

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

Visual Studio .NET: .NET Framework. Czarna księga

Autorzy: Julian Templeman, David Vitter
Tłumaczenie: Anna Konopka, Marek Konopka
ISBN: 83-7197-733-6

Tytuł oryginału: [Visual Studio .NET:
The .NET Framework. Black Book](#)

Format: B5, stron: 736
Zawiera CD-ROM



Niniejsza książka stanowi wprowadzenie do .NET Framework, złożonego i bardzo bogatego zestawu narzędzi służących do tworzenia aplikacji dla platformy .NET. Lektura tej książki sprawi, że poznasz główne elementy .NET Framework i nauczysz się tworzyć programy dla platformy .NET. Duża liczba przykładów – od tworzenia grafiki do obsługi baz danych – zilustrowanych wieloma linijkami kodu, ułatwi Ci tworzenie zaawansowanych aplikacji w pełni korzystających z nowych cech platformy .NET. Dołączony CD-ROM zawiera wiele gotowych do użycia narzędzi, które ułatwią Ci pracę.

Dzięki tej książce:

- Zrozumiesz architekturę .NET
- Dowiesz się, czym jest i co zawiera .NET Framework
- Poznasz główne przestrzenie nazw .NET
- Nauczysz się tworzyć aplikacje z graficznym interfejsem użytkownika dla platformy .NET korzystające z biblioteki Windows Forms
- Dowiesz się, jak programować usługi XML Web Services za pomocą biblioteki ASP.NET,
- Nauczysz się obsługiwać bazy danych za pomocą biblioteki ADO.NET
- Dowiesz się jak korzystać z obiektów COM i API systemu Windows
- Zrozumiesz mechanizm bezpieczeństwa platformy .NET
- Nauczysz się korzystać z SOAP i XML
- Poznasz technologię Remoting
- Dowiesz się jak korzystać z formularzy i kontrolek WWW
- Nauczysz się obsługiwać piórami, pędzlami, kolorami i innymi składowymi przestrzeni nazw Drawing

Książka przeznaczona jest dla programistów Visual Basica, C++, C# i Javy tworzących aplikacje dla Windows.



Spis treści

O Autorach.....	17
Wstęp	19
Rozdział 1. Wprowadzenie do .NET	23
Co to jest .NET?	23
Wstęp do kluczowych technologii.....	25
IL i specyfikacja CLS	26
Środowisko CLR.....	27
Biblioteka klas bazowych.....	27
ASP.NET	29
Windows Forms.....	30
XML.....	31
C#.....	32
Jak działa .NET?.....	34
IL i metadane.....	34
Kompilacja JIT	35
Kod nadzorowany a automatyczne zwalnianie pamięci	36
Przestrzenie nazw.....	36
Podzespoły	37
Dziedziny aplikacyjne.....	40
Wpływ .NET na Visual C++ i Visual Basic.....	41
Visual C++	41
Visual Basic.....	43
Oto C#.....	44
Co się stało z COM?.....	46
Rozdział 2. Model programowania w środowisku .NET	49
Teoria	49
Programowanie obiektowe z lotu ptaka.....	49
Co to jest obiekt?.....	50
Zapis klas i obiektów w kodzie.....	52
Dziedziczenie i polimorfizm	54
Mała dygresja na temat UML.....	57
Interfejsy	58
Klasy.....	59
Części składowe klasy.....	59
Modyfikatory klas.....	60
Typy referencyjne i bezpośrednie.....	61
Struktury	63
Dziedziczenie.....	65
Interfejsy.....	65
Delegacje.....	66
Zdarzenia.....	67

