także w czasie działania programu, a w przypadku ich przekroczenia wywoływany będzie wyjątek `ERangeError`.

Typy zakresów częściowych są specjalną funkcją dostępną wyłącznie w Delphi, dlatego środowisko CLR **nie** może wykrywać ewentualnych przekroczeń dopuszczalnego zakresu.

Typy logiczne

Typ logiczny (`boolean`) można traktować jak predefiniowany typ wyliczeniowy zawierający tylko dwie wartości. Są to wartości prawdy (`True`) i fałszu (`False`), które powstają w czasie określania wartości wyrażeń logicznych, takich jak `(x < MaxX) and (y < MaxY)`.

Typy porządkowe

Wszystkie typy proste z wyjątkiem typów zmiennoprzecinkowych i typu „wielkich liczb całkowitych" `Int64` określane są jeszcze jedną nazwą: *typy porządkowe* (ang. *Ordinal types*). Ich najważniejszą cechą wspólną jest to, że mogą być bardzo łatwo przekształcane w liczby całkowite, o ile już nimi nie są. Taką dodatkową wartość liczbową zmiennej porządkowej uzyskać można poprzez wywołanie funkcji `ord`. Za poszczególnymi typami porządkowymi ukrywają się następujące liczby:

- liczba porządkowa znaku z zestawu znaków ANSI lub zestawu znaków Unicode (dla typu `WideChar`),
- w przypadku zmiennych logicznych wartość `True` reprezentuje `1`, a wartość `False` — `0`,
- stałe danego typu wyliczeniowego standardowo numerowane są od zera w górę.

Oczywiście można też wykonywać operacje odwrotne i wartości liczbowe zamieniać w pozostałe typy porządkowe, na przykład wyrażenie `Boolean(1)` zwróci nam logiczną wartość prawdy — `True`.

Funkcje High i Low

Z typami porządkowymi wiążą się jeszcze dwie ważne funkcje: `High` i `Low`. Wywołując funkcję `High(`*`IdentyfikatorTypu`*`)` lub `High(`*`IdentyfikatorZmiennej`*`)`, otrzymujemy najwyższą wartość dostępną w zakresie wartości podanego typu lub zmiennej. Na przykład, wywołanie `High(Boolean)` zwraca wartość `True`, a funkcja `High` wywołana z przekazanym typem `TWindowState` na dzień dzisiejszy zwraca wartość `wsMaximized`. Odpowiednio, najmniejszą wartość zakresu uzyskać można wywołując funkcję `Low`.

Stosując te dwie funkcje można bardzo łatwo przejrzeć wszystkie wartości każdego typu wyliczeniowego, co pokazano na listingu 3.94.

Listing 3.94. *Wykorzystanie funkcji High i Low*

```
type
  TWartosci = (Zero, Jeden, Dwa);
   { Jeżeli powyższy typ zostanie zmieniony na przykład tak:
     "TWartosci = (Zero, Jeden, Dwa, Trzy, Cztery);"
     to przedstawiony niżej kod nie musi być modyfikowany }
var
  Licznik: TWartosci;
begin
  for Licznik := low(TWartosci) to high(TWartosci) do
  ... { Tutaj zmienna Licznik przyjmuje wartości 'Zero', 'Jeden' i 'Dwa' }
```

Reguły zgodności typów

W języku Object Pascal obowiązuje wiele reguł określających *zgodność typów* danych, które na przykład pozwalają w jednym wyrażeniu obliczeniowym stosować zmienne liczb całkowitych o kilku różnych wielkościach (na przykład `Byte` i `Integer`) albo

zmienne zmiennoprzecinkowe o różnych wielkościach. Pozostałe reguły dotyczą między innymi zgodności różnych typów procedur i rekordów, ale wykorzystywane są na tyle rzadko, że pozwolę sobie w tym zakresie odesłać czytelnika do specyfikacji języka Object Pascal.

Zupełnie innym i znacznie poważniejszym kryterium jest tu zgodność przypisań, która na przykład nie występuje w czasie, gdy próbujemy „upchnąć" wartość typu `Int64` w zmiennej typu `Byte`, ponieważ doprowadziłoby to do zniszczenia tej wartości, jeżeli byłaby ona większa niż `255` lub mniejsza niż `0`. W związku z tym kompilator zawsze dba o to, żeby w przypisaniach prawa strona zawsze pasowała do lewej strony (w razie konieczności dokonuje też automatycznej konwersji wartości całkowitych w wartości zmiennoprzecinkowe).

Konwersja wartości

Doskonałym przykładem konwersji wartości jest próba przypisania zawartości większej zmiennej do mniejszej zmiennej. Na przykład wiedząc, że w jednej zmiennej typu `Word` z całą pewnością nie znajdą się wartości większe niż `1000`, możemy podzielić jej wartość przez `10` i spokojnie dokonać konwersji na typ `Byte`. Docelowy typ konwersji stosuje się w takiej sytuacji podobnie jak funkcję, a konwertowaną wartość podaje się w nawiasach, tak jak na listingu 3.95.

Listing 3.95. *Konwersja typów wartości*

```
ZmiennaTypuByte := Byte(ZmiennaTypuWord div 10);
Znak := AnsiChar(KodZnaku);    // Zamiana liczby w znak
Falsz := Boolean(0);           // Zamiana liczby w wartość logiczną
```

Traktowanie zmiennych typów prostych jak obiektów

Na poziomie środowiska CLR obiektami są nawet najprostsze typy danych języka Object Pascal i w związku z tym dziedziczą na przykład metodę `ToString` pochodzącą z klasy `System.Object` lub `TObject`. I rzeczywiście, Delphi pozwala na wywołanie metody `ToString` na rzecz zmiennej typu `Integer`, a także wszystkich innych zmiennych pozostałych typów. Teoretycznie możliwe jest też wywoływanie takich metod na rzecz wartości podawanych bezpośrednio, przy czym najpierw trzeba wykonać jawną konwersję podawanej wartości do odpowiedniej klasy typu, na przykład: `Int16(145).ToString` lub `Double(0.944).ToString`.

Takie wywołania mają sens tylko wtedy, jeżeli podawane wartości mają być konwertowane na format ustalany dopiero w czasie działania programu, bo w przeciwnym wypadku można jawnie podać odpowiedni ciąg znaków, taki jak `'0.944'`. Więcej informacji na ten temat uzyskać można w dokumentacji Delphi, przeglądając opisy przeciążonych wersji metody `ToString` obsługujących parametry, jakie opisywać będę w podpunkcie „Formatowanie ciągów znaków" ze strony 373.

3.6.2. Operatory i wyrażenia

W języku Object Pascal operatory arytmetyczne czterech podstawowych działań matematycznych, a także operatory porównania wartości wyglądają dokładnie tak samo jak znaki, do których już wszyscy przywykliśmy. Jedynie operatory nierówności (<>), mniejszości lub równości (<=) i większości lub równości (=>) muszą być składane z dwóch znaków, co wynika z braku odpowiednich znaków na klawiaturze.

Większość pozostałych operatorów to słowa kluczowe, które są bardzo łatwe do zapamiętania z powodu nazw dokładnie opisujących pełnione przez nie funkcje.

Priorytety operatorów

W zależności od swojego priorytetu, operatory podzielone zostały na cztery poziomy priorytetów (priorytety te podaję w tabeli 3.7). Jeżeli w wyrażeniu stosowanych jest kilka operatorów o jednakowym priorytecie, to zapisane w nich operacje wykonywane są w kolejności od lewej do prawej. Jeżeli chcielibyśmy, aby obliczenia wykonywane były w innej kolejności niż ta standardowa, to należy skorzystać z odpowiednio umiejscowionych nawiasów.

Tabela 3.7. *Priorytety operatorów arytmetycznych*

Priorytet	Operator	Opis
1	`()`	nawiasy okrągłe (w języku Pascal nie są traktowane jak operatory)
2	`@, not`	operatory jednoargumentowe
3	`*, /, div, mod, and, shl, shr`	operatory mnożące
4	`+, -, or, xor`	operatory sumujące
5	`<>, <, >, =, <=, >=, in`	operatory porównujące

Specjalne operatory liczb całkowitych

Niklaus Wirth — twórca języka Pascal — wpadł na ciekawe rozwiązanie związane z dzieleniem liczb całkowitych. Do wykonania takiego dzielenia nie wykorzystuje się standardowego symbolu dzielenia (`/`), ale używać należy specjalnego operatora `div`. Resztę z tak wykonanego dzielenia uzyskać możemy za pomocą operatora `mod` (na przykład `7 mod 3 = 1`).

Operatory bitowe

Do łączenia ze sobą liczb za pomocą operacji wykonywanych na poziomie pojedynczych bitów wykorzystywane są operatory, których nazwa określa już rodzaj wykonywanej operacji. Wszystkie rodzaje operatorów bitowych podaję w tabeli 3.8.

Tabela 3.8. *Operatory bitowe w języku Pascal*

Operator	Połączenie	Przykład
`not`	bitowa negacja	`not $F0 = $0F`
`and`	bitowy iloczyn logiczny (operacja i)	`$0F and $11 = $01`
`or`	bitowa suma logiczna (operacja lub)	`$40 or $04 = $44`
`xor`	alternatywa wykluczająca	`$A8 xor $A0 = $08`
`shl`	przesuwa wszystkie bity w lewo	`$01 shl 4 = $10`
`shr`	przesuwa wszystkie bity w prawo	`$20 shr 2 = $08`

Operatory logiczne

Operatory `not`, `and`, `or` i `xor` można też wykorzystać do łączenia ze sobą wartości logicznych i w tym zastosowaniu oznaczają operacje negacji, iloczynu i sumy logicznej, a operator `xor` oznacza mniej więcej operację „jeden z dwóch". W wyniku tych operacji na podstawie dwóch wartości logicznych tworzona jest trzecia wartość logiczna. Na przykład wyrażenie:

```
Button1.Checked and Button2.Checked
```

sprawdza, czy zaznaczone zostały dwa przyciski specjalne, i może być wykorzystane w instrukcji kontrolującej przebieg programu albo zostać przypisane do zmiennej logicznej.

Wyliczanie wartości wyrażeń logicznych

Wyliczanie wartości wyrażeń logicznych może być realizowane na kilka sposobów. Jeżeli częścią wyrażenia są wywołania funkcji, a w przedstawianych tu przykładach każdy identyfikator oznaczać będzie właśnie funkcję, to wyliczanie takiego wyrażenia może mieć wielki wpływ na prędkość działania programu:

```
if True or PolgodzinneObliczenia then
  ...
```

…albo na logikę programu:

```
if PustyNosnik and PozwolenieUzytkownika and FormatujNosnik
  then WyswietlenieKomunikatu('Nośnik został sformatowany.');
```

Jeżeli prawdziwy jest pierwszy operand operacji `or`, tak jak w pierwszym przykładzie, to z góry wiadomo, że całe wyrażenie będzie miało wartość prawdy, nawet jeżeli drugi operand będzie fałszywy. W takim razie, jeżeli w drugiej części wyrażenia wykonywane mają być czasochłonne, ale w takiej sytuacji niepotrzebne już obliczenia, to dobrym rozwiązaniem jest zaniechanie wykonywania tych obliczeń i zwrócenie wyłącznie wartości `True` z pierwszej części wyrażenia, przez co wykonane zostaną instrukcje po słowie kluczowym `then`.

W drugim przykładzie pełne wyliczanie wartości wyrażenia mogłoby mieć też katastrofalne skutki. Jeżeli użytkownik w ostatniej chwili rozmyśliłby się i chciał zablokować formatowanie nośnika, w wyniku czego funkcja `PozwolenieUzytkownika` zwróciłaby

wartość `False`, ale program mimo to wykonałby ostatnią funkcję tego wyrażenia (`FormatujNosnik`), to mimo protestów użytkownika nośnik i tak zostałby sformatowany.

Metoda wyliczania wyrażeń stosowana przez kompilator uzależniona jest od ustawień opcji *Complete boolean eval* dostępnej w oknie dialogowym opcji kompilatora lub ustawienia dyrektywy kompilatora `$B`. Domyślnie kompilator stara się zakończyć wyliczanie wyrażenia tak szybko, jak tylko jest to możliwe, przez co oba podane wyżej przykłady okazują się bardzo sensowne.

Czasami przydaje się też włączenie pełnego wyliczania wartości wyrażenia. Podany niżej przykład wyrażenia kompilowany jest z założeniem pełnego wyliczania wartości, co wynika z zastosowania opcji kompilatora `$B+`:

```
{$B+}
  CalyTestOK := CDROM_Dziala and TwardyDyskOK;
```

W podanym przykładzie testowane mają być napędy CD-ROM oraz twarde dyski zainstalowane w systemie. Jeżeli w czasie testowania napędu CD-ROM i jego sterownika wykryta zostanie jakaś nieprawidłowość, to co prawda ogólny wynik testów nie może już ulec zmianie, ale użytkownik mimo to otrzyma jeszcze informacje o prawidłowym (lub nie) działaniu dysków twardych (nastąpi to w efekcie wykonania funkcji `TwardyDyskOK`).

Operatory porównania

Operatory porównania również zwracają w wyniku wartości logiczne. Trzeba jednak pamiętać o tym, że mają one niższy priorytet niż przedstawione wyżej operatory logiczne, w związku z czym w przykładzie przedstawionym na listingu 3.96 konieczne jest zastosowanie nawiasów, w przeciwnym razie bowiem kompilator połączyłby najpierw wartości zmiennych `MaxX` i `Y` operacją sumy logicznej (`or`), a potem zgłosił błąd wynikający z nieprawidłowego zastosowania operatora większości (`>`).

