helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

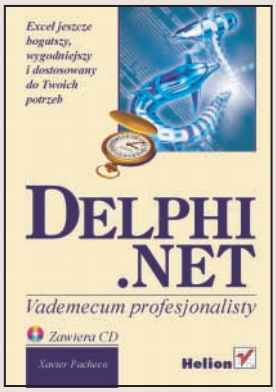
TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Delphi dla .NET. Vademecum profesjonalisty

Autor: Xavier Pacheco
Tłumaczenie: Rafał Jońca, Szymon Kobalczyk, Mikołaj Szczepaniak
ISBN: 83-7361-631-4
Tytuł oryginału: Delphi for .net Developers Guide
Format: B5, stron: 944



Platforma .NET staje się coraz bardziej popularna. Powstaje coraz więcej aplikacji realizowanych właśnie pod jej kątem. Udostępniane przez platformę .NET mechanizmy pozwalają na szybkie tworzenie aplikacji, co przysparza jej wielu zwolenników. Do stworzenia aplikacji nie wystarczą jednak tylko mechanizmy, nawet najlepsze. Niezbędne jest wygodne i uniwersalne środowisko programowania, jakim niewątpliwie jest Delphi. Jego najnowsza wersja umożliwia pełne wykorzystanie potencjału platformy .NET.

„Delphi dla .NET. Vademecum profesjonalisty" to podręcznik przedstawiający możliwości tworzenia aplikacji .NET za pomocą narzędzia programistycznego firmy Borland. W książce zamieszczono praktyczne przykłady, omówienie ciekawych technik oraz przydatne wskazówki na temat efektywnego korzystania z potencjału platformy .NET Framework. Książka zawiera dokładne omówienie języka programowania Delphi, zaawansowanych zagadnień związanych z programowaniem dla platformy .NET (w tym z zarządzaniem pamięcią), mechanizmów COM-Interop i Reflection, biblioteki GDI+, wytwarzania komponentów typu Windows Forms oraz Web Forms i wiele innych. Znajdziesz tu także solidną analizę kluczowych technologii platformy .NET, takich jak ADO.NET i ASP.NET, włącznie z mnóstwem przykładów demonstrujących ich możliwości.

- Podstawowe wiadomości o platformie .NET i rodzaje aplikacji .NET
- Przegląd elementów platformy .NET
- Delphi for .NET — środowisko i język programowania
- Biblioteka klas platformy .NET
- Korzystanie z biblioteki GDI+
- Środowisko Mono
- Programowanie wielowątkowe
- Usługi COM Interop i Platform Invocation Service
- Programowanie aplikacji bazodanowych
- Tworzenie stron WWW w technologii ASP.NET

Jeśli szukasz książki poświęconej technologii .NET i programowaniu w języku Delphi aplikacji zgodnych z tą technologią, trafiłeś najlepiej, jak tylko mogłeś.

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

Spis treści

O Autorze **19**

Wprowadzenie **21**

Część I Podstawy technologii .NET **25**

Rozdział 1. Wprowadzenie do technologii .NET **27**

Koncepcja .NET 28
Wizja .NET 28
Składniki platformy .NET Framework
— środowisko Common Language Runtime (CLR) i biblioteki Class Libraries 31
Rodzaje aplikacji .NET 32
Czym jest biblioteka VCL for .NET? 33
Rozproszone wytwarzanie oprogramowania za pośrednictwem usług Web Services 34
Definicja usług Web Services 35
Klienty usług Web Services 37
Narzędzia programowania usług Web Services 38

Rozdział 2. Przegląd platformy .NET Framework **39**

Od tworzenia do uruchamiania 39
Środowisko Common Language Runtime (CLR) 40
Moduły zarządzane 40
Podzespoły 41
Kod zarządzany i niezarządzany 42
Kompilowanie i uruchamianie kodu MSIL i JIT 42
System Common Type System (CTS) 45
Typy wartościowe 45
Typy referencyjne 46
Specyfikacja Common Language Specification (CLS) 46
Platforma .NET Framework i biblioteka Base Class Library (BCL) 47
Przestrzenie nazw 47
Przestrzeń nazw System 47
Główne podprzestrzenie przestrzeni nazw System 47

Część II Język programowania Delphi for .NET **53**

Rozdział 3. Wprowadzenie do języka Delphi for .NET i nowego środowiska IDE **55**

Omówienie Delphi for .NET 55
Wprowadzenie do zintegrowanego środowiska programowania (IDE) 56
Strona powitalna 57
Obszar projektowania 57
Formularze 60

Paleta narzędzi i fragmenty kodu ... 61
Inspektor obiektów ... 62
Edytor kodu ... 63
Menedżer projektu ... 65
Widok modelu ... 66
Eksplorator danych ... 67
Repozytorium obiektów ... 67
Eksplorator kodu ... 68
Lista zadań do wykonania ... 68

Rozdział 4. Programy, moduły i przestrzenie nazw ... 71
Struktury oparte na modułach zarządzanych ... 71
Przestrzenie nazw ... 71
Struktura modułu ... 73
Składnia klauzuli uses ... 75
Cykliczne odwołania do modułów ... 76
Przestrzenie nazw ... 77
Deklaracja przestrzeni nazw ... 77
Stosowanie przestrzeni nazw ... 79
Klauzula namespaces ... 79
Identyfikowanie ogólnych przestrzeni nazw ... 79
Aliasy modułów ... 80

Rozdział 5. Język Delphi ... 81
Wszystko o technologii .NET ... 81
Komentarze ... 82
Procedury i funkcje ... 82
Nawiasy w wywołaniach ... 83
Przeciążanie ... 83
Domyślne wartości parametrów ... 83
Zmienne ... 85
Stałe ... 86
Operatory ... 88
Operatory przypisania ... 88
Operatory porównania ... 89
Operatory logiczne ... 89
Operatory arytmetyczne ... 90
Operatory bitowe ... 91
Procedury zwiększania i zmniejszania ... 92
Operatory typu „zrób i przypisz” ... 92
Typy języka Delphi ... 93
Obiekty, wszędzie tylko obiekty! ... 93
Zestawienie typów ... 94
Znaki ... 95
Typy wariantowe ... 95
Typy definiowane przez użytkownika ... 99
Tablice ... 100
Tablice dynamiczne ... 101
Rekordy ... 103
Zbiory ... 104
„Niebezpieczny” kod ... 106
Wskaźniki ... 107
Klasy i obiekty ... 110
Aliasy typów ... 111
Rzutowanie i konwersja typów ... 112
Zasoby łańcuchowe ... 113

Testowanie warunków ... 113
Instrukcja if ... 114
Stosowanie instrukcji case ... 114
Pętle ... 115
Pętla for ... 115
Pętla while ... 116
Pętla repeat-until ... 117
Instrukcja Break ... 117
Instrukcja Continue ... 117
Procedury i funkcje ... 118
Przekazywanie parametrów ... 119
Zakres ... 122
Moduły i przestrzenie nazw ... 123
Klauzula uses ... 124
Cykliczne odwołania do modułów ... 125
Pakiety i podzespoły ... 125
Programowanie obiektowe ... 126
Stosowanie obiektów Delphi ... 127
Deklaracja i tworzenie egzemplarza ... 128
Destrukcja ... 129
Przodek wszystkich obiektów ... 129
Pola ... 129
Metody ... 130
Typy metod ... 131
Referencje do klas ... 134
Właściwości ... 135
Zdarzenia ... 136
Specyfikatory widoczności ... 138
Klasy zaprzyjaźnione ... 140
Klasy pomocnicze ... 140
Typy zagnieżdżone ... 141
Przeciążanie operatorów ... 142
Atrybuty ... 142
Interfejsy ... 143
Ujednolicony mechanizm obsługi wyjątków ... 147
Klasy wyjątków ... 150
Przepływ sterowania działaniem ... 151
Ponowne generowanie wyjątków ... 153

Część III Praca z biblioteką klas platformy .NET Framework ... 155

Rozdział 6. Podzespoły .NET, biblioteki i pakiety ... 157

Podstawowe podzespoły ... 157
Przeglądanie zawartości podzespołów i występujących między nimi zależności ... 158
Mechanizm GAC ... 159
Konstruowanie podzespołów ... 160
Dlaczego stosujemy podzespoły .NET? ... 161
Stosowanie pakietów do budowy podzespołów ... 161
Stosowanie bibliotek do budowania podzespołów ... 166
Stosowanie podzespołów w języku Delphi ... 170
Stosowanie podzespołów z języka Delphi w programach C# ... 171
Instalacja pakietów w środowisku Delphi ... 171
Podzespoły ze ścisłą kontrolą nazw ... 172
Dynamicznie wczytywane podzespoły ... 173

Rozdział 7. Programowanie przy użyciu GDI+. Rysowanie w .NET 175
Pojęcia podstawowe 175
Przestrzenie nazw GDI+ 175
Klasa Graphics 176
Układ współrzędnych w systemie Windows 176
Rysowanie prostych 178
Klasy Pen i Brush 178
Rysowanie prostych 179
Końcówki linii 181
Łączenie linii — klasa GraphicsPath 183
Rysowanie krzywych 185
Krzywa sklejana typu cardinal 185
Krzywa sklejana Beziera 185
Rysowanie figur 189
Rysowanie prostokątów 189
Rysowanie elips 190
Rysowanie wielokątów 191
Rysowanie wycinków elips 191
Więcej o „pędzlu" LinearGradientBrush 193
Klasy GraphicsPath i Region 193
Rysowanie za pomocą klasy GraphicsPath 194
Rysowanie za pomocą klasy Region 195
Obszary przycinające 197
Praca z obrazami 199
Klasy Image 200
Wczytywanie i tworzenie bitmap 200
Zmiana rozdzielczości obrazu 201
Rysowanie obrazów 202
Interpolacja 203
Rysowanie efektu zwierciadła (lustra) 204
Stosowanie metod przekształcania obrazów 206
Tworzenie miniatur 210
Przegląd układów współrzędnych 211
Przykład animacji 213

Rozdział 8. Mono. Wieloplatformowe środowisko .NET 221
Cechy środowiska Mono 221
Historia Mono 222
Po co stworzono Mono? 223
Mapa drogowa Mono 224
Cele Mono 1.0 224
Cele Mono 1.2 225
Cele Mono 1.4 225
Instalacja i ustawienia 226
Instalacja środowiska uruchomieniowego Mono — program Red Carpet 226
Tworzenie naszego pierwszego programu Mono 229
Uruchamianie w środowisku Mono (w systemie Linux)
podzespołów wygenerowanych w Delphi 230
Wieloplatformowa technologia ASP.NET 234
Wdrażanie rozwiązań ASP.NET w środowisku Mono 236
Konfiguracja XSP 236
Parametry środowiska uruchomieniowego XSP 236
Kilka uwag i możliwych kierunków rozwoju rozszerzeń zaprezentowanego przykładu 238
Mono i technologia ADO.NET 239
Mono i serwer Apache 243
Mono i przestrzeń nazw System.Windows.Forms 245

Rozdział 9. Zarządzanie pamięcią i odśmiecanie 247
Sposób działania mechanizmu odzyskiwania pamięci 247
Pokoleniowy algorytm odzyskiwania pamięci 249
Wywoływanie mechanizmu odzyskiwania pamięci 252
Konstruktory 252
Finalizacja 253
Metoda bezpośredniego zwalniania zasobów — interfejs IDisposable 255
Przykład implementacji interfejsu IDisposable 255
Automatyczne implementowanie interfejsu IDisposable 257
Problemy z wydajnością w aspekcie finalizacji 258

Rozdział 10. Kolekcje 261
Interfejsy przestrzeni nazw System.Collections 261
Interfejs IEnumerable 262
Interfejs ICollection 263
Interfejs IList 263
Interfejs IDictionary 263
Interfejs IEnumeration 264
Klasy przestrzeni nazw System.Collections 264
Kolekcja typu Stack 265
Klasa Queue 268
Klasa ArrayList 271
Klasa HashTable 275
Tworzenie kolekcji ze ścisłą kontrolą typów 278
Dziedziczenie po klasie bazowej CollectionBase 278
Stosowanie kolekcji ze ścisłą kontrolą typów 282
Tworzenie słowników ze ścisłą kontrolą typów 283
Dziedziczenie po klasie bazowej DictionaryBase 283
Stosowanie kolekcji ze ścisłą kontrolą typów 286

Rozdział 11. Praca z klasami String i StringBuilder 287
Typ System.String 287
Niezmienność łańcuchów w środowisku .NET 288
Operacje na łańcuchach 290
Porównywanie łańcuchów 291
Klasa StringBuilder 295
Metody klasy StringBuilder 296
Stosowanie obiektów klasy StringBuilder 296
Formatowanie łańcuchów 297
Specyfikatory formatu 298
Specyfikatory formatów liczbowych 299
Specyfikatory formatów daty i czasu 301
Specyfikatory formatów typów wyliczeniowych 304

Rozdział 12. Operacje na plikach i strumieniach 307
Klasy przestrzeni nazw System.IO 307
Praca z systemem katalogów 309
Tworzenie i usuwanie katalogów 309
Przenoszenie i kopiowanie katalogów 310
Analizowanie informacji o katalogach 313
Praca z plikami 314
Tworzenie i usuwanie plików 314
Przenoszenie i kopiowanie plików 315
Analizowanie informacji o plikach 315

Strumienie ... 315
Praca ze strumieniami plików tekstowych ... 316
Praca ze strumieniami plików binarnych ... 319
Asynchroniczny dostęp do strumieni ... 321
Monitorowanie aktywności katalogów ... 324
Serializacja ... 326
Sposób działania serializacji ... 327
Formatery ... 328
Przykład serializacji i deserializacji ... 328

Rozdział 13. Tworzenie własnych kontrolek WinForms ... 333

Podstawy budowy komponentów ... 334
Kiedy należy tworzyć własne komponenty? ... 334
Etapy pisania komponentu ... 335
Wybór klasy bazowej ... 335
Tworzenie modułów komponentów ... 336
Tworzenie właściwości ... 339
Tworzenie zdarzeń ... 350
Tworzenie metod ... 356
Konstruktory i destruktory ... 356
Zachowanie w fazie projektowania ... 358
Testowanie komponentu ... 359
Dołączanie ikony komponentu ... 359
Przykładowe komponenty ... 360
ExplorerViewer — przykład komponentu dziedziczącego po klasie UserControl ... 360
SimpleStatusBars — przykład użycia dostawców rozszerzeń ... 368
Tworzenie komponentów użytkownika — kontrolka PlayingCard ... 373

Rozdział 14. Programowanie wielowątkowe w Delphi .NET ... 381

Procesy ... 381
Wątki ... 382
Wątki w stylu .NET ... 383
Domeny aplikacji ... 384
Przestrzeń nazw System.Threading ... 385
Klasa System.Threading.Thread ... 385
Typ wyliczeniowy System.Threading.ThreadPriority ... 389
Typ wyliczeniowy System.Threading.ThreadState ... 390
Typ wyliczeniowy System.Threading.ApartmentState ... 391
Klasa System.Threading.ThreadPool ... 391
Klasa System.Threading.Timer ... 393
Delegacje ... 394
Tworzenie bezpiecznego kodu wielowątkowego w stylu .NET ... 396
Mechanizmy blokujące ... 396
Zdarzenia ... 401
Lokalna pamięć wątków ... 402
Komunikacja międzyprocesowa Win32 ... 403
Bezpieczne wielowątkowe klasy i metody środowiska .NET ... 403
Kwestie dotyczące interfejsu użytkownika ... 404
Metoda System.Windows.Forms.Control.Invoke() ... 405
Właściwość System.Windows.Forms.Control.InvokeRequired ... 405
Metoda System.Windows.Forms.Control.BeginInvoke() ... 406
Metoda System.Windows.Forms.Control.EndInvoke() ... 406
Metoda System.Windows.Forms.Control.CreateCraphics() ... 407

Wyjątki w programach wielowątkowych ... 409
System.Threading.ThreadAbortException ... 409
System.Threading.ThreadInterruptedException ... 412
System.Threading.ThreadStateException ... 412
System.Threading.SynchronizationLockException ... 412
Odzyskiwanie pamięci a wielowątkowość ... 412

Rozdział 15. Mechanizm odzwierciedlania ... 413
Odzwierciedlanie podzespołów ... 413
Odzwierciedlanie modułów ... 416
Odzwierciedlanie typów ... 417
Dostęp do składowych typu podczas wykonywania (późne wiązanie) ... 419
Wydajny dostęp do składowych przez wywoływanie typów składowych ... 423
Kolejny przykład wywoływania składowych ... 423
Emitowanie kodu MSIL przy użyciu odzwierciedlania ... 427
Narzędzia — klasy do emitowania MSIL ... 427
Proces emitowania ... 428
Przykład użycia przestrzeni nazw System.Reflection.Emit ... 428

Rozdział 16. Współpraca z istniejącym kodem. Usługi COM Interop i Platform Invocation Service ... 433
Do czego służą mechanizmy współpracy z istniejącym kodem? ... 433
Powszechne problemy przy współpracy ... 434
Użycie obiektów COM w kodzie .NET ... 435
Automatyzacja z późnym dowiązywaniem ... 435
Parametry typów prostych, referencyjne i opcjonalne ... 438
Wcześnie dowiązywane obiekty COM ... 439
Podzespoły pośredniczące ... 442
Tworzenie podzespołu pośredniczącego ... 443
Zawartość pośredniczącej biblioteki typów ... 444
Użycie zdarzeń COM ... 445
Sterowanie długością życia obiektów COM ... 447
Obsługa błędów ... 447
Podstawowe podzespoły pośredniczące ... 448
Dostosowywanie zwykłych i podstawowych podzespołów pośredniczących ... 449
Użycie obiektów .NET w kodzie COM ... 451
Rejestrowanie podzespołu .NET dla automatyzacji ... 451
Automatyzacja z późnym dowiązywaniem ... 452
Pośredniczące biblioteki typów ... 453
Co zawiera pośrednicząca biblioteka typów? ... 454
Implementowanie interfejsów ... 455
Typy i szeregowanie parametrów ... 457
Obsługa błędów ... 459
Użycie procedur bibliotek Win32 w kodzie .NET ... 460
Tradycyjna składnia Delphi ... 461
Składnia wykorzystująca atrybuty ... 462
Typy i szeregowanie parametrów ... 464
Obsługa błędów ... 466
Kody błędów Win32 ... 467
Kody błędów HResult ... 469
Kwestie związane z wydajnością ... 471
Użycie procedur .NET w kodzie Win32 ... 475
Tradycyjna składnia Delphi ... 476
Typy i szeregowanie parametrów ... 477

Część IV Programowanie aplikacji bazodanowych z wykorzystaniem technologii ADO.NET 481

Rozdział 17. Przegląd ADO.NET 483

Założenia projektowe 483
Architektura danych odłączonych od źródła danych 483
Integracja z XML-em 484
Jednolita reprezentacja danych 484
Oparcie na platformie .NET Framework 484
Wykorzystanie wcześniejszych technologii 484
Obiekty ADO.NET 485
Klasy dostępu bezpośredniego 486
Klasy dostępu rozłączalnego 487
Dostawcy danych w .NET 487

Rozdział 18. Użycie obiektu Connection 489

Funkcje obiektu Connection 489
Konfiguracja właściwości ConnectionString 490
Ustawienia SqIConnection.ConnectionString 490
Ustawienia OleDbConnection.ConnectionString 491
Ustawienia OdbcConnection.ConnectionString 491
Ustawienia OracleConnection.ConnectionString 491
Otwieranie i zamykanie połączeń 492
Zdarzenia obiektu Connection 492
Buforowanie połączeń 495

Rozdział 19. Użycie obiektów Command i DataReader 497

Wykonywanie poleceń 497
Interfejs IDbCommand 497
Polecenia niezwracające wyników 498
Pobieranie pojedynczych wartości 500
Wykonywanie poleceń języka DDL 501
Podawanie parametrów przy użyciu klasy IDbParameter 503
Wykonywanie procedur składowanych 504
Odczytywanie parametrów 506
Pobieranie zbiorów wynikowych przy użyciu obiektu DataReader 508
Interfejs IDataReader 508
Pobranie zbioru wynikowego 508
Pobranie wielu zbiorów wynikowych przy użyciu obiektu DataReader 509
Użycie obiektu DataReader do pobierania danych typu BLOB 510
Użycie obiektu DataReader do pobierania informacji na temat schematu 512

Rozdział 20. Klasy DataAdapter i DataSet 515

Klasa DataAdapter 515
Struktura klasy DataAdapter 515
Tworzenie obiektu DataAdapter 517
Pobieranie wyników zapytania 518
Odwzorowywanie wyników zapytania 520
Praca z obiektami DataSet 523
Struktura klasy DataSet 523
Operacje klasy DataSet 525
Praca z obiektami DataTable 526
Definiowanie kolumn 526
Definiowanie kluczy głównych 528

Praca z ograniczeniami ... 528
Praca z obiektami DataRelation ... 531
Manipulowanie danymi — praca z obiektem DataRow ... 534
Wyszukiwanie, sortowanie i filtrowanie danych ... 536

Rozdział 21. Korzystanie z formularzy WinForms — obiekty DataView i dowiązywanie danych ... 539
Wyświetlanie danych za pomocą DataView i DataViewManager ... 539
Klasa DataView ... 540
Klasa DataViewManager ... 541
Przykładowe projekty wykorzystujące klasy DataView i DataViewManager ... 541
Dowiązywanie danych ... 552
Interfejsy dowiązywania danych ... 552
Dowiązanie proste i złożone ... 553
Klasy dowiązań danych z formularza WinForm ... 553
Tworzenie formularzy Windows z dowiązaniami danych ... 554

Rozdział 22. Zapis danych do źródła danych ... 565
Aktualizacja źródła danych za pomocą klasy SQLCommandBuilder ... 565
Aktualizacja źródła danych za pomocą własnej logiki aktualizacji ... 568
Korzystanie z klasy Command ... 568
Korzystanie z klasy SqlDataAdapter ... 575
Aktualizacja za pomocą zapamiętanych procedur ... 580
Obsługa współbieżności ... 586
Odświeżanie danych po ich aktualizacji ... 590

Rozdział 23. Transakcje i obiekty DataSet ze ścisłą kontrolą typów ... 593
Przetwarzanie transakcyjne ... 593
Przykład prostego przetwarzania transakcyjnego ... 594
Transakcje wykorzystujące obiekt DataAdapter ... 597
Poziomy izolacji ... 597
Znaczniki zapisu ... 599
Zagnieżdżone transakcje ... 599
Obiekty DataSet ze ścisłą kontrolą typów ... 600
Wady i zalety ... 600
Tworzenie obiektów DataSet ze ścisłą kontrolą typów ... 601
Analiza pliku .pas dla obiektu DataSet ze ścisłą kontrolą typów ... 602
Korzystanie z obiektów DataSet ze ścisłą kontrolą typów ... 609

Rozdział 24. Klasy dostawcy danych (Borland Data Provider) ... 611
Przegląd architektury ... 611
Klasy Borland Data Provider ... 612
Klasa BdpConnection ... 613
Klasa BdpCommand ... 614
Klasa BdpDataReader ... 615
Klasa BdpDataAdapter ... 616
Klasy BdpParameter i BdpParameterCollection ... 617
Klasa BdpTransaction ... 618
Elementy projektowe w środowisku programistycznym ... 619
Edytor połączeń ... 619
Edytor tekstów poleceń ... 620
Edytor kolekcji parametrów ... 620
Okno dialogowe konfiguracji łącznika danych ... 620

Część V Tworzenie aplikacji internetowych w ASP.NET 623

Rozdział 25. Podstawy ASP.NET 625

Technologie internetowe — jak one działają? 625
Omówienie protokołu HTTP 625
Pakiet żądania protokołu HTTP 626
Pakiet odpowiedzi protokołu HTTP 627
ASP.NET — jak działa? 628
Prosta aplikacja internetowa 629
Struktura strony ASP.NET 630
Komunikacja sterowana zdarzeniami 632
VIEWSTATE i utrzymywanie stanu 633
Kod poza sceną (CodeBehind) 634
Klasy ASP.NET 635
Klasa HTTPResponse 635
Klasa HTTPRequest 638
Klasa HTTPCookie 640
Obsługa wielokrotnego wysyłania danych 641

Rozdział 26. Tworzenie stron WWW w ASP.NET 643

Tworzenie stron WWW za pomocą kontrolek ASP.NET 643
Przykładowy formularz prośby o pobranie produktu 644
Układ graficzny strony 645
Tworzenie formularza 645
Przetworzenie zdarzenia załadowania 646
Zapis plików w aplikacjach ASP.NET 647
Kolejność przetwarzania zdarzeń w formularzach WWW 649
Wcześniejsze wypełnianie kontrolek list 649
Przeprowadzanie walidacji formularzy 650
Walidacja po stronie klienta a walidacja po stronie serwera 651
Klasa BaseValidator 651
Kontrolka RequiredFieldValidator 652
Kontrolka CompareValidator 653
Kontrolka RegularExpressionValidator 654
Kontrolka RangeValidator 656
Kontrolka CustomValidator 656
Kontrolka ValidationSummary 657
Formatowanie stron 658
Właściwości WebControl z typami ścisłymi 658
Kaskadowe arkusze stylów 659
Wykorzystanie klasy stylu 660
Przemieszczanie się między formularzami Web Forms 661
Przekazywanie danych dzięki mechanizmowi POST 662
Zastosowanie metody Response.Redirect() i tekstu zapytania 662
Wykorzystanie metody Server.Transfer() 663
Wykorzystanie zmiennych sesji 664
Wskazówki i ciekawe sztuczki 665
Użycie kontrolki Panel do symulacji wielu formularzy 665
Wysyłanie pliku przez klienta 667
Wysłanie listu e-mail z poziomu formularza 669
Wyświetlanie obrazów 670
Dynamiczne dodawanie kontrolek 671

Rozdział 27. Tworzenie aplikacji bazodanowych w ASP.NET 675
Dowiązywanie danych 675
Proste dowiązywanie danych 675
Złożone dowiązywanie danych 680
Dowiązywanie danych do kontrolek list 680
Kontrolka CheckBoxList 680
Kontrolka DropDownList 682
Kontrolka ListBox 685
Kontrolka RadioButtonList 687
Dowiązywanie danych do kontrolek iteracyjnych 689
Kontrolka Repeater 689
Kontrolka DataList 693
Korzystanie z elementu DataGrid 698
Stronicowanie kontrolki DataGrid 700
Edycja za pomocą kontrolki DataGrid 703
Dodanie elementów do kontrolki DataGrid 709
Sortowanie kontrolki DataGrid 709
Formularz prośby o pobranie pliku oparty na bazie danych i administracja systemem 710

Rozdział 28. Usługi sieciowe 715
Terminy związane z usługami sieciowymi 715
Konstrukcja usług sieciowych 716
Atrybut [WebService] 721
Zwracanie danych z usługi sieciowej 722
Atrybut [WebMethod] 723
Wykorzystywanie usług sieciowych 725
Odkrywanie usługi 725
Tworzenie klasy pośredniczącej 725
Korzystanie z klasy pośredniczącej 727
Pobranie obiektu DataSet z usługi sieciowej 730
Wywołanie asynchronicznej metody usługi sieciowej 733
Zabezpieczanie usług sieciowych 734