Metadane i atrybuty.....	68
Wyjątki	69
Refleksja i klasa Type.....	72
Gotowe rozwiązania	74
Definiowanie klas.....	74
Przeciążanie i przesłanianie metod.....	74
Definiowanie pól i metod należących do klasy.....	75
Definiowanie struktur	75
Konstruktory i destruktory w VB.....	77
Sprzątanie po obiektach .NET.....	78
Korzystanie z dziedziczenia.....	78
Przesłanianie metod.....	79
Definiowanie klas abstrakcyjnych.....	80
Definiowanie zapieczętowanych klas i metod.....	81
Definiowanie właściwości.....	81
Definiowanie interfejsów	83
Implementowanie interfejsów.....	83
Korzystanie z obiektu za pośrednictwem interfejsu.....	84
Definiowanie i używanie delegacji.....	86
Definiowanie i używanie zdarzeń.....	90
Jak dołączyć atrybuty do klas i składowych?.....	96
Jak definiuje się atrybuty użytkownika?.....	97
Jak odczytać wartość atrybutu?.....	100
Jak obsłużyć wyjątek?	102
Jak zgłosić wyjątek?.....	104
Jak otrzymać obiekt klasy Type z informacjami o typie?.....	104
Jak odczytać informacje o typie?	105
Dynamiczne tworzenie obiektów	107
Rozdział 3. Przestrzeń nazw System.....	109
Teoria	109
Typy podstawowe	109
Typy podstawowe a CLS.....	110
Typy zmiennopozycyjne.....	110
Konwersje.....	111
Interfejsy	112
Klasa Object.....	113
Równość obiektów.....	113
Finalizacja.....	115
Metoda GetHashCode().....	116
Metoda GetType()	116
Klonowanie i kopiowanie	116
ToString()	117
Tablice	118
Inne typy.....	119
String.....	119
DateTime i TimeSpan	119
TimeZone	119
Decimal	120
Wyliczenia.....	120
Wyjątki	121
Klasa Console	122
Klasa Math	123
Klasa Type	123
Inne klasy.....	124

Gotowe rozwiązania	124
W jaki sposób można skorzystać z klas zdefiniowanych w przestrzeni nazw System?	124
Co łączy typy danego języka programowania z typami przestrzeni System?	125
Jak zdefiniować nowy typ bezpośredni?	126
Jak sprawdzić, czy dwa obiekty są takie same?	130
Typy referencyjne	130
Typy bezpośrednie	131
Jak zrealizować kopiowanie płytkie i głębokie?	131
Jak zdefiniować własną metodę ToString()?	133
Indeksowanie tablic w języku Visual Basic	135
Jak posługiwać się typem Array?	135
Tworzenie tablic	136
Odczyt właściwości tablicy	136
Odczyt elementów tablicy i nadanie im wartości	137
Metody klasy Array	138
Jak posługiwać się typem String?	140
Tworzenie obiektu klasy String	140
Porównywanie napisów	141
Kopiowanie i modyfikacja napisów	142
Przeszukiwanie napisów	143
Konwersja napisów	144
Jak przedstawiać i posługiwać się datami i czasem?	144
Tworzenie obiektu klasy TimeSpan	144
Odczyt wartości obiektów TimeSpan	145
Operacje na obiektach TimeSpan	145
Tworzenie obiektu klasy DateTime	145
Wyrowadzenie daty i czasu	146
Odczyt wartości obiektów DateTime	146
Operacje na obiektach DateTime	147
Jak definiować i posługiwać się typami wyliczeniowymi?	148
Jak dowiedzieć się, jaki wyjątek oraz gdzie wystąpił?	149
Jak korzysta się z wyjątków wewnętrznych?	150
Czym różnią się metody Console.WriteLine() i Console.Out.WriteLine()?	151
Jak formatować wyprowadzane dane?	151
Wykorzystanie szablonów do formatowania	152
Metoda ToString()	153
Jak generuje się liczby losowe?	154
Rozdział 4. Przestrzeń nazw System.Collections.....	157
Teoria	157
Interfejsy zdefiniowane w System.Collections	158
Interfejs IEnumerable	158
Interfejs IEnumerator	159
Interfejs ICollection	159
Interfejs IList	160
Interfejs IComparer	160
Interfejs IDictionary	161
Interfejs IDictionaryEnumerator	162
Interfejs IHashCodeProvider	162
Klasa ArrayList	162
Klasa BitArray	163
Klasa Hashtable	163
Klasa NameValueCollection	165