Listing 3.96. *Operatory porównania mają mniejszy priorytet od operatorów operacji logicznych*

```
if (X > MaxX) or (Y > MaxY) then
  raise EIndexOutOfRange.Create('X lub Y'+
        ' poza dopuszczalnym zakresem wartości');
```

Operatory i inne typy

Część prezentowanych do tej pory operatorów ma naturę polimorficzną i mogą być traktowane podobnie do funkcji wirtualnych. Operatory te powodują wykonanie różnych operacji, w zależności od typu, na jakim mają działać. Z takimi operatorami mogą być łączone też typy danych niebędące zwykłymi typami porządkowymi albo zmiennoprzecinkowymi, takie jak zbiory lub ciągi znaków.

Operatory przeciążone w środowisku .NET

Środowisko CLR pozwala też na stosowanie przeciążonych operatorów wobec samodzielnie zdefiniowanych klas, a biblioteka FCL korzysta z tej możliwości dość często.

Przykładem może być tutaj klasa `DateTime`, w której przeciążone operatory porównania wykorzystane zostały do porównywania dwóch dat lub czasów, a oprócz tego zdefiniowane zostały też przeciążone operatory dodawania i odejmowania.

Język Object Pascal również oferuje pełną obsługę operatorów przeciążonych, dzięki czemu określenie daty, jaka będzie za sto dni, można zrealizować za pomocą kodu przedstawionego na listingu 3.97.

Listing 3.97. *Przeciążone operatory w operacjach na datach*

```
var
  date: DateTime;
  span: TimeSpan;
begin
  date := DateTime.Now;
  span := TimeSpan.FromDays(100);
  date := date + span;
  MessageBox.Show('Za sto dni będzie: '+date.ToString('dd.MM.yyyy'));
```

Działanie operatora dodawania wykorzystanego w powyższym przykładzie definiowane jest wyłącznie wewnątrz klasy `DateTime`. Wynika z tego, że przy każdym wykorzystaniu operatorów z dowolnymi obiektami należy skonsultować się z dokumentacją tej klasy, sprawdzając w niej działanie operatora.

3.6.3. Tablice

Tablice są „indeksowanym uszeregowaniem" w pamięci wartości tego samego typu. W języku Object Pascal dostępne są też indeksowane właściwości (mówiłem o nich w punkcie 3.2.4), które dla użytkownika wyglądają dokładnie tak samo jak tablice. Dostęp do poszczególnych elementów tablic i indeksowanych właściwości realizowany jest w taki sam sposób: za nazwą tablicy lub właściwości podawany jest indeks elementu zamknięty w nawiasach kwadratowych. Indeksy nie muszą być włącznie liczbami; tablice mogą być indeksowane dowolnymi typami porządkowymi, a w przypadku właściwości dopuszcza się indeksowanie dowolnym typem.

W przykładzie przedstawionym na listingu 3.98 definiowane są trzy tablice, wśród których pierwsza wykorzystuje chyba najczęściej stosowane indeksy typu `integer`, elementy drugiej indeksowane są typem wyliczeniowym, a w trzeciej użyty został typ zakresu częściowego.

Listing 3.98. *Do indeksowania tablic można stosować różne typy porządkowe*

```
type
  // dwa samodzielnie zdefiniowane typy porządkowe
  SkladowaKoloru = (czerwony, zielony, niebieski);
  IndeksyTabeli = 1..100;
var
  WartosciPomiarow: array[0..1000] of Double;
  Kolor: array[SkladowaKoloru] of Integer;
  ArkuszKalkulacyjny: array[IndeksyTabeli, IndeksyTabeli] of Integer;
```

Jak widać, pierwszemu elementowi tablicy można nadać indeks 0, 1 lub dowolny inny z dopuszczalnego zakresu wartości indeksów. Na listingu 3.99 przedstawiam zatem sposoby odwoływania się do jednego z elementów trzech tablic zdefiniowanych na listingu 3.98.

Listing 3.99. *Sposoby uzyskiwania dostępu do elementów różnie indeksowanych tablic*

```
WartosciPomiarow[0] := WartoscPoczatkowa;
SkladowaZielona := Kolor[zielony];
ArkuszKalkulacyjny[5, 6] := '4*b';
```

Tablice dynamiczne

Język Object Pascal oferuje szczególną obsługę tablic dynamicznych, czyli tablic, dla których pamięć rezerwowana jest dopiero w czasie działania programu. Tablica dynamiczna deklarowana jest tak:

```
var
  intArray1: array of Integer;
```

Podobnie jak w przypadku dynamicznych obiektów, które odkładane są na stertę, musimy się specjalnie upewnić, że na tablicę przygotowana zostanie wystarczająca ilość pamięci. Odpowiednią procedurę przedstawiam na listingu 3.100.

Listing 3.100. *Na tablicę dynamiczną trzeba jawnie zarezerwować pamięć*

```
begin
  intArray1[1] := 100;  // << Błąd, pamięć nie jest jeszcze zarezerwowana
  SetLength(intArray1, 100);
  intArray1[100] := 100;  // << Błąd, dostępne są indeksy od 0 do 99
  intArray1[0] := 100;  // Wartość zapisywana jest do pierwszego elementu
```

Najciekawsze w tym wszystkim jest to, że funkcję SetLength można wywoływać wielokrotnie i w ten sposób dostosowywać wielkość tablicy do aktualnych potrzeb. W czasie takich zmian wielkości tablicy jej zawartość może być „uszkodzona" tylko w przypadku zmniejszania jej rozmiaru.

Szczególną nowością wprowadzoną w środowisku .NET w związku z dynamicznymi tablicami jest fakt, że teraz już funkcje samego „systemu operacyjnego" (w tym miejscu mam na myśli bibliotekę FCL) zwracają łatwe w obsłudze tablice dynamiczne, a nie wiele powiązanych ze sobą wskaźników na różne obszary pamięci, które zwracane są przez funkcje Windows API. Na przykład w punkcie 1.5.4 mówiliśmy o funkcji Process.GetProcessesByName zwracającej wartość typu array of Process.

Klasa Array

Środowisko CLR traktuje każdą tablicę jak obiekt, podobnie jak i wszystkie inne typy proste. W środowisku .NET klasą bazową dla wszystkich tablic jest klasa System.Array, jednak zawartość takiej tablicy nie jest dostępna poprzez indeksy zapisane w nawiasach

prostokątnych, a wyłącznie poprzez wywoływanie metody `GetValue` i `SetValue`. Pozostałe metody klasy `Array` pozwalają na odczytanie indeksów granicznych, przeszukiwanie i sortowanie tablicy, kopiowanie i usuwanie wartości, a także odwracanie kolejności elementów (`Reverse`). Dokładne dane na temat tych metod uzyskać można w dokumentacji Delphi.

Dynamiczną tablicę języka Object Pascal można też przedstawiać jako obiekt klasy `System.Array`. W przykładowym kodzie z listingu 3.101 przygotowywany jest specjalny, abstrakcyjny widok `AbstractArray` opisujący istniejącą tablicę dynamiczną `ProcessArray`.

Listing 3.101. *Reprezentowanie tablicy dynamicznej jako obiektu klasy System.Array*

```
var
  ProcessArray: array of Process;
  AbstractArray: System.Array;
begin
  ProcessArray := Process.GetProcessesByName('Idle');
  AbstractArray := ProcessArray;
```

W klasie `System.Array` implementowane są interfejsy `IList`, `ICollection` i `Ienumerable`. W związku z tym wszystkie tablice dynamiczne tworzone w Delphi można obsługiwać tak samo jak kolekcje przedstawiane w punkcie 2.2.5.

Tablice wielowymiarowe

Wszystkie przedstawione do tej pory zasady można łatwo przenieść na tablice wielowymiarowe. Musimy tylko wiedzieć, jak deklarowane są takie tablice:

```
var
 IntMatrix: array of array of Int64;
```

… i jak ustalać ich wielkość wywołaniami procedury `SetLength`:

```
SetLength(IntMatrix, 100, 100);  // rezerwuje pamięć na 100*100 liczb typu Int64
```

Jeżeli wyobrazimy sobie, że deklarację `array of Int64` „wyciągniemy za nawias", to okaże się, że deklaracja zmiennej `IntMatrix` jest właściwie deklaracją tablicy jednowymiarowej, której wielkość ustalać można następującym wywołaniem:

```
SetLength(IntMatrix, 100);  // Rezerwuje sto tablic wartości typu Int64
```

Każdy ze stu elementów takiej tablicy można traktować tak samo jak zwyczajną zmienną typu `array of Int64`. Oznacza to, że każdy element takiej dynamicznej tablicy może być osobno inicjowany poprzez przypisanie lub wywołanie procedury `SetLength`. Jak wiemy, tablice dynamiczne mogą mieć różne wielkości, wobec czego każdemu elementowi takiej tablicy możemy przypisać tablicę o innej wielkości, przez co tablica `IntMatrix` nie byłaby prostokątna, ale na przykład trójkątna lub całkowicie nieregularna.

Stałe tablicowe

Tablice mogą być też deklarowane jako stałe, które przy okazji można inicjować wartościami podawanymi bezpośrednio. W deklaracjach takich tablic należy wykorzystywać następującą składnię:

```
ArrayConst: array[0..10] of Byte = (0, 0, 0, 100, 1, 100, 0, 0, 0, 1, 0);
```

3.6.4. Różne typy ciągów znaków

Rodzaje ciągów znaków dostępnych w Delphi są bardzo podobne do rodzajów pojedynczych znaków typu `Char`: Ciągi składające się ze znaków typu `AnsiChar` noszą nazwę `AnsiString`, natomiast te składające się ze znaków `WideChar` nazywają się `WideString`. Ogólny typ `String` w środowisku .NET odpowiada typowi `WideString` i jednocześnie zdefiniowanej w środowisku .NET klasie `System.String`. Z kolei typ `AnsiString` zdefiniowany jest w module `Bornald.Delphi.System` (w środowisku Win32 ogólny typ `String` odpowiada typowi `AnsiString`).

Typ `String` pod pewnymi względami podobny jest do ogólnych typów `Integer` i `Char`, ponieważ jego wewnętrzna budowa jest nieco inna w środowiskach Win32 i CLR, ale zastosowanie w obu środowiskach jest takie samo.

Typ System.String a typ String języka Pascal

Delphi umożliwia stosowanie tych samych operacji, typowych dla języka Pascal, na obu rodzajach ciągów znaków. Oprócz tego, dla każdego ciągu znaków typu `String` (lub `WideString`) wywoływać można metody udostępniane przez klasę `System.String`. Trzeba przy tym pamiętać, że klasa `System.String` stosuje nieco inny sposób indeksowania ciągów znaków: pozycja pierwszego znaku otrzymuje indeks zerowy, podobnie jak ma to miejsce w tablicach dynamicznych, natomiast w języku Pascal od zawsze pierwszy znak ciągu miał indeks o wartości `1`. Podobne różnice zauważyć można na przykład w działaniu funkcji `Copy` pochodzącej z języka Pascal i metody `String.SubString` dostępnej w środowisku .NET. W przykładzie przedstawionym na listingu 3.102 obie funkcje wywoływane są z tymi samymi parametrami, ale zwracają nieco inne wyniki.

Listing 3.102. *Różnice w indeksowaniu ciągów znaków w języku Pascal i środowisku .NET*

```
// Aplikacja konsolowa (menu File/New/Other/Console application)
var
  MyString: String
begin
  MyString := 'Test';
  writeln(Copy(MyString, 1, 3));      // wypisuje 'Tes'
  writeln(MyString.SubString(1, 3)); // wypisuje 'est'
```

Operatory działające na ciągach znaków

Pascalowe operatory działające na ciągach znaków dostępne są również w Delphi dla .NET. Do połączenia ze sobą dwóch ciągów znaków stosowany jest operator dodawania (+), a do porównania alfabetycznej kolejności ciągów wykorzystywane są operatory mniejszości (<) lub większości (>). Z kolei operator równości (=) pozwala na wykonywanie szybkich porównań ciągów znaków. Operatory porównujące wykonują dokładne porównanie ciągów znaków i w związku z tym odpowiadają metodzie środowiska .NET `System.String.CompareOrdinal`.

Jeżeli w czasie porównywania ciągów znaków nie mają być brane pod uwagę różnice między wielkimi i małymi znakami, to można w tym celu wykorzystać funkcję `CompareText` dostępną również w innych wersjach Delphi albo statyczną metodę `System.String.Compare`, podając im w dwóch pierwszych parametrach porównywane ciągi znaków, a w trzecim parametrze o nazwie `ignoreCase` — wartość `True`.

Operacje na ciągach znaków

W tym podpunkcie przyjrzymy się innym operacjom wykonywanym na ciągach znaków w języku Pascal i porównamy je z podobnymi operacjami wykonywanymi przez metody klasy `System.String`.

We wszystkich operacjach przedstawionych na poniższej liście trzeba zwracać uwagę na różny sposób indeksowania znaków. Jak pamiętamy, indeks numer `1` w języku Pascal oznacza pierwszy znak w ciągu znaków, ale w klasie `System.String` ten sam indeks wskazuje na drugi znak ciągu.