Rozdział 29. Technologia .NET Remoting i Delphi 739
Dostępne obecnie technologie zdalne 739
Gniazda 739
RPC 740
Java RMI 740
CORBA 740
XML-RPC 741
DCOM 741
Com-Interop 741
SOAP 741
.NET Remoting 742
Architektury rozproszone 743
Architektura klient-serwer 743
Architektura typu „równy z równym” 744
Architektury wielowarstwowe 744
Zalety tworzenia aplikacji wielowarstwowych 745
Skalowalność i odporność na błędy 745
Tworzenie i wdrażanie 747
Bezpieczeństwo 747
Podstawy technologii .NET Remoting 747
Architektura 748
Domeny aplikacji 748

Przestrzeń nazw System.Runtime.Remoting ... 749
Obiekty zdalności ... 750
Aktywacja obiektu ... 751
Dzierżawcy i sponsorzy ... 753
Pośrednicy ... 753
Kanały ... 754
Pierwsza aplikacja wykorzystująca .NET Remoting ... 754
Przygotowanie projektu ... 754
Dodawanie referencji ... 756
Plik BankPackage.dll — kontakt między klientami i serwerami ... 757
Implementacja serwera ... 759
Implementacja klienta ... 763

Rozdział 30. Technologia .NET Remoting w akcji ... 767
Projekt szablonu ... 767
Śledzenie komunikatów ... 768
Analiza pakietów SOAP ... 770
Aktywacja kliencka ... 772
Wzorzec fabryki ... 773
Testowanie przykładu ... 779
Problemy związanie z CAO ... 780
Zarządzanie czasem życia obiektów ... 781
Nieudane odnowienie wynajmu ... 784
Pliki konfiguracyjne ... 785
Konfiguracja serwera ... 786
Konfiguracja klienta ... 788
Przejście z komunikacji HTTP na TCP ... 794
Przejście z formatu SOAP na format binarny ... 794
Różnice w kodowaniu binarnym i SOAP ... 796

Rozdział 31. Bezpieczeństwo w aplikacjach .NET ... 799
Rodzaje bezpieczeństwa w ASP.NET ... 799
Uwierzytelnianie ... 799
Konfiguracja modelu uwierzytelniania w ASP.NET ... 800
Uwierzytelnianie Windows ... 800
Uwierzytelnianie bazujące na formularzach ... 802
Uwierzytelnianie Passport ... 809
Autoryzacja ... 810
Autoryzacja plikowa ... 810
Autoryzacja URL — sekcja <authorization> ... 811
Autoryzacja bazująca na rolach ... 812
Podszywanie się ... 814
Wylogowywanie się ... 815

Rozdział 32. Wdrażanie i konfiguracja ASP.NET ... 817
Wdrażanie aplikacji ASP.NET ... 817
Kwestie związane z prostym wdrażaniem ... 817
Wdrażanie z wykorzystaniem polecenia XCOPY ... 821
Ustawienia konfiguracji ... 821
Plik machine.config ... 822
Plik web.config ... 822
Wskazówki konfiguracyjne ... 827
Obsługa przekierowania błędów ... 828
Ponowne uruchomienie procesu wykonawczego ... 829
Zwiększenie wydajności przez buforowanie wyjścia ... 831

Monitorowanie procesu ASP.NET ... 831
Śledzenie aplikacji ... 833
Dodawanie i pobieranie własnych ustawień konfiguracji ... 837
Dodanie lub odczytanie sekcji <appSettings> ... 837
Dodawanie i odczyt własnych sekcji konfiguracyjnych ... 838

Rozdział 33. Buforowanie stron i zarządzanie stanem w aplikacjach ASP.NET ... 839
Buforowanie stron aplikacji ASP.NET ... 839
Buforowanie stron ... 839
Buforowanie fragmentów stron ... 844
Buforowanie danych ... 844
Zależności buforowania ... 848
Rozszerzenie zależności plików w celu ich użycia z serwerem SQL Server ... 849
Metody wywołań zwrotnych bufora ... 850
Zarządzanie stanem w aplikacjach ASP.NET ... 853
Zarządzanie stanem za pomocą cookies ... 853
Korzystanie z komponentu ViewState ... 855
Zarządzanie stanem sesji ... 858
Przechowywanie danych sesji na serwerze stanów sesji ... 860
Przechowywanie danych sesji w serwerze SQL Server ... 860
Zdarzenia sesji ... 861
Zarządzanie stanem aplikacji ... 863
Buforowanie a stan aplikacji ... 864

Rozdział 34. Tworzenie własnych kontrolek serwerowych ASP.NET ... 867
Kontrolki użytkownika ... 868
Bardzo prosta kontrolka użytkownika ... 868
Omówienie prostej kontrolki ... 871
Kontrolka użytkownika dotycząca logowania ... 873
Kontrolki WWW ... 875
Tworzenie bardzo prostej kontrolki WWW ... 875
Wartości trwałe ... 878
Dodanie własnego renderingu ... 879
Określenie rodzaju bloku HTML ... 882
Obsługa danych żądań zwrotnych ... 883
Kontrolka TPostBackInputWebControl ... 884
Kontrolki złożone ... 888
Implementacja kontrolki złożonej — TNewUserInfoControl ... 888

Dodatki ... 895

Skorowidz ... 897

Rozdział 5.
Język Delphi

Autor: Steve Taixeira

W niniejszym rozdziale zajmiemy się językiem wykorzystywanym w zintegrowanym środowisku Delphi — językiem programowania Object Pascal (od pewnego czasu nazywanym po prostu językiem Delphi). Najpierw zaprezentujemy podstawy tego języka, takie jak jego zasadnicze reguły i konstrukcje, a następnie niektóre z bardziej zaawansowanych aspektów programowania w Delphi, jak klasy czy obsługa wyjątków. Zakładamy przy tym, że masz już pewne doświadczenie z innymi wysokopoziomowymi językami programowania. Nie będziemy w związku z tym tracić czasu na wprowadzanie najbardziej podstawowych pojęć dotyczących tego typu języków, a zamiast tego skupimy się na wyjaśnianiu elementów typowych dla Delphi. Po uważnym przeczytaniu rozdziału powinieneś rozumieć, jak w tym języku można stosować takie elementy jak zmienne, typy, operatory, pętle, instrukcje warunkowe, wyjątki i obiekty oraz które z tych elementów mają związek z docelową platformą .NET Framework. Aby zapewnić Ci jak najlepsze podstawy do pracy nad aplikacjami dla tej platformy, spróbujemy także porównać możliwości języka Delphi z jego bardziej popularnymi kuzynami z rodziny .NET: językami C# i Visual Basic .NET.

Wszystko o technologii .NET

Środowisko Delphi 8 generuje aplikacje, które są w pełni zgodne z platformą Microsoft .NET Framework. Oznacza to, że funkcjonalność i cechy kompilatora środowiska Delphi 8 muszą być zgodne z funkcjonalnością i właściwościami platformy .NET Framework. Związane z tym wymagania mogą nie być do końca jasne dla tych programistów, którzy przez lata funkcjonowali w świecie rdzennego kodu (np. dla platformy Win32). Kompilator takiego kodu może bowiem robić z kodem źródłowym „co zechce" — możliwości takiego kompilatora są ograniczane tylko przez założenia przyjęte przez jego producenta. W świecie aplikacji .NET dosłownie każdy fragment programu — nawet coś tak trywialnego jak sumowanie dwóch liczb całkowitych — musi przejść przez kompilator generujący kod zgodny z właściwościami i typami platformy .NET Framework.

Zamiast generować rdzenny kod dla jednej platformy, kompilator .NET środowiska Delphi 8 musi wytwarzać kod w formacie nazywanym językiem pośrednim firmy Microsoft (ang. *Microsoft Intermediate Language — MSIL*), który jest reprezentacją kodu źródłowego aplikacji najniższego poziomu.



W rozdziale 2. omówiliśmy stosowaną na platformie .NET Framework kompilację „w locie" (ang. *Just in Time — JIT*). Teraz jest dobry moment, aby na chwilę wrócić do informacji przekazanych w tamtym rozdziale.

Komentarze

Język Delphi obsługuje trzy typy komentarzy: komentarze w nawiasach klamrowych, komentarze w nawiasach okrągłych z gwiazdkami oraz komentarze poprzedzone dwoma znakami ukośnika. Poniżej przedstawiono przykłady tych trzech typów komentarzy:

```
{ Komentarz otoczony nawiasami klamrowymi }
(* Komentarz otoczony nawiasami okrągłymi i gwiazdkami *)
// Komentarz poprzedzony dwoma lewymi ukośnikami
```

Pierwsze dwa typy komentarzy są do siebie bardzo podobne. Kompilator uznaje za komentarz cały tekst znajdujący się pomiędzy symbolem otwierającym a odpowiadającym mu symbolem zamykającym. W przypadku komentarzy z dwoma ukośnikami sytuacja wygląda nieco inaczej — komentarzem jest cały tekst znajdujący pomiędzy tymi znakami a najbliższym znakiem podziału wiersza.



W języku Delphi nie można zagnieżdżać komentarzy tego samego typu. Chociaż z punktu widzenia składni języka zagnieżdżanie komentarzy różnych typów jest dozwolone, w praktyce stosowanie takich konstrukcji nie jest zalecane. Oto kilka przykładów:

```
{ (* To jest poprawne *) }
(* { To jest poprawne } *)
(* (* To jest niepoprawne *) *)
{ { To jest niepoprawne } }
```

Inną wygodną techniką wyłączania wybranych części kodu źródłowego, w szczególności w przypadku stosowania w tym kodzie różnych typów komentarzy, jest stosowanie dyrektywy kompilatora `$IFDEF`. Przykładowo w poniższym fragmencie kodu wykorzystano dyrektywę `$IFDEF` do „wzięcia w komentarz" bloku kodu, który ma zostać pominięty w procesie kompilacji:

```
{$IFDEF NIEKOMPILUJMNIE}
// wyobraź sobie, że w tym miejscu znajduje się jakiś blok kodu
{$ENDIF}
```

Ponieważ identyfikator `NIEKOMPILUJMNIE` nie został zdefiniowany, kod pomiędzy dyrektywami `$IFDEF` i `$ENDIF` nie będzie przetwarzany.

Procedury i funkcje

Ponieważ procedury i funkcje występują niemal we wszystkich językach programowania i powinny być znane każdemu programiście, nie będziemy związanych z nimi zagadnień szczegółowo w tej książce omawiali. Chcielibyśmy jedynie zwrócić uwagę na kilka unikalnych lub mało znanych kwestii pojawiających się w tym obszarze.



Funkcje, które nie zwracają żadnych wartości (w języku C# są deklarowane ze zwracanym typem `void`) są nazywane procedurami (ang. *procedures*), natomiast funkcje zwracające jakieś wartości są nazywane właśnie funkcjami (ang. *functions*). W języku angielskim często stosuje się pojęcie *routine*, które odnosi się zarówno do procedur, jak i funkcji, natomiast terminu metoda (ang. *method*) używamy do określania funkcji i procedur należących do klas (termin ten jest więc charakterystyczny dla programowania obiektowego).

Nawiasy w wywołaniach

Jedną z najmniej znanych cech języka programowania Delphi jest możliwość opcjonalnego stosowania nawiasów okrągłych w wywołaniach bezparametrowych procedur i funkcji. Oznacza to, że oba poniższe przykłady są poprawne składniowo:

```
Form1.Show;
Form1.Show();
```

Możliwość opcjonalnego umieszczania nawiasów bezpośrednio za nazwami bezparametrowych funkcji i procedur dla większości programistów nie ma oczywiście żadnego znaczenia, jednak może być ważna dla tych osób, które dzielą swój czas pracy pomiędzy język Delphi i taki język programowania jak np. C#, gdzie stosowanie owych nawiasów jest konieczne. Jeśli równolegle pracujesz w wielu językach programowania, nie musisz — dzięki tej właściwości Delphi — pamiętać o stosowaniu różnych reguł składniowych dla wywołań funkcji i procedur w tych językach.

Przeciążanie

Język programowania Delphi umożliwia stosowanie techniki nazywanej przeciążaniem funkcji, czyli techniki polegającej na deklarowaniu wielu procedur lub funkcji z tą samą nazwą, ale innymi listami parametrów. Wszystkie przeciążone metody muszą być deklarowane z dyrektywą `overload`, czyli tak jak w poniższym przykładzie:

```
procedure Hello(I: Integer); overload;
procedure Hello(S: string); overload;
procedure Hello(D: Double); overload;
```

Zwróć uwagę na fakt, że reguły przeciążania metod należących do klas są nieco inne niż odpowiednie reguły dla procedur i funkcji deklarowanych poza klasami — wyjaśnimy to w podrozdziale „Przeciążanie metod”.

Domyślne wartości parametrów

Język Delphi obsługuje także wygodną w wielu przypadkach możliwość deklarowania domyślnych wartości parametrów — czyli możliwość określania wartości domyślnej dla parametru funkcji lub procedury i brak konieczności przekazywania tego parametru w późniejszych wywołaniach tej funkcji lub procedury. Aby zadeklarować funkcję lub procedurę z domyślnymi wartościami parametrów, za typem wybranego parametru umieść znak równości i jego domyślną wartość; ilustruje to poniższy przykład:

```
procedure HasDefVal(S: string; I: Integer = 0);
```

Procedura `HasDefVal()` może być wywoływana na dwa sposoby. Po pierwsze, w wywołaniu tej procedury możemy określić wartości obu parametrów:

```
HasDefVal('witaj', 26);
```

Po drugie, możemy określić tylko parametr `S` i — tym samym — użyć domyślnej wartości dla parametru `I`:

```
HasDefVal('witaj');  // dla parametru I użyto wartości domyślnej
```

Jeśli zdecydujesz się na stosowanie domyślnych wartości parametrów, musisz pamiętać o przestrzeganiu kilku ważnych zasad:

- Parametry ze zdefiniowanymi wartościami domyślnymi muszą występować na końcu listy parametrów. Na liście parametrów procedury lub funkcji parametry bez wartości domyślnych nie mogą być deklarowane za parametrami z takimi wartościami.
- Domyślne wartości parametrów mogą mieć postać liczb całkowitych, łańcuchów, liczb zmiennoprzecinkowych, wskaźników lub zbiorów. W języku Delphi są obsługiwane także takie typy jak klasy, interfejsy, tablice dynamiczne oraz referencje do klas, jednak tylko w przypadku, gdy domyślną wartością jest `nil`.
- Parametry z zadeklarowanymi wartościami domyślnymi muszą być przekazywane przez wartość lub jako stałe (ze słowem `const`). Nie mogą być referencjami (`var`, `out`) ani parametrami bez typów.

Jedną z największych korzyści wynikających z możliwości deklarowania domyślnych wartości parametrów jest zwiększanie funkcjonalności istniejących funkcji i procedur bez utraty ich zgodności z dotychczasowymi wywołaniami, a więc bez konieczności modyfikowania istniejących wywołań. Przypuśćmy na przykład, że udostępniamy moduł z „rewolucyjną” funkcją nazwaną `AddInts()`, która dodaje dwie liczby całkowite:

```
function AddInts(I1, I2: Integer): Integer;
begin
  Result := I1 + I2;
end;
```

Po jakimś czasie stwierdzamy, że musimy zaktualizować tę funkcję w taki sposób, by umożliwiała sumowanie trzech liczb całkowitych. Nie jesteśmy jednak przekonani co do słuszności takiego posunięcia, ponieważ dodanie jeszcze jednego parametru uniemożliwi kompilowanie istniejącego kodu, w którym ta funkcja jest wywoływana. Na szczęście okazuje się, że dzięki domyślnym parametrom możemy rozszerzyć funkcjonalność funkcji `AddInts()`, nie powodując żadnych niezgodności w istniejącym kodzie. Oto przykład takiego rozwiązania:

```
function AddInts(I1, I2: Integer; I3: Integer = 0);
begin
  Result := I1 + I2 + I3;
end;
```



W ogólności, jeśli chcesz zwiększyć funkcjonalność funkcji lub procedur i jednocześnie zachować zgodność z dotychczasowymi wywołaniami, powinieneś raczej stosować funkcje i procedury przeciążone zamiast domyślnych wartości parametrów. Wykonywanie procedur i funkcji przeciążonych jest nie tylko bardziej efektywne, ale także zapewnia większą zgodność z pozostałymi językami programowania platformy .NET, ponieważ domyślne wartości parametrów nie są obsługiwane w takich językach jak C# czy zarządzany C++.

Zmienne

Być może jesteś przyzwyczajony do deklarowania zmiennych „na zawołanie", czyli zgodnie z zasadą, że jeśli w danym miejscu potrzebujesz kolejnej liczby całkowitej, deklarujesz ją w środku bloku kodu, bezpośrednio przed wyrażeniem, w którym jest Ci potrzebna. Takie przyzwyczajenia występują bardzo często wśród programistów, którzy przez lata wykorzystywali inne języki programowania, takie jak C# czy Visual Basic .NET. Jeśli taki sposób deklarowania zmiennych nie jest Ci obcy, będziesz musiał się przyzwyczaić do zupełnie innego modelu wykorzystywania zmiennych w języku Delphi. W tym języku programowania wszystkie zmienne muszą być deklarowane w wyznaczonym do tego celu bloku poprzedzającym właściwy kod procedury, funkcji lub programu. Być może do tej pory podchodziłeś do problemu lokalizowania deklaracji zmiennych bardzo swobodnie i tworzyłeś funkcje podobne do poniższej:

```
public void foo()
{
  int x = 1;
  x++;
  int y = 2;
  float f;
  // ... etc ...
}
```

W języku Delphi taki kod musi być uporządkowany i dostosowany do odpowiedniej struktury — w tym przypadku nasza procedura powinna wyglądać następująco:

```
procedure Foo;
var
  x, y: Integer;
  f: Double;
begin
  x := 1;
  inc(x);
  y := 2;
  // ... etc ...
end;
```

Różnicowanie małych i wielkich liter oraz używanie wielkich liter w kodzie źródłowym

W języku programowania Delphi — podobnie jak w języku Visual Basic .NET, ale inaczej niż w języku C# — małe i wielkie litery są traktowane tak samo. W Delphi różna wielkość liter ma na celu jedynie ułatwienie czytania kodu, zatem pełni podobną rolę jak style wykorzystywane w książkach. Jeśli identyfikator funkcji, procedury, zmiennej lub innego elementu składa się z wielu połączonych słów, powinniśmy pamiętać o stosowaniu wielkiej litery na początku każdego słowa składającego się na taki identyfikator. Przykładowo poniższa nazwa procedury jest niejasna i trudna do odczytania:

```
procedure takanazwaproceduryniemasensu;
```

Taki kod jest być może interesujący, jednak z punktu widzenia osoby czytającej, znacznie lepsza jest następująca postać:

```
procedure TakaNazwaProceduryJestZnacznieBardziejCzytelna;
```

Być może zastanawiasz się, jaki jest cel stosowania tak ścisłej struktury i jakie korzyści z tego płyną. Po jakimś czasie programowania w tym języku z pewnością zgodzisz się z tezą, że styl języka programowania Delphi z jednoznaczną strukturą deklarowania zmiennych ułatwia czytanie i konserwację kodu, a także pozwala uniknąć wielu błędów, które często występując w językach pozbawionych tak surowych wymagań.

Zwróć uwagę na możliwy w języku Delphi sposób grupowania w jednym wierszu więcej niż jednej zmiennej (w tym przypadku większej liczby parametrów) tego samego typu, zgodnie z następującą regułą składniową:

```
VarName1, VarName2: SomeType;
```

Dzięki temu nasz kod opracowany w języku programowania Delphi może być znacznie bardziej skondensowany i czytelny niż podobne deklaracje w innych językach, np. C#, w których każda zmienna lub parametr musi mieć osobno określony typ.

Pamiętaj, że kiedy deklarujesz zmienną w języku Delphi, nazwa tej zmiennej musi występować przed jej typem, a pomiędzy zmiennymi i typami koniecznie musi się znajdować znak dwukropka. W przypadku zmiennych lokalnych inicjalizacja zmiennej jest zawsze oddzielona od jej deklaracji.

Język programowania Delphi zezwala na inicjalizowanie zmiennych globalnych już w bloku ich deklaracji (`var`). Oto kilka przykładów demonstrujących składnię takich operacji:

```
var
  i: Integer = 10;
  S: string  = 'Witaj świecie';
  D: Double  = 3.141579;
```



Preinicjalizacja zmiennych jest dozwolona wyłącznie w przypadku zmiennych globalnych — nie jest możliwa w przypadku lokalnych zmiennych wykorzystywanych w procedurach lub funkcjach.

Inicjalizacja wartością zero

Specyfikacja środowiska CLR zakłada, że wszystkie zmienne są automatycznie inicjalizowane wartością 0. Kiedy uruchamiana jest nasza aplikacja lub wywoływana jest jedna z naszych funkcji, wszystkie zmienne całkowitoliczbowe będą reprezentowały wartość `0`, wszystkie zmienne zmiennoprzecinkowe będą reprezentowały wartość `0.0`, wszystkie obiekty będą miały wartość `nil`, wszystkie łańcuchy będą puste itd. Oznacza to, że inicjalizowanie w kodzie źródłowym zmiennych wartością zero mija się z celem.

Inaczej niż w wersjach Delphi dla platformy Win32 automatyczne inicjalizowanie zmiennych dotyczy zarówno zmiennych lokalnych, jak i zmiennych globalnych.

Stałe

W języku Delphi stałe są definiowane za pomocą klauzuli `const`, której znaczenie jest podobne do znaczenia słowa kluczowego `const` znanego z języka programowania C#. Oto przykład trzech deklaracji stałych w języku C#:

```
public const float ADecimalNumber = 3.14;

public const int i = 10;

public const String ErrorString = "Niebiezpieczeństwo, Niebiezpieczeństwo,
Niebiezpieczeństwo!";
```

Główna różnica pomiędzy stałymi definiowanymi w języku C# a stałymi języka Delphi polega na tym, że w tym drugim języku (podobnie jak w języku Visual Basic .NET) nie jest wymagane podanie typu stałej w jej deklaracji. Kompilator Delphi automatycznie przydziela właściwy typ dla stałej w oparciu o jej wartość lub — w przypadku stałych skalarnych (np. liczb całkowitych typu `Integer`) — kompilator zachowuje jedynie wartości stałych, bez przydzielania im odpowiedniej przestrzeni w pamięci. Oto przykład deklaracji stałych w języku Delphi:

```
const
  ADecimalNumber = 3.14;
  i = 10;
  ErrorString = 'Niebiezpieczeństwo, Niebiezpieczeństwo, Niebiezpieczeństwo!';
```

Możemy także wprost określać typy stałych w bloku ich deklaracji. W ten sposób możemy w pełni kontrolować sposób traktowania tych stałych przez kompilator Delphi:

```
const
  ADecimalNumber: Double = 3.14;
  i: Integer = 10;
  ErrorString: string = 'Niebiezpieczeństwo, Niebiezpieczeństwo, Niebiezpieczeństwo!';
```

Bezpieczeństwo typów stałych z określonymi typami

Stałe z określonymi typami mają jedną zasadniczą przewagę nad stałymi z typami automatycznie przypisywanymi przez kompilator — w przypadku stałych bez jawnie zadeklarowanych typów (z typami przypisywanymi dopiero w fazie kompilacji) nie jest możliwe wywoływanie metod odpowiednich obiektów. Przykładowo, poniższy fragment kodu jest prawidłowy:

```
const
  I: Integer = 19710704;
  S: string;
begin
  S := I.ToString;
```

Natomiast fragment przedstawiony poniżej jest błędny (nie zostanie skompilowany):

```
const
  I = 19710704;
  S: string;
begin
  S := I.ToString;
```

Reguły języka Delphi zezwalają na stosowanie w deklaracjach `const` i `var` (odpowiednio stałych i zmiennych) tzw. funkcji czasu kompilacji. Do tego zbioru należą takie funkcje jak `Ord()`, `Chr()`, `Trunc()`, `Round()`, `High()`, `Low()`, `Abs()`, `Pred()`, `Succ()`, `Length()`, `Odd()` oraz `SizeOf()`. Przykładowo, cały poniższy kod jest prawidłowy:

```
type
  A = array[1..2] of Integer;
```

```
const
  w: Word = SizeOf(Byte);

var
  i: Integer = 8;
  j: SmallInt = Ord('a');
  L: LongInt = Trunc(3.14159);
  x: ShortInt = Round(2.71828);
  B1: Byte = High(A);
  B2: Byte = Low(A);
  C: Char = Chr(46);
```



Aby zapewnić zgodność z wcześniejszymi wersjami, kompilator Delphi udostępnia przełącznik umożliwiający przypisywanie wartości stałym z jawnie zadeklarowanymi typami (a więc operowanie na tych stałych w taki sam sposób jak na zmiennych typów). Odpowiedni przełącznik jest dostępny na zakładce *Compiler* (kompilator) okna dialogowego opcji projektu lub poprzez dyrektywę kompilatora `$WRITEABLECONST` (lub `$J`). Pamiętaj jednak, że stosowanie tej opcji nie jest zalecane, co oznacza, że powinieneś jej unikać i — jeśli nie jest to konieczne — nie włączać tej opcji kompilatora.



Podobnie jak języki C# i Visual Basic .NET, także język programowania Delphi nie wykorzystuje tzw. preprocesora, który jest stosowany np. w języku C. W języku Delphi nie istnieje pojęcie makra, zatem nie ma możliwości stosowania konstrukcji równoważnych np. znanym z języka C deklaracjom stałych `#define`. Chociaż w języku Delphi możemy wykorzystywać dyrektywę kompilatora `#define` dla kompilacji warunkowych (a więc podobnie do odpowiedniego zastosowania dyrektywy `#define` w języku C), dyrektywa ta nie może służyć definiowaniu stałych. Wszędzie tam, gdzie w języku C zadeklarowałbyś stałą za pomocą dyrektywy `#define`, w języku Delphi powinieneś stosować klauzulę `const`.

Operatory

Operatory są symbolami wykorzystywanymi w kodzie programu do manipulowania wszystkimi rodzajami danych. Przykładowo, istnieją operatory dodawania, odejmowania, mnożenia i dzielenia danych numerycznych. Istnieją także operatory odwoływania się do konkretnych elementów tablic. W tym podrozdziale wyjaśnimy niektóre spośród operatorów dostępnych w języku programowania Delphi i porównamy je z ich odpowiednikami w języku C# i środowisku CLR.

Operatory przypisania

Jeśli nigdy nie pracowałeś w języku programowania Pascal, wykorzystywany w tym języku (a więc także w środowisku i języku Delphi) operator przypisania może być tym elementem, do którego najtrudniej będzie Ci się przyzwyczaić. Aby przypisać danej zmiennej jakąś wartość, w Delphi używamy operatora `:=`; do tej samej operacji w językach C# i Visual Basic .NET użylibyśmy operatora `=`. W odniesieniu do tego operatora programiści Delphi używają niekiedy określenia *otrzymuje* lub *jest przypisywana*, zatem wyrażenie w postaci:

```
Number1 := 5;
```

czytają: „zmienna `Number1` *otrzymuje* wartość `5`" lub „wartość `5` *jest przypisywana* do zmiennej `Number1`".

Operatory porównania

Jeśli programowałeś już w języku Visual Basic .NET, nie powinieneś mieć żadnych problemów z operatorami porównania stosowanymi w języku Delphi, ponieważ operatory wykorzystywane w obu językach są niemal identyczne. Operatory tego typu są standardem prawie we wszystkich językach programowania, zatem nie będziemy ich w tym podrozdziale szczegółowo omawiali.