Klasa Queue	165
Klasa SortedList	166
Klasa Stack	166
Klasy StringCollection i StringDictionary	167
Gotowe rozwiązania	167
Której klasy kolekcji użyć?	167
Które kolekcje są wielobieżne?	168
Jak zbudować iterację dla elementów kolekcji?	169
Jak posługiwać się klasą ArrayList?	170
Tworzenie i wypełnianie obiektu klasy ArrayList	170
Usuwanie elementów	171
Operacje na obiektach ArrayList	172
Korzystanie z metod opakujących	173
Jak przechowywać dane identyfikowane kluczami?	174
Tworzenie i wypełnianie obiektu klasy Hashtable	174
Wyszukiwanie kluczy i wartości w obiekcie klasy Hashtable	175
Usuwanie elementów z obiektu klasy Hashtable	176
Korzystanie z metod opakujących obiekt klasy Hashtable	176
Korzystanie z klasy SortedList	177
Tworzenie i wypełnianie obiektu klasy SortedList	177
Pobieranie elementów w obiekcie klasy SortedList	179
Modyfikacja elementów w obiekcie klasy SortedList	179
Usuwanie elementów w obiekcie klasy SortedList	179
Korzystanie z obiektów SortedList przez wiele wątków	180
Przechowywanie listy elementów w obiekcie klasy Queue	180
Jak posługiwać się klasą Stack?	181
Jak przechowywać znaczniki w obiekcie klasy BitArray?	182
Przechowywanie napisów w obiekcie klasy StringCollection	183
Przechowywanie napisów w obiekcie klasy NameValueCollection	184
Wyszukiwanie i pobieranie elementów	185
Usuwanie elementów	186
Jak określić własną kolejność sortowania?	186
Jak zdefiniować własną kolekcję?	188
Rozdział 5. Przestrzeń nazw XML	191
Teoria	191
XML z lotu ptaka	191
Co to jest XML?	192
Budowa dokumentu XML	193
Atrybuty	195
Poprawność dokumentu XML	195
Przestrzeń nazw	196
Przetwarzanie dokumentów XML	197
Wykorzystanie arkuszy stylów XSL do przekształceń dokumentów XML	198
Przestrzeń nazw System.Xml	200
Klasa XmlTextReader	200
Klasa XmlValidatingReader	202
Klasa XmlTextWriter	203
Klasa XmlDocument	203
Klasa XmlNode	205
Klasa XmlElement	205
Składowe klasy XmlDocument	206
XSL i XPath	207
Klasa XPathNavigator	208

Gotowe rozwiązania	209
Której klasy należy użyć do obsługi XML?	209
Przetwarzanie dokumentu XML za pomocą klasy XmlTextReader	211
Tworzenie obiektu klasy XmlTextReader	212
Odczyt elementów	212
Korzystanie z atrybutów	214
Obsługa przestrzeni nazw	215
Przetwarzanie dokumentu ze sprawdzaniem poprawności	215
Zapis dokumentu XML za pomocą klasy XmlTextWriter	218
Wyprowadzanie instrukcji przetwarzania i komentarzy	220
Obsługa przestrzeni nazw	221
Korzystanie z klasy XPathNavigator	221
Tworzenie obiektu	221
Poruszanie się po drzewie	222
Nawigacja wśród atrybutów	224
Obsługa drzewa DOM za pomocą klasy XmlDocument	225
Ładowanie dokumentu XML	225
Nawigacja	226
Przetwarzanie węzłów potomnych	226
Tworzenie i modyfikacja węzłów	229
Korzystanie z klasy XPath	231
Kompilacja wyrażeń XPath	232
Przekształcanie dokumentu XML za pomocą klasy XslTransform	233
Rozdział 6. Przestrzenie nazw klas wejścia-wyjścia i sieciowych	235
Teoria	235
Strumienie	235
Klasa Stream	235
Klasa FileStream	237
Klasa MemoryStream	239
Inne klasy do obsługi strumieni	239
Tekstowe operacje wejścia-wyjścia	240
Podklasy TextWriter	241
Podklasy TextReader	243
Pliki i katalogi	244
Klasa FileSystemInfo	244
Klasa File	245
Klasa FileInfo	247
Klasa Directory	248
Klasa DirectoryInfo	249
Klasa Path	249
Klasa FileSystemWatcher	250
Przestrzeń nazw System.Net	251
Klasy: IPAddress, IPEndPoint i Dns	252
Podklasy WebRequest i WebResponse	252
Przestrzeń nazw System.Net.Sockets	253
Czym są gniazda?	253
Jak korzystać z gniazd?	254
Gotowe rozwiązania	256
Binarne operacje wejścia-wyjścia z użyciem strumieni	256
Odczyt i zapis w plikach tekstowych	258
Zapis w pliku	259
Odczyt z pliku	261