- `Copy(s, index, count)` — Tworzy nowy ciąg znaków (nie zmienia przy tym ciągu *s*) zawierający wycinek podanego w parametrze ciągu *s*, rozpoczynający się od pozycji *index* i składający się z *count* znaków. Odpowiednik tej funkcji z klasy `System.String` ma nieco czytelniejszą nazwę — `SubString`. Oczywiście dostępna jest też metoda `String.Copy`, ale robi ona dokładnie to, co sugeruje nazwa, czyli tworzy dokładną kopię pełnego ciągu znaków.
- `Pos(substr, s)` — Wyszukuje w ciągu znaków *s* podciąg *substr* i zwraca pozycję wyszukanego ciągu znaków lub wartość zera, jeżeli podciąg nie został znaleziony. Podobną metodą jest metoda `System.IndexOf`, która pozwala dodatkowo na nieco bardziej elastyczne działanie, ponieważ umożliwia określenie pozycji startowej, od której rozpocząć ma się szukanie. Alternatywnym rozwiązaniem jest też metoda `String.LastIndexOf`, która przeszukuje podany ciąg znaków od tyłu. Obie metody — `IndexOf` i `LastIndexOf` — w przeciwieństwie do funkcji `Pos` — w przypadku nieznalezienia szukanego podciągu zwracają wartość `-1`, a nie `0`.
- `Delete(s, index, count)` — Usuwa *count* znaków z podanego ciągu znaków *s*, rozpoczynając od pozycji *index*. W przeciwieństwie do metody `String.Remove`, funkcja `Delete` nie zwraca odpowiednio zmodyfikowanej kopii ciągu znaków, ale dokonuje modyfikacji bezpośrednio w ciągu znaków przekazanym w parametrze *s*.

- `Insert(substr, s, index)` — Do ciągu znaków `s` wstawia na pozycji `index` dodatkowy ciąg znaków `substr`. Jej odpowiednikiem jest metoda `String.Insert`, która jednak nie modyfikuje źródłowego ciągu znaków, ale zwraca ciąg wynikowy.

Z kolei poniższe procedury są całkowicie niezależne od stosowanej metody indeksowania:

- `Length(s)` — Zwraca długość ciągu znaków liczoną w znakach (odpowiada właściwości `String.Length`).
- `IntToStr(int)` i `IntToHex(int)` — Obie funkcje zamieniają podaną w parametrze liczbę w ciąg znaków zawierający jej reprezentację dziesiętną lub szesnastkową. W klasie `String` podobne efekty uzyskać można wywołując metody `Integer(int).ToString` lub `Integer(int).ToString('X')`.
- `StrToInt(str)` — Zwraca liczbę całkowitą tworzoną na podstawie zapisu przekazanego w parametrze `str`. Jeżeli ciągu znaków `str` nie da się przekształcić w liczbę, to wywoływany jest wyjątek `EConvertError`. Biblioteka FCL udostępnia nieco bardziej złożoną, ale za to bardziej wszechstronną metodę zamiany ciągu znaków `str` w liczbę całkowitą `i`, którą przedstawiam na listingu 3.103.

Listing 3.103. *Konwersja ciągu znaków na liczbę całkowitą*

```
// Wersja prosta:
i := Int16.Parse(str);  // Int16 = klasa środowiska .NET odpowiadająca typowi integer
// Wersja złożona, bardziej ogólna i wszechstronna:
i := Integer(TypeDescriptor.GetConverter(i.GetType).ConvertFromString(str));
```

- `LowerCase` i `UpperCase` — Zamieniają wszystkie litery w ciągu na litery wielkie lub małe i odpowiadają metodom `ToLower` i `ToUpper` z klasy `String`.

Manipulacje na ciągach znaków na poziomie pojedynczych znaków

Funkcją typową dla języka Pascal jest możliwość odwoływania się do poszczególnych znaków w ciągu, tak jakby były one elementami tablicy:

```
DziesiatyZnak := s[10];
s[9] := DziesiatyZnak;
```

Te znaki, które leżą poza aktualnym końcem ciągu znaków, mogą być zapisywane dopiero po jawnym powiększeniu długości ciągu znaków. Początkowo ciąg znaków ma długość zerową, ponieważ w Delphi inicjowany jest ciągiem pustym. Jeżeli teraz chcielibyśmy zapisać piąty znak ciągu, to moglibyśmy to wykonać za pomocą kodu przedstawionego na listingu 3.104.

Listing 3.104. *Zapisywanie pojedynczych znaków w ciągu*

```
var
  s: String;
begin
  // Ciąg znaków s ma długość zerową
  SetLenght(s, 5);
  // Ciąg znaków s ma długość pięciu znaków
  s[5] := NowyZnak;
```

W środowisku .NET takie bezpośrednie manipulacje na pojedynczych znakach ciągu powodują wykonywanie długotrwałych operacji, ponieważ klasa `System.String` pochodząca ze środowiska .NET nie pozwala na zmianę swojej zawartości. Z tego wynika, że przedstawiony wyżej kod Delphi zamieniający tylko jeden znak w ciągu powoduje utworzenie całkowicie nowego ciągu znaków, który od poprzedniego ciągu znaków różni się tylko jednym, zmienionym w danej instrukcji znakiem. Zmiennej typu `string` przypisywany jest nowy ciąg znaków, a stary ciąg po pewnym czasie usuwany jest z pamięci przez mechanizm oczyszczania pamięci. Jeżeli chcemy składać dłuższy ciąg znaków z pojedynczych znaków, to powinniśmy skorzystać z klasy `StringBuilder` oferowanej przez środowisko .NET do takich właśnie operacji. W klasie tej można bezpośrednio zmieniać poszczególne znaki ciągu (poprzez właściwość `Chars`), a na zakończenie przygotowany ciąg znaków zmienić w faktyczną wartość typu `String` wywołując metodę `ToString`.

Ciągi znaków o stałej długości

Osoby znające starsze wersje Delphi będą się teraz zapewne zastanawiać, czy w Delphi dla .NET nadal funkcjonują stare ciągi znaków języka Pascal. Owszem, ciągi te są nadal dostępne i wewnętrznie posługują się specyficznym formatem: Po pierwsze, są one tablicą znaków, której indeksowanie zaczyna się od wartości `0`. Na pozycji zerowej zapisywany jest bajt długości, który określa, z ilu znaków składa się dany ciąg. W Delphi dla .NET nie ma jednak bezpośredniego dostępu do tego bajtu długości, a jego wartość można modyfikować i odczytywać wyłącznie za pomocą metod `SetLength` i `Length`. W tym rodzaju ciągu znaków, niezależnie od rzeczywistej długości ciągu, ilość zajętej przez niego pamięci jest stała. Na przykład, za pomocą poniższej deklaracji rezerwowanych jest 101 bajtów pamięci (włącznie z bajtem długości) przeznaczonych na nowy ciąg znaków:

```
var
  StuznakowyCiag: string[100];
```

Jeżeli takiej zmiennej przypiszemy teraz ciąg znaków krótszy niż zadeklarowane 100 znaków, to pozostała pamięć będzie niezagospodarowana.

Ten rodzaj ciągu znaków wykorzystywany jest dokładnie tak samo jak długie ciągi znaków, a dodatkowo ciągi znaków starego typu można łatwo zamieniać na typ `System.String`.

Ciągi znaków o stałej długości na poziomie CLR implementowane są w postaci typów wartości — ich zamiana na typ `System.String` do pewnego stopnia odpowiada opisywanemu w punkcie 3.6.6 mechanizmowi Boxingu.

Formatowanie ciągów znaków

W czasie tworzenia ciągu znaków na podstawie kilku danych wejściowych mamy możliwość skorzystania z metody oferowanej przez Delphi, która dostępna jest też w Delphi dla Win32 i w pakiecie Kylix, a także z metody środowiska .NET funkcjonującej wyłącznie z biblioteką FCL.

Funkcja formatująca dostępna w Delphi umieszczona została w module `SysUtils` i nazywa się `Format`. W jej pierwszym parametrze podawać należy formatujący ciąg znaków, na przykład taki:

```
Wynik := Format('Przebieg %3d - Diagnoza: %s.',[Count, Diag]);
```

Formatujący ciąg znaków składa się z tekstowego szablonu, który z całą pewnością zostanie zapisany do zmiennej `Wynik`, oraz ze znaczników (ang. *Placeholders*), zamiast których do tekstu wstawiane będą inne parametry w określonym formacie. Każdy taki znacznik zaczyna się od znaku procentu (`%`), a kończy się znakiem określającym typ wstawianego w tym miejscu parametru. Na przykład litera `d` oznacza liczbę całkowitą, `e` — liczbę zmiennoprzecinkową, a `s` — ciąg znaków. Pomiędzy znakiem początkowym i końcowym znajdować się mogą też informacje o tym, ile znaków zajmować ma w ciągu wstawiany parametr.

Do funkcji `Format` oprócz formatującego ciągu znaków należy też przekazać, zamkniętą w nawiasach kwadratowych, tablicę wartości umieszczanych w formatowanym ciągu znaków w miejsce znaczników. W przedstawionym wyżej przykładzie w takiej tablicy znalazła się wartość liczby całkowitej reprezentowana przez zmienną `Count` oraz ciąg znaków zapisany w zmiennej `Diag`.

Wynikowy ciąg znaków generowany przez przykładowe wywołanie funkcji formatującej mógłby wyglądać tak: `Przebieg  33 - Diagnoza: Brak błędów.`, przy czym przed liczbą `33` zapisana zostałaby dodatkowa spacja dopełniająca długość wstawianego parametru do wymaganych trzech znaków.

Klasa `System.String` również oferuje metodę o nazwie `Format`, która także współpracuje z formatującym ciągiem znaków zawierającym znaczniki zamieniane wartościami parametrów. Tym razem jednak znaczniki podawane są w nawiasach klamrowych i nie jest konieczne określanie typu danych wstawianego parametru, jako że biblioteka FCL sama rozpozna jego typ. W stosowanych przez tę metodę znacznikach podawać należy indeks elementu tablicy podawanej za formatującym ciągiem znaków, dzięki czemu kolejność znaczników umieszczanych w ciągu znaków nie musi być identyczna z kolejnością parametrów wstawianych do tablicy. Poniżej przedstawiam wywołanie metody `String.Format` równoważne z przedstawionym wyżej wywołaniem metody `Format`:

```
Wynik := System.String.Format('Przebieg {0,3} - Diagnoza: {1}.',[Count, Diag]);
```

Formatowanie liczb zmiennoprzecinkowych i dat

Dokładniejszą kontrolę nad sposobem tworzenia ciągu znaków na podstawie wartości liczbowych oraz dat uzyskać można, wykorzystując procedury `FormatDateTime`, `FormatFloat` oraz inne oferowane przez Delphi. Dwie wymienione funkcje działają właściwie dokładnie tak samo jak funkcja `Format`, z tym, że pobierają w parametrze formatujący ciąg znaków tworzony zgodnie z nieco innym wzorcem i tylko jedną wartość przeznaczoną do formatowania.

W bibliotece FCL wszystkie operacje dotyczące formatowania wartości liczbowych i informacji o datach wykonywane są przez wymienioną przed chwilą funkcję `System.String.Format`.

3.6.5. Typy strukturalne

W języku Pascal klasycznym typem strukturalnym jest rekord. Deklaracja rekordu od zawsze składa się z ze słowa kluczowego `record`, listy elementów danych rekordu i końcowego słowa `end`, za którym musi znaleźć się jeszcze średnik. Przykładową deklarację podaję na listingu 3.105.

Listing 3.105. *Przykładowa deklaracja rekordu*

```
type
  TPlik = record
  Nazwa: String;
  Sciezka: String;
  Wielkosc: Int64;
end;
var
 Plik: TPlik;
begin
  Plik.Nazwa := 'Zycie bez komputerow.doc'
```

W środowisku .NET Delphi implementuje rekordy w postaci klasy, która jest automatycznie wywodzona z klasy `System.ValueType` (na temat typu `System.ValueType` mówić będę w punkcie 3.6.6) i pozwala też na definiowanie dla tej klasy nowych metod. Oto podstawowe różnice tej klasy w stosunku do „normalnych" klas:

- Rekordy nie muszą być tworzone wywołaniem konstruktora (wszystkie zmienne typu rekordowego automatycznie otrzymują przypisane obszary pamięci, a zmienne lokalne są automatycznie tworzone na stosie).
- Rekordy nie mogą korzystać z destruktorów i nie mogą pokrywać metody `Finalize`.
- W czasie przekazywania rekordu w parametrze metody tworzona jest jego kopia, chyba że dany parametr definiowany jest jako parametr referencyjny (na ten temat mówił będę w punkcie 3.8.1).

Wszystkie te różnice powodują, że konieczne jest tu rozróżnianie typów referencyjnych i wartości, o których dokładniej będę mówił w punkcie 3.6.6. Poza tym rekordów używać można jak zwyczajnych obiektów.

Stałe rekordów

Język Object Pascal dopuszcza też możliwość definiowania rekordów jako wartości stałych, tak jak pokazano na listingu 3.106.

Listing 3.106. *Definicja stałej rekordu*

```
const
  ObjectConst: TPlik = (Nazwa: 'FileDump.exe'; Sciezka: 'd:\Programy\BinUtils');
```

Jak widać, nie trzeba inicjować wszystkich elementów rekordu, ale elementy inicjowane muszą być podawane we właściwej kolejności. Ważne jest też to, że za zapisem inicjalizacji ostatniego elementu rekordu nie może znaleźć się znak średnika.

Zbiory

Zajmiemy się teraz typem, który w bibliotece FCL stosowany jest nader często — typem zbiorów. Ogólnie, składnia przedstawiona na listingu 3.107 służy do deklarowania typu zbioru oraz określania wartości, jakie w tym zbiorze mają się znajdować.