W języku Delphi do logicznego porównania dwóch wyrażeń lub wartości wykorzystuje się operator `=`. Stosowany w tym języku operator `=` odpowiada stosowanemu w języku C# operatorowi `==`, co oznacza, że instrukcja warunkowa C# w postaci:

```
if (x == y)
```

w języku Delphi wyglądałaby następująco:

```
if x = y
```



Pamiętaj, że w języku Delphi operator `:=` jest wykorzystywany do przypisywania wartości zmiennym, natomiast operator `=` służy jedynie porównywaniu wartości dwóch operandów (nigdy operacji przypisania).

Operatorem nierówności w języku Delphi jest `<>`. Znaczenie tego operatora jest identyczne jak znaczenie operatora `!=` w języku programowania C#. Aby określić, czy dwa wyrażenia (lub wartości) są od siebie różne, w języku Delphi wykorzystujemy następujący kod:

```
if x <> y then DoSomething;
```

Operatory logiczne

W języku programowania Delphi operatory logiczne „i" oraz „lub" wyraża się za pomocą słów odpowiednio `and` i `or` — tę samą funkcję w języku C# pełnią odpowiednio symbole `&&` oraz `||`. Operatory logiczne `and` i `or` najczęściej wykorzystuje się w instrukcjach warunkowych `if` oraz pętlach (patrz dwa poniższe przykłady):

```
if (Warunek 1) and (Warunek 2) then
  DoSomething;

while (Warunek 1) or (Warunek 2) do
  DoSomething;
```

Logicznym operatorem negacji w języku programowania Delphi jest słowo `not` — słowo to jest wykorzystywane do zaprzeczania wyrażeniom logicznym definiowanym w tym języku. W języku C# tę samą rolę pełni symbol `!`. Słowo `not` jest często wykorzystywane w instrukcjach warunkowych `if`. Oto przykład:

```
if not (warunek) then (zrób coś);   // jeśli warunek if jest fałszywy, wówczas...
```

W tabeli 5.1 zestawiono operator przypisania, operatory porównania i operatory logiczne stosowane w języku programowania Delphi wraz z ich odpowiednikami wykorzystywanymi w językach C# i Visual Basic .NET.

Tabela 5.1. *Operator przypisania, operatory porównania i operatory logiczne*

Operator	Delphi	C#	Visual Basic .NET
Przypisanie	:=	=	=
Porównanie	=	==	= lub Is[1]
Różne	<>	!=	<>
Mniejsze	<	<	<
Większe	>	>	>
Mniejsze lub równe	<=	<=	<=
Większe lub równe	>=	>=	>=
Logiczne „i”	and	&&	And
Logiczne „lub”	or	||	Or
Logiczne „nie”	not	!	Not
Logiczne „lub wyłączające”	xor	^	Xor

Operatory arytmetyczne

Już teraz powinieneś dobrze znać większość operatorów arytmetycznych stosowanych w języku Delphi, ponieważ w większości przypadków te same operatory wykorzystuje się w innych popularnych językach programowania. W tabeli 5.2 przedstawiono wszystkie operatory arytmetyczne Delphi wraz z ich odpowiednikami stosowanymi w językach C# i Visual Basic .NET.

Tabela 5.2. *Operator przypisania, operatory porównania i operatory logiczne*

Operator	Delphi	C#	Visual Basic .NET
Dodawanie	+	+	+
Odejmowanie	-	-	-
Mnożenie	*	*	*
Dzielenie zmiennoprzecinkowe	/	/	/
Dzielenie całkowite	div	/	\
Modulo	mod	%	Mod
Potęga	Brak	Brak	^

[1] W języku programowania Visual Basic .NET Is jest operatorem porównania wykorzystywanym dla obiektów, natomiast = jest operatorem porównania stosowanym dla wszystkich pozostałych typów.

Być może zwróciłeś uwagę na to, że w językach Delphi i Visual Basic .NET wykorzystuje się różne operatory dla operacji dzielenia zmiennoprzecinkowego i dzielenia całkowitoliczbowego, chociaż w języku C# dla obu tych operacji stosuje się ten sam operator. Operator `div` automatycznie obcina ewentualną resztę z dzielenia dwóch wyrażeń całkowitoliczbowych.



Pamiętaj o stosowaniu właściwych operatorów dzielenia dla wykorzystywanych wyrażeń różnych typów. Kompilator Delphi wygeneruje komunikat o błędzie, jeśli spróbujesz podzielić dwie liczby zmiennoprzecinkowe za pomocą operatora dzielenia całkowitego `div` lub jeśli podejmiesz próbę przypisania zmiennej całkowitoliczbowej wyniku dzielenia dwóch liczb całkowitych za pomocą operatora `/`; ilustruje to poniższy fragment kodu:

```
var
  i: Integer;
  d: Double;
begin
  i := 4 / 3;         // Ten wiersz kodu spowoduje błąd kompilatora.
  d := 3.4 div 2.3;   // Także ten wiersz spowoduje błąd.
end;
```

Jeśli musisz podzielić dwie liczby całkowite za pomocą operatora `/`, otrzymany wynik możesz przypisać zmiennej całkowitoliczbowej tylko w sytuacji, gdy uprzednio przekonwertujesz to wyrażenie na liczbę całkowitą za pomocą funkcji `Trunc()` (obcięcie części ułamkowej) lub `Round()` (zaokrąglenie).

Operatory bitowe

Operatory bitowe umożliwiają nam modyfikowanie pojedynczych bitów danej zmiennej całkowitoliczbowej. Do najczęściej stosowanych operatorów binarnych należą operatory przesunięcia bitów w lewo lub w prawo, a także logiczne operatory „i", „nie", „lub" oraz „lub wyłączającego" dla par liczb całkowitych. W języku Delphi operatorami przesunięcia w lewo i przesunięcia w prawo są odpowiednio symbole `shl` i `shr` — ich działanie jest identyczne jak w przypadku znanych z języka C# operatorów << i >>. Pozostałe operatory binarne stosowane w języku programowania Delphi są bardzo łatwe do zapamiętania: `and`, `not`, `or` i `xor`. Wszystkie operatory bitowe tego języka (wraz z odpowiednimi operatorami wykorzystywanymi w językach C# i Visual Basic .NET) przedstawiono w tabeli 5.3.

Tabela 5.3. *Operatory bitowe*

Operator	Delphi	C#	Visual Basic .NET
I	`and`	`&`	`And`
Nie	`not`	`-`	`Not`
Lub	`or`	`|`	`Or`
Lub wyłączające	`xor`	`^`	`Xor`
Przesunięcie w lewo	`shl`	<<	Brak
Przesunięcie w prawo	`shr`	>>	Brak

Procedury zwiększania i zmniejszania

Operatory zwiększania i zmniejszania są wygodnym narzędziem wykorzystywanym do dodawania i odejmowania wartości od zmiennych całkowitoliczbowych. Język Delphi nie obsługuje co prawda operatorów zwiększania i zmniejszania podobnych do bardzo popularnych wśród programistów języka C# operatorów ++ i --, ale udostępnia procedury `Inc()` i `Dec()`, które działają w ten sam sposób.

Procedury `Inc()` i `Dec()` możemy wywoływać z jednym lub z dwoma parametrami. Przykładowo, zademonstrowane poniżej dwa wiersze kodu odpowiednio zwiększają i zmniejszają zmienną `variable` o `1`:

```
Inc(variable);

Dec(variable);
```

Dwa poniższe wiersze kodu odpowiednio zwiększają i zmniejszają zmienną `variable` o `3`:

```
Inc(variable, 3);

Dec(variable, 3);
```

W tabeli 5.4 zestawiono operatory zwiększania i zmniejszania wykorzystywane w różnych językach programowania.

Tabela 5.4. *Operatory zwiększania i zmniejszania*

Operator	Delphi	C#	Visual Basic .NET
Zwiększenie	`Inc()`	++	Brak
Zmniejszenie	`Dec()`	--	Brak

Zwróć uwagę na różnicę w sposobie działania operatorów — w języku C# operatory ++ i -- zwracają wartość, czego nie robią wykorzystywane w języku Delphi procedury `Inc()` i `Dec()`. Wartości są zwracane przez dostępne w języku Delphi funkcje `Succ()` i `Pred()`, jednak żadna z tych funkcji nie modyfikuje otrzymanego parametru.

Operatory typu „zrób i przypisz”

Język Delphi nie oferuje wygodnych operatorów typu „zrób i przypisz”, które znamy np. z języka programowania C#. Takie operatory (np. += i *=) wykonują operacje arytmetyczne (w tym przypadku odpowiednio dodawanie i mnożenie) jeszcze przed przeprowadzeniem operacji przypisania. W języku Delphi tego typu operacje muszą być wykonywane przez zastosowanie dwóch osobnych operatorów. Oznacza to, że poniższy fragment kod napisanego w języku C#:

```
x += 5;
```

miałby następującą postać w języku Delphi (pozbawionym wygodnych operatorów typu „zrób i przypisz”):

```
x := x + 5;
```

Typy języka Delphi

Jedną z największych zalet języka programowania Delphi jest duża liczba dostępnych typów, która sprawia, że język ten doskonale nadaje się do programowania aplikacji dla platformy .NET Framework. Założenia przyjęte przez twórców tej platformy przewidują konieczność zapewniania zgodności rzeczywistych typów zmiennych przekazywanych do procedur i funkcji z formalnymi identyfikatorami tych parametrów w definicjach procedur lub funkcji — w praktyce oznacza to, że sam musisz pamiętać o konwersji typów. Właściwości języka Delphi w zakresie obsługi typów umożliwiają przeprowadzania odpowiednich testów kodu źródłowego, który ma na celu zapewnienie zgodności stosowanych typów. W końcu najłatwiejsze do usunięcia są te błędy, o których poinformuje nas kompilator!

Obiekty, wszędzie tylko obiekty!

Jednym z najważniejszych zagadnień związanych z podstawowymi typami obsługiwanymi przez kompilator Delphi for .NET jest możliwość konwersji wszystkich typów wartościowych do postaci odpowiednich klas. Konwersja zmiennej typu wartościowego do postaci obiektu i konwersja obiektu w zmienną typu wartościowego w terminologii przyjętej przez programistów aplikacji .NET jest określana odpowiednio jako opakowywanie (ang. *boxing*) oraz odpakowywanie (ang. *unboxing*). Liczby całkowite, łańcuchy, liczby zmiennoprzecinkowe i wszystkie inne typy wartościowe nie są już implementowane w postaci typów prostych kompilatora (tak jak w przypadku platformy Win32), ale są odwzorowywane do typów wartościowych obsługiwanych albo przez platformę .NET Framework, albo przez bibliotekę RTL lub VCL firmy Borland. Inaczej niż w wersjach Delphi dla platformy Win32 te typy wartościowe mogą mieć własne procedury i funkcje, które rozszerzają funkcjonalność metod dostępnych w odpowiednich klasach wykorzystywanych podczas realizowanego w tle opakowywania i odpakowywania tych typów. Taki mechanizm umożliwia stosowanie konstrukcji składniowych, które nie były wykorzystywane w rdzennych kompilatorach (chociaż wydają się zupełnie naturalne dla programistów, którzy mieli do czynienia z takimi językami jak Java czy SmallTalk):

```
var
  S: string;
  I: Integer;
begin
  I := 42;
  S := I.ToString;   // Możemy w ten sposób wywołać metodę dla typu Integer!
```



Typ `string` jest wykorzystywany podczas tworzenia aplikacji szczególnie często. Nie oznacza to, że inne typy są mniej ważne, jednak wielu programistów ocenia języki programowania właśnie pod kątem możliwości w zakresie przetwarzania łańcuchów. Łańcuchom poświęcimy cały rozdział 11., zatem nie będziemy ich szczegółowo omawiali w tym miejscu.

Zestawienie typów

Język programowania Delphi wyróżnia się przede wszystkim dużą liczbą typów prostych obsługiwanych w środowisku uruchomieniowym CLR. W tabeli 5.5 zestawiono i porównano ze sobą podstawowe typy występujące w języku Delphi, w języku C# oraz obsługiwane w środowisku CLR. Tabela zawiera także informacje o zgodności poszczególnych typów ze specyfikacją CLS.

Tabela 5.5. *Zestawienie typów stosowanych w języku Delphi, w języku C# i obsługiwanych w środowisku CLR*

Zakres zmiennych	Delphi	C#	CLR	Zgodność ze specyfikacją CLS
8-bitowa liczba całkowita ze znakiem	`ShortInt`	`sbyte`	`System.SByte`	Nie
8-bitowa liczba całkowita bez znaku	`Byte`	`byte`	`System.Byte`	Tak
16-bitowa liczba całkowita ze znakiem	`SmallInt`	`short`	`System.Int16`	Tak
16-bitowa liczba całkowita bez znaku	`Word`	`ushort`	`System.UInt16`	Nie
32-bitowa liczba całkowita ze znakiem	`Integer`	`int`	`System.Int32`	Tak
32-bitowa liczba całkowita bez znaku	`Cardinal`	`uint`	`System.UInt32`	Nie
64-bitowa liczba całkowita ze znakiem	`Int64`	`long`	`System.Int164`	Tak
64-bitowa liczba całkowita bez znaku	`UInt64`	`ulong`	`System.UInt164`	Nie
Liczba zmiennoprzecinkowa pojedynczej precyzji	`Single`	`float`	`System.Single`	Tak
Liczba zmiennoprzecinkowa podwójnej precyzji	`Double`	`double`	`System.Double`	Tak
Stałoprzecinkowa liczba dziesiętna	Brak	`decimal`	`System.Decimal`	Tak
Stałoprzecinkowa liczba dziesiętna Delphi	`Currency`	Brak	Brak	Nie
Data-czas	`TDateTime`[2]	Brak	`System.DateTime`	Tak
Wariant	`Variant`, `OleVariant`	Brak	Brak	Nie
1-bajtowy znak	`AnsiChar`	Brak	Brak	Nie
2-bajtowy znak	`Char`, `WideChar`	`char`	`System.Char`	Tak
Łańcuch bajtowy stałej długości	`ShortString`	Brak	Brak	Nie
Dynamiczny łańcuch 1-bajtowy	`AnsiString`	Brak	Brak	Nie
Dynamiczny łańcuch 2-bajtowy	`string`, `WideString`	`string`	`System.String`	Tak
Wartość logiczna	`Boolean`	`bool`	`System.Boolean`	Brak

[2] Typ `TDateTime` jest rekordem dołączającym do klasy `System.DateTime` metody i przeciążone operatory, które tylko w minimalnym stopniu wpływają na zachowanie `System.DateTime`, ale jednocześnie zapewniają zgodność z typem `TDateTime` stosowanym w języku Delphi dla platformy Win32.

Znaki

Język Delphi obsługuje trzy typy znakowe:

- `WideChar` — ten typ znakowy (zgodny ze specyfikacją CLS) ma rozmiar dwóch bajtów i reprezentuje pojedynczy znak w formacie Unicode.
- `Char` — ten typ jest jedynie aliasem typu `WideChar`. W wersji Delphi dla platformy Win32 typ `Char` był aliasem typu `AnsiChar`.
- `AnsiChar` — ten typ reprezentuje pojedynczy znak w przestarzałym, jednobajtowym formacie ANSI.

W swoim kodzie nigdy nie powinieneś z góry zakładać rozmiaru zmiennej typu `Char` (ani żadnego innego typu). Zamiast stałych wartości odczytanych ze specyfikacji w odpowiednich miejscach zawsze powinieneś stosować funkcję `SizeOf()`.



Standardowa procedura `SizeOf()` zwraca wyrażany w bajtach rozmiar typu lub zmiennej.

Typy wariantowe

Klasa `Variant` jest ciekawą implementacją idei pojemnika dla wielu innych typów. Oznacza to, że w czasie wykonywania programu możemy zmieniać rodzaj reprezentowanych danych w ramach wariantu. Przykładowo, poniższy fragment kodu zostanie prawidłowo skompilowany i wykonany:

```
var
  V: Variant;
  I: IInterface;
begin
  V := 'Delphi jest super!';  // wariant przechowuje łańcuch
  V := 1;                     // wariant przechowuje teraz liczbę całkowitą
  V := 123.34;                // wariant przechowuje teraz liczbę zmiennoprzecinkową
  V := True;                  // wariant przechowuje teraz wartość logiczną
  V := I;                     // wariant przechowuje teraz interfejs
end;
```

Warianty obsługują szeroki zakres typów, włącznie z tak skomplikowanymi typami jak tablice czy interfejsy. Poniższy fragment kodu skopiowany z modułu `Borland.Vcl.Variants` dobrze pokazuje mnogość obsługiwanych typów, które są identyfikowane za pomocą specjalnych wartości, tzw. typów `TVarTypes`:

```
type
  TVarType = Integer;

const                    // Dziesiętna | Alternatywne nazwy typów
  varEmpty     = $0000; // = 0        | Unassigned, Nil
  varNull      = $0001; // = 1        | Null, System.DBNull
  varSmallInt  = $0002; // = 2        | I2, System.Int16
  varInteger   = $0003; // = 3        | I4, System.Int32
  varSingle    = $0004; // = 4        | R4, System.Single
  varDouble    = $0005; // = 5        | R8, System.Double
  varCurrency  = $0006; // = 6        | Borland.Delphi.System.Currency
```

```
  varDate      = $0007; // = 7        | Borland.Delphi.System.TDateTime
  varString    = $0008; // = 8        | WideString, System.String
  varError     = $000A; // = 10       | Exception, System.Exception
  varBoolean   = $000B; // = 11       | Bool, System.Boolean
  varObject    = $000C; // = 12       | TObject, System.Object
  varDecimal   = $000E; // = 14       | System.Decimal
  varShortInt  = $0010; // = 16       | I1, System.SByte
  varByte      = $0011; // = 17       | U1, System.Byte
  varWord      = $0012; // = 18       | U2, System.UInt16
  varLongWord  = $0013; // = 19       | U4, System.UInt32
  varInt64     = $0014; // = 20       | I8, System.Int64
  varUInt64    = $0015; // = 21       | U8, System.UInt16
  varChar      = $0016; // = 22       | WideChar, System.Char
  varDateTime  = $0017; // = 23       | System.DateTime;

  varFirst     = varEmpty;
  varLast      = varDateTime;

  varArray     = $2000; // = 2     | System.Array, Dynamic Arrays
  varTypeMask  = $0FFF; // = 2
  varUndefined = -1;
```

W razie wystąpienia takiej konieczności warianty umożliwiają przekształcanie samych siebie w dowolne obsługiwane typy w czasie wykonywania funkcji przypisania wartości. Przykładowo:

```
var
  V: Variant;
  I: Integer;
begin
  V := '1';  // V przechowuje '1'
  I := V;    // Niejawna konwersja do typu Integer; I reprezentuje teraz wartość 1
end;
```

Rzutowanie typów wariantowych

W języku Delphi możemy wprost rzutować typy wyrażeń na typ `Variant`. Przykładowo, wynikiem wywołania w postaci:

```
Variant(X);
```

jest typ `Variant`, którego kod typu (`TVarTypes`) odpowiada wynikowi wyrażenia `X` — może to być liczba całkowita, liczba zmiennoprzecinkowa, stałoprzecinkowa liczba dziesiętna Delphi (`Currency`), łańcuch, znak, data-czas, klasa lub wartość logiczna.

Możemy także przeprowadzać rzutowanie typów w drugą stronę — od wariantu do odpowiedniego typu prostego. Przykładowo, jeśli mamy w kodzie instrukcję przypisania w postaci:

```
V := 1.6;
```

gdzie `V` jest zmienną typu `Variant`, po wykonaniu poniższych instrukcji przypisania otrzymamy wyniki zgodne z treścią zawartą w komentarzach:

```
S := string(V);    // zmienna S będzie zawierała łańcuch '1.6'
// zmienna I jest zaokrąglana do najbliższej liczby całkowitej (w tym przypadku 2)
```

```
I := Integer(V);
B := Boolean(V);   // zmienna B zawiera wartość True, ponieważ V jest różne od 0
D := Double(V);    // zmienna D zawiera wartość 1.6
```

Otrzymane rezultaty wynikają z konkretnych reguł konwersji typów przyjętych dla typów wariantowych. Reguły konwersji szczegółowo zdefiniowano w dokumentacji języka Delphi.

Nawiasem mówiąc, w powyższych przykładach instrukcji przypisania niepotrzebnie zastosowaliśmy rzutowanie typów wariantowych na pozostałe typy danych. Poniższy fragment kodu będzie działał tak samo:

```
V := 1.6;
S := V;
I := V;
B := V;
D := V;
```

Kiedy w jednym wyrażeniu zastosujemy wiele zmiennych typu `Variant`, może się okazać, że w związku z niejawnymi (wykonywanymi w tle) operacjami wymuszania odpowiednich typów faktycznie wykonywanych jest znacznie więcej instrukcji niż zdefiniowaliśmy w naszym wyrażeniu. Jeśli w takich przypadkach masz absolutną pewność co do typów przechowywanych w wariantach, lepszym rozwiązaniem będzie zastosowanie w wyrażeniu jawnego rzutowania typów, które przyspieszy jego przetwarzanie.

Warianty w wyrażeniach

Warianty możemy stosować w wyrażeniach z następującymi operatorami: `+`, `-`, `*`, `/`, `div`, `mod`, `shl`, `shr`, `and`, `or`, `xor`, `not`, `:=`, `=`, `<>`, `<`, `>`, `<=` oraz `>=`. Także standardowe funkcje `Inc()`, `Dec()`, `Trunc()` i `Round()` są poprawnymi składnikami wyrażeń ze zmiennymi wariantowymi.

Kiedy stosujemy warianty w wyrażeniach, kompilator Delphi podejmuje decyzje odnośnie wykonywanych operacji w oparciu o typy przechowywane w tych wariantach. Przykładowo, jeśli dwa warianty, `V1` i `V2`, zawierają liczby całkowite, wynikiem wyrażenia `V1 + V2` będzie suma tych dwóch liczb całkowitych (a więc także liczba całkowita). Jeśli jednak zmienne wariantowe `V1` i `V2` zawierają łańcuchy, wynikiem takiego samego wyrażenia będzie konkatenacja tej pary łańcuchów. Jaki jednak będzie wynik podobnego wyrażenia, jeśli użyte warianty `V1` i `V2` zawierają dwa różne typy danych? W Delphi wykorzystuje się specjalne reguły opisujące sposób postępowania w tego typu sytuacjach. Przykładowo, jeśli zmienna `V1` zawiera łańcuch `'4.5'`, a zmienna `V2` zawiera liczbę zmiennoprzecinkową, zmienna `V1` zostanie przekonwertowana do postaci liczby zmiennoprzecinkowej i następnie dodana do liczby reprezentowanej przez zmienną `V2`. Ilustruje to poniższy fragment kodu:

```
var
  V1, V2, V3: Variant;
begin
  V1 := '100';  // łańcuch
  V2 := '50';   // łańcuch
  V3 := 200;    // liczba całkowita
  V1 := V1 + V2 + V3;
end;
```

Zgodnie z tym, o czym wspomnieliśmy odnośnie specjalnych reguł języka Delphi, na pierwszy rzut oka wydaje się, że efektem wykonania powyższego kodu będzie przypisanie zmiennej `V1` liczby całkowitej równej `350`. Jeśli jednak przeanalizujemy ten fragment nieco dokładniej, okaże się, że w tym przypadku będzie zupełnie inaczej. Ponieważ domyślna kolejność wykonywania działań przewiduje pierwszeństwo skrajnie lewego operatora, najpierw zostanie wykonana operacja dodawania `V1 + V2`. Ponieważ oba operandy są wariantami reprezentującymi łańcuchy, wykonana zostanie ich konkatenacja, a zatem otrzymamy łańcuch `'10050'`. Do tego wyniku zostanie następnie dodana wartość przechowywana w wariancie `V3`. Ponieważ `V3` jest liczbą całkowitą, wynik pierwszej operacji (`'10050'`) zostanie przekonwertowany do postaci liczby całkowitej (`10050`) i dodany do wartości `V3`, a więc efektem całej operacji przypisania będzie wartość `10250` w wariancie `V1`.

W języku Delphi warianty są przekształcane do postaci największych typów numerycznych użytych w wyrażeniu, co w większości przypadków gwarantuje poprawność otrzymanego wyniku. Jeśli jednak wyrażenie zawiera dwa warianty, których kompilator Delphi nie może w żaden sensowny sposób do siebie dopasować, generowany jest wyjątek `EVariantTypeCast`. Taką sytuację ilustruje poniższy fragment kodu:

```
var
  V1, V2: Variant;
begin
  V1 := 77;
  V2 := 'witaj';
  V1 := V1 / V2;  // operacja spowoduje wygenerowanie wyjątku
end;
```

Jak już wspomnieliśmy, w niektórych sytuacjach warto jawnie rzutować typ wariantowy na konkretny typ danych — dotyczy to tych przypadków, w których dokładnie wiemy, jakie typy powinny być zastosowane w wyrażeniu i jakie typy faktycznie w tym wyrażeniu występują. Przeanalizujmy poniższy wiersz kodu:

```
V4 := V1 * V2 / V3;
```

Zanim możliwe będzie wygenerowanie wyniku tego wyrażenia, każda operacja jest obsługiwana przez funkcję czasu wykonywania, która analizuje ją na wszystkie sposoby celem określenia zgodności typów reprezentowanych przez parę operandów (w tym przypadku zmiennych wariantowych). Dopiero po przeprowadzeniu wnikliwej analizy wykonywane są operacje konwersji do właściwych typów. Taka technika przetwarzania wyrażeń wiąże się oczywiście z dodatkowymi kosztami czasowymi i zwiększonym rozmiarem kodu. Lepszym rozwiązaniem jest więc w takim przypadku rezygnacja ze stosowania wariantów. Jeśli jednak wykorzystanie wariantów jest w danym programie konieczne, możemy także zastosować dla tych zmiennych jawne rzutowanie typów, dzięki czemu wszystkie niezbędne operacje uzgadniania typów będą wykonywane już w fazie kompilacji:

```
V4 := Integer(V1) * Double(V2) / Integer(V3);
```

Pamiętaj, że rozwiązanie z jawnym rzutowaniem typów wymaga od Ciebie znajomości typów danych, do których użyte warianty mogą być bezpiecznie konwertowane.

Warianty puste i wartość Null

Dwie specjalne wartości zmiennych wariantowych wymagają krótkiego omówienia. Pierwszą z tych wartości jest `Unassigned` (nieprzypisana), która oznacza, że danemu wariantowi nie przypisano jeszcze żadnej wartości. `Unassigned` jest wartością początkową każdego wariantu. Drugą z tych wartości jest `Null`, która tym się różni od wartości `Unassigned`, że reprezentuje przypisaną wartość, a nie brak jakiejkolwiek wartości. Rozróżnienie pomiędzy brakiem wartości a wartością `Null` jest szczególnie ważne w przypadku ich stosowania w tabelach bazy danych. Więcej informacji na temat programowania aplikacji baz danych znajdziesz w części IV, „Programowanie aplikacji bazodanowych z wykorzystaniem technologii ADO.NET".

Kolejna różnica pomiędzy tymi wartościami jest widoczna w momencie wykonywania operacji na reprezentujących je wariantach — zastosowanie jakiejkolwiek operacji dla wariantu zawierającego wartość `Unassigned` powoduje przekształcenie tej wartości w liczbę `0` (w przypadku operacji numerycznych) lub łańcuch pusty (w przypadku operacji na łańcuchach). Inaczej jest w przypadku wariantów z wartością `Null` — kiedy wykorzystamy taki wariant w wyrażeniu, próba wykonania na nim operacji może spowodować wygenerowanie wyjątku `EVariantTypeCast`.