Przetwarzanie plików i katalogów	262
Założenie projektu	263
Odczyt listy napędów	263
Obsługa wyboru innego napędu	264
Przetworzenie katalogu	265
Wyświetlenie informacji o plikach i katalogach	266
Zmiana katalogu	268
Przejsięcie do góry	269
Śledzenie zmian plików i katalogów	269
Założenie projektu	270
Zdefiniowanie interfejsu użytkownika	270
Korzystanie z gniazd	273
Program klienta	274
Podłączenie do gniazda	275
Pobranie strumienia	275
Wysłanie danych do gniazda	275
Program serwera	276
Rozdział 7. Bezpieczeństwo w .NET	279
Teoria	279
Model bezpieczeństwa w .NET	280
Współpraca mechanizmów bezpieczeństwa .NET i systemu Windows	280
Uprawnienia dostępu do kodu	281
Uprawnienia związane z tożsamością	281
Uprawnienia związane z rolami	283
Zasady bezpieczeństwa	283
Definiowanie zasad bezpieczeństwa	285
Kodowanie operacji na uprawnieniach	289
Klasa <code>CodeAccessPermission</code>	289
Sprawdzanie uprawnień	291
Ograniczanie uprawnień	293
Gwarantowanie uprawnień	294
Gotowe rozwiązania	295
Nadawanie podzespołom nazw silnych	295
Nadanie nazwy silnej za pomocą Visual Studio .NET	296
Nadanie nazwy silnej za pomocą programu <code>al.exe</code>	296
Żądanie dostępu do zasobów	297
Ograniczanie dostępu do plików i katalogów	298
Umożliwienie wykonywania metody tylko wskazanym użytkownikom	300
Sprawdzenie bezpośrednie	300
Użycie atrybutów określających tożsamość	301
Użycie klas <code>WindowsIdentity</code> i <code>WindowsPrincipal</code>	302
Rozdział 8. Przestrzeń nazw System.Web	305
Teoria	305
Wprowadzenie do ASP.NET	305
Od ASP do ASP.NET	306
Jak działa sieć WWW?	307
Stosowanie technologii ASP.NET w aplikacjach	310
Formularze WWW	311
Jak działają formularze WWW?	312
Kod schowany	313
Zdarzenia ASP.NET	314
Formularze WWW w projekcie aplikacji	316

Kontrolki WWW.....	316
Kontrolki HTML.....	316
Kontrolki WWW.....	317
Zdarzenia zgłaszane przez kontrolki.....	318
Usługi XML Web Services.....	319
Wstęp do usług XML Web Services.....	320
Przykłady usług XML Web Services.....	321
Jak działają usługi XML Web Services?.....	323
Zmiana paradygmatów projektowych.....	324
Usługi XML Web Services w projekcie aplikacji.....	325
Tworzenie usług XML Web Services.....	327
Wywoływanie usług XML Web Services.....	327
Gotowe rozwiązania.....	331
Utworzenie formularza WWW.....	331
Dodanie kontrolek WWW do formularza WWW.....	333
Sposoby rozmieszczania kontrolek na formularzu WWW.....	333
Dodanie kontrolek i wybór sposobu ich rozmieszczania.....	333
Dodanie kodu obsługującego zdarzenia zgłaszane przez kontrolki WWW.....	334
Wykrycie przesłania zwrotnego w obsłudze zdarzenia Page_Load.....	335
Opóźniona obsługa zdarzeń.....	337
Korzystanie z kontrolki WWW DataGrid.....	340
Korzystanie z kontrolek sprawdzających.....	343
Przechowywanie danych w obiekcie sesji serwera WWW.....	344
Testowanie i debugging formularzy WWW.....	345
Punkty kontrolne i inne narzędzia.....	345
Właściwość Trace formularzy WWW.....	347
Utworzenie usługi XML Web Service.....	347
Usługa BookService.....	348
Zamiana biblioteki klas na usługę XML Web Service.....	350
Wykrywanie usług XML Web Services i korzystanie z plików WSDL.....	351
Wykrywanie usług XML Web Services.....	351
Dokument WSDL.....	352
Wywołanie usługi XML Web Service z aplikacji.....	354
Testowanie i debugging usług XML Web Services.....	355
Debugging usługi XML Web Service w Visual Studio .NET.....	355
Punkty kontrolne i inne narzędzia.....	356
Korzystanie ze zdalnych usług XML Web Services.....	356
Rozdział 9. Formularze Windows Forms.....	357
Teoria.....	357
Formularze i kontrolki.....	357
Anatomia aplikacji typu Windows Forms.....	358
Szkielet aplikacji.....	358
Kod programu i polecenie wywołujące kompilator.....	360
Klasa Form.....	361
Właściwości formularza.....	361
Związki między formularzami.....	364
Formularze MDI.....	366
Okna dialogowe.....	369
Obsługa zdarzeń.....	370
Klasa Application.....	372
Dziedziczenie wizualne.....	373
Powszechnie stosowane okna dialogowe.....	375