Listing 3.107. *Deklarowanie zbioru*

```
var
  CharSet: set of AnsiChar;
begin
  CharSet := ['a'..'z'];
```

Typ bazowy zbioru (czyli typ podawany za słowem kluczowym `of`) musi być typem porządkowym, który dodatkowo nie może przyjmować więcej niż 256 różnych wartości. W zmiennej zbioru dla każdej wartości rezerwowany jest jeden bit, który ma określać, czy dana wartość jest obecna w zbiorze, czy też nie. Przedstawiony powyżej kod oznacza, że zdefiniowany w nim zbiór znaków ma wielkość 32 bajtów.

Typ zbioru jest oczywiście interesujący tylko wtedy, gdy można do niego szybko dodawać i usuwać elementy oraz możliwe jest łatwe sprawdzanie, czy dany element jest zawarty w zbiorze. Operacje dodawania elementów do zbioru i usuwania ich z niego ułatwiają dwie funkcje standardowe przedstawione na listingu 3.108.

Listing 3.108. *Standardowe funkcje obsługujące dodawanie i usuwanie elementów ze zbioru*

```
{ Dopisywanie elementu do zbioru: }
Include(CharSet, 'd');
{ Usuwanie elementu ze zbioru: }
Exclude(CharSet, 'x');
```

Funkcji tych nie można jednak używać z właściwościami, ponieważ w parametrach przyjmują wyłącznie zmienne. Do właściwości zbioru nową wartość dopisać można na przykład tak:

```
BorderIcons := BorderIcons+[biMaximize];
```

Do obsługi zbiorów, obok operatora dodawania (+), można wykorzystywać jeszcze inne operatory, wypisane w tabeli 3.9.

3.6.6. Kategorie typów w CLR

System typów stosowany w środowisku .NET (CTS — ang. *Common Type System*) przede wszystkim rozróżnia dwa rodzaje typów: typy wartości i typy referencji. W tym punkcie wyjaśnię, na czym polegają różnice miedzy tymi rodzajami typów i jak dopasowywane są typy języka Object Pascal to tego podziału.

Tabela 3.9. *Operatory obsługujące operacje na zbiorach*

Operator	Funkcja	Wynik	Przykład
+	Łączenie	Zbiór	`[1..3,5,9]+[4..6]=[1..6,9]`
-	Różnica	Zbiór	`['c'..'h']-['a'..'z']=[]`
*	Część wspólna	Zbiór	`[1,3,5,6,]*[1,2,3]=[1,3]`
=	Równość	Boolean	`([niebieski]=[niebieski])=True`
<>	Nierówność	Boolean	
<=	Podzbiór	Boolean	`([5]<=[1,5,10]=True)`
>=	Nadzbiór	Boolean	
<	Rzeczywisty podzbiór	Boolean	`([czerwony]<[czerwony])=False`
>	Rzeczywisty nadzbiór	Boolean	
in	Zawieranie elementu	Boolean	`(zielony in [czerwony, zielony])=True`

Typy referencji

Typy referencji (nazywa są też typami wskaźnikowymi) charakteryzują się tym, że wewnętrznie realizowane są one wyłącznie poprzez wskaźnik. Zmienna typu referencyjnego przechowuje wyłącznie wskazanie na obiekt, który został dynamicznie utworzony na stercie. Dzięki tej właściwości tego rodzaju zmiennych, kilka zmiennych może wskazywać na ten sam obiekt. Jeżeli obiekt ten ulegnie jakiejkolwiek zmianie, to zmiana ta widoczna będzie natychmiast we wszystkich tych zmiennych. Załóżmy, że zmienna `ListView1` zadeklarowana została jako typ referencyjny wskazujący na kontrolkę *ListView*. W takich warunkach wywołanie:

```
MojObiekt := MojFormularz.ListView1;
```

spowoduje zapisanie do zmiennej `MojObiekt` wyłącznie referencji zapisanej w zmiennej `ListView1`, ale w systemie będzie nadal istniał tylko jeden egzemplarz kontrolki *ListView*. (Oprócz tego powiększona zostanie jeszcze wartość licznika referencji kontrolki stosowanego przez mechanizm automatycznego zwalniania pamięci). Podobnie, w czasie przekazywania typu referencyjnego w parametrze metody kopiowana jest zawsze wyłącznie referencja, co odpowiada metodzie przekazywania parametrów „przez referencję". Jak widać, nie da się przekazać zawartości typu referencyjnego „przez wartość".

W środowisku .NET wszystkie typy są typami referencji, oprócz tych typów, które wywiedzione są z klasy `System.ValueType`.

W języku Object Pascal typami referencji są:

- klasy zadeklarowane słowem kluczowym `class`,
- tablice,
- wskaźniki metod.

Typy wartości

Typy wartości nie są przechowywane na stercie, ale na stosie. Klasa typu wartości zawsze musi być klasą wywiedzioną z klasy `System.ValueType`. W języku Object Pascal kryteria te spełniają następujące typy:

- typy proste, takie jak `integer`, `boolean` lub `double` (typy te odpowiadają typom środowiska .NET wymienianym w różnych miejscach tego rozdziału; wszystkie te typy zostały w środowisku .NET wywiedzione z klasy `System.ValueType`),
- typy wyliczeniowe i typy zakresów częściowych (wywodzą się z klasy `System.Enum`, a klasa ta została wywiedziona z klasy `System.ValueType`),
- rekordy, które na swój sposób również reprezentują klasy, ale definiowane są słowem kluczowym `record` i w związku z tym są zawsze niejawnie wywodzone z klasy `System.ValueType`.

Każda zmienna typu wartości ma przydzieloną swoją własną pamięć na stosie i własną kopię typu wartości (wyjątkiem są parametry metod, które zadeklarowane zostały jako parametry referencyjne). Przypisując jedną zmienną typu wartości do drugiej, zawsze tworzymy pełną kopię danych zapisanych w tej zmiennej, na przykład:

```
Integer2 := Integer1;
Record2 := Record1;
```

Po wykonaniu powyższych przypisań można zmieniać wartości zmiennych `Integer2` i `Record2`, bez jednoczesnego naruszania zawartości zmiennych `Integer1` i `Record1`.

W środowisku CLR funkcjonuje jeszcze jedna klasa, która obok klasy `ValueType` stanowi podstawową różnicę pomiędzy rodzajami typów danych — `MarshalByRefObject`. Obiekty wywodzące się z tej klasy przekazywane są jako referencja w przypadku wywołań pochodzących z innej domeny aplikacji, z kodu niezarządzanego albo zewnętrznego komputera, podczas gdy wszystkie pozostałe obiekty przekazywane są jako kopia wartości. Podana tutaj różnica nie ma nic wspólnego z różnicami pomiędzy omawianymi typami referencji i typami wartości. Należy tu jednak zaznaczyć, że z klasą `MarshalByRefObject` współpracować mogą wyłącznie typy referencyjne.

Mechanizm Boxingu

W środowisku CLR mechanizm Boxingu tworzy możliwość przekształcenia typu wartości w typ referencji, poprzez przekształcenie jej w klasę `TObject`, tak jak na listingu 3.109.

Listing 3.109. *Przykład wykorzystania mechanizmu Boxingu*

```
var
  o: System.Object;
begin
  o := System.Object(Integer(4));
```

Zastosowany przy tym mechanizm jest tutaj całkowicie logiczny i wynika z innych reguł obowiązujących w języku programowania. Klasa `Object` nie została wywiedziona z klasy `System.ValueType` i w związku z tym nie jest typem wartości. Klasa ta jest jednak klasą nadrzędną w stosunku do klasy `System.ValueType`, co oznacza, że typ `ValueType` może być przekształcony w typ `Object`.

W czasie działania programu całość nie przedstawia się jednak aż tak łatwo, ponieważ wartości typu `ValueType` są odkładane na stosie i w czasie ich przekształcania do typu referencyjnego konieczne jest rezerwowanie pamięci na stercie, a zawartość stosu musi zostać skopiowana do sterty. W podanym wyżej przykładzie powstający tak obiekt zapisywany jest w zmiennej `o`, i w związku z tym może być ona dowolnie przekazywana dalej, a sam obiekt zostanie usunięty z pamięci dopiero po wyzerowaniu wartości licznika jego referencji.

3.7. Instrukcje

W języku Object Pascal rozróżnia się instrukcje proste i instrukcje strukturalne. Do instrukcji prostych zalicza się operacje przypisania za pomocą operatora przypisania (`:=`) oraz wywołania procedur, funkcji i metod. Większość instrukcji strukturalnych służy do kontrolowania przebiegu programu; zalicza się do nich pętle, sprawdzenia warunków i skoki. W języku Pascal funkcjonuje jeszcze instrukcja `with`, która przeznaczona jest przede wszystkim do ułatwiania tworzenia tekstu programu i nie ma nic wspólnego z przepływem programu.

Średniki kończące instrukcje

W języku Pascal średniki służą do rozdzielania od siebie dwóch następujących po sobie instrukcji, co oznacza, że ostatnia instrukcja w bloku nie musi być zamykana średnikiem. Na szczęście język obsługuje też instrukcje puste, dlatego ostatnie instrukcje w bloku również mogą być zakończone średnikiem (a nawet kilkoma). W programach przedstawionych w tej książce dla uproszczenia średniki umieszczałem za wszystkimi instrukcjami.

Pętla for

Najprostszą strukturą sterującą jest pętla `for`, wielokrotnie wykonująca jedną lub kilka instrukcji. Pętla ta występuje w dwóch wariantach: w pierwszym z nich podać należy przedział wartości, w jakim pętla ma się wykonywać, a liczba obiegów pętli znana jest jeszcze przed wykonaniem pierwszego obiegu. Przykład takiej pętli podaję na listingu 3.110.

Listing 3.110. *Przykładowa pętla for*

```
x := a + b
for i := a to b do
   PowtarzanaInstrukcja;
```

Na początku pętla ta ustala wartość zmiennej `i` na wartość pobraną ze zmiennej `a` i po każdym obiegu pętli o jeden powiększa wartość zmiennej `i`. Trwa to tak długo, aż zmienna `i` osiągnie wartość, jaką miała zmienna `b` przed rozpoczęciem pętli. Jak widać, zmiany wartości zmiennych `a` i `b` wykonywane już w czasie działania pętli nie mają żadnego wpływu na jej funkcjonowanie.

Pętla for…in

Drugi wariant pętli `for` przygotowany został specjalnie dla Delphi dla .NET, ale firma Borland wprowadziła go dopiero do Delphi 2005. Pętla ta służy do przeglądania kolekcji różnych wartości za pomocą jednej zmiennej. Jako takie „kolekcje" wykorzystywane mogą być tablice, ciągi znaków, zbiory oraz obiekty kolekcji. W przykładzie przedstawionym na listingu 3.111 przeglądana jest „kolekcja" ciągów znaków zgromadzona we właściwości `TextBox1.Lines` (tablica). W procesie tym wykorzystywana jest zmienna `s` typu `String`, a całkowita długość wszystkich ciągów znaków jest sumowana i wypisywana w oknie programu.

Listing 3.111. *Przykład zastosowania pętli for…in*

```
procedure TWinForm2.TextBox1_TextChanged(sender: System.Object; e:
System.EventArgs);
var
  s: String;
  OgolnaDlugosc: Integer;
begin
  for s in TextBox1.Lines do
    inc(OgolnaDlugosc, s.Lenght);
  Text := OgolnaDlugosc.ToString;
end;
```

W podobny sposób można przeglądać wszystkie znaki ciągu znaków, wykorzystując przy tym zmienną typu `char`. Do przeglądania elementów zbioru zastosować trzeba zmienną o takim samym typie, jaki mają poszczególne elementy tego zbioru.

Przeglądanie kolekcji odbywa się mniej więcej tak, jak przeglądanie tablic, co wynika z tego, że w środowisku .NET każda tablica **jest** kolekcją (klasa `System.Array` implementuje interfejs `ICollection`, a każda dynamiczna tablica w Delphi wywodzi się z klasy `System.Array`). Przykład przeglądania zawartości kolekcji za pomocą pętli `for…in` znaleźć można w punkcie 2.2.5, w którym omawiany jest też wewnętrzny mechanizm funkcjonowania tego rodzaju pętli.

Bloki

Wszystkie struktury sterujące mogą wykonywać nie tylko pojedyncze instrukcje, ale również całe bloki kodu. We wszystkich przypadkach z wyjątkiem pętli `repeat…until` konieczne jest zamykanie takiego bloku kodu pomiędzy słowami kluczowymi `begin` i `end`, co doskonale widać na przykładzie pętli `for` przedstawionej na listingu 3.112.

Listing 3.112. *Przykład pętli for wykonującej blok kodu*

```
for i := 1 to 100 do begin
  Instrukcja1;
  Instrukcja2;
  ...
end;
```

Warunki

Wszystkie pozostałe instrukcje dotyczące kontroli przepływu programu operują na jawnie sprecyzowanych warunkach. W czasie oceny takiego warunku w języku Object Pascal zawsze powstaje wartość `True` (jeżeli warunek został spełniony) lub `False` (w przypadku niespełnienia warunku). Taki wynik logiczny może zostać zwrócony przez funkcje lub operatory (operatory porównujące i operatory logiczne). Wszystkie operatory dostępne w języku Pascal oraz ich priorytety omawiałem już w punkcie 3.6.2.