Jeśli chcemy zastosować operację przypisania lub porównania wariantu z jedną z tych dwóch specjalnych wartości, możemy wykorzystać odpowiednie warianty zdefiniowane w module `System.Vcl.Variants` (nazwane odpowiednio `Unassigned` i `Null`).

Kontrolowanie zachowania wariantów

Ustawienie wartości `True` dla właściwości `Variant.NullStrictOperations` umożliwia nam kontrolę nad generowaniem wyjątków w przypadku wykonywania operacji arytmetycznych na wariantach z wartością `Null`. Poniżej wymieniono pozostałe przełączniki (flagi) umożliwiające kontrolowanie zachowań wariantów z wartością `Null` w przypadku ich konwersji do postaci łańcuchów i wartości logicznych.

```
NullEqualityRule: TNullCompareRule;
NullMagnitudeRule: TNullCompareRule;
NullAsStringValue: string;
NullStrictCOnvert: Boolean;
BooleanToStringRule: TBooleanToStringRule;
BooleanTrueAsOrdinalValue: Integer;
```

Typy definiowane przez użytkownika

Liczby całkowite, łańcuchy i liczby zmiennoprzecinkowe często nie są wystarczającymi środkami do właściwego reprezentowania zmiennych wykorzystywanych podczas rozwiązywania rzeczywistych problemów, przed jakimi stają programiści. W podobnych przypadkach konieczne jest tworzenie własnych typów, które będą lepiej reprezentowały zmienne niezbędne do rozwiązania danego problemu. W języku programowania Delphi takie typy definiowane przez użytkownika (lub po prostu typy użytkownika) zwykle przyjmują postać rekordów lub klas — definiujemy je za pomocą słowa kluczowego `Type`.

Tablice

Język programowania Delphi umożliwia nam tworzenie tablic dla dowolnych typów zmiennych. Przykładowo, zmienna zadeklarowana jako tablica ośmiu liczb całkowitych będzie miała następującą postać:

```
var
  A: Array[0..7] of Integer;
```

Powyższa instrukcja jest równoważna następującej deklaracji napisanej w języku programowania C#:

```
int A[8];
```

Istnieje także deklaracja równoważna w języku Visual Basic .NET:

```
Dim A(8) as Integer
```

Tablice stosowane w języku Delphi mają specjalną własność, która odróżnia je od podobnych struktur stosowanych w innych językach programowania — indeksy tablic Delphi nie muszą się rozpoczynać od z góry określonej liczby (w innych językach jest to liczba `0` lub `1`). Oznacza to, że możemy zadeklarować trójelementową tablicę, która będzie się rozpoczynała od indeksu `28`, oto przykład:

```
var
  A: Array[28..30] of Integer;
```

Ponieważ nie mamy gwarancji, że tablice stosowane w języku Delphi rozpoczynają się od indeksu `0` lub `1`, musimy zachować ostrożność podczas iteracyjnego przeglądania tych tablic w pętli `for`. Kompilator Delphi oferuje wbudowane funkcje nazwane `High()` i `Low()`, które zwracają odpowiednio górną i dolną granicę dla zmiennej lub typu tablicowego. Stosując te funkcje podczas kontrolowania zakresu dla pętli `for`, możesz uniknąć wielu błędów i bardzo ułatwić konserwowanie swojego kodu źródłowego:

```
var
  A: array[28..30] of Integer;
  i: Integer;
begin
  for i := Low(A) to High(A) do  // Staraj się tworzyć elastyczne pętle for!
    A[i] := i;
end;
```

Do tworzenia tablic wielowymiarowych w języku Delphi wykorzystuje się listy zakresów oddzielone przecinkami:

```
var
  // dwuwymiarowa tablica liczb całkowitych (typu Integer)
  A: array[1..2, 1..2] of Integer;
```

Aby uzyskać dostęp do elementu wielowymiarowej tablicy, należy użyć przecinków oddzielających indeksy poszczególnych wymiarów:

```
I := A[1, 2];
```

Tablice dynamiczne

Tablice dynamiczne są strukturami z dynamicznie przydzielaną pamięcią, a więc są to takie tablice, których wymiary nie są znane w czasie kompilacji programu. Aby zadeklarować tablicę dynamiczną, musimy po prostu użyć znanej nam deklaracji tablic, ale bez określania konkretnych wymiarów, oto przykład:

```
var
  // dynamiczna tablica łańcuchów:
  SA: array of string;
```

Zanim będziemy mogli użyć tak zadeklarowanej tablicy, musimy zastosować procedurę `SetLength()`, która przydzieli tej tablicy odpowiednią przestrzeń w pamięci:

```
begin
  // przydziel przestrzeń dla 33 elementów:
  SetLength(SA, 33);
```

Po przydzieleniu pamięci możesz już w normalny sposób (identyczny jak w przypadku tablic statycznych) uzyskiwać dostęp do elementów tablicy dynamicznej:

```
SA[0] := 'Kubuś Puchatek lubi miód';
InnyŁańcuch := SA[0];
```



Tablice dynamiczne zawsze są indeksowane począwszy od wartości `0`.

Tablice dynamiczne są zarządzane w czasie wykonywania przez środowisko uruchomieniowe platformy .NET, zatem nie ma potrzeby (a w rzeczywistości nie ma nawet możliwości) zwalniania zajmowanej przez nie pamięci, ponieważ i tak w pewnym momencie (po opuszczeniu pewnego zakresu kodu) zostaną usunięte przez mechanizm odzyskiwania (odśmiecania) pamięci. Możesz jednak mieć do czynienia z sytuacją, w której chciałbyś zażądać od środowiska uruchomieniowego .NET usunięcia dynamicznej tablicy z pamięci jeszcze przed opuszczeniem danego zakresu kodu (np. w przypadku, gdy taka tablica wykorzystuje zbyt dużą ilość pamięci). W tym celu wystarczy przypisać takiej zmiennej tablicowej wartość `nil`:

```
SA := nil;  // żądanie zwolnienia pamięci zajmowanej przez tablicę SA
```

Zauważ, że przypisanie wartości `nil` nie powoduje automatycznego zwolnienia pamięci przydzielonej wcześniej tablicy `SA` — w ten sposób zwalniamy jedynie jedną referencję do tej tablicy, ponieważ może istnieć więcej niż jedna zmienna wskazująca na tablicę, na którą wskazywała także zmienna `SA`. Dopiero po zwolnieniu ostatniej referencji do tej tablicy dynamicznej wykorzystywany w środowisku uruchomieniowym .NET mechanizm odzyskiwania pamięci zwolni pamięć zajmowaną przez tę tablicę (ale dopiero w kolejnym cyklu działania tego mechanizmu).

Na tablicach dynamicznych operujemy zgodnie z regułami semantycznymi przyjętymi dla referencji, a nie według zasad semantycznych dla zmiennych wartościowych, w tym tablic statycznych. Oto szybki test: Jaka będzie wartość elementu `A1[0]` po wykonaniu poniższego fragmentu kodu?

```
var
  A1, A2: array of Integer;
begin
  SetLength(A1, 4);
  A2 := A1;
  A1[0] := 1;
  A2[0] := 26;
```

Poprawna odpowiedź brzmi `26`, ponieważ przypisanie `A2 := A1` nie tworzy nowej tablicy, a jedynie sprawia, że `A1` jest referencją do tej samej tablicy, na którą wskazuje zmienna tablicowa `A2`. Oznacza to, że wszystkie modyfikacje zmiennej `A2` spowodują zmiany w zmiennej `A1`. Jeśli zamiast tworzenia nowej referencji chcesz skopiować tablicę `A1` do zmiennej `A2`, powinieneś użyć standardowej funkcji `Copy()`:

```
A2 := Copy(A1);
```

Po wykonaniu powyższego wiersza kodu, zmienne `A2` i `A1` będą dwiema osobnymi tablicami, które początkowo będą zawierały te same dane. Zmiany w jednej z tych tablic nie będą powodowały zmian w drugiej. Dodatkowo możesz wykorzystać opcjonalne parametry funkcji `Copy()` do określenia elementu początkowego i łącznej liczby elementów do skopiowania; poniżej przedstawiono przykład takiego zastosowania tej funkcji:

```
// skopiuj 2 elementy, począwszy od pierwszego:
A2 := Copy(A1, 1, 2);
```

Podobnie jak tablice statyczne, także tablice dynamiczne mogą być wielowymiarowe. Aby zdefiniować więcej wymiarów, w deklaracji tablicy dla każdego z wymiarów użyj dodatkowego wyrażenia `array of`:

```
var
  // dwuwymiarowa tablica dynamiczna liczb całkowitych:
  IA: array of array of Integer;
```

Aby przydzielić pamięć takiej wielowymiarowej tablicy, przekaż rozmiary dla poszczególnych wymiarów w postaci dodatkowych parametrów procedury `SetLength()`:

```
begin
  // IA będzie tablicą liczb całkowitych o wymiarach 5 × 5
  SetLength(IA, 5, 5);
```

Uzyskiwanie dostępu do elementów dynamicznych tablic wielowymiarowych niczym nie różni się od sposobu uzyskiwania takiego samego dostępu do wielowymiarowych tablic statycznych — indeksy dla poszczególnych wymiarów umieszczamy wewnątrz jednej pary nawiasów kwadratowych i oddzielamy przecinkami:

```
IA(0,3] := 28;
```

W języku programowania Delphi jest także obsługiwana składnia dostępu do tablic wielowymiarowych znana z rodziny języków C (C, C++, C#):

```
IA[0][3] := 28;
```

Rekordy

Struktura zdefiniowana przez użytkownika jest w języku programowania Delphi nazywana rekordem i odpowiada konstrukcji `struct` znanej z języka C# oraz konstrukcji `Type` stosowanej w języku Visual Basic .NET. Przykładowo, poniżej przedstawiono definicję rekordu w Delphi wraz z równoważnymi definicjami w językach C# i Visual Basic .NET:

```
{ Delphi }
Type
  MyRec = record
    i: Integer;
    d: Double;
  end;

/* C# */
public struct MyRec
{
  int i;
  double d;
}

'Visual Basic
Type MyRec
  i As Integer
  d As Double
End Type
```

Pracując ze strukturami tego typu, wykorzystujemy symbol kropki do uzyskiwania dostępu do poszczególnych pól rekordu. Oto przykład:

```
var
  N: MyRec;
begin
  N.i := 23;
  N.d := 3.4;
end;
```

Dla rekordów definiowanych w języku Delphi for .NET możliwe jest stosowanie metod, mechanizmu przeciążania operatorów oraz implementacji interfejsów. Takich możliwości nie mieliśmy we wcześniejszych wersjach kompilatora Delphi (dla platformy Win32). Wszystkie te elementy omówimy bardziej szczegółowo w dalszej części rozdziału — podczas analizy typów `class`. W poniższym przykładowym kodzie przedstawiono składnię dla wymienionych elementów stosowanych dla rekordów (zademonstrowana składnia różni się od omawianej później składni deklarowania klas):

```
IBlah = interface
  procedure bar;
end;

Foo = record(IBlah)  // ten rekord implementuje interfejs IBlah
  AField: Integer;
  procedure bar;
  class operator Add(a, b: Foo): Foo;  // przeciążenie + operator
end;
```

Zbiory

Zbiory są unikalnym typem występującym tylko w języku programowania Delphi — nie mają swoich odpowiedników ani w języku C#, ani w języku Visual Basic .NET. Zbiory są efektywnym i wygodnym narzędziem reprezentowania wielu elementów typów porządkowych, znaków typu `AnsiChar` lub wartości wyliczeniowych. Nowy typ zbioru możemy deklarować za pomocą słów kluczowych `set of` poprzedzających typ elementów zbioru lub podzakres możliwych wartości zbioru. Oto przykład takiej deklaracji:

```
type
  TCharSet = set of AnsiChar;      // możliwe liczby: #0 - #255

  TEnum = (Monday, Tuesday, Wednesday, Thursday, Friday);
  TEnumSet = set of TEnum;  // zbiór może zawierać dowolną kombinację elementów TEnum

  TSubrangeSet = set of 1..10; // możliwe liczby: 1 - 10
```

Zwróć uwagę na ograniczenie odnośnie maksymalnej liczby elementów w zbiorze, która wynosi 255. Dodatkowo należy pamiętać, że po słowach kluczowych `set of` mogą występować wyłącznie identyfikatory typów porządkowych. Oznacza to, że poniższe deklaracje są niepoprawne:

```
type
  TIntSet = set of Integer;  // Niepoprawne: zbyt wiele elementów
  TStrSet = set of string;   // Niepoprawne: string nie jest typem porządkowym
```

Zbiory przechowują swoje elementy w wewnętrznym formacie bitowym, dzięki czemu są bardzo efektywne zarówno w obszarze szybkości przetwarzania, jak i ilości wykorzystywanej pamięci.



Jeśli przenosisz kod z platformy Win32 na platformę .NET Framework, miej na uwadze istniejące różnice w mechanizmie reprezentowania znaków — w świecie .NET wykorzystuje się format 2-bajtowy, natomiast w aplikacjach dla platformy Win32 stosowano format 1-bajtowy. Oznacza to, że deklaracja `set of Char` jest przez kompilator .NET zamieniana na deklarację `set of AnsiChar`, co w niektórych przypadkach może prowadzić do zmiany pierwotnego znaczenia kodu źródłowego. Kompilator wygeneruje wówczas odpowiednie ostrzeżenie z zaleceniem zamiany tej deklaracji na jawną deklarację zbioru `set of AnsiChar`.

Stosowanie zbiorów

Podczas konstruowania wartości zbioru na podstawie jednego lub więcej elementów powinieneś używać nawiasów kwadratowych. Poniższy fragment kodu demonstruje sposób deklarowania zmiennych typu `set` oraz przypisywania im wartości:

```
type
  TCharSet = set of AnsiChar;      // możliwe liczby: #0 - #255

  TEnum = (Monday, Tuesday, Wednesday, Thursday, Friday);
  TEnumSet = set of TEnum;  // zbiór może zawierać dowolną kombinację elementów TEnum


var
  CharSet: TCharSet;
```

```
  EnumSet: TEnumSet;
  SubrangeSet: set of 1..10; // możliwe liczby: 1 - 10
  AlphaSet: set of 'A'..'z'; // możliwe elementy: 'A' - 'z'

begin
  CharSet := ['A'..'J', 'a', 'm'];
  EnumSet := [Saturday, Sunday];
  SubrangeSet := [1, 2, 4..6];
  AlphaSet := [];  // zbiór pusty - brak elementów
end;
```

Operatory dla zbiorów

Język programowania Delphi oferuje wiele operatorów, które można stosować do manipulowania na zbiorach. Możemy używać tych operatorów do określania elementów należących do zbiorów, do łączenia (sumowania) zbiorów, do określania różnicy oraz do wyznaczania części wspólnej pary zbiorów.

Przynależność. Operator `in` służy do określania, czy dany element należy do konkretnego zbioru. Przykładowo, poniższy fragment kodu można wykorzystać do sprawdzenia, czy wspomniany przed chwilą zbiór `CharSet` zawiera literę `'S'`:

```
if 'S' in CharSet then
  // zrób coś
```

Poniższy fragment kodu ma na celu określenie, czy zbiór `EnumSet` *nie* zawiera elementu `Monday`:

```
if not (Monday in EnumSet) then
  // zrób coś
```

Suma i różnica. Za pomocą operatorów + i - lub pary procedur `Include()` i `Exclude()` możemy dodawać i usuwać elementy do i ze zmiennej typu `set` (czyli do i ze zbioru):

```
Include(CharSet, 'a');           // dodaje do zbioru element 'a'
CharSet := CharSet + ['b'];      // dodaje do zbioru element 'b'
Exclude(CharSet, 'x');            // usuwa ze zbioru element 'x'
CharSet := CharSet - ['y', 'z']; // usuwa ze zbioru elementy 'y' i 'z'
```



Wszędzie tam, gdzie jest to możliwe, staraj się stosować parę procedur `Include()` i `Exclude()` odpowiednio do dodawania i odejmowania pojedynczych elementów, ponieważ wspomniane procedury są bardziej efektywne od operatorów + i -.

Iloczyn zbiorów. Do wyznaczania iloczynu (części wspólnej) dwóch zbiorów możemy używać operatora *. Wynikiem wyrażenia w postaci `Set1 * Set2` jest zbiór zawierający wszystkie te elementy ze zbioru `Set1`, które należą także do zbioru `Set2`. Przykładowo, poniższy fragment kodu może być użyty w roli efektywnego narzędzia do określania, czy dany zbiór zawiera powtarzające się elementy:

```
if ['a', 'b', 'c'] * CharSet = ['a', 'b', 'c'] then
  // zrób coś
```

„Niebezpieczny" kod

Na tym etapie programiści, którzy mają doświadczenie w pracy z językiem Delphi dla platformy Win32, mogą stwierdzić: „do tej pory wszystko się zgadza, ale co się stało ze wskaźnikami?". Chociaż w języku programowania Delphi for .NET wskaźniki nadal są dostępne, z punktu widzenia specyfikacji platformy .NET Framework są traktowane jak konstrukcje „niebezpieczne", ponieważ umożliwiają bezpośredni dostęp do pamięci. Oznacza to, że wykorzystanie wskaźników w tworzonych aplikacjach .NET będzie wymagało odpowiedniego ustawienia kompilatora — zezwolenia na akceptowanie „niebezpiecznego" kodu. Aby umożliwić pisanie i kompilowanie takiego kodu, musisz wykonać następujące kroki:

1. Dołączyć dyrektywę `{$UNSAFECODE ON}` do modułu zawierającego „niebezpieczny" kod.
2. Oznaczyć słowem kluczowym `unsafe` wszystkie funkcje, które zawierają „niebezpieczny" kod.

Oto przykład. Poniższy „niebezpieczny" kod zostanie pomyślnie skompilowany przez kompilator Delphi for .NET.

```
{$UNSAFECODE ON}

procedure RunningWithScissors; unsafe;
var
  A: array[0..31] of Char;
  P: PChar;   // PChar jest "niebezpiecznym" typem
begin
  A := 'safety first';  // wypełnia tablicę znaków
  P := @A[0];           // wskazuje na pierwszy element
  P[0] := 'S';          // zmienia pierwszy element
  MessageBox.Show(A);   // wyświetla zmienioną tablicę
end;
```

Zwróć uwagę na sposób wykorzystania dostępnego w języku Delphi operatora `@` (tzw. operatora adresu) w celu uzyskania adresu wybranej struktury danych.



Stosowanie „niebezpiecznego" kodu jest bardzo niemile widziane w aplikacjach dla platformy .NET Framework, ponieważ tak zbudowane aplikacje nie mogą przejść testów stosowanego na tej platformie narzędzia PEVerify, przez co mogą być objęte jeszcze poważniejszymi ograniczeniami w zakresie bezpieczeństwa. Z drugiej strony, „niebezpieczny" kod może być doskonałym sposobem na wypełnienie przepaści dzielącej świat platformy Win32 od świata platformy .NET. Przenoszenie całych dużych aplikacji z platformy Win32 na platformę .NET w jednym kroku i tak jest bardzo trudne, jednak zastosowanie „niebezpiecznego" kodu umożliwia nam w miarę sprawne przenoszenie przynajmniej fragmentów takich aplikacji i sukcesywne dostosowywanie poszczególnych modułów do wymagań stawianych przez platformę docelową.

Możesz zakładać, że fragmenty kodu prezentowane w tej części rozdziału muszą być kompilowane z dyrektywą `{$UNSAFECODE ON}` i słowami kluczowymi `unsafe` (z oczywistych względów oba te elementy nie we wszystkich przykładach są widoczne).

Wskaźniki

Wskaźnik (ang. *pointer*) jest zmienną reprezentującą konkretną lokalizację w pamięci. Przykład wskaźnika (typ `PChar`) miałeś już okazję zobaczyć wcześniej w tym rozdziale. Stosowany w języku Delphi wskaźnik ogólny został zręcznie nazwany `Pointer`. Często mówi się, że `Pointer` jest wskaźnikiem bez typu, ponieważ reprezentuje wyłącznie adres w pamięci i kompilator nie utrzymuje żadnych dodatkowych informacji na temat danych wskazywanych przez ten wskaźnik. Pojęcie wskaźnika bez typu stoi jednak w sprzeczności z jedną z podstawowych cech języka Delphi, jaką jest gwarancja bezpieczeństwa typów, zatem wszędzie tam, gdzie jest to możliwe, powinniśmy w naszym kodzie stosować wskaźniki z typami.



W środowisku .NET do reprezentowania wskaźników bez typów jest wykorzystywany typ `System.IntPtr`.

Wskaźniki z typami są deklarowane za pomocą operatora ^ (tzw. operatora wskaźnika) w znajdującej się na początku programu części `Type`. Wskaźniki z typami ułatwiają kompilatorowi dokładne śledzenie rodzaju typów wskazywanych przez poszczególne wskaźniki, a więc umożliwiają kompilatorowi śledzenie tego, co robisz (i co możesz zrobić) ze swoimi zmiennymi wskaźnikowymi. Oto kilka przykładów typowych deklaracji wskaźników:

```
Type
  PInt = ^Integer;          // PInt jest teraz wskaźnikiem na liczbę całkowitą
  Foo = record              // deklaracja rekordu
    GobbledyGook: string;
    Snarf: Double;
  end;
  PFoo = ^Foo;              // PFoo jest wskaźnikiem na zmienną typu Foo
var
  P: Pointer;               // wskaźnik bez typu
  P2: PFoo;                 // egzemplarz PFoo
```



Programiści C/C++ z pewnością zwrócili uwagę na podobieństwo pomiędzy stosowanym w języku Delphi operatorem ^, a znanym z języków C i C++ operatorem *. Dostępny w języku Delphi typ `Pointer` (wskaźnika bez typu) odpowiada używanemu w językach C i C++ typowi `void *`.

Pamiętaj, że zmienna wskaźnikowa przechowuje jedynie adres w pamięci. Za przydzielenie odpowiedniej ilości pamięci dla struktur wskazywanych przez wskaźniki zawsze odpowiada programista. Poprzednie wersje języka programowania Delphi oferowały wiele funkcji, za pomocą których programiści mogli przydzielać i zwalniać pamięć; ponieważ jednak bezpośrednie przydzielanie pamięci jest operacją wykonywaną bardzo rzadko w aplikacjach .NET, zazwyczaj wykorzystuje się do tego celu metody udostępniane przez klasę `System.Runtime.InteropServices.Marshal`. Poniższy fragment kodu demonstruje sposób wykorzystania tej klasy do tworzenia i zwalniania bloków pamięci, a także kopiowania danych do i z tego bloku.

```
{$UNSAFECODE ON}

type
  TArray = array[0..31] of Char;
```

```
procedure ArrayCopy; unsafe;
var
  A1: TArray;
  A2: array of char;
  P: IntPtr;
begin
  A1 := 'Bezpieczeństwo przede wszystkim';  // wypełnia tablicę znaków
  SetLength(A2, High(TArray) + 1);
  P := Marshal.AllocHGlobal(High(TArray) + 1);
  try
    Marshal.Copy(A1, 0, P, High(TArray) + 1);   // kopiuje A1 do temp
    Marshal.Copy(P, A2, 0, High(TArray) + 1);   // kopiuje temp do A2
    MessageBox.Show(A2);   // wyświetla zmienioną tablicę
  finally
    Marshal.FreeHGlobal(P);
  end;
end;
```

Jeśli chcesz uzyskać dostęp do danych wskazywanych przez konkretny wskaźnik, użyj operatora ^ bezpośrednio za nazwą zmiennej wskaźnikowej. Tę technikę nazywa się niekiedy dereferencją (usuwaniem pośredniości) wskaźnika. Właśnie taką metodę pracy ze wskaźnikami zademonstrowano w poniższym przykładzie:

```
procedure PointerFun; unsafe;
var
  I: Integer;
  PI: ^Integer;
begin
  I := 42;
  PI := @1;                      // wskazuje na I
  PI^ := 24;                     // zmienia I
  MessageBox.Show(I.ToString);
end;
```

Kompilator Delphi wykorzystuje mechanizm ścisłego kontrolowania typów wskaźnikowych. Przykładowo, kompilator wykaże brak zgodności typów dla tak zadeklarowanych zmiennych wskaźnikowych `a` i `b`:

```
var
  a: ^Integer;
  b: ^Integer;
```

Okazuje się, że równoważne deklaracje zmiennych wskaźnikowych w języku C nie powodują podobnych problemów w kompilatorze tego języka:

```
int  *a;
int  *b
```

Wynika to z faktu, że język Delphi tworzy unikalny typ dla każdej deklaracji wskaźnika do typu — oznacza to, że jeśli chcemy bez problemów przypisywać wartości ze zmiennej `a` do zmiennej `b`, musimy stworzyć i nazwać specjalny typ dla tych zmiennych:

```
type
  PtrInteger = ^Integer;  // tworzy nazwany typ

var
  a: PtrInteger;
  b: PtrInteger;   // typy zmiennych a i b są teraz zgodne
```



Kiedy wskaźnik na nic nie wskazuje (jego wartość jest równa `0`), programiści Delphi mówią, że jest równy `nil`, a o samym wskaźniku mówią, że jest wskaźnikiem pustym.

Łańcuchy zakończone znakiem Null

Wcześniej w tym rozdziale wspominaliśmy o trzech obsługiwanych w języku Delphi rodzajach łańcuchów zakończonych znakiem pustym: `PChar`, `PAnsiChar` oraz `PWideChar`. Zgodnie z tym każdy z nich reprezentuje łańcuch znaków innego formatu (wspominaliśmy już, że w języku Delphi istnieją trzy formaty reprezentowania znaków). W języku Delphi for .NET typ `PChar` jest aliasem typu `PWideChar`; w języku Delphi dla platformy Win32 ten sam typ był aliasem typu `PAnsiChar`. W tym rozdziale będziemy stosować pewne uproszczenie — do każdego z tych typów łańcuchowych będziemy się odwoływali za pomocą nazwy `PChar`. W języku Delphi for .NET udostępniono typ `PChar` przede wszystkim w celu zapewnienia zgodności z aplikacjami napisanymi we wcześniejszych wersjach tego języka. Typ ten jest definiowany jako wskaźnik na łańcuch zakończony znakiem pustym (zerem). Ponieważ `PChar` jest surowym, niezarządzanym i „niebezpiecznym" typem wskaźnikowym, pamięć dla tych typów nie jest automatycznie ani przydzielana, ani zarządzana przez środowisko uruchomieniowe .NET.

W wersji języka Delphi dla platformy Win32 zmienne wskaźnikowe typu `PChar` były zgodne (przynajmniej w obszarze operacji przypisania) ze zmiennymi typu `string`. Jednak w wersji dla platformy .NET zrezygnowano ze zgodności tych typów, co znacznie ogranicza możliwości ich stosowania w świecie aplikacji .NET.

Rekordy wariantowe

Język programowania Delphi obsługuje także rekordy wariantowe, które umożliwiają wzajemne nakładanie się różnych struktur danych w tej samej części pamięci wykorzystywanej przez dany rekord. Rekordów tego typu nie należy mylić z wariantowym typem danych (`Variant`) — rekordy wariantowe umożliwiają niezależny dostęp do wszystkich nachodzących na siebie pól danych. Jeśli masz doświadczenie w programowaniu w języku C, z pewnością dostrzegasz podobieństwo pomiędzy stosowanymi w Delphi rekordami wariantowymi a uniami (`union`) wewnątrz struktur (`struct`) definiowanych w języku C. Przedstawiony poniżej fragment kodu pokazuje przykład takiego rekordu wariantowego, w którym wszystkie użyte zmienne (typów `Double`, `Integer` i `Char`) są przechowywane w tej samej przestrzeni pamięciowej:

```
type
  TVariantRecord = record
    NullStrField: PChar;
    IntField: Integer;
    case Integer of
      0: (D: Double);
      1: (I: Integer);
      2: (C: Char);
  end;
```



Reguły języka programowania Delphi zakładają, że część wariantowa rekordu nie może zawierać żadnych typów zarządzanych w czasie wykonywania aplikacji. Do zabronionych typów należą klasy, interfejsy, warianty, tablice dynamiczne oraz łańcuchy.