Gotowe rozwiązania	376
Jak utworzyć aplikację typu Windows Forms?	376
Jak zdefiniować i wyświetlić nowy formularz?	379
Utworzenie formularza MDI	380
Utworzenie i wyświetlenie okna dialogowego	381
Utworzenie okna dialogowego	381
Wyświetlenie okna dialogowego	382
Wyświetlenie okna komunikatu	383
Jak obsługiwać menu?	384
Obsługa zdarzeń zgłaszanych przez menu	385
Kodowanie operacji na menu	385
Jak dodać do formularza menu kontekstowe?	387
Wyświetlenie okna dialogowego „Otwieranie”	388
Utworzenie formularza pochodzącego od innego formularza	390
Użycie kontrolki Splitter	392
Rozdział 10. Kontrolki i formularze Windows Forms	393
Teoria	393
Formularze i kontrolki	393
Klasa Control	394
Styl obiektu klasy Control	396
Odświeżenie kontrolki	396
Zastosowanie kontroltek	398
Kontrolki Label i LinkLabel	398
Przyciski	399
Kontrolki CheckBox i RadioButton	401
Kontrolka ListBox	402
Kontrolka CheckedListBox	405
Kontrolka ComboBox	406
Pola tekstowe	406
Kontrolka DataGridView	411
Kontrolka DateTimePicker	412
Kontrolka MonthCalendar	415
Kontrolki UpDown	416
Kontrolka GroupBox	416
Kontrolka Panel	417
Paski przewijania i kontrolka TrackBar	418
Kontrolka ImageList	419
Kontrolki ListView i TreeView	420
Menu	425
Kontrolka PictureBox	426
Kontrolka ProgressBar	426
Kontrolka StatusBar	427
Kontrolka Toolbar	428
Klasa SystemInformation	429
Kontrolka TabControl	430
Kontrolka Timer	433
Kontrolki dostawcze	433
Gotowe rozwiązania	434
Rozmieszczenie kontroltek na formularzu	434
Ustalenie kolejności dostępu do kontroltek	436
Wykorzystanie etykiet do przemieszczania się między kontrolkami	437
Symulacja hiperłączy	437
Jak utworzyć grupę przycisków opcji?	438
Korzystanie z pól tekstowych	439

Odczyt i ustawienie zawartości.....	439
Pola tekstowe jednowierszowe i wielowierszowe.....	439
Operacje na zaznaczonym tekście.....	439
Zmiana wielkości liter.....	440
Skąd wiadomo, że zmieniła się zawartość pola tekstowego?.....	440
Zamaskowanie hasła wprowadzanego w polu tekstowym.....	440
Jak umożliwić użytkownikowi wybór jednego z napisów przechowywanych w tablicy?.....	440
Jak wyświetlić bieżącą wartość kontrolki TrackBar?.....	441
Jak używa się kontrolek: ListBox, CheckedListBox i ComboBox?.....	441
Ustawienie właściwości.....	442
Dodanie elementów.....	442
Ustalenie, który element listy jest wybrany.....	443
Obsługa zdarzenia wyboru elementu.....	444
Korzystanie z kontrolki CheckedListBox.....	445
Korzystanie z kontrolki ComboBox.....	445
Korzystanie z kontrolki StatusBar.....	446
Tekst i panele.....	446
Korzystanie z kontrolki ToolBar.....	447
Zdefiniowanie paska narzędziowego.....	448
Obsługa zdarzeń zgłaszanych przez przyciski.....	449
Style przycisków.....	450
Korzystanie z kontrolki TreeView.....	450
Utworzenie kontrolki TreeView.....	450
Tworzenie wierzchołków.....	450
Właściwości kontrolki TreeView określające jej wygląd.....	451
Obsługa zdarzeń.....	452
Korzystanie z kontrolki ListView.....	452
Utworzenie kontrolki ListView.....	453
Tworzenie elementów.....	453
Obsługa zdarzeń.....	454
Tworzenie formularzy z zakładkami.....	455
Korzystanie z kontrolki Timer.....	457
Jak na formularzu umieścić kontrolkę ActiveX?.....	457
Jak zdefiniować własną kontrolkę?.....	458
Zdefiniowanie właściwości kontrolki.....	458
Przesłonięcie metody OnPaint().....	459
Obsługa zdarzeń myszy.....	459
Testowanie kontrolki.....	460
Rozdział 11. Przestrzenie nazw Drawing.....	463
Teoria.....	463
Podstawowe funkcje podsystemu GDI+.....	463
Klasa Graphics.....	463
Podstawowe struktury danych.....	464
Kolory.....	468
Przybory do rysowania: pióra i pędzle.....	469
Szczegółowe informacje o klasie Graphics.....	475
Odświeżenie wyświetlanej grafiki.....	478
Czcionki.....	479
Obsługa obrazów.....	479
Klasa Image.....	481
Klasa Bitmap.....	482
Klasy Icon i SystemIcons.....	482