Instrukcja if

Instrukcja `if` składa się z wyrażenia, którego wartość musi zostać sprawdzona i maksymalnie dwóch alternatywnych instrukcji lub bloków instrukcji. Instrukcja ocenia podany warunek i w przypadku, gdy zwróci on wartość `True`, wykonuje pierwszą instrukcję lub blok instrukcji, a jeżeli warunek zwróci wartość `False`, instrukcja wykona instrukcję lub blok instrukcji zapisany po słowie kluczowym `else`. Przykład takiej postaci instrukcji `if` przedstawiam na listingu 3.113.

Listing 3.113. *Przykładowa instrukcja if*

```
if Wyrazenie
  then jestPrawdziwe
else jestFałszywe;
```

Nie chcę tu wprowadzać wykładu na temat różnic instrukcji `if` w języku Pascal w stosunku do podobnych instrukcji w innych językach programowania (na przykład w porównaniu do języków wielkiej rodziny C). Muszę jednak zaznaczyć, że w języku Pascal instrukcja `if` traktowana jest w całości jako jedna instrukcja, razem z jej częścią `else`. Pamiętamy, że znak średnika nie może być umieszczany wewnątrz instrukcji, w związku z czym nie można też umieszczać średnika przed słowem kluczowym `else`, tak jak mają to w zwyczaju języki C# i Java. Na tej samej zasadzie opiera się też rozpoznawanie zagnieżdżeń instrukcji `if`, takie jak na listingu 3.114.

Listing 3.114. *Zagnieżdżone instrukcje if*

```
if Alternatywa1 then
  if Alternatywa2 then
    Instrukcja1
  else Instrukcja2;
```

W powyższym kodzie powstaje pytanie, do której instrukcji `if` należy instrukcja `Instrukcja2`. W języku Pascal kompilator oczekuje, że po pierwszym słowie kluczowym `then` zapisana będzie pełna instrukcja, a ta nie kończy się jeszcze wraz z końcem instrukcji

Instrukcja1, ponieważ nie został za nią umieszczony znak średnika (jeżeli średnik znalazłby się w tym miejscu, to spowodowałby błąd, ponieważ przerywałby instrukcję zewnętrzną). Z tego wynika, że w języku Pascal instrukcja Instrukcja2 zaliczona zostanie jako część drugiej instrukcji if. Podany wyżej przykład można by uzupełnić o jeszcze jedno słowo kluczowe else należące do pierwszej instrukcji if, ale wtedy konieczne byłoby usunięcie średnika zamykającego instrukcję Instrukcja2.

Te wszystkie wątpliwości z całą pewnością nie będą się pojawiać, jeżeli instrukcje przypisane do różnych części poszczególnych instrukcji if zamykać będziemy pomiędzy słowami kluczowymi begin i end. Na listingu 3.115 pokazano, że takie rozwiązanie pozwoli też na zmianę przypisania części else do innej instrukcji if niż na listingu 3.114.

Listing 3.115. *Słowa kluczowe begin i end pozwalają precyzyjnie umieścić instrukcje w programie*

```
if Alternatywa1 then begin
  if Alternatywa2 then
    Instrukcja1;
end
else Instrukcja2;
```

Pętla repeat

Pętla repeat…until tak długo wykonuje instrukcje zapisane pomiędzy tymi dwoma słowami kluczowym, aż warunek zapisany za słowem kluczowym until będzie prawdziwy. Na listingu 3.116 przedstawiam nieskończoną pętlę, która sama w sobie nie wykonuje żadnych sensownych operacji.

Listing 3.116. *Przykładowa pętla repeat…until*

```
repeat
  if a > 5 then a := a div 2;
  a := a + 1;
until a > 9;
```

Pętla while

Cechą pętli repeat…until jest to, że warunek jej wykonania sprawdzany jest dopiero po wykonaniu zawartych w niej instrukcji. Fakt ten czyni tę pętlę przypadkiem specjalnym najczęściej stosowanej w języku Pascal pętli while. Ta ostatnia już na samym początku sprawdza, czy zawartość pętli w ogóle powinna być wykonana. Przykład pętli while przedstawiam na listingu 3.117.

Listing 3.117. *Przykładowa pętla while*

```
{ Opróżnianie stosu }
while not Stack.Empty do
  Stack.Pop;
```

W tym miejscu obowiązują oczywiście reguły zapisu bloków instrukcji oraz zasady tworzenia warunku umieszczanego za słowem kluczowym while.

Instrukcja case

Instrukcja case pozwala na uniknięcie bardzo złożonych konstrukcji z instrukcjami if, tworząc tabelę możliwych wartości zwracanych i powiązanych z nimi funkcji. Przykład zastosowania instrukcji case podaję na listingu 3.118.

Listing 3.118. *Przykład instrukcji case*

```
case RadioGroup.ItemIndex of
  0: Instrukcja1;
  1: begin
    ... // kilka instrukcji
  end;
  2, 3: Instrukcja2;
  else ...
end;
```

Zmienna zapisywana za słowem kluczowym case musi być zmienną typu porządkowego. Jak widać na przykładzie, kilka warunków rozdzielanych w kodzie przecinkami może być obsługiwanych przez ten sam wycinek kodu. Wszystkie warunki niewymienione w kodzie instrukcji case mogą być obsługiwane przez kod umieszczony za słowem kluczowym else.

Jeżeli jeden z warunków zostanie odnaleziony na przygotowanej liście, to wykonywane są instrukcje z bloku kodu znajdującego się bezpośrednio za warunkiem. Wynika z tego, że nie ma potrzeby stosowania znanej z języka C instrukcji break.

Skoki

Język Object Pascal może realizować cztery rodzaje skoków:

- Stosując wywołanie Exit, które nie jest słowem kluczowym, powodujemy natychmiastowe wyjście z procedury lub funkcji, co jest równoznaczne ze skokiem do słowa kluczowego end oznaczającego koniec funkcji. Dzięki takiemu rozwiązaniu uniknąć można stosowania jednego lub kilku poziomów instrukcji if.
- Wyjątki są najnowocześniejszym rodzajem skoków, które po uwzględnieniu innych struktur sterujących pozwalają na łatwe wyeliminowanie z kodu skoków wykonywanych instrukcjami exit i goto. Wyjątkom poświęcony zostanie podrozdział 3.9.
- Skoki w pętlach. Za pomocą instrukcji continue wykonywany jest skok na początek pętli, tak jakby już w tym miejscu zakończył się jej poprzedni obieg. Z kolei instrukcja break powoduje natychmiastowe wyjście z pętli.
- W końcu, nadal dostępne są instrukcje goto, ale są one już tylko reliktem z bardzo starych czasów i w tej książce nie odgrywają żadnej roli.

Pod pewnymi względami, z instrukcjami break i exit spokrewniona jest też instrukcja halt. Powoduje ona natychmiastowe zakończenie programu i dlatego wewnętrznie jest znacznie bardzie rozbudowana niż zwyczajny skok, ponieważ po jej wywołaniu muszą być między innymi wykonane sekcje finalization wszystkich modułów programu.

Instrukcja with

Jedyną strukturą sterującą języka Object Pascal, której odpowiednika nie znajdziemy w żadnym z języków wywodzących się z języka C, jest instrukcja `with`. W instrukcji tej podawany jest zakres widoczności, który w podanym bloku kodu ma być przez kompilator uznawany za zakres domyślny. Korzystając z instrukcji `with` można zamiast zapisu:

```
DefaultFontStruct.Name := 'Phantom';
DefaultFontStruct.Height := 20;
DefaultFontStruct.UseRaytracing := True;
```

… zastosować znacznie prostszy wariant:

```
with DefaultFontStruct do begin
  Name := 'Phantom';
  Height := 20;
  UseRaytracing := True;
end;
```

Jak można się domyślać, instrukcja `with` pozwala przesłonić pewne identyfikatory aktualnego zakresu widoczności. Co więcej, za słowem kluczowym `with` podawać można kilka zakresów widoczności rozdzielanych przecinkami, a nawet zagnieżdżać bloki tworzone tymi instrukcjami. Te dwie ostatnie możliwości są jednak bardzo rzadko wykorzystywane praktycznie.

3.8. Procedury i funkcje

W języku Object Pascal wszystkie metody dzielone są na dwie kategorie: procedur i funkcji. Funkcja różni się od procedury wyłącznie sposobem zwracania wartości wynikowych. Wewnątrz ciała funkcji zwracana wartość musi być przypisywana do specjalnej zmiennej o nazwie `Result`, a typ wartości zwracanej przez funkcję podawany jest za dwukropkiem umieszczonym za listą parametrów funkcji. Przykład takiej funkcji podaję na listingu 3.119.

Listing 3.119. *Przykładowa funkcja i procedura*

```
function Math.WyliczTangens(f: Double): Double;
begin
  Result := sin(f) / cos(f);
end;

procedure Math.WyswietlWartosc(f: Double);
begin
  MessageBox.Show(FloatToStr(f));
end;
```

Z reguły listy parametrów są identyczne w funkcjach i procedurach. Składnia parametrów jest identyczna ze stosowaną przy deklarowaniu zmiennych, za słowem kluczowym `var` (słowo to może pojawiać się też na liście parametrów, ale ma tam inne znacznie, które omawiane będzie za chwilę).

Wywołanie procedury wymaga zapisania jej nazwy i podania w nawiasach listy jej parametrów, na przykład tak:

```
WyswietlWartosc(3.3);
```

Przy wywoływaniu procedur niepobierających żadnych parametrów w języku Object Pascal zazwyczaj nie zapisuje się pustych nawiasów za nazwą procedury (jest to typowe dla języków wywodzących się z języka C), choć w Delphi 2005 zapisywanie pustych nawiasów zostało dopuszczone.

To samo dotyczy też funkcji, a jedyną różnicą jest to, że wynik jej działania powinien zostać w jakiś sposób zagospodarowany, tak jak w podanym wyżej przykładzie wykorzystane zostały wyniki zwracane przez funkcje `cos` i `sin`. Oczywiście, wartości zwracane przez funkcje mogą być też ignorowane, przez co wywołanie funkcji upodabnia się do wywołania procedury (musi być jednak włączona opcja kompilatora `$X` (*EntendedSyntax*), która włączona jest standardowo).

Deklaracje uprzedzające

Jeżeli dwie funkcje z jednego modułu, niebędące metodami i niezadeklarowane w części interfejsu modułu, wykorzystują się wzajemnie, to o jednej z nich trzeba uprzedzająco poinformować kompilator. Przykład takiej uprzedzającej deklaracji funkcji podaję na listingu 3.120.

Listing 3.120. *Przykład uprzedzającej deklaracji funkcji*

```
procedue ProcBuzywaA; forward; { deklaracja uprzedzająca }
procedue ProcAuzywaB;
begin
  ProcBuzywaA;
end;
procedue ProcBuzywaA;
  { definicja procedury z deklaracji uprzedzającej }
begin
  ProcAuzywaB;
...
```

Jeżeli nie przygotowalibyśmy dla kompilatora deklaracji uprzedzającej, to pierwsza funkcja nie mogłaby wywoływać drugiej funkcji.

Takie deklaracje uprzedzające nie są potrzebne w przypadku metod, ponieważ deklaracja klasy jest niejako jedną wielką deklaracją uprzedzającą dla wszystkich metod.

3.8.1. Typy parametrów

Na początek muszę podać wyjaśnienia pewnych pojęć: parametry, jakie podawane są w deklaracji funkcji, nazywane są *parametrami formalnymi* (ang. *Formal parameters*), natomiast parametry, z którymi funkcja pracuje w czasie swojego działania, nazywane są *parametrami aktualnymi* (ang. *Actual parameters*).

Parametry wartości

W języku Pascal, podobnie jak i w większości innych języków programowania, istnieją dwie możliwości przekazywania parametrów. W aktualnych parametrach funkcji znaleźć mogą się oryginalne zmienne przekazane bezpośrednio do funkcji albo tylko ich kopie. W tym drugim przypadku wywoływana funkcja może dowolnie zmieniać zawartość otrzymanego parametru, bez wpływu na wartość przechowywaną w funkcji wywołującej. Takie parametry nazywane są parametrami wartości, a ich przykład podaję na listingu 3.121.

Listing 3.121. *Przykład parametrów wartości przekazywanych do funkcji*

```
procedure PaintCells(od, do: Integer);
begin
  for od := od to do do begin
    ...
  end;
end;

var
  Start, Stop: Integer;
begin
  Start := 0;
  Stop := 10;
  PaintCells(Start, Stop);
```

W powyższym przykładzie zawartość zmiennej `Start` nie zostanie zmieniona, ponieważ funkcja `PaintCells` otrzyma tylko kopię tej wartości zapisaną w zmiennej `od`. Zmienna ta utworzona jest na stosie i zostanie zniszczona, gdy tylko funkcja `PaintCells` zakończy swoją pracę.

Jeżeli parametr wartości jest typu referencyjnego, czyli na przykład jest obiektem, to nie jest tworzona **kopia** tego obiektu, ale do funkcji przekazywana jest tylko kopia wskazania na ten obiekt. Dzięki temu wewnątrz procedury można przypisać do parametru inny obiekt, nie wpływając tym samym na obiekt otrzymany z procedury wywołującej. Jeżeli jednak zmienimy wartości właściwości obiektu otrzymanego w parametrze, to zmiany te widoczne będą również w procedurze wywołującej.