Poniżej przedstawiono deklarację struktury w języku C, która jest równoważna powyższej deklaracji rekordu wariantowego języka Delphi:

```
struct TUnionStruct
{
  char * StrField;
  int IntField;
  union u
  {
    double D;
    int i;
    char c;
  };
};
```

Ponieważ rekordy wariantowe zakładają jawny dostęp do rozplanowania pamięci, tego rodzaju konstrukcje są także uważane za typy „niebezpieczne".



Rozkład pamięci wykorzystywanej przez rekord może być kontrolowany przez programistę aplikacji .NET za pośrednictwem atrybutów `StructLayout` i `FieldOffset`.

Klasy i obiekty

Klasa jest typem wartościowym, który może zawierać dane, właściwości, metody i operatory. Model obiektowy zastosowany w języku programowania Delphi omówimy znacznie bardziej szczegółowo w dalszej części tego rozdziału, w podrozdziale „Stosowanie obiektów Delphi" — na tym etapie skupimy się jedynie na podstawowych regułach składniowych dla deklaracji klas języka Delphi. Klasę definiuje się w następujący sposób:

```
Type
  TChildObject = class(TParentObject)
  public
    SomeVar: Integer;
    procedure SomeProc;
  end;
```

Powyższa deklaracja jest równoważna następującej deklaracji stworzonej w języku programowania C#:

```
public class TChildObject: TParentObject
{
  public int SomeVar;
  public void SomeProc() {};
}
```

Metody klas są definiowane w sposób niemal identyczny jak normalne procedury i funkcje (patrz podrozdział „Procedury i funkcje") — jedyną różnicą jest dodatkowo umieszczana przed nazwą procedury lub funkcji nazwa klasy wraz z oddzielającą kropką:

```
procedure TChildObject.SomeProc;
begin
  { tutaj znajduje się kod procedury }
end;
```

Stosowany w języku Delphi symbol kropki ma podobne znaczenie jak wykorzystywany w językach C# i Visual Basic .NET identyczny operator służący do odwoływania się do składowych klas.

Aliasy typów

Język programowania Delphi daje możliwość tworzenia nowych nazw (tzw. aliasów) dla wcześniej zdefiniowanych typów. Przykładowo, jeśli chcemy dla typu `Integer` stworzyć nową nazwę, w tym przypadku `MyReallyNiftyInteger`, możemy użyć w naszym kodzie następującej deklaracji:

```
type
  MyReallyNiftyInteger = Integer;
```

Nowo zdefiniowany alias typu jest pod każdym względem zgodny z typem, który reprezentuje. Oznacza to, że w tym przypadku możemy używać typu `MyReallyNiftyInteger` wszędzie tam, gdzie moglibyśmy użyć standardowego typu `Integer`.

W języku Delphi istnieje jednak możliwość definiowania aliasów z mocniejszą kontrolą typów, które z punktu widzenia kompilatora są zupełnie nowymi, unikalnymi typami. Aby zdefiniować taki alias, użyj słowa zastrzeżonego `type` w następujący sposób:

```
type
  MyOtherNeatInteger = type Integer;
```

Dzięki zastosowaniu takiej składni, typ `MyOtherNeatInteger` będzie w razie konieczności konwertowany na typ Integer (np. w celu prawidłowego wykonania operacji przypisania), jednak typ ten nie będzie zgodny ze standardowym typem `Integer` w roli parametrów `var` i `out`. Oznacza to, że poniższy fragment kodu jest syntaktycznie prawidłowy:

```
var
  MONI: MyOtherNeatInteger;
  I: Integer;
begin
  I := 1;
  MONI := I;
```

Z drugiej jednak strony skompilowanie poniższego fragmentu jest niemożliwe:

```
procedure Goon(var Value: Integer);
begin
  // jakiś kod
end;

var
  M: MyOtherNeatInteger;
begin
  M := 29;
  Goon(M);  // Błąd: M nie jest zmienną zgodną z typem Integer
```

Poza wspomnianymi problemami z wymuszaną przez kompilator ścisłą zgodnością typów okazuje się, że podczas kompilacji dla każdego aliasu z mocniejszą kontrolą typów automatycznie są generowane odpowiednie informacje czasu wykonywania. Dzięki temu możliwe jest tworzenie dla poszczególnych typów unikalnych edytorów właściwości — patrz rozdział 8., „Mono. Wieloplatformowe środowisko. NET”.

Rzutowanie i konwersja typów

Rzutowanie typów jest techniką, dzięki której możesz wymusić na kompilatorze traktowanie zmiennej pewnego typu jak zmiennej innego typu. Z uwagi na stosowaną w języku programowania Delphi ścisłą kontrolę typów, z pewnością zauważysz bardzo małą tolerancję kompilatora tego języka w obszarze dopasowywania formalnych i rzeczywistych typów parametrów wywołań funkcji. W efekcie będziesz niekiedy zmuszony do rzutowania zmiennej jednego typu na postać zmiennej innego typu, aby z jednej strony osiągnąć założony cel i jednocześnie „zadowolić" kompilator. Przypuśćmy na przykład, że musimy przypisać wartość zmiennej znakowej (typu `char`) do zmiennej typu `Word`:

```
var
  c: char;
  w: Word;
begin
  c := 's';
  w := c;   // kompilator wskaże na błąd w tym wierszu
end.
```

W poniższym przykładzie zastosowany mechanizm rzutowania typów jest wymagany do konwersji zmiennej `c` na typu `Word`. Efektem wykorzystania tego mechanizmu jest zasygnalizowanie kompilatorowi tego, że programista wie, co robi, i rzeczywiście chce przekonwertować jeden typ na drugi:

```
var
  c: Char;
  w: Word;
begin
  c := 's';
  w := Word(c);   // ten wiersz kodu nie spowoduje błędu kompilacji
end.
```



Rzutowanie zmiennej z jednego na inny typ jest możliwe tylko w sytuacji, w której rozmiar zmiennych obu typów jest identyczny. Przykładowo, nie możemy rzutować liczby zmiennoprzecinkowej typu `Double` na liczbę całkowitą typu `Integer`. Konwersja liczby zmiennoprzecinkowej na liczbę całkowitą wymaga zastosowania funkcji `Trunc()` (przycinania części ułamkowej) lub `Round()` (zaokrąglania). Przekształcenie odwrotne — zamiana liczby całkowitej na liczbę zmiennoprzecinkową — wymaga tylko operatora przypisania: `FloatVar := IntVar`. Możemy także jawnie lub niejawnie konwertować typy za pomocą specjalnych operatorów konwersji zdefiniowanych dla wykorzystywanych klas (więcej informacji na temat przeciążania operatorów znajdziesz w dalszej części tego rozdziału).

Język programowania Delphi obsługuje także specjalny mechanizm rzutowania typów obiektowych w oparciu o operator `as`. Operator ten działa identycznie jak standardowe funkcje rzutowania typów z tym wyjątkiem, że w przypadku błędu zamiast generować wyjątek, zwraca wartość `Null`.

Zasoby łańcuchowe

Język Delphi oferuje klauzulę `resourcestring`, dzięki której programista ma możliwość umieszczania zasobów łańcuchowych bezpośrednio w kodzie źródłowym tworzonej aplikacji. Zasoby łańcuchowe są po prostu stałymi łańcuchowymi (zwykle tymi, które są wyświetlane przed użytkownikiem), które fizycznie należą do zasobów dołączonych do aplikacji lub biblioteki, a więc nie są wbudowane w jej kod źródłowy. W naszym kodzie źródłowym powinniśmy się jednak odwoływać do zasobów łańcuchowych, a nie do z góry zdefiniowanych stałych tego typu. Oddzielając zawartość tych łańcuchów od kodu źródłowego, możemy ułatwić przyszłe tłumaczenie naszej aplikacji na inne języki — przy zastosowaniu odpowiedniej konstrukcji aplikacji takie tłumaczenie będzie dotyczyło wyłącznie zasobów łańcuchowych. Zasoby łańcuchowe są deklarowane w klauzuli `resourcestring` za pomocą wyrażeń w postaci *identyfikator = stała łańcuchowa*, oto przykład:

```
resourcestring
  ReString1 = 'Zasób łańcuchowy nr 1';
  ReString2 = 'Zasób łańcuchowy nr 2';
  ReString3 = 'Zasób łańcuchowy nr 3';
```

Zgodnie z regułami syntaktycznymi zasoby łańcuchowe mogą być wykorzystywane w naszym kodzie źródłowym w sposób, przypominający stosowanie funkcji zwracających łańcuchy:

```
resourcestring
  ReString1 = 'witaj';
  ReString2 = 'świecie';

var
  String1: string;

begin
  String1 := ResString1 + ' ' + ResString2;
  .
  .
  .
end;
```

Testowanie warunków

W tym podrozdziale spróbujemy porównać stosowane w języku programowania Delphi konstrukcje `if` i `case` z ich odpowiednikami wykorzystywanymi w językach C# i Visual Basic .NET. Zakładamy, że w swojej pracy stosowałeś już tego typu konstrukcje programistyczne, zatem nie będziemy poświęcać zbyt dużo czasu na wyjaśnianie ich znaczenia i funkcjonowania.

Instrukcja if

Instrukcja warunkowa `if` umożliwia nam określanie, czy dane warunki są spełnione jeszcze przed wykonaniem konkretnego bloku kodu. Poniżej zaprezentowano przykładową instrukcję warunkową `if` w języku programowania Delphi wraz z jej odpowiednikami w językach C# i Visual Basic .NET:

```
{ Delphi }
if x = 4 then y := x;

/* C# */
if (x == 4) y = x;

'Visual Basic
If x = 4 Then y = x
```



Jeśli wykorzystujesz w swoim kodzie instrukcję warunkową `if`, w której sprawdzasz więcej niż jeden warunek, dla jasności upewnij się, że każdy z tych warunków znajduje się w osobnej parze nawiasów okrągłych:

```
if (x = 7) and (y = 8) then
```

Unikaj jednocześnie konstrukcji podobnych do tej z poniższego przykładu (w tym przypadku kompilator zasygnalizuje błąd):

```
if x = 7 and y = 8 then
```

W języku programowania Delphi stosuje się słowa kluczowe `begin` i `end` niemal tak samo jak w języku C# wykorzystywane są nawiasy klamrowe { i }. Przykładowo, poniższą konstrukcję możemy zastosować, jeśli chcemy wykonać więcej niż jeden wiersz kodu w przypadku spełnienia warunku instrukcji `if` (w tym przypadku `x = 6`):

```
if x = 6 then begin
  DoSomething;
  DoSomethingElse;
  DoAnotherThing;
end;
```

Możemy także łączyć wiele wzajemnie sprzecznych warunków za pomocą konstrukcji `if-else`:

```
if x = 100 then
  SomeFunction;
else if x = 200 then
  SomeOtherFunction;
else begin
  SomethingElse;
  Entirely;
end;
```

Stosowanie instrukcji case

Instrukcja `case` w języku programowania Delphi ma niemal identyczne znaczenie jak instrukcja `switch` w języku C#. Instrukcja `case` jest wygodnym narzędziem wybierania jednego warunku spośród wielu możliwości bez konieczności stosowania rozbudowanych

konstrukcji `if-else if-else` itd. Poniżej przedstawiono przykład instrukcji `case` zbudowanej w języku Delphi:

```
case SomeIntegerVariable of
  101 : DoSomething;
  202 : begin
      DoSomething;
      DoSomethingElse;
    end;
  303 : DoAnotherThing;
  else DoTheDefault;
end;
```



Selektor instrukcji `case` musi być zmienną typu porządkowego. Stosowanie w roli selektora zmiennej typów nieporządkowych, np. łańcuchów, jest traktowane w języku Delphi jak błąd. Pozostałe języki programowania dla platformy .NET, w tym język C#, zezwalają na wykorzystywanie łańcuchów w roli selektorów.

Poniżej przedstawiono instrukcję `switch` z języka C#, która jest równoważna zaprezentowanej powyżej instrukcji `case`:

```
switch (SomeIntegerVariable)
{
  case 101: DoSomething(); break;
  case 202: DoSomething();
            DoSomethingElse(); break;
  case 303: DoAnotherThing(); break;
  default: DoTheDefault();
}
```

Pętle

Pętla jest konstrukcją umożliwiającą wielokrotne wykonywanie określonych działań. Konstrukcje pętli dostępnych w języku programowania Delphi są bardzo podobne do swoich odpowiedników, które powinieneś doskonale znać z innych języków, zatem poświęcanie w tym podrozdziale zbyt wiele miejsca na wprowadzanie podstaw pętli nie ma większego sensu. Poniżej krótko omówimy różne konstrukcje pętli wykorzystywane w języku Delphi.

Pętla for

Zastosowanie pętli `for` jest idealnym rozwiązaniem w sytuacji, gdy musimy powtórzyć jakieś działanie taką liczbę razy, którą można określić z góry. Poniżej przedstawiono przykładowy fragment kodu z pętlą `for`, która dziesięć razy dodaje do zmiennej `X` swój indeks (w praktyce stosowanie tego kodu oczywiście nie miałoby sensu):

```
var
  I, X: Integer;
begin
  X := 0;
  for I := 1 to 10 do
    Inc(X, I);
end.
```

Poniżej przedstawiono odpowiednik tego programu zbudowany w języku programowania C#:

```
void main()
{
  int x = 0;
  for (int i=1; i<=10; i++)
    x += i;
}
```

Poniższy przykład przedstawia tę samą pętlę, tyle że napisaną w języku programowania Visual Basic .NET:

```
Dim X As Integer
for I = 1 to 10
  X = X + I
Next I
```

Pętla while

Konstrukcję pętli `while` stosujemy w sytuacjach, gdzie jakaś część naszego kodu musi być wykonana wielokrotnie, o ile spełniony jest jakiś warunek. Warunki pętli `while` są testowane jeszcze przed wykonaniem jej pierwszej iteracji — klasycznym przykładem zastosowania pętli `while` jest wielokrotne przeprowadzanie tych działań na otwartym pliku aż do osiągnięcia jego końca. Poniżej przedstawiono przykład pętli `while`, w której odczytujemy kolejne wiersze (po jednym w każdej iteracji) z pliku tekstowego i wyświetlamy je na ekranie:

```
program FileIt;

{$APPTYPE CONSOLE}

var
  f: TextFile;  // plik tekstowy
  s: string;
begin
  AssignFile(f, 'foo.txt');
  Reset(f);
  try
    while not EOF(f) do
    begin
      readln(f, S);
      writeln(S);
    end;
  finally
    CloseFile(f);
  end;
end.
```

Stosowana w języku Delphi pętla `while` działa tak samo jak odpowiadające jej konstrukcje dostępne w językach programowania C# (także pętla `while`) oraz Visual Basic .NET (pętla `Do While`).

Pętla repeat-until

Pętla `repeat-until` jest wykorzystywana do rozwiązywania problemów należących do tej samej klasy co problemy rozwiązywane za pomocą pętli `while` — obie konstrukcje różnią się jednak podejściem do problemu. Pętla `repeat-until` powtarza wykonywanie danego bloku kodu tylko do momentu, w którym pewien warunek stanie się prawdziwy. Inaczej niż w przypadku pętli `while` blok kodu zawarty w pętli `repeat-until` jest zawsze wykonywany przynajmniej raz, ponieważ warunek pętli jest sprawdzany dopiero na jej końcu. Stosowana w języku programowania Delphi pętla `repeat-until` jest w ogólności równoważna znanej z języka C# konstrukcji `do-while` z tą różnicą, że odwrócony jest warunek przerwania wykonywania pętli.

Przykładowo, w poniższym fragmencie kodu instrukcja zwiększająca licznik `x` o jeden jest powtarzana w pętli `repeat-until` do momentu, w którym wartość tego licznika przekroczy `100`:

```
var
  x: Integer;
begin
  x := 1;
  repeat
    inc(x);
  until x > 100;
end.
```

Instrukcja Break

Wywołanie instrukcji `Break` z wnętrza pętli `while`, `for` lub `repeat-until` powoduje natychmiastowe przejście na koniec aktualnie wykonywanej pętli. Ta metoda opuszczania bloku kodu wewnątrz pętli jest szczególnie przydatna w sytuacjach, w których z uwagi na pewne okoliczności wykryte wewnątrz pętli musimy przerwać jej wykonywanie. Dostępna w języku Delphi instrukcja `Break` jest odpowiednikiem znanej z języka programowania C# instrukcji `break` oraz stosowanej w języku Visual Basic .NET instrukcji `Exit`. W poniższej pętli wykorzystano instrukcję `Break` do przerwania wykonywania pętli już po pięciu iteracjach:

```
var
  i: Integer;
begin
  for i := 1 to 1000000 do
  begin
    MessageBeep(0);          // powoduje wydanie sygnału dźwiękowego
    if i = 5 then Break;
  end;
end;
```

Instrukcja Continue

Instrukcję `Continue` wywołujemy wewnątrz pętli w sytuacji, gdy chcemy pominąć następujący po niej blok kodu i przekazać sterowanie do kolejnej iteracji pętli. Zwróć uwagę na sposób wykorzystania tej instrukcji w poniższym przykładzie kodu — część

kodu występująca po instrukcji `Continue` zostanie pominięta w pierwszej iteracji użytej pętli `for`:

```
var
  i: Integer;
begin
  for i := 1 to 3 do
  begin
    writeln(i, ', Przed funkcją continue');
    if i = 1 then continue;
    writeln(i, ', Po funkcji continue');
  end;
end;
```

Procedury i funkcje

Jako programista z pewnym doświadczeniem w pracy z językami programowania powinieneś już teraz znać podstawy stosowania procedur i funkcji. Procedura jest wyodrębnioną częścią programu, która po swoim wywołaniu wykonuje określone zadanie i zwraca sterowanie do części kodu, w której została wywołana. Funkcja działa w ten sam sposób z jedną różnicą — po zakończeniu działania wraz ze sterowaniem przekazywanym do wywołującej części programu funkcja zwraca także swoją wartość (patrz listing 5.1).

Listing 5.1. *Przykładowy program wykorzystujący funkcję i procedurę*

```
1:  program FuncProc;
2:
3:  {$APPTYPE CONSOLE}
4:
5:  procedure BiggerThanTen(I: Integer);
6:  { Wypisuje na ekranie komunikat, jeśli I jest większe od 10. }
7:  begin
8:    if I > 10 then
9:      writeln('Tchórzliwy.');
10: end;
11:
12: function IsPositive(I: Integer): Boolean;
13: { Zwraca wartość True, jeśli I jest równe 0 lub jest liczbą dodatnią,
    albo wartość False, jeśli I jest liczbą ujemną }
14: begin
15:   Result := I >= 0;
16: end;
17:
18: var
19:   Num: Integer;
20: begin
21:   Num := 23;
22:   BiggerThanTen(Num);
23:   if IsPositive(Num) then
24:     writeln(Num, ' Jest pozytywne.');
25:   else
26:     writeln(Num, ' Jest negatywne.');
27: end.
```

Zmienna Result

Wykorzystana w funkcji `IsPositive()` zmienna lokalna `Result` wymaga krótkiego omówienia. Każda funkcja języka programowania Delphi zawiera deklarowaną niejawnie zmienną lokalną nazwaną `Result`, która przechowuje wartość zwracaną przez tę funkcję. Zauważ, że w przeciwieństwie do znanej z języka C# instrukcji `return` przypisanie wartości do zmiennej `Result` nie jest równoważne z zakończeniem działania danej funkcji.

Jeśli chcesz otrzymać w języku Delphi podobną funkcjonalność, jaką w języku C# daje instrukcja `return`, możesz bezpośrednio po przypisaniu wartości do zmiennej `Result` wywołać instrukcję `Exit`, która powoduje natychmiastowe opuszczenie bieżącej funkcji lub procedury.

Alternatywnym sposobem zwracania wartości funkcji jest użycie wewnątrz jej kodu operacji przypisania wartości do nazwy funkcji. Jest to standardowa składnia języka programowania Delphi, która pochodzi jeszcze ze starszych wersji języka Borland Delphi. Jeśli zdecydujesz się użyć nazwy funkcji w jej ciele, musisz pamiętać o zasadniczej różnicy pomiędzy jej umieszczeniem po lewej stronie operatora przypisania, a dowolnym innym miejscem w tworzonym kodzie. Nazwa funkcji wykorzystana w roli lewego operandu operatora przypisania oznacza, że przypisujemy funkcji zwracaną później wartość. Wykorzystanie tej nazwy w dowolnym innym miejscu kodu będzie oznaczało rekurencyjne wywołanie danej funkcji!

Pamiętaj także, że wykorzystywanie niejawnie deklarowanej zmiennej `Result` nie będzie akceptowane przez kompilator, jeśli na zakładce *Compiler* okna dialogowego ustawień projektu (wywoływanego przez wybór opcji *Project/Options*) wyłączymy opcję *Extended Syntax* lub jeśli w naszym kodzie użyjemy dyrektywy `{$EXTENDEDSYNTAX OFF}` (bądź dyrektywy `{$X-}`).

Przekazywanie parametrów

Język programowania Delphi umożliwia nam przekazywanie parametrów funkcji i procedur przez wartości lub przez referencje. Przekazywane przez nas parametry mogą być zmiennymi typu prostego, zmiennymi typu użytkownika lub tablicami otwartymi (tablice otwarte omówimy w dalszej części tego rozdziału). Jeśli dana funkcja lub procedura nie modyfikuje otrzymywanych parametrów, możemy je przekazać w formie wartości stałych.

Parametry wartościowe

Stosowanie parametrów wartościowych jest w języku Delphi domyślnym trybem przekazywania parametrów do funkcji i procedur. Kiedy przekazujemy jakiś parametr przez wartość, automatycznie tworzona jest lokalna kopia tej zmiennej, na której możemy potem operować z poziomu kodu funkcji lub procedury. Przeanalizujmy teraz poniższy przykład:

```
procedure Foo(I: Integer);
```

Kiedy wywołamy tak zadeklarowaną procedurę, zostanie utworzona lokalna kopia przekazanej zmiennej całkowitoliczbowej `I` — właśnie na tej kopii będzie operował kod procedury `Foo()`. Oznacza to, że możemy z poziomu tej procedury modyfikować wartość lokalnej kopii zmiennej `I`, nie zmieniając przy tym oryginalnej zmiennej przekazanej do procedury `Foo()`.

Parametry referencyjne

Język Delphi umożliwia przekazywanie do funkcji i procedur zmiennych przez ich referencje; parametry przekazywane w ten sposób są także nazywane parametrami zmiennymi. Przekazywanie przez referencje oznacza, że funkcja lub procedura otrzymująca daną zmienną może zmodyfikować jej oryginalną wartość. Przekazywanie zmiennych przez referencje wymaga użycia słowa kluczowego `var` na liście parametrów w instrukcji wywołania procedury lub funkcji:

```
procedure ChangeMe(var x: longint);
begin
  x := 2;  { zmienna x procedury wywołującej została zmieniona }
end;
```

Zamiast tworzyć kopię zmiennej `x`, wskutek użycia słowa kluczowego `var` do procedury `ChangeMe()` jest przekazywana kopia adresu tego parametru, dzięki czemu możemy w kodzie tej procedury bezpośrednio modyfikować tę zmienną.

Technika polegająca na stosowaniu w języku Delphi słowa kluczowego `var` dla parametrów przekazywanych przez referencje ma swoje odpowiedniki w języku C# (słowo kluczowe `ref`) oraz języku Visual Basic .NET (dyrektywa `ByRef`).

Parametry wyjściowe

Podobnie jak parametry zmienne (deklarowane ze słowem kluczowym `var`), parametry wyjściowe (deklarowane ze słowem kluczowym `out`) są jedną z metod zwracania z funkcji i procedur wartości do kodu wywołującego za pośrednictwem parametrów. Różnica polega jednak na tym, że parametry zmienne muszą być inicjalizowane poprawną wartością jeszcze przed wywołaniem funkcji lub procedury — takie wymaganie nie dotyczy parametrów wyjściowych, poprawność przekazywanej wartości w ogóle nie jest weryfikowana. W przypadku typów referencyjnych oznacza to, że w momencie wywołania procedury lub funkcji taka referencja jest usuwana.

```
procedure ReturnMe(out O: TObject);
begin
  O := SomeObject.Create;
end;
```

Oto próba przedstawienia reguły stosowania tych parametrów w największym skrócie: słowo kluczowe `var` stosuje się dla parametrów wejścia-wyjścia, natomiast słowo kluczowe `out` stosuje się wyłącznie dla parametrów wyjściowych.

Parametry stałe

Jeśli nie chcesz, aby wartość przekazywana do funkcji lub procedury była zmieniana, możesz ją zadeklarować ze słowem kluczowym `const`. Poniżej przedstawiono przykładową deklarację procedury, która otrzymuje w wywołaniu stały parametr łańcuchowy:

```
procedure Goon(const s: string);
```

Otwarte parametry tablicowe

Otwarte parametry tablicowe dają nam możliwość przekazywania do funkcji i procedur zmieniających się liczb argumentów. Możemy deklarować albo otwarte tablice wybranego jednolitego typu, albo stałe tablice różnych typów. Przykładowo, za pomocą poniższego kodu deklarujemy funkcję przyjmującą otwartą tablicę liczb całkowitych:

```
function AddEmUp(A: array of Integer): Integer;
```

Do funkcji i procedur przyjmujących otwarte parametry tablicowe możemy przekazywać zmienne, stałe oraz wyrażenia reprezentujące dowolne typy tablic (dynamiczne, statyczne lub otwarte). Poniższy kod demonstruje wywołanie zadeklarowanej przed chwilą funkcji `AddEmUp()` z przekazaniem otwartego parametru tablicowego w postaci zestawu czterech różnych elementów (odpowiednio zmiennej, wyrażenia i dwóch stałych):

```
var
  I, Rez: Integer;
const
  J = 23;
begin
  I := 8;
  Rez := AddEmUp([I, I + 50, J, 89]);
```

Do funkcji lub procedury przyjmującej otwarty parametr tablicowy możemy także bezpośrednio przekazywać istniejącą zmienną (lub stałą) tablicową:

```
var
  A: array of integer;
begin
  SetLength(A, 10);
  for i := Low(A) to High(A) do
    A[i] := i;
  Rez := AddEmUpConst(A);
```

Podczas przetwarzania otwartych tablic wewnątrz danej funkcji lub procedury możemy wykorzystywać funkcje `High()`, `Low()` i `Length()`, za pomocą których bez trudu można uzyskać podstawowe informacje na temat przekazanej tablicy. Ilustruje to poniższy fragment kodu, w którym zaimplementowano funkcję `AddEmUp()` zwracającą sumę wszystkich liczb całkowitych należących do przekazanej tablicy `A`:

```
function AddEmUp(A: array of Integer): Integer;
var
  i: Integer;
begin
  Result := 0;
  for i := Low(A) to High(A) do
    inc(Result, A[i]);
end;
```

Język programowania Delphi obsługuje także tablice stałych elementów (`array of const`), które umożliwiają przekazywanie do funkcji i procedur tablic z heterogenicznymi typami danych. Składnia definiowania funkcji i procedur przyjmujących takie tablice ma następującą postać:

```
procedure WhatHaveIGot(A: array of const);
```

Możemy wywołać powyższą procedurę np. za pomocą następującej instrukcji:

```
WhatHaveIGot(['Tabasko', 90, 5.6, 3.14159, True, 's']);
```

Kompilator przekazuje każdy z elementów tego parametru tablicowego w postaci obiektu klasy `System.Object`, dzięki czemu można te elementy stosunkowo łatwo przetwarzać w docelowej funkcji lub procedurze. Przykładem pracy z tablicami stałych elementów jest poniższa implementacja procedury `WhatHaveIGot()`, w której zastosowano pętlę `for` iteracyjnie przetwarzającą kolejne elementy tej tablicy i wyświetlającą na ekranie komunikaty wskazujące na typy danych przekazanych na poszczególnych pozycjach tabeli:

```
procedure WhatHaveIGot(A: array of const);
var
  i: Integer;
begin
  for i := Low(A) to High(A) do
    WriteLn('Indeks ', I, ': ', A[i].GetType.FullName);
  ReadLn;
end;
```

Zakres

Zakres (zasięg) jest pewną częścią Twojego programu, w której dana funkcja lub zmienna jest „widoczna" dla kompilatora. Przykładowo, stała globalna mieści się w zakresie (jest zasięgu) wszystkich punktów programu, natomiast zdefiniowana w danej procedurze zmienna lokalna mieści się tylko w zakresie tej procedury. Przeanalizuj poniższy listing 5.2.