Drukowanie	484
Klasa PrintDocument	484
Klasy przechowujące ustawienia: PrinterSettings i PageSettings	485
Klasa PrintController	486
Zdarzenie PrintPage	487
Gotowe rozwiązania	488
Jak narysować coś na formularzu?	488
Korzystanie ze składowych klasy Graphics	489
Korzystanie z kolorów	490
Konwersje kolorów	492
Korzystanie z piór i pędzli	492
Tworzenie i korzystanie z piór	492
Tworzenie i korzystanie z pędzli	494
Korzystanie z przekształceń	496
Reprezentacja przekształceń	498
Jak obsłużyć odświeżanie?	499
Korzystanie z czcionek	500
Tworzenie obiektów klasy Font	500
Rysowanie tekstu	501
Rysowanie konturów napisów	502
Rysowanie obróconego napisu	502
Wykaz dostępnych czcionek	503
Jak wyświetlić obraz na formularzu?	504
Jak zrealizować drukowanie?	506
Wyszukiwanie i wybór drukarki	506
Inicjalizacja obiektu klasy PrintDocument	507
Kontroler wydruku	508
Drukowanie dokumentów wielostronicowych	510
Rozdział 12. Inne przestrzenie nazw	515
Teoria	515
Inne przestrzenie nazw .NET	515
Przetwarzanie wielowątkowe	515
Co to jest wątek?	516
Klasa Thread	518
Klasy stosowane do synchronizacji	521
Globalizacja	524
Informacje o kulturze	524
Informacje o kalendarzu	525
Informacje o formatach	527
Usługi systemu Windows	529
Sterowanie usługami	530
Architektura procesu usługi	531
Przestrzeń nazw System.ServiceProcess	532
Przestrzeń nazw System.Diagnostics	537
Zastosowanie asercji do sprawdzenia poprawności działania programu	537
Klasy Trace i Debug	538
Dziennik zdarzeń	539
Korzystanie z „Dziennika zdarzeń” w .NET	541
Przestrzeń nazw Text	541
Klasy reprezentujące sposoby kodowania znaków	542
Klasa StringBuilder	542
Wyrażenia regularne	543

Gotowe rozwiązania	545
Programy wielowątkowe	545
Utworzenie aplikacji	545
Inicjalizacja	546
Funkcja wątku	547
Tworzenie nowych wątków	549
Sterowanie pracą wątków	550
Tworzenie usługi systemu Windows	551
Zdefiniowanie usługi	551
Zdefiniowanie funkcji realizującej zadania usługi	553
Utworzenie i wystartowanie wątku	553
Sterowanie pracą wątku	554
Przygotowanie komponentów instalujących usługę	555
Instalacja usługi	556
Korzystanie z asercji	558
Śledzenie działania programu	559
Sterowanie śledzeniem	561
Korzystanie z „Dziennika zdarzeń”	561
Zapis w „Dzienniku zdarzeń”	561
Odczyt z „Dziennika zdarzeń”	563
Korzystanie z klasy StringBuilder	564
Używanie wyrażeń regularnych do wyszukiwania napisów w tekście	567
Bardziej zaawansowany przykład	570
Rozdział 13. Remoting — zdalne korzystanie z obiektów	573
Teoria	573
Podstawy technologii Remoting	573
Technologie zdalnego korzystania z obiektów	574
Zdalny klient i serwer	575
Aktywacja i czas życia	576
Porównanie technologii Remoting z DCOM	577
Porównanie technologii Remoting z XML Web Services	578
Zastosowanie technologii Remoting w aplikacjach wielowarstwowych	578
Kanały komunikacyjne	579
Kanał TCP	579
Kanał HTTP	579
Ujścia	580
Porty	580
Rejestracja kanału	581
Komunikacja między obiektami w technologii Remoting	581
Wiadomości	582
Szeregowanie danych	582
Formatowanie wiadomości	583
Obiekty pośredniczące	584
Kontekst wywołania	584
Zastosowanie SOAP w Remoting	585
Zdalne serwery w Remoting	585
Projekt zdalnego serwera	586
Aplikacja macierzysta	586
Konfiguracja zdalnego serwera	587
Określenie konfiguracji w programie	588
Rejestracja obiektu serwerowego	589
Obsługa wersji obiektu serwerowego	590