Parametry referencyjne

Jeżeli chcielibyśmy, żeby w poprzednim przykładzie wywoływana procedura otrzymywała faktyczną wartość zmiennej `Start`, to musielibyśmy następująco zmienić deklarację procedury `PaintCells`:

```
procedure PaintCells(var od: Integer; do: Integer);
```

Słowo kluczowe `var` umieszczone w liście parametrów funkcji informuje kompilator, że do funkcji przekazywany ma być wskaźnik na oryginał parametru. Dzięki temu nazwa `od` opisywałaby dokładnie ten sam wycinek pamięci, co nazwa zmiennej `Start`.

Oczywiście, takie parametry referencyjne (w nomenklaturze języka Pascal nazywane są one parametrami zmiennymi) pozwalają nie tylko na zmiany wartości zmiennych, ale też na oszczędności pamięci stosu i przyspieszenie wywołań procedur i funkcji. Przykład takiego przyspieszenia wywołania funkcji podaję na listingu 3.122.

Listing 3.122. *Parametry referencyjne pozwalają na oszczędność czasu i pamięci*

```
type
  TBigArray: array[0..100] of Longint;
procedure CheckArray(var A: TBigArray);
```

Jeżeli w deklaracji funkcji `CheckArray` nie byłoby słowa kluczowego `var`, to przy każdym jej wywołaniu w programie rezerwowanych byłoby 400 bajtów w pamięci stosu, do których kopiowana byłaby zawartość tablicy. Dzięki słowu kluczowemu `var` do funkcji przekazywany jest tylko wskaźnik na tablicę, który ma wielkość czterech bajtów. Niebezpieczeństwo polega na tym, że w tym rozwiązaniu procedura może zmieniać wartości zapisane w tablicy. Ta wada parametrów referencyjnych została usunięta dzięki wprowadzeniu do listy parametrów słowa kluczowego `const`.

Ostatni przykład zadziałał tylko dlatego, że typ `TBigArray` jest typem wartości języka Object Pascal, a nie dynamiczną tablicą środowiska CLR. Takie tablice dynamiczne są po prostu typami referencji, które zawsze przekazywane są przez referencję, i w związku z tym **nie** powinny być deklarowane jako parametry ze słowem kluczowym `var`. Zadeklarowanie w taki sposób parametru typu referencyjnego ma taką zaletę, że wywoływana funkcja może w otrzymanym parametrze zapisywać wartości zwracane.

Parametry stałe

Język Object Pascal umożliwia też zadeklarowanie parametrów z wykorzystaniem słowa kluczowego `const`. Kompilator nie pozwala wtedy na zmianę tak zadeklarowanych parametrów wewnątrz wywoływanej procedury. Jeżeli w parametrze i tak przekazywana miałaby być tylko kopia wartości, to takie ograniczenie nie miałoby sensu, dlatego parametry zadeklarowane ze słowem kluczowym `const` traktowane są jak parametry referencyjne. Jest to bardzo efektywne rozwiązanie w przypadku zmiennych typów wartości, które zajmują wiele pamięci, ponieważ zamiast kopiowania całej zmiennej, do procedury przekazywany jest tylko wskaźnik.

Tablice otwarte

Ostatnim wariantem parametru przekazywanego do procedury są *tablice otwarte* (ang. *Open array*), a w przypadku ciągów znaków — *otwarte ciągi znaków* (ang. *Open string*). Z tego rodzaju parametrów korzystać można pod warunkiem, że nie została wyłączona opcja kompilatora `$P+` (*Open parameters* — standardowo jest ona włączona). Parametry te nazywane są otwartymi, ponieważ w parametrach formalnych (czyli w deklaracji) nie określa się, ile elementów ma mieć tablica albo z ilu znaków składać się ma ciąg znaków. Oznacza to, że w (aktualnych) parametrach funkcji przekazywane mogą być tablice i ciągi znaków o dowolnej wielkości. W przypadku tablic otwartych wystarczy,

że podany zostanie typ elementów tablicy, a przy okazji można też wybierać tryb przekazywania tablicy do funkcji: normalny, `var` lub `const` (odpowiednie deklaracje przykładowe przedstawiam na listingu 3.123).

Listing 3.123. *Przykładowe deklaracje parametrów otwartych tablic*

```
function SumujTablice_Kopia(a: array of Integer): Integer;
function SumujTablice_Bezposrednio(const a: array of Integer): Integer;
procedure SortujTablice(var a: array of Integer);
```

Różnice w wywołaniach funkcji z parametrami deklarowanymi normalnie i ze słowami kluczowymi `var` lub `const` są identyczne z tymi opisywanymi w przypadku zmiennych. Oznacza to, że funkcja `SumujTablice_Kopia` jest w czasie działania programu nieco mniej wydajna niż funkcja `SumujTablice_Bezposrednio`, ponieważ nie wymaga ewentualnego kopiowania dużych ilości danych tablicy, której wielkości nie można przewidzieć, ze względu na jej otwarty charakter.

Formalny parametr tablicy otwartej można wypełnić parametrem aktualnym na dwa sposoby, przedstawione na listingu 3.124.

Listing 3.124. *Przekazywanie tablic w parametrach tablic otwartych*

```
var
  ParamArray: array[0..10000] of Integer;
begin
  SortujTablice(ParamArray); { przekazując zmienną tablicową }
  SumujTablice_Kopia([322, 223, 443, 988]); { przekazując tablicę tymczasową }
```

Funkcja `SortujTablice` pobiera w parametrze dowolną tablicę liczb całkowitych, dlatego bez żadnych kłopotów można przekazywać jej zmienną `ParamArray`. W wywołaniu drugiej funkcji nie była stosowana zmienna tablicowa, ale konstrukcja opisywana w następnym podpunkcie.

Konstruktory tablic otwartych

W wywołaniu drugiej funkcji z poprzedniego przykładu zastosowany został konstruktor tablicy otwartej (ang. *Open array constructor*), będący bardzo wydajnym mechanizmem języka Object Pascal, który w tekście programu zapisywany jest w postaci niewyróżniających się nawiasów kwadratowych. W nawiasach tych podawana jest lista wartości zapisywanych do tworzonej tablicy tymczasowej, która następnie przekazywana jest do wywoływanej funkcji. W funkcji `SumujTablice_Kopia` przekazana tablica wygląda tak, jakby została ona przygotowana za pomocą kodu przedstawionego na listingu 3.125.

Listing 3.125. *Hipotetyczny kod konstruktora tablicy otwartej*

```
var
  TempArray: array of Integer;
begin
  SetLength(TempArray, 4);
  TempArray[0] := 322; TempArray[1] := 223;
  TempArray[2] := 443; TempArray[3] := 988;
```

Jak można się domyślać, przekazywanie tymczasowej tablicy utworzonej konstruktorem tablic jako parametru referencyjnego nie ma większego sensu, ponieważ wartości, jakie w tablicy mogłaby zmienić wywoływana funkcja, zostaną utracone natychmiast po jej zakończeniu.

Praca z tablicami otwartymi

Właściwa praca z tablicami otwartymi możliwa jest tylko wtedy, gdy wewnątrz procedury mamy możliwość sprawdzenia, jaka jest rzeczywista wielkość przekazywanej tablicy. W związku z tym wewnątrz procedury można wykorzystywać funkcje przedstawione na listingu 3.126, które pozwalają na sprawdzenie wielkości tablicy.

Listing 3.126. *Funkcje sprawdzające wielkość tablicy otwartej*

```
StartIndex := Low(a);  { indeks pierwszego elementu tablicy (zawsze 0) }
EndIndex := High(a);  { indeks ostatniego elementu tablicy }
TotalArraySize := SizeOf(a);
```

Pierwszy element tablicy zawsze ma indeks zerowy, dlatego ogólna liczba elementów zapisanych w tablicy wynosi `EndIndex+1`. Funkcje `Low` i `High` są w Delphi funkcjami niezależnymi od platformy, które umożliwiają określenie granicznych indeksów tablicy. W środowisku .NET dostępna jest niezależna od języka metoda `System.Array.Length`, którą można też wywoływać dla każdej tablicy otwartej języka Object Pascal w celu uzyskania informacji o ogólnej długości tablicy (włączając w nią element o indeksie zerowym).

3.8.2. Przeciążanie metod i parametry standardowe

Jeżeli pracujemy z metodą, w której pewne parametry mogą otrzymywać wartość standardową, to w zapisie wywołania takiej metody możemy te parametry pominąć. W takim wypadku metoda otrzyma w parametrach taką wartość, która została zapisana jeszcze w jej deklaracji. Przykład takiej deklaracji podaję na listingu 3.127.

Listing 3.127. *Przykład deklaracji funkcji z parametrami standardowymi*

```
// Lista ciągów znaków składana jest w jeden długi ciąg znaków.
// Opcjonalnie podać można maksymalną długość poszczególnych ciągów znaków
// oraz znak rozdzielający te ciągi:
function MakeString(Strings: TStringList; SeparatorChar: char = ';';
  MaxLen: Integer = 0) // 0 oznacza brak ograniczenia długości
  :String;
```

W deklaracji tej najważniejsze jest to, że wszystkie parametry, które mają pewną wartość standardową, muszą być podawane na końcu listy parametrów. Dzięki temu kompilator przy wywoływaniu metody może określić, ile parametrów ma otrzymać wartość standardową, kierując się wyłącznie liczbą parametrów wpisanych w wywołaniu. Wynika z tego, że przykładową funkcję z listingu 3.127 można wywoływać na trzy sposoby:

```
Str := MakeString(Lista);           // Tworzenie ciągu znaków z ustawieniami domyślnymi
Str := MakeString(Lista, ',');      // Do rozdzielania ciągów stosowany jest przecinek
Str := MakeString(Lista, ',', 30);  // Dodatkowo, żaden z wejściowych ciągów znaków
                                    // nie może być dłuższy niż 30 znaków.
```

W wywołaniach takiej funkcji obowiązuje ta sama zasada, która obowiązywała w czasie jej deklarowania: nie można opuszczać dowolnego z parametrów, ale wyłącznie te znajdujące się na końcu listy parametrów. Oznacza to, że chcąc podać specjalną wartość trzeciego parametru, nie możemy pominąć drugiego parametru, nawet jeżeli może on otrzymywać wartość standardową.

Parametry standardowe doskonale sprawdzają się też w przypadkach, gdy często stosowaną metodę chcemy uzupełnić o dodatkowy parametr, a jednocześnie nie chcemy zmieniać istniejących już wywołań tej metody. Taki dodatkowy parametr można dopisać na końcu listy parametrów metody i uzupełnić o wartość standardową, dzięki czemu nie będzie trzeba wymieniać go w wywołaniach metody, a to oznacza, że nie będzie konieczne modyfikowanie istniejących już wywołań tej metody.

Przeciążanie metod

Przeciążanie metod pozwala na stosowanie tej samej nazwy dla wielu różnych metod, różniących się od siebie pobieranymi parametrami. W czasie tworzenia takich metod trzeba zawsze zwracać uwagę na następujące rzeczy:

- Każdy wariant przeciążanej metody musi być opatrzony dyrektywą `overload`.
- Wszystkie przeciążane funkcje otrzymujące tę samą nazwę muszą się jednoznacznie odróżniać od pozostałych typami swoich parametrów lub typem wartości zwracanej.
- Parametry standardowe mogą utrudnić rozróżnianie przeciążanych metod, ponieważ funkcja zadeklarowana z jednym parametrem standardowym może być wywoływana z dwoma zestawami parametrów.

W bibliotece FCL znaleźć można wiele przykładów przeciążanych metod; należą do nich między innymi metody rysujące, które zależnie od wyboru programisty pracować mogą ze współrzędnymi podawanym pojedynczo albo zgromadzonymi w ramach struktury `Rectangle`. Z kolei same współrzędne mogą być podawane jako liczby całkowite lub liczby zmiennoprzecinkowe. W Delphi metody `DrawRectangle` pochodzące z klasy `Graphics` można próbować przedstawić tak jak na listingu 3.128.

Listing 3.128. *Rekonstrukcja deklaracji metod DrawRenctangle w Delphi*

```
type
  Graphics = class
    procedure DrawRectangle(aPen: Pen; aRectangle: Rectangle); overload;
    procedure DrawRectangle(aPen: Pen; x, y, w, h: Integer); overload;
    procedure DrawRectangle(aPen: Pen; x, y, w, h: Double); overload;
  end;

implementation
```

```
// W implementacji nie stosujemy już oznaczenia overload
procedure Graphics.DrawRectangle(aPen: Pen; aRectangle: Rectangle);
begin
  ...
```

3.8.3. Wskaźniki metod

Dzięki typom proceduralnym już w historycznych wersjach Delphi możliwe było przekazywanie procedur i funkcji w parametrach lub zapisywanie ich wewnątrz zmiennych. W poniższym przykładzie deklarowany jest typ wskaźnika na funkcję przyjmującą w parametrze liczbę całkowitą i podobnie zwracającą taką liczbę w wyniku swojego działania:

```
type
  TIntFunction = function(x: Integer): Integer;
```

Deklaracja różni się od zwyczajnej deklaracji funkcji tym, że słowo kluczowe `function` nie znajduje się na jej początku, ale wypisywane jest dopiero po znaku równości. Deklarowana w ten sposób nazwa `TIntFunction` jest nazwą **typu** funkcji, który może być podawany jako typ parametru przekazywanego do innej funkcji:

```
procedure RysujKrzywa(Function: TIntFunction);
```

Teraz możemy założyć, że w programie działać będą dwie konkretne funkcje (`Funkcja1` i `Funkcja2`) zgodne z zadeklarowanym typem:

```
function Funkcja1(x: Integer): Integer;
function Funkcja2(x: Integer): Integer;
```

W związku z tym możemy każdą z tych funkcji przekazać w parametrze do przedstawianej przed chwilą procedury `RysujKrzywa`:

```
RysujKrzywa(Funkcja1);
RysujKrzywa(Funkcja2);
```

Obiektowym wariantem typów proceduralnych są typy metod. Jeżeli w podanym wyżej przykładzie zamiast zwyczajnej funkcji w parametrze ma być przekazywana metoda pewnego obiektu, to musi zostać przygotowana specjalna deklaracja typu:

```
type
  TIntMethod = function(X: Integer): Integer of object;
```

Typy metod nie są zgodne z typami proceduralnymi, ponieważ metody wymagają podania dodatkowego, niewidocznego parametru `self`, będącego wskaźnikiem na obiekt, na rzecz którego wywoływana jest metoda (była o tym mowa w punkcie 3.2.3). Dodatkowo, wskaźniki metod obok wskazania samej metody zapisują też dane obiektu, z którego pochodzą, i które będą przekazywane metodzie w parametrze `self` (jest to ogromna przewaga tych wskaźników nad podobnymi wskaźnikami metod stosowanymi w języku C++).