Listing 5.2. *Przykładowy program wykorzystujący funkcję i procedurę*

```
1:  program Foo;
2:
3:  {$APPTYPE CONSOLE}
4:
5:  const
6:    SomeConstant = 100;
7:
8:  var
9:    SomeGlobal: Integer;
10:   D: Double;
11:
12: procedure SomeProc;
13: var
14:   D, LocalD: Double;
15: begin
16:   LocalD := 10.0;
17:   D := D - LocalD;
18: end;
19:
20: begin
21:   SomeGlobal := SomeConstant;
22:   D := 4.593;
23:   SomeProc;
24: end.
```

Stała `SomeConstant` oraz zmienne `SomeGlobal` i `D` mają zasięg globalny — ich wartości są „znane" kompilatorowi we wszystkich punktach powyższego programu `Foo`. Procedura `SomeProc()` definiuje dwie zmienne (`D` i `LocalD`), których zasięg ogranicza się tylko do tej procedury. Jeśli spróbujesz uzyskać dostęp do zmiennej `LocalD` poza kodem procedury `SomeProc()`, kompilator wyświetli błąd nieznanego identyfikatora. Jeśli natomiast wykorzystasz zmienną `D` wewnątrz procedury `SomeProc()`, będziesz się odwoływał do lokalnej wersji (nie do zmiennej `D` zdefiniowanej globalnie); jeśli jednak użyjesz zmiennej `D` poza tą procedurą, będziesz się odwoływał do wersji globalnej (jedynej, jaka jest „widoczna" dla kompilatora procedurą `SomeProc()`).

Moduły i przestrzenie nazw

Moduły są wyodrębnionymi częściami kodu źródłowego, które składają się na program napisany w języku Delphi. Moduł jest dobrym miejscem do grupowania funkcji i procedur, które będą później wywoływane z poziomu kodu źródłowego głównego programu. Każdy moduł musi się składać co najmniej z trzech części:

- Instrukcji `unit` (nagłówka) — każdy moduł musi w swoim pierwszym wierszu (bez poprzedzających komentarzy, spacji czy pustych wierszy) zawierać instrukcję określającą, że dany plik jest modułem, i jednocześnie definiującą jego nazwę (jednak bez rozszerzenia pliku). Przykładowo, w module `FooBar.pas` taki nagłówek miałby postać:

  ```
  unit FooBar;
  ```

- Części `interface` (interfejsu) — zaraz po instrukcji `unit` kolejny funkcjonalny wiersz kodu powinien zawierać instrukcję `interface`. Wszystkie elementy znajdujące się po tej instrukcji, ale przed instrukcją `implementation`, powinny być deklaracjami tych typów, stałych, zmiennych, procedur i funkcji, które chcemy udostępnić naszemu głównemu programowi i innym modułom. Pamiętaj, że w części interfejsu mogą się znajdować wyłącznie deklaracje — nigdy ciała udostępnianych funkcji i procedur. Instrukcja `interface` powinna mieć postać pojedynczego słowa w osobnym wierszu:

  ```
  interface
  ```

- Części `implementation` (implementacji) — część implementacji następuje w kodzie modułu po części interfejsu. Chociaż w części `implementation` umieszcza się przede wszystkim procedury i funkcje danego modułu, można w tej części także deklarować dowolne typy, stałe i zmienne, których nie chcemy udostępniać poza tym modułem. Właśnie w części implementacji powinniśmy zdefiniować wszystkie funkcje i procedury, które zadeklarowaliśmy wcześniej w części interfejsu. Instrukcja `implementation` powinna mieć postać pojedynczego słowa w osobnym wierszu:

  ```
  implementation
  ```

  Moduł może także zawierać dwie części opcjonalne:

- Część `initialization` (inicjalizacji) — ta część modułu umieszczana blisko końca jego pliku zawiera kod inicjalizacji zdefiniowany dla danego modułu. Kod należący do tej części jest wykonywany tylko raz, przed rozpoczęciem wykonywania kodu głównego programu.

- Część `finalization` (finalizacji) — ta część modułu umieszczana pomiędzy częścią inicjalizacji a końcem modułu (oznaczonym za pomocą słowa `end.`) zawiera zdefiniowany dla danego modułu kod finalizacji (nazywany często kodem „czyszczącym"), który jest wykonywany w momencie kończenia pracy głównego programu. Część finalizacji została wprowadzona w wersji 2.0 języka programowania Delphi. W wersji 1.0 do finalizowania modułów można było wykorzystać specjalne procedury wyjścia dodawane za pomocą funkcji `AddExitProc()`. Jeśli dostosowujesz aplikację napisaną w języku Delphi 1.0 do jego nowszej wersji, powinieneś przenieść kod zawarty w procedurach wyjścia do części finalizacji swoich modułów.

Moduły definiują także przestrzenie nazw. Przestrzeń nazw umożliwia organizowanie aplikacji lub biblioteki w logiczną i hierarchiczną strukturę. Do tworzenia zagnieżdżonych przestrzeni nazw można wykorzystać notację z kropką, która zapobiega ewentualnym konfliktom identycznych identyfikatorów. Bardzo często spotyka się nazwy zagnieżdżone na trzech poziomach: `firma.produkt.dziedzina`; przykładowo: `Borland.Delphi.System` lub `Borland.Vcl.Controls`.



Jeśli więcej niż jeden użyty (dołączony) moduł zawiera kod zdefiniowany w części inicjalizacji i (lub) finalizacji, odpowiednie bloki kodu są wykonywane w kolejności napotykania dołączanych modułów przez kompilator (najpierw wykonywany jest kod pierwszego modułu klauzuli `uses` głównego programu, potem pierwszy moduł uwzględniony w klauzuli `uses` tego modułu itd.). Pamiętaj także, że nie należy w modułach tworzyć takiego kodu inicjalizującego i (lub) finalizującego, którego działanie jest uzależnione od tej kolejności, ponieważ w takim przypadku stosunkowo niewielka zmiana w klauzuli `uses` może być źródłem trudnego do zlokalizowania błędu.

Klauzula uses

W klauzuli `uses` umieszczamy listę przestrzeni nazw, które chcemy dołączyć do tworzonego programu lub modułu. Przykładowo, jeśli chcemy zbudować program nazwany `FooProg`, który będzie wykorzystywał funkcje i typy należące do dwóch różnych przestrzeni nazw (`UnitA` i `UnitB`), powinniśmy zastosować instrukcję `program` i klauzulę `uses` w sposób następujący:

```
program FooProg;

uses UnitA, UnitB;
```

Moduły mogą zawierać dwie klauzule `uses` — jedną w części interfejsu i jedną w części implementacji.

Poniżej przedstawiono przykład prostego modułu z dwiema klauzulami `uses`:

```
unit FooBar;

interface

uses BarFoo;
  { tutaj powinny się znaleźć deklaracje publiczne }

implementation
```

```
uses BarFly;
  { tutaj powinny się znaleźć deklaracje prywatne }
  { definicje funkcji i procedur zadeklarowanych w części interface }

initialization
  { tutaj powinien się znaleźć kod inicjalizacji }
finalization
  { tutaj powinien się znaleźć tzw. kod czyszczący }
end.
```



W klauzuli `uses` możemy wykorzystywać pełne kwalifikatory dołączanych przestrzeni nazw lub — co jest dopuszczalne w języku Delphi — tylko najbardziej wewnętrzne identyfikatory przestrzeni nazw. Ta druga możliwość wynika z troski autorów tego języka programowania o zgodność z jego wcześniejszymi wersjami (przykładem jest przestrzeń nazw *Controls*). Odpowiednie ustawienia dotyczące prefiksów przestrzeni nazw są dostępne we właściwym oknie dialogowym (wyświetlanym przez wybór opcji *Tools/Options/Delphi Options/Library*).

Cykliczne odwołania do modułów

Możesz się spotkać z sytuacją, w której moduł `UnitA` odwołuje się w klauzuli `uses` do modułu `UnitB`, który z kolei w swojej klauzuli `uses` odwołuje się do modułu `UnitA`. Nazywamy to cyklicznymi odwołaniami do modułów. Wystąpienia tego typu sytuacji z reguły wskazują na błędy popełnione w fazie projektowania aplikacji. Powinieneś unikać tworzenia tego typu struktur w swoich programach. Optymalnym rozwiązaniem tego problemu jest w większości przypadków przeniesienie tej części danych, do której odwołują się oba moduły, do nowego, trzeciego modułu. Okazuje się jednak, że podobnie jak wielu innych niewygodnych konstrukcji, także cyklicznych odwołań do modułów nie zawsze można uniknąć w tak prosty sposób. Pamiętaj, że takie odwołania są niepoprawne, jeśli w obu modułach znajdują się w części implementacji lub w obu modułach należą do części interfejsu. Wobec tego w wielu przypadkach najlepszym rozwiązaniem jest przeniesienie odpowiedniej klauzuli `uses` w jednym module do części implementacji i pozostawienie cyklicznej klauzuli `uses` drugiego modułu w dotychczasowej lokalizacji (lub odwrotnie). Zazwyczaj w ten sposób można rozwiązać problem cyklicznych odwołań do modułów[3].

Pakiety i podzespoły

Pakiety Delphi umożliwiają nam umieszczanie wybranych części naszej aplikacji w wyodrębnionych modułach, które — w postaci tzw. podzespołów .NET — mogą być następnie współdzielone przez wiele aplikacji opracowanych dla tej platformy.

Pakiety i podzespoły .NET omówimy bardziej szczegółowo w rozdziale 6., „Podzespoły .NET, biblioteki i pakiety".

[3] Więcej informacji na temat cyklicznego odwołania do modułów znajdzie Czytelnik w rozdziale 4., „Programy, moduły i przestrzenie nazw" — *przyp. red.*

Programowanie obiektowe

Na temat koncepcji programowania obiektowego, nazywanego także programowaniem zorientowanym obiektowo (ang. *Object-Oriented Programming — OOP*), napisano już mnóstwo książek. Programowanie obiektowe jest często traktowane bardziej jak religia niż jedna z metodologii projektowania — wielu programistów stara się w sposób sztuczny wymyślać i prezentować niezliczone zalety tej techniki, nie dopuszczając do siebie żadnych argumentów jej przeciwników. Nie jesteśmy ortodoksyjnymi zwolennikami programowania obiektowego i nie mamy zamiaru prezentować w tej książce faktycznych bądź wyimaginowanych zalet tej metodologii — chcemy jedynie uczciwie przedstawić najważniejsze podstawy, na których opiera się język programowania Delphi.

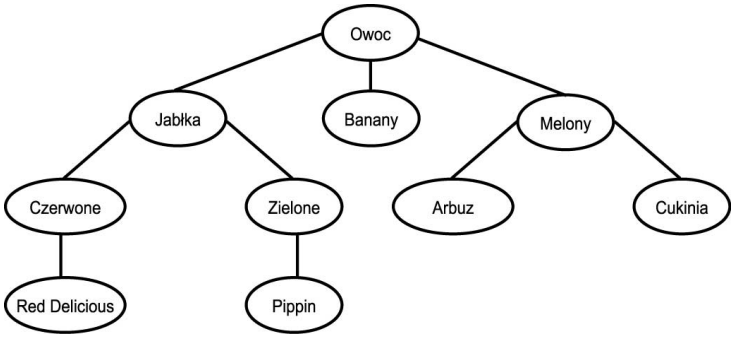
Programowanie obiektowe jest metodologią przewidującą stosowanie wyodrębnionych obiektów — zawierających zarówno dane, jak i kod — w roli bloków wykorzystywanych podczas budowy aplikacji. Chociaż metodologia OOP niekoniecznie musi prowadzić do uproszczenia procesu tworzenia kodu, efektem jej stosowania w praktyce tradycyjnie jest otrzymywanie kodu łatwiejszego do konserwacji. Połączenie danych i kodu obiektów upraszcza proces wyszukiwania błędów w aplikacjach, naprawiania tych błędów i ograniczania do minimum ich wpływu na pozostałe obiekty — a więc przyczynia się do udoskonalenia tworzonych aplikacji. Tradycyjnie każdy język obiektowy zawiera implementację przynajmniej trzech zasadniczych właściwości koncepcji programowania obiektowego:

- *Hermetyzacja*[4] — jest związana z właściwym łączeniem uzależnionych od siebie pól danych i jednoczesnym ukrywaniem odpowiednich szczegółów implementacyjnych. Do korzyści płynących z zapewniania hermetyczności należy możliwość dzielenia programów na odseparowane moduły oraz łatwość izolowania od siebie poszczególnych części kodu.
- *Dziedziczenie* — wiąże się z możliwością tworzenia nowych obiektów w oparciu o właściwości i funkcjonalność ich obiektów macierzystych. Dziedziczenie umożliwia nam konstruowanie hierarchii obiektów podobnej do struktury biblioteki VCL, w której mamy jeden najbardziej ogólny obiekt oraz jego obiekty potomne, które na każdym kolejnym poziomie są coraz bardziej szczegółowe.

  Zaletą dziedziczenia jest możliwość współdzielenia tego samego kodu. Na rysunku 5.1 przedstawiono przykład dziedziczenia, gdzie jeden obiekt bazowy (*Owoc*) jest przodkiem wszystkich obiektów reprezentujących owoce, w tym obiektu *Melon*. Obiekt *Melon* jest w przodkiem wszystkich obiektów reprezentujących „melonowate”, w tym dla obiektu *Arbuz*. Przeanalizuj teraz ten rysunek.
- *Polimorfizm* — polimorfizm dosłownie oznacza „wielopostaciowość”. Wywołania wirtualnych metod zmiennej obiektowej powodują wywołania kodu właściwego dla rzeczywistego egzemplarza tego obiektu, który aktualnie (w czasie wykonywania programu) jest reprezentowany przez tę zmienną.

[4] Możesz się również spotkać z określeniem enkapsulacja — *przyp. red.*

Rysunek 5.1.
Przykładowa struktura dziedziczenia



Zwróć uwagę na brak możliwości stosowania znanego z języka C++ mechanizmu wielokrotnego (wielobazowego) dziedziczenia w środowisku uruchomieniowym .NET CLR — takiej możliwości nie daje także język programowania Delphi for .NET.

Zanim przejdziemy do omawiania podstawowych pojęć związanych z metodologią programowania obiektowego, musimy się upewnić, że dobrze rozumiesz następujące terminy nieodłącznie związane z tą metodologią:

- *Pole* — pola są zmiennymi reprezentującymi dane przechowywane wewnątrz obiektów. Pola w obiektach pełnią identyczną rolę jak pola stosowane w rekordach definiowanych w języku Delphi. W języku C# pola są niekiedy nazywane danymi składowymi lub atrybutami.
- *Metoda* — jest to nazwa używana dla procedur i funkcji należących do obiektów. W języku C# metody są czasami nazywane funkcjami składowymi.
- *Właściwość* — jest to konstrukcja pełniąca rolę akcesora do danych i kodu zawartego w danym obiekcie, stanowi więc połączenie koncepcji pola i metody. Właściwości przypominają pola, ponieważ określają taki sposób dostępu do pól i metod, który ukrywa przed użytkownikiem szczegóły implementacji danego obiektu.



W ogólności uważa się, że bezpośrednie udostępnianie pól obiektu jest złym stylem programowania obiektowego, ponieważ utrudnia przyszłe zmiany szczegółów implementacyjnych obiektu. Zamiast bezpośrednich odwołań do pól obiektu powinieneś używać odpowiednich właściwości dostępowych, które z jednej strony stworzą standardowy interfejs obiektu, a z drugiej nie będą zmuszały użytkownika do zagłębiania się w strukturę jego implementacji. Właściwości zostaną omówione w punkcie „Właściwości” w dalszej części podrozdziału.

Stosowanie obiektów Delphi

Jak już wspomnieliśmy, klasy są strukturami, które mogą zawierać zarówno dane, jak i kod (metody). Obiekty są tworzonymi w czasie wykonywania programu egzemplarzami tych klas. Klasy w języku Delphi dają nam wszystkie możliwości wynikające z podstawowych cech koncepcji programowania obiektowego, a więc dziedziczenie, hermetyczność i polimorfizm.

Deklaracja i tworzenie egzemplarza

Zanim będziemy mogli stosować obiekty w naszym kodzie, oczywiście musimy go najpierw zdefiniować za pomocą słowa kluczowego `class`. Wspominaliśmy już w tym rozdziale, że klasy są deklarowane w części `type` modułu lub programu:

```
type
  TFooObject = class;
```

Poza samą deklaracją klasy zazwyczaj będziemy dodatkowo wykorzystywali zmienną tego typu (egzemplarz tej klasy) zadeklarowaną w części `var`:

```
var
  FooObject: TFooObject;
```

W języku programowania Delphi egzemplarze obiektu tworzy się przez wywołanie jednego z jej konstruktorów. Konstruktor odpowiada nie tylko za stworzenie egzemplarza naszego obiektu, ale także za przydzielenie mu odpowiedniej ilości pamięci lub inicjalizację wszystkich niezbędnych pól, dzięki którym obiekt będzie gotowy do użycia po opuszczeniu konstruktora. Obiekty w języku Delphi muszą zawierać przynajmniej jeden konstruktor nazwany `Create()` — możemy jednak tworzyć dla poszczególnych obiektów więcej konstruktorów. W zależności od typu obiektu konstruktor `Create()` może pobierać różną liczbę parametrów wejściowych. W tym rozdziale skupimy się na najprostszym przypadku, w którym konstruktor nie pobiera żadnych parametrów.

Konstruktory obiektów w języku Delphi nie są wywoływane automatycznie — za ich wywoływanie zawsze odpowiada programista. Składnia takich wywołań jest następująca:

```
FooObject := TFooObject.Create;
```

Zwróć uwagę na unikalność składni stosowanej w wywołaniach konstruktorów. Odwołujemy się do konstruktora klasy (metody `Create()`) przez typ, a nie przez egzemplarz, jak w przypadku pozostałych metod zdefiniowanych w wykorzystywanych klasach. Takie rozwiązanie na pierwszy rzut oka może się wydawać nieco dziwne, jednak już po dokonaniu krótkiej analizy nie będziesz miał wątpliwości, że jest to technika sensowna. Zmienna `FooObject` jest w czasie tego wywołania niezdefiniowana, a typ `TFooObject` jest statyczną strukturą przechowywaną w pamięci. Statyczne wywołanie jej konstruktora `Create()` jest więc w pełni poprawne.

Proces wywołania konstruktora celem stworzenia egzemplarza obiektu jest często nazywany konkretyzowaniem obiektu lub po prostu tworzeniem egzemplarza (ang. *instantiation*).



Kiedy wykorzystujemy konstruktor do stworzenia egzemplarza obiektu, odpowiednie mechanizmy środowiska uruchomieniowego CLR upewniają się, że wszystkie pola naszego obiektu są inicjalizowane wartością zero. Możesz bez obaw zakładać, że wszystkie zmienne liczbowe będą miały wartość `0`, wszystkie obiekty będą równe `nil`, wszystkie zmienne logiczne przyjmą wartość `False` oraz że wszystkie łańcuchy będą puste.

Destrukcja

Wszystkie klasy wykorzystywane w aplikacjach dla platformy .NET Framework dziedziczą funkcję nazwaną `Finalize()`, która może zostać przykryta w kodzie klasy i wykonywać wszystkie operacje odzyskiwania zasobów, które programista uzna za konieczne. Metoda `Finalize()` jest wywoływana automatycznie dla każdego egzemplarza klasy przez stosowany w środowisku CLR mechanizm odzyskiwania pamięci. Pamiętaj jednak, że nigdy nie ma gwarancji co do tego, kiedy metoda `Finalize()` faktycznie zostanie wywołana, oraz czy — w niektórych okolicznościach — w ogóle zostanie wywołana. Z tych powodów nie zaleca się zwalniania krytycznych lub ograniczonych zasobów (takich jak ogromne bufory pamięci, połączenia z bazą danych lub uchwyty systemu operacyjnego) za pomocą metody `Finalize()`. Zamiast tego programiści Delphi powinni przykrywać destruktor `Destroy()` swoją wersją tej metody, która zwolni wszystkie cenne zasoby. Więcej informacji na ten temat znajdziesz w rozdziale 9., „Zarządzanie pamięcią i odśmiecanie".

Przodek wszystkich obiektów

Być może zadajesz sobie pytanie, jak to możliwe, że wszystkie te metody mieszczą się w Twoim niewielkim obiekcie. Z pewnością sam tych metod nie deklarowałeś w swoim kodzie źródłowym, prawda? Omawiane metody w rzeczywistości są dziedziczone po standardowej klasie `System.Object` udostępnianej wszystkim aplikacjom platformy .NET. Wykorzystywana w języku Delphi klasa `TObject` jest jedynie aliasem ogólnodostępnej klasy `System.Object`. W tworzonych w języku Delphi aplikacjach .NET wszystkie obiekty zawsze są bezpośrednimi lub pośrednimi potomkami klasy `TObject`, niezależnie od tego, czy jawnie zadeklarowałeś to dziedziczenie. Oznacza to, że deklaracja:

```
type
  TFoo = class
  end;
```

jest równoważna deklaracji:

```
type
  TFoo = class(TObject)
  end;
```

Pola

Podczas dodawania pól do klasy wykorzystuje się składnię bardzo podobną do tej, która jest stosowana dla deklaracji zmiennych w bloku `var`. Przykładowo, poniższy fragment kodu dodaje do klasy `TFoo` po jednym polu typu `Integer`, `string` i `Double`:

```
type
  TFoo = class(TObject)
    I: Integer;
    S: string;
    D: Double;
  end;
```

Język programowania Delphi obsługuje także pola statyczne czyli takie, które reprezentują dane współdzielone przez wszystkie egzemplarze danej klasy. Pola tego typu można dodawać do deklaracji klasy za pomocą jednego lub więcej bloków `class var`. Ilustruje to poniższy fragment kodu, w którym do klasy `TFoo` dodajemy trzy pola statyczne:

```
type
  TFoo = class(TObject)
    I: Integer;
    S: string;
    D: Double;
    class var
      I_Static: Integer;
      S_Static: string;
      D_Static: Double;
  end;
```

Warto pamiętać, że wewnątrz definicji klasy można umieścić (choć z punktu widzenia reguł syntaktycznych nie jest to konieczne) blok `var` definiujący normalne pola klasy.

Odpowiednikiem stosowanego w języku Delphi bloku `class var` jest znane z języka programowania C# słowo kluczowe `static`. Pamiętaj, że definiowane wewnątrz klas bloki `class var` i `var` kończą się na następujących elementach:

- innym bloku `class var` lub `var`,
- deklaracji własności,
- dowolnej deklaracji metody,
- specyfikatorze widoczności.

Metody

Metody są funkcjami i procedurami należącymi do danego obiektu — zadaniem metod jest zapewnienie obiektom możliwości działania, dzięki czemu nie są to struktury przeznaczone wyłącznie do reprezentowania danych. Specjalnymi typami metod są omówione przed chwilą konstruktory i destruktory. W naszych obiektach możemy jednak tworzyć własne metody, które będą odpowiadały za wykonywanie rozmaitych zadań.

Tworzenie metody jest procesem dwuetapowym. W pierwszej kolejności musimy zadeklarować nową metodę w deklaracji typu obiektu, a dopiero potem powinniśmy tę metodę zdefiniować we właściwej części kodu. Poniższy fragment kodu demonstruje proces deklarowania i definiowania przykładowej metody:

```
type
  TBoogieNights = class
    Dance: Boolean;
    procedure DoTheHustle;
  end;

procedure TBoogieNights.DoTheHustle;
begin
  Dance := True;
end;
```

Zauważ, że podczas definiowania ciała metody konieczne jest stosowanie jej pełnej nazwy — składającej się z nazw klasy i metody. Warto także zwrócić uwagę na bezpośrednią dostępność pola `Dance` z poziomu ciała definiowanej metody.

Typy metod

W klasach możemy deklarować następujące typy metod: normalne, statyczne (`static`), wirtualne (`virtual`), klasowe statyczne (`class static`), klasowe wirtualne (`class virtual`), dynamiczne (`dynamic`) oraz obsługujące komunikaty (`message`). Przeanalizujmy poniższy przykład deklaracji obiektu:

```
TFoo = class
  procedure IAmNormal;
  class procedure IAmAClassMethod;
  class procedure IAmAStatic; static;
  procedure IAmAVirtual; virtual;
  class procedure IAmAVirtualClassMethod; virtual;
  procedure IAmADynamic; dynamic;
  procedure IAmAMessage(var M: TMessage); message WM_SOMEMESSAGE;
end;
```

Metody normalne

`IAmNormal()` jest normalną metodą Delphi. Jest to domyślny typ metod w tym języku — metody takie jak `IAmNormal()` działają podobnie jak zwykłe procedury i funkcje. Kompilator „zna" adresy metod tego typu, dzięki czemu w momencie wywołania metody statycznej może statycznie dołączyć odpowiednie informacje do wykonywanego kodu. Metody statyczne są w związku z tym wykonywane najszybciej, jednak ze względu na brak możliwości ich przykrywania nasze konstrukcje oparte na tych metodach faktycznie tracą własność polimorfizmu.

Metody klasowe

`IAmAClassMethod()` jest specjalnym, specyficznym dla języka Delphi rodzajem metody statycznej. Metody klasowe mogą być wywoływane nawet bez uprzedniego stworzenia egzemplarza danej klasy, a w przypadku istnienia takich egzemplarzy implementacja tych metod jest współdzielona przez wszystkie te egzemplarze. Metody klasowe zawierają jednak specjalny i ukryty parametr `Self`, który jest przekazywany przez kompilator celem zapewnienia możliwości wywoływania polimorficznych (wirtualnych) metod klasowych. Dodatkowo metody klasowe mogą być wirtualne. Elementy nieklasowe lub niestatyczne są niedostępne z poziomu ciała metody klasowej.