Zdalne klienty w Remoting.....	591
Korzystanie z obiektu serwerowego.....	591
Plik konfiguracyjny obiektu klienckiego	593
Bezpieczeństwo w Remoting.....	594
Bezpieczeństwo komunikacji.....	594
Bezpieczeństwo obiektu	595
Gotowe rozwiązania	595
Utworzenie zdalnego serwera	595
Konfiguracja serwera w kodzie programu.....	598
Utworzenie aplikacji klienckiej.....	599
Konfiguracja klienta w kodzie programu	602
Zastosowanie kanału HTTP w komunikacji ze zdalnym obiektem.....	603
Określenie czasu życia.....	604
Utworzenie obiektu aktywowanego przez klienta i określenie czasu jego życia	605
Szyfrowanie wiadomości przesyłanych przez zdalne obiekty.....	606
Rozdział 14. SOAP i XML.....	607
Teoria	607
Zaawansowany XML	607
XML i ADO.NET.....	607
Zastosowanie XML-a do trwałego przechowywania danych	608
XPath.....	608
Klasa XmlConvert	609
Schematy XML.....	609
Budowa schematu XML.....	610
Schematy wewnętrzne.....	614
Schematy zewnętrzne.....	615
Przekształcenia XML.....	615
Klasa XslTransform.....	616
Protokół SOAP	616
Koperta SOAP	617
SOAP i usługi XML Web Services.....	618
SOAP w Visual Studio .NET.....	618
DCOM a SOAP.....	619
Gotowe rozwiązania	619
Tworzenie dokumentu XML w Visual Studio .NET	619
Wyświetlenie konspektu dokumentu XML w Visual Studio .NET	622
Tworzenie schematu XSD w Visual Studio .NET.....	622
Tworzenie schematu XSD na podstawie istniejącego dokumentu XML.....	624
Sprawdzenie poprawności dokumentu XML za pomocą schematu XSD.....	626
Tworzenie pliku z przekształceniami XSLT	627
Przekształcenie dokumentu XML za pomocą XSLT	629
Rozdział 15. ADO.NET	631
Teoria	631
Wprowadzenie do ADO.NET.....	631
Porównanie ADO z ADO.NET	632
Warstwy dostępu do danych w ADO.NET	634
Klasa DataSet	636
Klasa DataTable	636
Związki między tabelami w obiekcie DataSet.....	639
Obiekty DataSet beztypowe i określonego typu	640
Ograniczenia	640

Połączenie ze źródłem danych	641
Obiekt DataAdapter	641
Obiekt Connection	642
Obiekt Command	642
Obiekt DataReader	642
Korzystanie z obiektu DataSet	643
Napełnienie obiektu DataSet danymi	643
Trzy wersje danych	643
Modyfikacja danych przechowywanych w DataSet	644
Właściwość RowState	644
Zatwierdzenie i wycofanie zmian	644
Obsługa XML w ADO.NET	644
Zapis zawartości DataSet w formacie XML	645
Odczyt XML	645
Schematy XML	645
Narzędzia bazodanowe w Visual Studio .NET	646
Korzystanie z komponentów Data	646
Generowanie za pomocą narzędzia Server Explorer kodu korzystającego z danych	647
Projekty typu Database i projektant kwerend	648
Kreator formularza operującego na danych	648
Zaawansowane zagadnienia ADO.NET	649
Zdarzenia w ADO.NET	649
Obsługa błędów w ADO.NET	650
Korzystanie w ADO.NET z procedur przechowywanych	651
Gotowe rozwiązania	651
Zbudowanie obiektu DataSet w kodzie programu	651
Zdefiniowanie związku między tabelami w obiekcie DataSet	653
Zdefiniowanie połączenia z bazą danych w oknie Server Explorer	654
Szybki dostęp do danych za pomocą komponentów Data	655
Korzystanie z komponentów Data z okna Toolbox	655
Korzystanie z komponentów z okna Server Explorer	657
Napełnienie obiektu DataSet danymi odczytywanymi z bazy danych	658
Modyfikacja danych przechowywanych w obiekcie DataSet	659
Dodanie i usunięcie wierszy	659
Wyszukiwanie danych w obiekcie DataTable	660
Zatwierdzenie i wycofanie zmian	661
Zapisanie w bazie danych zmian wykonanych w obiekcie DataSet	661
Utworzenie obiektu DataSet o określonym typie	662
Zdefiniowanie schematu dla obiektu DataSet	662
Dodanie schematu do obiektu DataSet	664
Utworzenie dokumentu XML za pomocą obiektu DataSet	665
Napełnienie obiektu DataSet zawartością dokumentu XML	666
Odczyt danych za pomocą obiektu DataReader	667
Wykonywanie instrukcji języka SQL	668
Wykonanie procedury przechowywanej	668
Korzystanie ze zdarzeń ADO.NET	670
Wykrywanie błędów w ADO.NET	671
Definiowanie kwerend za pomocą Query Designera	672
Rozdział 16. Współpraca z obiektami COM i korzystanie z Win32 API	675
Teoria	675
Współpraca z obiektami COM	676
Co to jest COM?	676
Korzystanie z obiektów COM w kodzie .NET	678