Typy metod stosowane są przede wszystkim do definiowania zdarzeń i w związku z tym dokładnie omówione zostaną w punkcie 6.2.1.

3.9. Wyjątki

Wyjątki najczęściej stosowane są jako mechanizm obsługi błędów, choć w rzeczywistości mogą one być stosowane do obsługiwania wszystkich rodzajów **sytuacji wyjątkowych** (angielskie słowo *Exception* oznacza właśnie wyjątek). W momencie wystąpienia wyjątku przerywany jest aktualny przepływ programu i poszukiwana jest metoda, która mogłaby przechwycić ten wyjątek. Może to być też ta sama metoda, w której wyjątek wystąpił. Jeżeli jednak metoda ta **nie** przechwyci wyjątku, to jej wykonywanie jest przerywane, a kontrola przekazywana jest to metody wywołującej. Jeżeli ona również nie przechwyci wyjątku, to następuje przerwanie także jej działania. Dzieje się tak do czasu, aż znaleziona zostanie metoda obsługująca dany wyjątek, przy czym każda przerywana metoda ma jeszcze szanse na końcową obsługę wyjątku (blok `finally`), w której może wykonać ważne prace czyszczące.

Jeżeli żadna z wywołujących metod nie będzie w stanie obsłużyć wyjątku, to jest on obsługiwany w środowisku CLR, które wyświetla specjalny komunikat i pozwala użytkownikowi wybrać pomiędzy zakończeniem programu a uruchomieniem debugera w celu sprawdzenia programu.

Wyjątki są w Delphi obsługiwane od pierwszej wersji pakietu, a dzisiaj, w środowisku .NET, stały się preferowaną metodą obsługi błędów w programie.

3.9.1. Wywoływanie wyjątków

Na początek zajmiemy się źródłem powstawania wyjątków. Jako przykład przyjmiemy tutaj zdefiniowany w środowisku .NET wyjątek `ArgumentException` opisujący sytuację, w której dana metoda wywołana została z nieprawidłowymi wartościami parametrów. Przygotujmy sobie przykładową metodę pobierającą w parametrze kontrolkę, przy czym przekazywana jej kontrolka musi być częścią jakiegokolwiek formularza. Jeżeli metoda stwierdzi, że przekazana jej kontrolka nie spełnia tych podstawowych założeń, to może wywołać odpowiedni wyjątek stosując kod przedstawiony na listingu 3.129.

Listing 3.129. *Wywołanie wyjątku wewnątrz metody*

```
procedure MojaKlasa.UzyjKontrolki(Param:Control);
begin
  if not ((Param as Control).TopLevelControl is Form)
    then raise ArgumentException.Create('Parametr musi być kontrolką '
                                        +'będącą częścią formularza.');
```

Tworzenie obiektu wyjątku przebiega dokładnie tak samo jak w przypadku każdego innego obiektu, czyli poprzez wywołanie konstruktora `Create` odpowiedniej klasy. Do rozpoczęcia specjalnego postępowania związanego z obsługą wyjątków konieczne jest zastosowanie słowa kluczowego `raise`. Znaczenie tego słowa (podnieść) doskonale opisuje to, co dzieje się w momencie wywołania wyjątku. Przerywane jest wykonywanie aktualnej metody, a wyjątek przenoszony jest na wyższy poziom stosu wywołań. Możliwe sposoby reakcji metody wywołującej przedstawiam w punkcie 3.9.4.

3.9.2. Klasy wyjątków

Zamiast dla każdego możliwego błędu przygotowywać kod obsługi, z którym najprawdopodobniej powiązany będzie jakiś specjalny tekst komunikatu, w przestrzeniach nazw biblioteki FCL i w modułach biblioteki VCL przygotowano dla każdego rodzaju błędu osobną klasę wyjątku. Każda z takich klas została bezpośrednio lub pośrednio wywiedziona z klasy `System.Exception`.

Klasa `Exception` deklaruje te wszystkie elementy, które są potrzebne we wszystkich innych klasach wyjątków: ciąg znaków z komunikatem o błędzie, który można odczytać z właściwości `Exception.Message`, a także inne dane, które są automatycznie przygotowywane przez środowisko CLR, takie jak dane metody, w której wystąpił wyjątek oraz dane obiektu, na rzecz którego metoda ta była wywoływana.

Definiowanie własnych wyjątków

Jeżeli w naszym programie może występować nowy rodzaj błędu, który chcielibyśmy uwzględniać w procedurach obsługi błędów, to możemy przygotować dla niego specjalną klasę wyjątku. Najprawdopodobniej będziemy chcieli umieścić ją gdzieś w istniejącej już hierarchii klas wyjątków, dlatego jej deklaracja może przypominać tę podaną na listingu 3.130.

Listing 3.130. *Deklaracja nowej klasy wyjątków*

```
type
    // Klasa ApplicationException jest jedną z klas wyjątków
    // zdefiniowanych w przestrzeni nazw System biblioteki FCL
ETableOverflow = class(ApplictationException)
  public
    TableSize: LongInt;
end;
```

Wyjątek ten może być wywoływany w programie w momencie, gdy w wewnętrznej tablicy stosowanej w programie nie będzie już miejsca na nowe elementy (niezależnie do tego, jak może dojść do takiego ograniczenia). W takiej sytuacji można też skorzystać z klasy `ApplicationException` albo nawet bezpośrednio z klasy `Exception`, jeżeli nie byłby potrzebny zadeklarowany w nowej klasie element `TableSize`. W takiej sytuacji pozbawiamy się jednak możliwości odróżnienia w programie naszego własnoręcznie zdefiniowanego wyjątku od pozostałych (o przechwytywaniu wyjątków mówić będę w punkcie 3.9.4).

Podany wyżej przykład wyjaśnia też, dlatego klasy wyjątków przechowują tak niewielkie ilości danych. Zdefiniowany w naszej klasie wyjątku element `TableSize` najprawdopodobniej nie będzie w programie w ogóle potrzebny, ponieważ może on w inny sposób ustalić, jaka jest aktualna wielkość tablicy.

3.9.3. Zabezpieczanie kodu z wykorzystaniem sekcji finally

Przyjrzymy się tutaj ogólnemu przypadkowi metody, która **nie** przeprowadza końcowej obsługi wyjątków, ale przekazuje wyjątki do funkcji nadrzędnych w stosie wywołań. Jak się okazuje, w wielu przypadkach nie wystarcza proste przerwanie działania funkcji. Jeżeli funkcja w czasie swojego działania rezerwowała jakieś zasoby, to powinna je zwalniać, gdy tylko wywołany zostanie jakikolwiek wyjątek. Bez wyjątków kod takiej metody mógłby wyglądać na przykład tak, jak pokazano na listingu 3.131.

Listing 3.131. *Kod metody nieobsługującej wyjątków*

```
Plik := FileStream.Create(...);  // Rezerwowanie zasobu
if WywolaniePowodujaceBlad = -1 then begin  // Źródło błędu!!!
  Plik.Close;  // "Awaryjne" zwolnienie zasobu
  exit;
end;
... pozostałe instrukcje ...
Plik.Close;  // normalne zwolnienie zasobów
```

W przykładzie tym założono, że funkcja `WywolaniePowodujaceBlad` zwraca informację o wystąpieniu błędu w postaci wartości `-1`. Przedstawiony na powyższym listingu kod samodzielnie decyduje, czy może kontynuować pracę, czy też powinien zakończyć działanie funkcji wywołaniem `exit`. Ten ostatni wariant wiąże się oczywiście z koniecznością zamknięcia otwartego pliku.

Jeżeli jednak funkcja `WywolaniePowodujaceBlad` nie przekazywałaby informacji o błędzie w zwracanej wartości, ale generowałaby wyjątek, to kod podanej wyżej metody musiałby zostać zmieniony tak, jak pokazano na listingu 3.132, aby metoda nadal mogła zwalniać zarezerwowany plik w przypadku awarii.

Listing 3.132. *Kod metody z obsługą wyjątków*

```
Plik := FileStream.Create(...);  // Rezerwowanie zasobu
try
  WywolaniePowodujaceBlad;       // Źródło błędu!!!
... pozostałe instrukcje ...
finally
 Plik.Close;  // zwolnienie zasobu
end;
```

W przypadku wystąpienia wyjątku wszystkie instrukcje zapisane pomiędzy wywołaniem funkcji `WywolaniePowodujaceBlad`, a słowem kluczowym `finally` zostaną całkowicie pominięte w wykonywanej metodzie. Jak widać, tutaj program nie ma możliwości wyboru, takiej jak w pierwszym wariancie.

Kod w konstrukcji `try…finally…end` wykonywany jest następująco: jeżeli w czasie wykonywania instrukcji umieszczonych pomiędzy słowami kluczowymi `try` i `finally` wystąpi jakikolwiek wyjątek, to blok jest natychmiast opuszczany, a program przystępuje do wykonywania instrukcji zapisanych pomiędzy słowami kluczowymi `finally` i `end`.

Po tym wszystkim metoda jest opuszczana bez wykonywania jakichkolwiek instrukcji znajdujących się za blokiem `finally`. Jedynym wyjątkiem jest sytuacja, w której blok `try … finally` zamknięty jest wewnątrz innego bloku `try…finally`. W takim wypadku wykonywane są jeszcze instrukcje zapisane w sekcji `finally` bloku zewnętrznego.

Jeżeli w czasie pracy metody nie wystąpi żaden wyjątek, to jej kod wykonywany jest tak, jakby w ogóle nie było w nim słów kluczowych `try` i `finally`: najpierw wykonywany jest cały blok `try`, a następnie cały blok `finally`.

Zasoby chronione

Z powodu tego, że zasoby zarezerwowane w bloku `try` z całą pewnością zostaną zwolnione w bloku `finally`, blok `try` bardzo często nazywany jest też *blokiem chronionym*. Jeżeli wewnątrz tego bloku rezerwowane są jeszcze inne zasoby, które również muszą być odpowiednio chronione, to konieczne jest zagnieżdżanie struktury sterującej, tak jak pokazano na listingu 3.133.

Listing 3.133. *Zagnieżdżone bloki chronione*

```
AllocateRes1 { Zasób pierwszy }
try  { blok chroniony dla zasobu pierwszego }
  ...  { Pozycja (*) }
  AllocateRes2 { zasób drugi }
  try  { blok chroniony dla zasobu drugiego }
    ...
  finally
    FreeRes2; { Zwolnienie zasobu drugiego }
  end;
finally
  FreeRes1; { Zwolnienie zasobu pierwszego }
end;
```

Jeżeli w podanym kodzie wyjątek wystąpi w pierwszym, ale jeszcze poza drugim blokiem chronionym (pozycja oznaczona znakiem gwiazdki `(*)`), to wykonany zostanie wyłącznie blok `finally` związany z zewnętrznym blokiem chronionym. Wewnętrzny blok `finally` nie musi być wykonywany, ponieważ drugi zasób nie został jeszcze zarezerwowany. Blok `finally` włączany jest do przewidywanego przepływu programu dopiero wtedy, gdy wykonana zostanie pierwsza instrukcja z powiązanego z nim bloku `try`, a wtedy wykonywany jest nieodwołalnie, nawet jeżeli wyjdziemy z bloku `try` wywołując instrukcję `exit`.

3.9.4. Obsługa wyjątków

W poprzednim punkcie opisywane były reakcje na wystąpienie wyjątku, które jednak nie powodowały jego zniesienia. Po wykonaniu instrukcji z bloku `finally` obiekt wyjątku tak długo przekazywany jest do funkcji wywołujących na wyższych poziomach, aż w którejś z nich znajdzie się blok `except` obsługujący ten wyjątek. Wszystkie funkcje, które nie będą miały takiego bloku, zostaną natychmiast przerwane i wykonany zostanie w nich jedynie ewentualny blok `finally`.

Blok except (obsługujący wyjątki) służy do przygotowania takiej obsługi błędu, żeby program mógł poprawnie wznowić swoje działanie. Początek chronionego tak bloku kodu rozpoczyna się od słowa kluczowego try, podobnie jak w blokach chronionych słowem kluczowym finally.

W przykładzie podanym na listingu 3.134 informacja o wyjątku przekazywana jest bezpośrednio do użytkownika.