Metody statyczne

`IAmAStatic()` jest prawdziwą, zgodną ze specyfikacją platformy .NET metodą statyczną. Podobnie jak pola statyczne implementacja metod tego typu jest współdzielona przez wszystkie egzemplarze danej klasy. Oznacza to, że elementy niestatyczne są niedostępne z poziomu ciała metody statycznej. Do metod statycznych nie jest przekazywany parametr `Self`, zatem metody niestatyczne nie mogą być wywoływane przez metody statyczne.

Metody wirtualne

`IAmAVirtual()` jest metodą wirtualną. Metody tego typu są wywoływane w taki sam sposób jak metody statyczne, jednak z uwagi na brak możliwości przykrywania metod wirtualnych, kompilator „nie zna" adresu konkretnej metody wirtualnej w momencie jej wywołania w kodzie programu. W związku z tym stosowany w środowisku .NET kompilator JIT buduje specjalną tablicę metod wirtualnych (ang. *Virtual Method Table — VMT*), która w czasie wykonywania aplikacji jest przeszukiwana pod kątem adresów wywoływanych funkcji wirtualnych. Przez tę tablicę muszą przejść wszystkie pojawiające się w czasie wykonywania programu wywołania metod wirtualnych. Tablica VMT dla obiektu zawiera nie tylko informacje na temat wszystkich deklarowanych w tym obiekcie funkcji wirtualnych, ale dane o wszystkich takich funkcjach deklarowanych w jego przodkach.

Metody dynamiczne

`IAmADynamic()` jest metodą dynamiczną. Inaczej niż kompilator Win32 (który zapewniał osobny mechanizm obsługi metod dynamicznych), kompilator .NET odwzorowuje metody dynamiczne w odpowiednie metody wirtualne.

Metody obsługujące komunikaty

`IAmAMessage()` jest metodą obsługującą komunikaty. Użyta dyrektywa `message` powoduje utworzenie metody, która może odpowiadać na dynamicznie pojawiające się komunikaty. Wartość zadeklarowana bezpośrednio za słowem `message` określa, na jakie komunikaty dana metoda będzie odpowiadała. W komponentach udostępnianych przez bibliotekę VCL metody obsługujące komunikaty są wykorzystywane do generowania automatycznych odpowiedzi na komunikaty systemu Windows, zatem w ogólności nie są bezpośrednio wywoływane przez programistów.

Przykrywanie metod

Przykrywanie metod jest oferowaną w języku Delphi implementacją koncepcji polimorfizmu — jednego z podstawowych elementów metodologii programowania obiektowego. Dzięki technice przykrywania metod możemy modyfikować działanie metod na poszczególnych poziomach dziedziczenia. Język programowania Delphi umożliwia przykrywanie tylko tych metod, które zostały wcześniej zadeklarowane jako wirtualne (`virtual`), dynamiczne (`dynamic`) lub jako metody obsługujące komunikaty (`message`). Aby przykryć metodę wirtualną lub dynamiczną, wystarczy w typie obiektu potomnego zamiast słowa dyrektywy `virtual` lub `dynamic` użyć dyrektywy `override`. Aby przykryć metodę obsługującą komunikaty, w klasie potomnej należy powtórzyć dyrektywę `message` wraz z tym samym identyfikatorem komunikatu, który użyto w klasie macierzystej (bazowej). Przykładowo, za pomocą poniższej deklaracji klasy potomnej `TFooChild` możemy przykryć metody `IAmAVirtual()`, `IAmADynamic()` i `IAmAMessage()` zadeklarowane wcześniej w klasie bazowej `TFoo`:

```
TFooChild = class(TFoo)
  procedure IAmAVirtual; override;
  procedure IAmADynamic; override;
  procedure IAmAMessage(var M: TMessage); message WM_SOMEMESSAGE;
end;
```

Dyrektywa `override` zastępuje w tablicy VMT wpis dotyczący oryginalnej metody informacjami o nowej metodzie. Gdybyśmy ponownie zadeklarowali metody `IAmAVirtual()` i `IAmADynamic()` ze słowami kluczowymi `virtual` lub `dynamic` zamiast dyrektywy `override`, stworzylibyśmy nowe metody, zamiast przykryć odpowiednie metody klasy bazowej `TFoo`. Taki zabieg zazwyczaj będzie powodował wygenerowanie ostrzeżenia kompilatora, chyba że wcześniej dodamy do deklaracji tych metod dyrektywę `reintroduce` (omówioną krótko w dalszej części tego podrozdziału). Należy także pamiętać, że próba przykrycia w typie potomnym standardowej metody spowoduje, że metoda statyczna w nowej klasie sprawi, że metoda ta nie będzie dostępna dla użytkowników klasy potomnej.

Przeciążanie metod

Podobnie jak zwykłe funkcje i procedury języka Delphi, także metody mogą być przeciążane, co oznacza, że pojedyncza klasa może zawierać wiele metod o tej samej nazwie i różnych listach parametrów. Przeciążane metody muszą być oznaczane dyrektywą `overload`, chociaż stosowanie tej dyrektywy dla pierwszego wystąpienia danej nazwy metody w hierarchii klas jest opcjonalne. Poniższy fragment kodu jest przykładem deklaracji klasy zawierającej trzy metody przeciążone:

```
type
  TSomeClass = class
    procedure AMethod(I: Integer); overload;
    procedure AMethod(S: string); overload;
    procedure AMethod(D: Double); overload;
  end;
```

Ponownie wprowadzanie nazw metod

Może się zdarzyć, że będziesz chciał w taki sposób dodać metodę do jednej ze swoich klas, aby zastąpić metodę wirtualną o takiej samej nazwie, którą zadeklarowano w wykorzystywanej klasie bazowej. W takim przypadku nie powinieneś przeciążać oryginalnej metody udostępnianej przez klasę bazową — lepszym rozwiązaniem jest jej całkowite ukrycie i zastąpienie nową metodą. Jeśli po prostu dodasz nową metodę i skompilujesz swój program, kompilator wygeneruje ostrzeżenie wyjaśniające, że nowa metoda ukrywa identycznie nazwaną metodę klasy bazowej. Aby przerwać generowanie tego typu ostrzeżeń, w deklaracji metody w klasie potomnej powinieneś użyć dyrektywy `reintroduce`. Poniższy fragment kodu przedstawia przykład prawidłowego wykorzystania tej dyrektywy:

```
type
  TSomeBase = class
    procedure Cooper; virtual;
  end;

  TSomeClass = class
    procedure Cooper; reintroduce;
  end;
```

Parametr Self

Niejawnie deklarowana zmienna nazwana `Self` jest dostępna wewnątrz wszystkich metod obiektów. `Self` jest referencją do tego egzemplarza klasy, za pośrednictwem którego wywołano daną metodę. Zmienna `Self` jest przekazywana przez kompilator do wszystkich metod w postaci ukrytego parametru. Odpowiednikiem zmiennej referencyjnej `Self` w języku C# jest zmienna `this`, natomiast w języku Visual Basic .NET jest to zmienna `Me`.

Referencje do klas

Chociaż zwykłe zmienne mogą przechowywać referencje do obiektów, w języku Delphi istnieje także pojęcie referencji do klasy, czyli referencji do zdefiniowanego typu obiektowego. Za pomocą referencji do klas możemy nie tylko wywoływać metody klasowe i statyczne, ale także tworzyć egzemplarze tych klas. Poniższy fragment kodu ilustruje składnię deklaracji klasy (nazwanej `SomeClass`) i referencji do nowego typu obiektowego:

```
type
  SomeClass = class
   constructor Create; virtual;
   class procedure Foo;
  end;
  SomeClassRef = class of SomeClass;
```

Możemy wykorzystać te typy do wywołania metody klasowej `SomeClass.Foo()` za pośrednictwem naszej nowej referencji do klasy `SomeClass` czyli `SomeClassRef` — oto przykład takiego zastosowania tej referencji:

```
var
  SCRef: SomeClassRef;
begin
  SCRef := SomeClass;
  SCRef.Foo;
```

Podobnie, w oparciu o zdefiniowaną referencję, możemy stworzyć egzemplarz klasy `SomeClass`:

```
var
  SCRef: SomeClassRef;
  SC: SomeClass;
begin
  SCRef := SomeClass;
  SC := SCRef.Create;
```

Zauważ, że tworzenie egzemplarzy za pośrednictwem referencji do klasy wymaga zdefiniowania w tej klasie przynajmniej jednego wirtualnego konstruktora. Takie konstruktory są unikalnymi strukturami stosowanymi wyłącznie w języku programowania Delphi — dzięki nim możemy tworzyć klasy przez referencje, a więc w sytuacji, gdy konkretny typ klasy nie jest znany w czasie kompilacji.

Właściwości

W zrozumieniu sensu istnienia i funkcjonowania właściwości może pomóc założenie, że są to specjalne pola dostępowe, które umożliwiają nam modyfikowanie danych i wykonywanie kodu zawartego w naszych klasach. W przypadku komponentów to właśnie właściwości są tymi elementami, które są wyświetlane w panelu *Object Inspector* po ich zaznaczeniu (np. na tworzonym formularzu). Poniższy przykład ilustruje uproszczony obiekt z jedną własnością:

```
TMyObject = class
private
  SomeValue: Integer;
  procedure SetSomeValue(AValue: Integer);
public
  property Value: Integer read SomeValue write SetSomeValue;
end;

procedure TMyObject.SetSomeValue(AValue: Integer);
begin
  if  SomeValue <> AValue then
    SomeValue := AValue;
end;
```

`TMyObject` jest obiektem zawierającym następujące składowe: jedno pole (liczba całkowita `SomeValue`), jedną metodę (procedura nazwana `SetSomeValue`) oraz jedną właściwość nazwaną `Value`. Zasadniczym zadaniem procedury `SetSomeValue` jest ustawienie wartości pola `SomeValue`. Własność `Value` w rzeczywistości nie zawiera żadnych danych — jest jedynie akcesorem zdefiniowanym dla pola `SomeValue`. Dopiero kiedy zadamy właściwości `Value` pytanie o reprezentowaną przez nią wartość, właściwość ta odczyta wartość ze zmiennej `SomeValue`. Kiedy podejmiemy próbę ustawienia wartości właściwości `Value`, właściwość ta wywoła metodę `SetSomeValue`, która zmodyfikuje wartość zmiennej `SomeValue`. Takie rozwiązanie jest korzystne z kilku powodów. Po pierwsze, umożliwia programiście tworzenie naturalnego mechanizmu pobierania i ustawiania właściwości (np. podczas ponownego wyznaczania wartości równania lub odświeżania widoku kontrolki). Po drugie, umożliwia nam prezentowanie przed użytkownikami naszej klasy prostej zmiennej bez konieczności zasypywania ich szczegółami związanymi z implementacją tej klasy. I wreszcie po trzecie, możemy zezwolić użytkownikom na przykrywanie metod dostępowych (tzw. akcesorów) w klasach potomnych zgodnie z zasadą polimorfizmu obiektów.

Podobnie jak w przypadku omówionych wcześniej pól i metod język programowania Delphi obsługuje także właściwości statyczne, które także deklaruje się ze słowem kluczowym `class`. Poniższy fragment kodu demonstruje klasę ze statyczną właściwością zapewniającą dostęp do pola statycznego:

```
TMyClass = class(TObject)
  class var FValue: Integer;
  class procedure SetValue(Value: Integer); static;
  class property Value: Integer read FValue write SetValue;
end;
```

Pamiętaj, że statyczne właściwości mogą obsługiwać dostęp tylko do pól statycznych i wykorzystywać jedynie statyczne metody dostępowe.

Zdarzenia

Język programowania Delphi obsługuje dwa różne typy zdarzeń: pojedyncze i grupowe.

Zdarzenia pojedyncze są obsługiwane w języku Delphi od jego pierwszej wersji. Są deklarowane w postaci właściwości, których typami są typy procedur z akcesorami odczytu (`read`) i zapisu (`write`). Zdarzenia pojedyncze mogą mieć zdefiniowane najwyżej po jednej procedurze nasłuchującej. Do łączenia tych procedur ze zdarzeniami wykorzystuje się zwykły operator przypisania — przypisanie zdarzeniu wartości `nil` jest równoważne z rozłączeniem tego zdarzenia i jego dotychczasowej procedury nasłuchującej. Na listingu 5.3 przedstawiono przykładowy kod deklarujący i wykorzystujący zdarzenie pojedyncze.

Listing 5.3. *Przykładowy program wykorzystujący zdarzenie pojedyncze*

```
1:  program  singleevent;
2:
3:  {$APPTYPE  CONSOLE}
4:
5:  type
6:    TMyEvent = procedure (Sender: TObject; Msg: string) of object;
7:
8:    TClassWithEvent = class
9:    private
10:     FAnEvent: TMyEvent;
11:   public
12:     procedure FireEvent;
13:     property AnEvent: TMyEvent read FAnEvent write FAnEvent;
14:   end;
15:
16:   TListener = class
17:     procedure EventHandler(Sender:  TObject; Msg: string);
18:   end;
19:
20: { TClassWithEvent }
21:
22: procedure TClassWithEvent.FireEvent;
23: begin
24:   if Assigned(FAnEvent) then
25:     FAnEvent(Self, '*zdarzenie pojedyńcze*');
26: end;
27:
28: { TListener }
29:
30: procedure TListener.EventHandler(Sender: TObject; Msg: string);
31: begin
32:   WriteLn('Zdarzenie zostało wywołane. Wiadomość to: ', Msg);
33: end;
34:
35: var
36:   L: TListener;
37:   CWE: TClassWithEvent;
38: begin
39:   L := TListener.Create;           // tworzy obiekty
40:   CWE := TClassWithEvent.Create;
```

```
41:    CWE.AnEvent := L.EventHandler;   // przypisuje procedurę obsługi zdarzenia
42:    CWE.FireEvent;                   // powoduje wywołanie zdarzenia
43:    CWE.AnEvent := nil;              // rozłącza procedurę obsługi zdarzenia
44:    ReadLn;
45:  end.
```

Oto dane wyjściowe wygenerowane przez program z listingu 5.3:

```
Zdarzenie zostało wywołane. Wiadomość to: *zdarzenie pojedyńcze*
```

Zdarzenia grupowe zostały dodane do najnowszej wersji języka Delphi, aby zapewnić zgodność ze specyfikacją platformy .NET, która przewiduje możliwość stosowania wielu procedur nasłuchujących dla pojedynczego zdarzenia. Zdarzenie grupowe jest własnością, której typem jest typ proceduralny i która wymaga zdefiniowania akcesorów `add` i `remove`. Do dodawania i usuwania procedur nasłuchujących dla zdarzenia grupowego wykorzystuje się odpowiednio procedury `Include()` i `Exclude()`.

Listing 5.4 zawiera przykładowy kod deklarujący i wykorzystujący zdarzenie grupowe.

Listing 5.4. *Przykładowy program wykorzystujący zdarzenie grupowe*

```
1:   program multievent;
2:
3:   {$APPTYPE CONSOLE}
4:
5:   uses
6:     SysUtils;
7:
8:   type
9:     TMyEvent = procedure (Sender: TObject; Msg: string) of object;
10:
11:    TClassWithEvent = class
12:    private
13:      FAnEvent: TMyEvent;
14:    public
15:      procedure FireEvent;
16:      property AnEvent: TMyEvent add FAnEvent remove FAnEvent;
17:    end;
18:
19:    TListener = class
20:      procedure EventHandler(Sender: TObject; Msg: string);
21:    end;
22:
23:  { TClassWithEvent }
24:
25:  procedure TClassWithEvent.FireEvent;
26:  begin
27:    if Assigned(FAnEvent) then
28:      FAnEvent(Self, '*zdarzenie grupowe*');
29:  end;
30:
31:  { TListener }
32:
33:  procedure TListener.EventHandler(Sender:  TObject;  Msg: string);
34:  begin
35:    WriteLn('Zdarzenie zostało wywołane. Wiadomość to: ', Msg);
36:  end;
```

```
37:
38:  var
39:    L1, L2: TListener;
40:    CWE: TClassWithEvent;
41:  begin
42:    L1 := TListener.Create;                 // tworzy obiekty
43:    L2 := TListener.Create;
44:    CWE := TClassWithEvent.Create;
45:    Include(CWE.AnEvent, L1.EventHandler);  // przypisuje procedurę obsługi
     zdarzenia
46:    Include(CWE.AnEvent, L2.EventHandler);  // przypisuje procedurę obsługi
     zdarzenia
47:    CWE.FireEvent;                          // powoduje wywołanie zdarzenia
48:    Exclude(CWE.AnEvent, L1.EventHandler);  // rozłącza procedurę obsługi zdarzenia
       Exclude(CWE.AnEvent, L2.EventHandler);  // rozłącza procedurę obsługi zdarzenia
49:    ReadLn;
     end.
50:
51:
```

Oto dane wyjściowe wygenerowane przez program z listingu 5.3:

```
Zdarzenie zostało wywołane. Wiadomość to: *zdarzenie grupowe*
Zdarzenie zostało wywołane. Wiadomość to: *zdarzenie grupowe*
```

Zwróć uwagę na fakt, że takie zastosowanie procedury `Include()`, w którym do listy procedur nasłuchujących dodaliśmy tę samą metodę więcej niż raz, spowoduje, że w przypadku wystąpienia obsługiwanego zdarzenia metoda ta zostanie wywołana wielokrotnie.

Aby zachować zgodność z pozostałymi językami środowiska uruchomieniowego CLR platformy .NET Framework, kompilator Delphi implementuje semantykę grupową także dla zdarzeń pojedynczych, tworząc dla nich akcesory dodawania i usuwania (odpowiednio `add` i `remove`). W przypadku zdarzeń pojedynczych wywołanie metody `add()` powoduje zastąpienie dotychczasowej wartości.

Specyfikatory widoczności

Język Delphi oferuje możliwości jeszcze dalej idącej kontroli zachowania naszych obiektów — umożliwia nam deklarowanie pól i metod z takimi dyrektywami jak `private` (prywatne), `strict private` (ściśle prywatne), `protected` (chronione), `strict protected` (ściśle chronione), `public` (publiczne) i `published` (publikowane). Poniższy przykład ilustruje składnię stosowaną dla tych dyrektyw:

```
TSomeObject = class
private
  APrivateVariable: Integer;
  AnotherPrivateVariable: Boolean;
strict private
  procedure AStrictPrivateMethod;
protected
  procedure AProtectedProcedure;
  function ProtectMe: Byte;
strict protected
  procedure AStrictProtectedMethod;
```

```
public
  constructor APublicContructor;
  destructor Destroy; override;   // publiczny destruktor
published
  property AProperty: Integer
  read APrivateVariable write APrivateVariable;
end;
```

W każdym z bloków wyznaczanych przez te dyrektywy możemy umieszczać dowolną liczbę pól i metod. Zgodnie ze stylem programowania powinieneś w tych blokach stosować takie same wcięcia jak w całym kodzie klasy (względem jej nazwy). Poniżej przedstawiono znaczenie poszczególnych dyrektyw z tej grupy:

- `private` — te składowe naszego obiektu są dostępne tylko z poziomu kodu znajdującego się wewnątrz tego samego modułu co implementacja danego obiektu. Dyrektywę `private` wykorzystujemy nie tylko do ukrywania szczegółów implementacji naszych obiektów przed ich użytkownikami, ale także w celu zapobiegania bezpośrednim modyfikacjom kluczowych składowych dokonywanym przez użytkowników.
- `strict private` — te składowe naszego obiektu są dostępne tylko wewnątrz klasy deklarującej — nie są dostępne w pozostałych częściach tego samego modułu. Dyrektywę `strict private` wykorzystujemy do zapewnienia jeszcze ściślejszej izolacji składowych niż w przypadku dyrektywy `private`.
- `protected` — chronione składowe naszego obiektu są dostępne dla jego obiektów potomnych. Dzięki temu możemy ukrywać szczegóły implementacji naszych obiektów przed ich użytkownikami, nie tracąc przy tym elastyczności niezbędnej podczas tworzenia efektywnych obiektów potomnych.
- `strict protected` — te składowe naszego obiektu są dostępne tylko wewnątrz klasy deklarującej i w jej potomkach, ale nie są dostępne z pozostałych bloków kodu modułów deklarujących te klasy. Dyrektywę `strict protected` wykorzystujemy do zapewnienia nieco ściślejszej izolacji składowych niż w przypadku dyrektywy `protected`.
- `public` — te pola i metody są dostępne ze wszystkich miejsc naszego programu. Konstruktory i destruktory obiektu zawsze powinny być deklarowane z dyrektywą `public`.
- `published` — z punktu widzenia dostępności składowych znaczenie tej dyrektywy jest identyczne jak w przypadku dyrektywy `public`. Dyrektywa `published` oferuje jednak dodatkową korzyść w postaci możliwości dodania atrybutu `[Browsable(true)]` do zawartych w tym bloku właściwości — w ten sposób powodujemy, że podczas pracy w trybie projektowania (*Designer*) tak zdefiniowane właściwości są widoczne w panelu *Object Inspector*. Atrybuty omówimy w dalszej części tego rozdziału.



Znaczenie dyrektywy `published` dobrze pokazuje subtelne odejście od koncepcji leżących u podstaw implementacji języka programowania Delphi dla platformy Win32. W tamtych wersjach tego języka dla właściwości zadeklarowanych z tą dyrektywą były generowane informacje o typach RTTI (od ang. *Runtime Type Information*). Odpowiednikiem mechanizmu RTTI jest odbicie, jednak okazuje się, że generowanie odbić jest możliwe dla wszystkich składowych klas, niezależnie od użytych specyfikatorów widoczności (dostępności).

Poniżej przedstawiono kod definiujący wprowadzoną już wcześniej klasę `TMyObject` — tym razem jednak zastosowano dyrektywy poprawiające spójność tego obiektu:

```
TMyObject = class
private
  SomeValue: Integer;
  procedure SetSomeValue(AValue: Integer);
published
  property Value: Integer read SomeValue write SetSomeValue;
end;

procedure TMyObject.SetSomeValue(AValue: Integer);
begin
  if SomeValue <> AValue then
    SomeValue := AValue;
end;
```

Teraz użytkownicy naszego obiektu nie będą już mogli bezpośrednio modyfikować wartości pola `SomeValue` — podczas modyfikowania danych tego obiektu będą musieli wykorzystywać specjalnie w tym celu zaprojektowany interfejs, który opiera się na właściwości `Value`.

Klasy zaprzyjaźnione

W języku C++ istnieje pojęcie tzw. klas zaprzyjaźnionych (czyli takich, które mają dostęp do prywatnych pól i metod należących do pozostałych klas). W języku programowania C++ można było stosować ten mechanizm za pomocą słowa kluczowego `friend`. Języki .NET, w tym Delphi i C#, oferują podobną możliwość, choć zaimplementowaną w zupełnie inny sposób. Wszystkie składowe klasy zadeklarowane w bloku rozpoczynającym się od dyrektywy `private` lub `protected` (ale bez dodatkowego specyfikatora `strict`) są widoczne i dostępne dla pozostałych klas i kodu zadeklarowanego wewnątrz tej samej przestrzeni nazw modułu.

Klasy pomocnicze

Klasy pomocnicze są wygodnym sposobem rozszerzenia funkcjonalności klas wykorzystywanych do tej pory bez konieczności ich modyfikowania. Zamiast wprowadzać zmiany, możemy stworzyć nową klasę pomocniczą (`helper`) i faktycznie przekazać jej metody do klasy oryginalnej. Dzięki temu użytkownicy naszej klasy oryginalnej mają możliwość wywoływania metod udostępnianych przez klasę `helper` w taki sam sposób, jak wywołują metody należące do klasy oryginalnej.

Poniższy kod jest przykładem utworzenia prostej klasy wraz z klasą pomocniczą; demonstruje także sposób wywoływania metody należącej do klasy `helper`:

```
program Helpers;

{$APPTYPE CONSOLE}

type
  TFoo = class
```

```
    procedure AProc;
  end;

  TFooHelper = class helper for TFoo
    procedure AHelperProc;
  end;

{ TFoo }

procedure TFoo.AProc;
begin
  WriteLn('TFoo.AProc');
end;

{ TFooHelper }

procedure TFooHelper.AHelperProc;
begin
  WriteLn('TFooHelper.AHelperProc');
  AProc;
end;

var
  Foo: TFoo;
begin
  Foo := TFoo.Create;
  Foo.AHelperProc;
end.
```



Klasy pomocnicze są interesującym mechanizmem, jednak w ogólności ich stosowanie nie jest potrzebne w przypadku dobrze zaprojektowanego oprogramowania. Utrzymano ten mechanizm przede wszystkim dlatego, że firma Borland starała się maksymalnie ukryć różnice pomiędzy standardowymi klasami platformy .NET a ich odpowiednikami tworzonymi w języku Delphi dla Win32. Właściwie przeprowadzona faza projektowania aplikacji powinna bardzo ograniczyć lub nawet wykluczyć konieczność stosowania klas pomocniczych.

Typy zagnieżdżone

Język programowania Delphi umożliwia umieszczanie klauzuli `type` wewnątrz deklaracji klasy, powodując tym samym zagnieżdżanie typów wewnątrz danej klasy. Do takich zagnieżdżonych typów możemy się odwoływać zgodnie ze składnią *`TypZewnętrzny`*`.`*`TypZagnieżdżony`* — ilustruje to poniższy przykładowy fragment kodu:

```
type
  OutClass = class
    procedure SomeProc;

    type
      InClass = class
        procedure SomeOtherProc;
      end;

  end;

  var
    IC: Outclass.InClass;
```

Przeciążanie operatorów

Język programowania Delphi obsługuje technikę przeciążania operatorów definiowanych dla klas i rekordów. Składnia przeciążania operatora jest równie prosta jak deklarowanie metody klasowej z konkretną nazwą i dodatkową dyrektywą. Pełna lista operatorów, które można przeciążać w budowanych klasach, jest dostępna na stronach pomocy internetowej dla języka Delphi pod hasłem *Operator Overloads*. Zademonstrowany poniżej przykładowy fragment kodu pokazuje sposób, w jaki można przeciążyć w kodzie klasy operatory dodawania i odejmowania:

```
  OverloadsOps = class
  private
    FField: Integer;
  public
    class operator Add(a, b: OverloadsOps): OverloadsOps;
    class operator Subtract(a, b: OverloadsOps): OverloadsOps;
  end;

class operator OverloadsOps.Add(a, b: OverloadsOps): OverloadsOps;
begin
  Result := OverloadsOps.Create;
  Result.FField := a.FField + b.FField;
end;

class operator OverloadsOps.Subtract(a, b: OverloadsOps): OverloadsOps;
begin
  Result := OverloadsOps.Create;
  Result.FField := a.FField - b.FField;
end;
```

Zauważ, że przeciążone operatory są deklarowane z dyrektywą `class operator` (operatorów klasowych) i pobierają klasę deklarującą w formie swoich parametrów. Ponieważ + i - są operatorami binarnymi, obie metody dodatkowo zwracają klasę deklarującą.

Po zadeklarowaniu operatorów można je wykorzystywać w sposób zbliżony do tego z poniższego przykładu:

```
var
  01, 02, 03: OverloadsOps;
begin
  01 := OverloadsOps.Create;
  02 := OverloadsOps.Create;
  03 := 01 + 02;
end;
```

Atrybuty

Jednym z najciekawszych elementów, jakie daje programistom platforma .NET Framework, jest możliwość tworzenia aplikacji opartych na atrybutach — nad taką koncepcją wytwarzania oprogramowania od wielu lat pracowali twórcy kilku różnych języków programowania. Atrybuty mają na celu wiązanie metadanych z takimi elementami języka jak klasy, właściwości, metody, zmienne i inne konstrukcje — wszystkie te zabiegi mają na celu zapewnienie klientom szerszego zbioru informacji na temat tych elementów.

Atrybuty są deklarowane za pomocą specjalnej notacji z nawiasami kwadratowymi. Przykładowo, poniższy wycinek kodu demonstruje sposób użycia atrybutu `DLLImport`, który sygnalizuje platformie .NET konieczność zaimportowania danej metody ze wskazanego pliku biblioteki DLL:

```
[DllImport('user32.dll')]
function MessageBeep(uType : LongWord) : Boolean; external;
```

A kodzie aplikacji dla platformy .NET Framework atrybuty można wykorzystywać do rozmaitych celów. Przykładowo, zdefiniowany dla właściwości atrybut `Browsable` określa, czy dana właściwość powinna być wyświetlana i udostępniana w panelu *Object Inspector* środowiska programowania Delphi:

```
[Browsable(True)]
property Foo: string read FFoo write FFoo;
```

System atrybutów platformy .NET jest z natury rzeczy rozszerzalny, ponieważ same atrybuty są implementowane w postaci klas. Dzięki temu możemy niemal bez ograniczeń rozbudowywać ten system —tworzyć własne atrybuty od podstaw lub wykorzystywać mechanizm dziedziczenia po klasach istniejących definiujących atrybuty i dalej stosować nasze nowe atrybuty w innych klasach.

Interfejsy

Język Delphi oferuje naturalną obsługę interfejsów, które — mówiąc najprościej — definiują zbiór funkcji i procedur wykorzystywanych przez użytkownika do operowania na obiektach. Definicja interfejsu jest znana zarówno dla części implementującej udostępniane elementy, jak i dla klienta tego interfejsu — interfejs pełni więc rolę „umowy" pomiędzy częścią implementacji a klientem, która z jednej strony określa sposób jego realizacji, a z drugiej strony definiuje sposób jego praktycznego wykorzystania. Pojedyncza klasa może implementować wiele interfejsów, oferując tym samym różne „oblicza" danej klasy, za pośrednictwem których klient może ją kontrolować.

Jak sama nazwa wskazuje, interfejs definiuje wyłącznie mechanizm pośredniczący w komunikacji klienta z obiektem. Za obsługę interfejsu i odpowiednią implementację każdej z jego funkcji i procedur odpowiada klasa.



Inaczej niż w języku Delphi dla platformy Win32, interfejsy definiowane w języku Delphi for .NET nie są niejawnymi potomkami interfejsów `IInterface` ani `IUnknown`. Oznacza to, że interfejsy definiowane w aplikacjach .NET nie implementują już takich elementów jak `QueryInterface()`, `_AddRef` czy `_Release()`. Rzutowanie typów jest teraz wykorzystywane do zapewniania zgodności typów, a mechanizm zliczania referencji jest elementem wbudowanym w platformie .NET.

Definiowanie interfejsów

Składnia definiowania interfejsu jest bardzo podobna do składni stosowanej w przypadku klas. Zasadnicza różnica dotyczy możliwości tworzenia opcjonalnego łącza pomiędzy interfejsem a identyfikatorem unikalnym globalnie (ang. *Globally Unique Identifier — GUID*), który pełni rolę unikalnego reprezentanta danego interfejsu. Poniższy kod definiuje nowy interfejs nazwany `IFoo`, który implementuje pojedynczą metodę nazwaną `F1()`:

```
type
  IFoo = interface
    function F1: Integer;
  end;
```

Pamiętaj, że stosowanie identyfikatorów GUID nie jest wymagane w definicjach interfejsów dla platformy .NET, choć były i są one konieczne w aplikacjach dla platformy Win32. Wykorzystywanie tych identyfikatorów jest więc zalecane tylko wtedy, gdy tworzony kod ma mieć charakter wieloplatformowy lub kiedy wykorzystujesz mechanizm .NET COM Interop zapewniający współpracę pomiędzy aplikacją .NET a obiektami COM.



Użycie kombinacji klawiszy *Ctrl + Shift + G* spowoduje automatyczne wygenerowanie przez środowisko programowania Delphi nowych identyfikatorów GUID dla Twoich interfejsów.

Poniższy fragment kodu definiuje nowy interfejs, który jest potomkiem zdefiniowanego przed chwilą interfejsu `IFoo`:

```
type
  IBar = interface(IFoo)
    function F2: Integer;
  end;
```

Implementowanie interfejsów

Poniższy fragment kodu demonstruje sposób, w jaki można zaimplementować interfejsy `IFoo` oraz `IBar` w naszej klasie nazwanej `TFooBar`:

```
TFooBar:

type
  TFooBar = class(TObject, IFoo, IBar)
    function F1: Integer;
    function F2: Integer;
  end;

function TFooBar.F1: Integer;
begin
  Result := 0;
end;

function TFooBar.F2: Integer;
begin
  Result := 0;
end;
```

Zwróć uwagę na możliwość wypisywania dowolnej liczby interfejsów bezpośrednio za klasą przodka w pierwszym wierszu deklaracji klasy — umieszczenie w tym miejscu więcej niż jednego identyfikatora oznacza, że będziemy implementowali wiele interfejsów. Proces wiązania funkcji zadeklarowanej w naszym interfejsie z konkretną funkcją należącą do klasy odbywa się w tym samym czasie, w którym kompilator dopasowuje sygnatury metod wymienionych w interfejsie do sygnatur metod zdefiniowanych w danej klasie. W przypadku braku możliwości znalezienia implementacji jednej lub więcej metod interfejsu w klasie implementującej ten interfejs kompilator wygeneruje odpowiedni komunikat o błędzie.

Metody interfejsu ułatwiają pracę

Interfejsy są oczywiście wspaniałym rozwiązaniem, jednak samodzielne wpisywanie kodu wewnątrz klasy, który jest niezbędny do zaimplementowania metod interfejsu, może być bardzo pracochłonne! Poniżej przedstawiono możliwości, jakie daje w tym zakresie środowisko Delphi — wykonując poniższe kroki, możesz implementować wszystkie metody interfejsu przez naciśnięcie kilku kombinacji klawiszy i kilkukrotne kliknięcie myszą:

1. Dodaj do deklaracji swojej klasy interfejs, który chcesz zaimplementować.
2. Umieść kursor w dowolnym miejscu wewnątrz klasy i naciśnij kombinację klawiszy *Ctrl + Spacja*, aby wywołać mechanizm automatycznego wykańczania kodu. W wyświetlonym oknie wykańczania kodu zostaną na czerwono wyświetlone niezaimplementowane jeszcze metody.
3. Zaznacz na liście wszystkie metody oznaczone kolorem czerwonym — przytrzymując wciśnięty klawisz *Shift*, użyj klawiszy strzałek lub myszy.
4. Naciśnij klawisz *Enter* — metody interfejsu zostaną automatycznie dodane do definicji klasy.
5. Naciśnij kombinację klawiszy *Ctrl + Shift + C*, aby wykończyć część implementacyjną dla nowo dodanych metod.
6. Teraz musisz już tylko odpowiednio wypełnić część implementacyjną każdej z dodanych metod!

Jeśli nasza klasa implementuje wiele interfejsów, które zawierają metody oznaczone takimi samymi sygnaturami, musimy dla tych metod stworzyć odpowiednie aliasy — ilustruje to przedstawiony poniżej krótki przykład kodu źródłowego:

```
type
  IFoo = interface
    function F1: Integer;
  end;

  IBar = interface
    function F1: Integer;
  end;

  TFooBar = class(TObject, IFoo, IBar)
    // metody z aliasami
    function IFoo.F1 = FooF1;
    function IBar.F1 = BarF1;
    // metody interfejsu
    function FooF1: Integer;
    function BarF1: Integer;
  end;

function TFooBar.FooF1: Integer;
begin
  Result := 0;
end;
function TFooBar.BarF1: Integer;
begin
  Result := 0;
end;
```



Stosowana w języku Delphi dla platformy Win32 dyrektywa `implements` nie jest już dostępna w aktualnej wersji kompilatora Delphi dla platformy .NET.

Stosowanie interfejsów

Ze stosowaniem w naszych aplikacjach zmiennych typu interfejsowego wiąże się kilka istotnych reguł językowych. Podobnie jak inne typy wykorzystywane w platformie .NET także interfejsy są zarządzane w czasie wykonywania programu. Mechanizm odzyskiwania pamięci zwolni pamięć zajmowaną przez obiekt dopiero wtedy, gdy zostaną zwolnione lub wyjdą poza bieżący zakres wszystkie referencje do tego obiektu i jego zaimplementowanych interfejsów. Przed użyciem typy interfejsowe są zawsze inicjalizowane wartością `nil`. Ręczne przypisanie wartości `nil` do zmiennej interfejsu powoduje zwolnienie referencji do odpowiedniego obiektu, który jest implementacją danego interfejsu.

Inną unikalną regułą dotyczącą zmiennych interfejsowych jest ich zgodność (w operacjach przypisania) z obiektami, które te interfejsy implementują. Należy jednak pamiętać, że ta zgodność występuje tylko w jedną stronę — możemy przypisywać referencji do obiektu referencję do interfejsu, ale nie możemy wykonać operacji odwrotnej. Przykładowo, poniższy fragment kodu jest poprawnym wykorzystaniem zdefiniowanej wcześniej klasy `TFooBar`:

```
procedure Test(FB: TFooBar)
var

  F: IFoo;
begin

  F := FB;  // obsługiwane, ponieważ FB jest implementacją interfejsu IFoo
  .
  .
  .
```

Gdyby zmienna `FB` nie była referencją do klasy implementującej interfejs `IFoo`, powyższy fragment kodu zostałby co prawda skompilowany, ale referencja do interfejsu miałaby wartość `nil`. W takim przypadku każda kolejna próba wykorzystania tej referencji powodowałaby w czasie wykonywania programu generowanie wyjątku `NullReferencedException`.

Język programowania Delphi umożliwia także stosowanie operatora rzutowania typów `as` do przekształcania danej zmiennej referencyjnej do jednego interfejsu w zmienną referencyjną wskazującą na inny interfejs tego samego obiektu. Ilustruje to poniższy fragment kodu:

```
var
  FB: TFooBar;
  F: IFoo;
  B: IBar;
begin
  FB := TFooBar.Create;
  F := FB;  // obsługiwane, ponieważ FB jest implementacją interfejsu IFoo
  B := F as IBar;  // rzutowanie dla IBar
  .
  .
  .
```

Gdyby docelowy typ rzutowania nie był zgodny z typem bieżącym, wynikiem wyrażenia byłaby wartość `nil`.

Ujednolicony mechanizm obsługi wyjątków

Mechanizm obsługi wyjątków (ang. *Structured Exception Handling — SEH*) jest zcentralizowaną i ustandaryzowaną metodą obsługi błędów, który oferuje zarówno nieinwazyjną obsługę wyjątków na poziomie kodu źródłowego aplikacji, jak i możliwość zgrabnego operowania na niemal wszystkich rodzajach uwarunkowań będących źródłem występowania błędów. Mechanizm SEH dostępny w języku programowania Delphi jest odwzorowaniem metod stosowanych w środowisku uruchomieniowym CLR platformy .NET.

Najkrócej mówiąc, wyjątki są po prostu klasami, które od czasu do czasu przechowują informacje o lokalizacji i naturze konkretnego błędu. Dzięki zastosowaniu takiego modelu wyjątki są bardzo łatwe w implementacji i stosowaniu nie tylko w naszych aplikacjach, ale także we wszystkich pozostałych klasach języka Delphi.

Platforma .NET udostępnia wiele predefiniowanych wyjątków reprezentujących mnóstwo rozmaitych błędów pojawiających się w programach dla tej platformy, w tym wyczerpanie pamięci, dzielenie przez zero, przekroczenie (w górę lub w dół) zakresu liczb czy błędy operacji wejścia-wyjścia na pliku. Firma Borland zaoferowała użytkownikom swojego zintegrowanego środowiska programowania Delphi dodatkowe klasy wyjątków umieszczone w bibliotekach RTL i VCL. Oczywiście nic nie stoi na przeszkodzie, abyśmy sami definiowali własne klasy wyjątków, które w jak największym stopniu będą odpowiadały potrzebom naszych aplikacji.

Na listingu 5.5 zademonstrowano sposób wykorzystania mechanizmu obsługi wyjątków dla operacji wejścia-wyjścia na pliku tekstowym.

Listing 5.5. *Operacje wejścia-wyjścia przeprowadzane na pliku z wykorzystaniem mechanizmu obsługi wyjątków*

```
1:  program FileIO;
2:
3:  {$APPTYPE CONSOLE}
4:
5:  uses System.IO;
6:
7:  var
8:    F: TextFile;
9:    S: string;
10: begin
11:   AssignFile(F, 'FOO.TXT');
12:   try
13:     Reset(F);
14:     try
15:       ReadLn(F, S);
16:       WriteLn(S);
17:     finally
18:       CloseFile(F);
19:     end;
20:   except
```

```
21:       on System.IO.IOException do
22:         WriteLn('Błąd w trakcie dostępu do pliku!');
23:     end;
24:     ReadLn;
25:   end.
```

Na listingu 5.5 wewnętrzny blok `try-finally` jest wykorzystywany do upewnienia się, że przetwarzany plik tekstowy zostanie zamknięty niezależnie od tego, czy podczas samego przetwarzania wystąpi jakiś wyjątek. Znaczenie tego bloku można by wyjaśnić w następujący sposób: „Spróbuj wykonać instrukcje pomiędzy słowami `try` i `finally`. Niezależnie od tego, czy uda się je wykonać czy też podjęta próba doprowadzi do wygenerowania wyjątku, wykonaj instrukcje pomiędzy słowami `finally` i `end`. Po wykonaniu tych instrukcji przejdź do kolejnego bloku obsługi wyjątków". Oznacza to, że przetwarzany plik tekstowy w każdym przypadku zostanie zamknięty, a ewentualny błąd będzie właściwie obsłużony niezależnie od tego, do jakiej kategorii będzie należał.



Instrukcje umieszczone po słowie `finally` w bloku `try-finally` są wykonywane niezależnie od ewentualnych wystąpień wyjątków. Upewnij się, że kod zdefiniowany w bloku `finally` nie zakłada wystąpienia jakiegokolwiek wyjątku. Ponieważ instrukcja `finally` w żaden sposób nie wstrzymuje przekazywania ewentualnego wyjątku, przepływ wykonywania programu będzie w normalny sposób kontynuowany w kolejnych instrukcjach obsługujących wyjątki.

Zewnętrzny blok `try-except` jest wykorzystywany do obsługi ewentualnych wyjątków występujących w czasie wykonywania programu. Po zamknięciu przetwarzanego pliku tekstowego w bloku `finally` kod zawarty w bloku `except` wyświetla na konsoli komunikat informujący użytkownika o wystąpieniu błędu operacji wejścia-wyjścia.

Jedną z kluczowych zalet takiego mechanizmu obsługi wyjątków (przynajmniej w porównaniu z tradycyjnymi metodami tego typu opartymi najczęściej na weryfikacji wartości zwracanych przez funkcję) jest możliwość pełnego oddzielenia kodu wykrywającego błędy od kodu korygującego wykryte błędy. Takie rozwiązanie jest korzystne przede wszystkim dlatego, że ułatwia czytanie i konserwowanie naszego kodu — pozwala skupić się w danym momencie tylko na konkretnym obszarze funkcjonalności analizowanej aplikacji.

Duże znaczenie ma fakt, że nie możemy za pomocą bloków `try-finally` „zastawiać pułapek" tylko na z góry określone, konkretne wyjątki. Kiedy stosujemy w naszym kodzie blok `try-finally`, oznacza to, że tak naprawdę nie interesuje nas, jakiego rodzaju wyjątki mogą wystąpić — chcemy jedynie mieć pewność, że pewne zadania zostaną prawidłowo wykonane niezależnie od ewentualnych wystąpień błędów. Blok `finally` jest idealnym miejscem do zwalniania przydzielonych wcześniej zasobów (takich jak pliki czy zasoby systemu Windows), ponieważ kod zawarty w tym bloku zostanie wykonany także w przypadku wystąpienia błędów. W wielu przypadkach musimy jednak zastosować taki mechanizm obsługi błędów, który zapewni możliwość różnego reagowania w zależności od rodzaju wykrytych błędów. Możemy „zastawiać pułapki" na konkretne wyjątki za pomocą bloków `try-except` — ilustruje to kod z listingu 5.6.

Listing 5.6. *Przykład zastosowania bloku obsługi wyjątków try-except*

```
1:  program HandleIt;
2:
3:  {$APPTYPE CONSOLE}
4:
5:  var
6:    D1, 02, D3: Double;
7:  begin
8:    try
9:      Write('Podaj liczbę: ');
10:     ReadLn(D1);
11:     Write('Podaj następną liczbę: ');
12:     ReadLn(D2);
13:     Writeln('Teraz podzielę liczbę pierwszą przez drugą...');
14:     D3 := D1 / D2;
15:     Writeln('Odpowiedź to: ', D3:5:2);
16:   except
17:     on System.OverflowException do
18:       Writeln('Przepełnienie stosuw trakcie dzielenia!');
19:     on System.DivideByZeroException do
20:       Writeln('Nie możesz dzielić przez zero!');
21:     on Borland.Delphi.System.EInvalidlnput do
22:       Writeln('Podana liczba nie jest prawidłowa!');
23:   end;
24: end.
```

Chociaż zastosowanie bloku `try-except` umożliwia nam „zastawianie pułapek" tylko na określone z góry wyjątki, możemy także przechwytywać i obsługiwać wszystkie pozostałe wyjątki, dodając do tej konstrukcji klauzulę `else`. Składnia konstrukcji `try-except-else` jest następująca:

```
try
  Instrukcje
except
  On ESomeException do Something;
else
  { domyślny kod obsługi pozostałych wyjątków }
end;
```



Stosując konstrukcję `try-except-else`, powinieneś pamiętać, że w części `else` będą przechwytywane i obsługiwane *wszystkie* wyjątki, włącznie z tymi, których możesz się w tym miejscu nie spodziewać — w tym błędów wyczerpania pamięci lub innych wyjątków biblioteki uruchomieniowej. W związku z tym używaj klauzuli `else` bardzo ostrożnie i staraj się robić to możliwie rzadko. Każdy przechwycony w ten sposób (a więc trochę przypadkowo) wyjątek powinieneś ponownie generować. Więcej informacji na ten temat znajdziesz w punkcie „Ponowne generowanie wyjątków".

Ten sam efekt, który uzyskujemy za pomocą konstrukcji `try-except-else`, możemy uzyskać także za pomocą uproszczonej wersji bloku `try-except`, w której nie określimy klasy wyjątku — oto przykład:

```
try
  Instrukcje
except
  HandleException  // takie samo znaczenie jak kod umieszczony w klauzuli else
end;
```

Klasy wyjątków

Wyjątki są po prostu specjalnymi egzemplarzami obiektów. Odpowiednie egzemplarze są tworzone w momencie występowania reprezentowanych przez nie wyjątków i są niszczone w chwili ich przechwycenia i obsłużenia w kodzie programu. Obiektem bazowym dla wszystkich wyjątków w aplikacjach platformy .NET jest `System.Exception`.

Jednym z ważniejszych elementów w obiekcie `Exception` jest właściwość `Message`, która reprezentuje łańcuch z dodatkowymi informacjami lub wyjaśnieniem danego wyjątku. Rodzaj informacji zawartych w tym łańcuchu zależy oczywiście od typu odpowiedniego wyjątku.



Jeśli zdecydujesz się na definiowanie własnego obiektu wyjątku, upewnij się, że Twój obiekt dziedziczy po znanym obiekcie wyjątku — po najbardziej ogólnym obiekcie bazowym `Exception` lub jednym z jego potomków. Dzięki temu ogólne procedury obsługi wyjątków będą mogły odpowiednio przechwytywać Twój wyjątek.

Kiedy obsługujemy w bloku `except` konkretny rodzaj wyjątku, ten sam blok będzie przechwytywał także wszystkie te wyjątki, które zostały zdefiniowane jako obiekty potomne względem wyjątku, który wskazaliśmy w tym bloku. Przykładowo, obiekt `System.ArithmeticException` jest przodkiem dla wielu wyjątków związanych z działaniami matematycznymi, w tym wyjątku dzielenia przez zero (`DivideByZeroException`), niezastosowania liczby skończonej (`NotFiniteNumberException`) czy przekroczenia zakresu (`OverflowException`). Możemy przechwytywać dowolne z tych wyjątków, ustawiając w bloku `except` klasę bazową `ArithmeticException` (patrz poniższy przykład):

```
try
  Instrukcje
except
  on EMathError do  // przechwyci EMathError i wszystkie wyjątki potomne
    HandleException
end;
```

Wszystkie pojawiające się wyjątki, których nie obsługujesz w swoim programie wprost (nie wymieniasz ich w bloku `except`), będą przechowywane na stosie aż do momentu ich obsłużenia. W aplikacjach typu WinForm i WebForm dla platformy .NET domyślny mechanizm obsługi wyjątków sam odpowiada za realizację zadań zmierzających do prezentowania informacji o wyjątkach przed użytkownikiem. W aplikacjach opartych na komponentach biblioteki VCL domyślny mechanizm obsługi wyjątków wyświetla specjalne okno dialogowe z komunikatem informującym użytkownika o zaistniałej sytuacji.

W naszym kodzie przechwytującym i obsługującym wyjątki musimy niekiedy uzyskać dostęp do egzemplarza obiektu wyjątku, aby uzyskać więcej informacji na jego temat — także tych udostępnianych przez właściwość `Message`. Istnieją dwa sposoby uzyskiwania takiego dostępu. Po pierwsze, metodą preferowaną jest wykorzystanie opcjonalnego identyfikatora już w konstrukcji `on` *`SomeException`*. Możemy także użyć funkcji `ExceptObject()` — nie jest to jednak sposób zalecany.

Do części `on` *`ESomeException`* bloku `except` możemy dodać opcjonalny identyfikator, który będzie reprezentował odpowiedni egzemplarz aktualnie obsługiwanego wyjątku.

Zgodnie ze składnią języka programowana Delphi taki identyfikator powinien poprzedzać nazwę typu wyjątku (oba elementy powinny być oddzielone dwukropkiem) — oto przykład:

```
try
  Cokolwiek
except
  on E:ESomeException do
    ShowMessage(E.Message);
end;
```

Identyfikator egzemplarza (w tym przypadku `E`) otrzymuje referencję do właśnie przechwyconego wyjątku. Taki identyfikator jest zawsze tego samego typu co poprzedzany przez niego wyjątek.

Składnia generowania wyjątków jest podobna do składni tworzenia egzemplarza obiektu. Aby wygenerować np. zdefiniowany przez użytkownika wyjątek `EBadStuff`, użylibyśmy następującej składni:

```
raise EBadStuff.Create('Some bad stuff happened.');
```

Przepływ sterowania działaniem

Po wywołaniu wyjątku sterowanie działaniem naszego programu jest przekazywane do kolejnej procedury obsługi wyjątków i pozostaje tam aż do momentu pełnego obsłużenia i zniszczenia danego egzemplarza wyjątku. Cały ten proces jest kontrolowany przez stos wywołań, dotyczy zatem całego programu (nie tylko pojedynczej procedury lub bieżącego modułu). Na listingu 5.7 przedstawiono moduł VCL, który dobrze ilustruje przepływ sterowania działaniem programu w przypadku wystąpienia wyjątku. Listing zawiera główny moduł aplikacji napisanej w języku Delphi, która składa się z pojedynczej formy zawierającej jeden przycisk. Kliknięcie tego przycisku powoduje, że metoda `Button1Click()` wywołuje procedurę `Proc1()`, która wywołuje procedurę `Proc2()`, która z kolei wywołuje procedurę `Proc3()`. Ponieważ wyjątek jest generowany w procedurze `Proc3()`, możemy prześledzić przepływ sterowania działaniem programu za pośrednictwem kolejnych bloków `try-finally` aż do momentu, w którym wygenerowany wyjątek zostanie ostatecznie obsłużony wewnątrz metody `Button1Click()`.

Listing 5.7. *Główny moduł programu demonstrującego przepływ sterowania jego działaniem*

```
1:   unit Main;
2:
3:   interface
4:
5:   uses
6:     Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
7:     Dialogs;
8:
9:   type
10:    TForm1 = class(TForm)
11:      Button1: TButton;
12:      procedure Button1Click(Sender: TObject);
13:    end;
14:
```

```
15:  var
16:    Form1: TForm1;
17:
18:  implementation
19:
20:  {$R *.nfm}
21:
22:  type
23:    EBadStuff = class(Exception);
24:
25:  procedure Proc3;
26:  begin
27:    try
28:      raise EBadStuff.Create('Idziemy do góry ze stosem!');
29:    finally
30:      ShowMessage('Wyjątek wywołany. Procedura Proc3 zobaczyła błąd');
31:    end;
32:  end;
33:
34:  procedure Proc2;
35:  begin
36:    try
37:      Proc3;
38:    finally
39:      ShowMessage('Procedura Proc2 zobaczyła błąd');
40:    end;
41:  end;
42:
43:  procedure Proc1;
44:  begin
45:    try
46:      Proc2;
47:    finally
48:      ShowMessage('Procedura Proc1 zobaczyła błąd');
49:    end;
50:  end;
51:
52:  procedure TForm1.Button1Click(Sender: TObject);
53:  const
54:    ExceptMsg = 'Wyjątek obsłużony w wywoływanej procedurze. Wiadomość to "%s"';
55:  begin
56:    ShowMessage('Ta metoda wywołuje procedurę Proc1 która wywołuje procedurę Proc2
     która wywołuje Proc3');
57:    try
58:      Proc1;
59:    except
60:      on E:EBadStuff do
61:        ShowMessage(Format(ExceptMsg, [E.Message]));
62:    end;
63:  end;
64:
65:  end.
```



Kiedy uruchomisz ten program w środowisku programowania Delphi, będziesz miał możliwość jeszcze lepszej obserwacji przepływu sterowania działaniem aplikacji, jeśli wyłączysz mechanizm obsługi wyjątków zintegrowanego z tym środowiskiem programu uruchomieniowego — usuń zaznaczenie opcji *Tools/Options/Debugger Options/Borland .NET Debugger/Language Exceptions/Stop on Language Exceptions*.

Ponowne generowanie wyjątków

Kiedy musimy użyć specjalnego mechanizmu obsługi ewentualnych wyjątków dla instrukcji znajdującej się wewnątrz istniejącego bloku `try-except`, nie powodując przy tym przerwania przepływu sterowania działaniem programu do domyślnej, zewnętrznej procedury obsługi wyjątków, możemy zastosować technikę nazywaną ponownym generowaniem wyjątków. Przykład użycia tej techniki zademonstrowano na listingu 5.8.

Listing 5.8. *Ponowne generowanie wyjątku*

```
1:   try              // to jest blok zewnętrzny
2:     { instrukcje }
3:     { instrukcje }
4:     (instrukcje }
5:     try            // to jest specjalny blok wewnętrzny
6:       { jakaś instrukcja wymagająca specjalnej obsługi wyjątków }
7:     except
8:       on ESomeException do
9:       begin
10:        { specjalna obsługa wyjątków dla instrukcji wewnątrz bloku  }
11:        raise;     // ponownie generuje wyjątek do zewnętrznego bloku
12:      end;
13:    end;
14:  except
15:    // zewnętrzny blok zawsze wykona domyślną procedurę obsługi wyjątków
16:    on ESomeException do Something;
17:  end;
```