Listing 3.134. *Przykładowa obsługa wyjątku w programie*

```
try
  stream := FileStream.Create(...);
except
  on FileNotFoundException do
    MessageBox.Show('Pliku nie znaleziono!');
end;
```

Sprawdzanie klasy wyjątku

Jak widać w przykładowym kodzie, bloki except są o wiele bardziej złożone niż bloki finally, ponieważ pozwalają na sprawdzanie klasy przechwyconego wyjątku i w swojej strukturze są nieco podobne do instrukcji case. Po słowie kluczowym on podać należy klasę wyjątku, którą chcielibyśmy obsłużyć. Jeżeli wyjątek ma podaną tu klasę lub klasę z niej wywiedzioną, to wykonywane będą instrukcje zapisane za słowem kluczowym do. Po ich wykonaniu, blok except zostanie zakończony, nawet jeżeli znajdują się w nim jeszcze inne klauzule on…do, które mogły obsłużyć przechwycony wyjątek. W kodzie z listingu 3.135 przedstawiam taką właśnie klauzulę, która nigdy nie zostanie wykonana, ponieważ przed nią zapisana jest obsługa wyjątków klasy bazowej jej wyjątku.

Listing 3.135. *Kolejność zapisywania klas wyjątków ma duże znaczenie*

```
except
  on SystemException do ...
  on FormatException do ...   // Klasa FormatException jest klasą wywiedzioną z klasy
SystemException
end;
```

Obie klauzule on…do muszą zostać zamienione miejscami, żeby przedstawiony kod mógł funkcjonować właściwie. Po takiej zamianie drugi warunek zapisany za słowem kluczowym on obsługiwałby wyłącznie wyjątki klasy SystemException, ale nie obsługiwałby wyjątków klasy FormatException.

Można tu też wymienić inne podobieństwa bloku obsługi wyjątków do instrukcji case: więcej niż jedna instrukcja musi zostać zamknięta wewnątrz bloku begin…end, po słowie kluczowym on podawać można więcej niż jedną klasę wyjątku, a poza tym przewidziana została „gałąź" else obsługująca niewymienione do tej pory klasy wyjątków. To ostatnie rozwiązanie nie jest raczej zalecanie, ponieważ w ten sposób obsługiwane będą też te wyjątki, których powodu wywołania nie będziemy znać. W takiej sytuacji nie będzie możliwe wychwycenie takiego wyjątku w miejscu do tego przewidzianym.

Przechwytywanie obiektu wyjątku

W przedstawionych wyżej przykładach sprawdzana była tylko klasa wyjątku. Można jednak przechwytywać też sam obiekt utworzony w instrukcji `raise` za pomocą konstruktora `Create`. W tym celu należy nadać mu jakąkolwiek nazwę i zapisać ją za słowem kluczowym `on`, stosując zapis podobny do deklaracji zmiennej. Następnie można korzystać z elementów obiektu wyjątku, na przykład odczytywać treść komunikatu ze zmiennej `Message`, która dostępna jest we wszystkich klasach wyjątków, lub zawartość zmiennej `TableSize` dostępnej wyłącznie w klasie wyjątku `ETableOverflow` zdefiniowanej w punkcie 3.9.2. Przykładowy kod korzystający z tej możliwości przedstawiam na listingu 3.136.

Listing 3.136. *Przechwytywanie obiektu wyjątku*

```
except
  on E: ETableOverflow do begin
    MessageBox.Show('Tablica jest zbyt mała. Aktualna wielkość = '
                    + E.TableSize.ToString);
  end;
end;
```

Istnieje jeszcze jedna możliwość odebrania obiektu zdarzenia, którą można wykorzystywać w gałęzi `else` bloku `except`. Dostępna w Delphi funkcja `ExceptObject` zwraca aktualny obiekt wyjątku w postaci referencji typu `System.Object` albo wartość `nil`, jeżeli aktualnie nie ma żadnego wyjątku.

Wznawianie wyjątku

W bloku obsługującym wyjątek można ponownie wykorzystać słowo kluczowe `raise`. Jeżeli zostanie ono wywołane bez żadnych parametrów, to powoduje ponowienie tego wyjątku, który jest aktualnie obsługiwany. Jeżeli w naszej funkcji możemy tylko częściowo obsłużyć stan błędu, to dzięki takiemu rozwiązaniu mamy możliwość wykonania odpowiednich prac w jednym bloku `except` i przekazania wyjątku do dalszej obróbki w funkcji wywołującej. Przykład takiej sytuacji przedstawiam na listingu 3.137.

Listing 3.137. *Przekazanie częściowo obsłużonego wyjątku do funkcji wywołującej*

```
try
  ...
except
  on Exception do begin
    { Zwalniane są wszystkie zasoby, które nie będą potrzebne tylko w przypadku
      wystąpienia pewnego wyjątku, ale w przypadku bezbłędnego wykonania
      metody powinny pozostać zarezerwowane. }
    ...
    raise; { Pozostała część obsługi błędu wykonywana jest w funkcji wywołującej }
  end;
end;
```

Kod przedstawiony w tym przykładzie w swoim działaniu podobny jest do bloku `finally`, który jednak wykonywany jest wyłącznie w przypadku wystąpienia wyjątku, ale w sytuacji bezbłędnego działania bloku `try` jest pomijany.

Jeżeli w bloku except wywołana zostanie instrukcja raise tworząca nowy obiekt wyjątku, to zastąpi on obsługiwany do tej pory wyjątek, chyba że zostanie on obsłużony w ramach tego samego bloku except. Takie działanie jest specjalnym przypadkiem nazywanym zagnieżdżaniem wyjątków.

Zagnieżdżanie wyjątków

W czasie wykonywania bloku except może oczywiście zdarzyć się sytuacja, w której wygenerowany zostanie kolejny wyjątek. Taki dodatkowy wyjątek może być obsługiwany wewnątrz kolejnej struktury try…except (lub finally), na przykład tak jak na listingu 3.138.

Listing 3.138. *Zagnieżdżona obsługa wyjątków*

```
try
except
  on Wyjatek1 do
   try
   except
    on Wyjatek2 do
   end;
end;
```

Jeżeli w pierwszym bloku except **nie** zostanie obsłużony wygenerowany w nim wyjątek, to przekazany zostanie on na poziom wyżej, zastępując tym samym aktualny obiekt wyjątku.

Zakończenie wyjątku

Wygenerowany w programie wyjątek uznawany jest za obsłużony w momencie wywołania bloku obsługi wyjątku, w którym nie został on odnowiony. Blok obsługujący wyjątek może przyjmować trzy różne postaci:

- Klauzuli on zgodnej z aktualną klasą wyjątku, która kończy stan wyjątkowy w sposób opisany powyżej.
- Części else bloku obsługi wyjątków, która może obsłużyć wszystkie pozostałe wyjątki.
- Bloku except, który zamiast listy obsługiwanych wyjątków ujętych w klauzulach on może zawierać tylko ciąg instrukcji. Tak zapisany blok obsługuje wszystkie przechwytywane wyjątki, co jest równoznaczne z blokiem zawierającym wyłącznie część za słowem kluczowym else i nieposiadającym żadnej klauzuli on.

Funkcje nieposiadające części obsługi wyjątków stają się dla wyjątków tylko elementem pośredniczącym w przekazywaniu wyjątku do wyższego poziomu obsługi. Jeżeli w programie nie znajdzie się żaden blok obsługujący aktualny wyjątek, to przekazywany jest on ostatecznie do środowiska CLR, które wyświetla na ekranie komunikat o wystąpieniu wyjątku i zamyka program.

Standardowa obsługa wyjątków wykonywana w środowisku CLR różni się w wielu miejscach od schematu obsługi stosowanego w bibliotece VCL. Aplikacje VCL po wystąpieniu wyjątku najczęściej kontynuują swoją pracę, ponieważ biblioteka VCL wyświetla co prawda komunikat o błędzie dla wszystkich wyjątków występujących w czasie pracy aplikacji i głównego formularza, ale poza tym ignoruje nieobsłużone wyjątki. IDE Delphi jest doskonałym przykładem programu, który może całkiem sprawnie pracować po wystąpieniu pewnych rodzajów wyjątków. W środowisku .NET również można ignorować wszystkie występujące w aplikacji wyjątki, wstawiając w krytycznym miejscu w programie pusty blok except. W takiej sytuacji należy uznać, że rozwiązanie stosowane przez środowisko CLR jest właściwszym, tym bardziej, że dostępna jest w nim również stosowana w bibliotece VCL metoda „przymykania oczu" na pojawiające się wyjątki, choć nie jest ona uznawana za rozwiązanie domyślne.

3.9.5. Asercja

Na zakończenie zajmiemy się techniką wyszukiwania błędów i diagnozowania problemów, w której wykorzystywane są również wyjątki, a konkretnie klasa wyjątków `Borland.System.EAssertionFailed`.

Korzystając z funkcji debugera opisywanych w podrozdziale 1.7 możemy sprawdzać w czasie działania programu, czy wszystko przebiega zgodnie z oczekiwaniami, na przykład to, czy pętla `while` prawidłowo „schodzi" do wartości zera. Mówiąc bardziej ogólnie — możemy kontrolować, czy spełniane są pewne **warunki** działania programu. Muszę tu zaznaczyć, że Delphi pozwala na wykorzystywanie specjalnych mechanizmów kontrolujących takie warunki pracy programu:

- Stosując standardową instrukcję `Assert` można sprawdzać pewne warunki, które muszą być spełnione w prawidłowo działającym programie. Jeżeli określony warunek nie jest spełniony, to instrukcja `Assert` podnosi wyjątek `EAssertionFailed`.
- Szczególny status instrukcji `Assert` polega jednak na tym, że w opcjach kompilatora można ustalić, czy instrukcja ta ma być uwzględniana w kodzie źródłowym programu, czy też nie (menu *Project/Options/Compiler*, przełącznik *Assertions*).

W czasie tworzenia programu, częste korzystanie z instrukcji `Assert` i związanych z nią testów warunków umożliwia szybkie wyszukiwanie wielu błędów. Po zakończeniu tworzenia programu można bardzo łatwo wyłączyć wszystkie testy wykonywane instrukcjami `Assert`. W takich okolicznościach można zakładać, że w gotowym programie wszystkie podstawowe warunki jego działania zostaną spełnione, wobec czego wyłączenie testów wykonywanych instrukcjami `Assert` ma jeszcze tę zaletę, że nieznacznie podnosi prędkość działania całej aplikacji i powoduje zmniejszenie rozmiarów pliku wykonywalnego. Na listingu 3.139 przedstawiam wyczerpujący przykład.

Listing 3.139. *Przykład wykorzystywania instrukcji Assert*

```
i := 1;
repeat
  Assert(i < 100, 'Warunek i < 100 nie jest spełniany!');
  i := i + 2;
until i = 100;
```

W podanym kodzie przykładowym chcemy się upewnić, że pętla rzeczywiście wykonywana będzie tak długo, jak długo zmienna `i` będzie miała wartość mniejszą od `100`. Zmienna `i` przyjmuje jednak wyłącznie wartości nieparzyste, przez co nigdy nie zostanie spełniony warunek końcowy `i = 100`, ponieważ zmienna `i` osiągnąwszy wartość `101` nie spełni warunku zakończenia pętli. W tym miejscu wkroczy do działania instrukcja `Assert`, wskazując na przyczynę błędu.

Co prawda instrukcja `Assert` języka Object Pascal ma w porównaniu do makrodefinicji `Assert` z języka C++ tę wadę, że nie wyświetla automatycznie w oknie komunikatu pełnej treści warunku, ale za to pozwala uniknąć stosowania schematów kodu, takich jak przedstawiony na listingu 3.140, który musiałby się wielokrotnie pojawiać w kodzie źródłowym programu.

Listing 3.140. *Mało praktyczna metoda wyłączania sprawdzania warunków działania programu*

```
{$ifdef AsercjaAktywna}
  if not (i < 100) then
    // Wyświetlenie komunikatu lub wywołanie wyjątku
{$endif}
```

Środowisko .NET oferuje też metodę `Debug.Assert`, będącą pewną alternatywą w stosunku do dostępnej w Delphi instrukcji `Assert`. Użycie tej metody wymaga dołączenia do modułu przestrzeni nazw `System.Diagnostics`. Jak się okazuje, metoda `Debug.Assert` działa na zupełnie innej zasadzie niż instrukcja `Assert` z Delphi. Nie wywołuje żadnych wyjątków, ale standardowo wyświetla tylko okno z komunikatem, przy czym komunikat ten można też skierować do debugera. Ignorowanie wyjątku (czyli pozwolenie na dalszą pracę programu) jest tu znacznie łatwiejsze niż w przypadku instrukcji `Assert` z Delphi, ponieważ wymaga tylko kliknięcia odpowiedniego przycisku w oknie komunikatu. Z drugiej strony, nieco trudniejsze jest debugowanie programu, ponieważ po przejściu do debugera program zostanie zatrzymany głęboko wewnątrz procedur biblioteki FCL, a nie w miejscu, w którym nastąpiło wykrycie nieprawidłowości warunków pracy programu przez metodę `Debug.Assert`.

Podobnie jak w przypadku instrukcji `Assert` z Delphi, możliwe jest grupowe włączanie i wyłącznie kontroli wykonywanych przez metodę `Debug.Assert`, a dodatkowo można dokonywać też zmian innych ustawień opisanych dokładnie w dokumentacji pakietu SDK, na stronie opisującej klasę `System.Diagnosics.Debug`. Stosowanie instrukcji `Assert` z Delphi zalecane jest wszędzie tam, gdzie aplikacja nie powinna być uzależniona od środowiska .NET, czyli na przykład w aplikacjach VCL.NET, które mają być kompilowane również w środowisku Win32.