Korzystanie z kontrolki ActiveX w kodzie .NET.....	679
Korzystanie z obiektów .NET jak z obiektów COM.....	679
Korzystanie z API systemu Win32.....	680
Wybór zbioru znaków.....	681
Nadanie funkcji z biblioteki DLL innej nazwy.....	682
Gotowe rozwiązania.....	682
Użycie obiektu COM w projekcie .NET.....	682
Korzystanie z późno wiązanych obiektów COM.....	684
Korzystanie z obiektów COM w nadzorowanym C++.....	687
Użycie kontrolki ActiveX w projekcie .NET.....	688
Wywołanie nienadzorowanej funkcji z biblioteki DLL za pomocą mechanizmu PInvoke.....	690
Przykład w Visual Basicu.....	691
Przykład w języku C#.....	693
Skorowidz.....	695

Rozdział 2.

Model programowania w środowisku .NET

Teoria

Zanim przejdziemy do omówienia biblioteki klas, konieczne jest przedstawienie modelu programowania w CLR oraz języka IL, który jest bardzo nietypowym kodem bajtowym. Kody bajtowe to na ogół proste języki, czego przykładem może być kod bajtowy Javy. Zazwyczaj instrukcje takiego kodu odpowiadają instrukcjom procesora lub maszyny wirtualnej. Oznacza to, że struktura programu w języku wysokiego poziomu nie jest zachowana po przejściu do kodu bajtowego.

Natomiast IL jest językiem obiektowym. Dzięki temu konstrukcje charakteryzujące obiektowe języki wysokiego poziomu mogą być teraz użyte w każdym języku, który będzie kompilowany na IL. Jak już wspomniano, VB7 zawiera konstrukcje obiektowe. W rzeczywistości możliwości VB7 to odbicie możliwości IL. To samo dotyczy języka C# i nadzorowanego kodu w C++. W rezultacie wszystkie języki platformy .NET implementują model obiektowy zastosowany w IL. Ten właśnie model jest tematem niniejszego rozdziału. Jak się przekonamy, IL, oprócz tradycyjnych cech języków obiektowych, ma również wiele nowych właściwości. Pokażemy również, jak wszystkie te cechy znajdują odzwierciedlenie w językach platformy .NET, a zwłaszcza w C# i VB. Każdy z języków programowania ma własne osobliwości i zawiłości składni (co jest szczególnie widoczne w językach o długiej historii, takich jak VB). W związku z tym niektóre cechy obiektowe platformy .NET w jednych językach łatwiej zaimplementować niż w innych. Co więcej — platforma ta ma również takie cechy, których w pewnych językach nie da się zapisać.

Programowanie obiektowe z lotu ptaka

Niniejszy punkt to wprowadzenie do programowania obiektowego. Informacje tu zawarte mogą być przydatne dla osób nieznających tej techniki programowania i powinny umożliwić jej zrozumienie, ale nie zastąpią dobrego podręcznika. Osoby znające temat mogą pobieżnie przejrzeć ten punkt i kontynuować czytanie od punktu „Klasy”.

Programowanie obiektowe nie jest czymś nowym, gdyż powstało w środowisku akademickim w latach sześćdziesiątych. Mimo swego słusznego wieku dopiero ostatnio techniki i języki obiektowe stosowane są powszechniej. Stało się tak z kilku powodów. Po pierwsze, wczesne języki obiektowe były wykorzystywane wyłącznie w środowiskach akademickich, gdzie nacisk kładziono na techniki programowania obiektowego, a pomijano zagadnienia wydajności i łatwości użycia. Po drugie, języki te dostępne były tylko na dużych uniwersyteckich komputerach typu mainframe, więc były poza zasięgiem większości programistów.

W latach siedemdziesiątych szersze uznanie zaczął zdobywać pogląd, że podejście obiektowe może pomóc w rozwiązaniu niektórych problemów powstających podczas wytwarzania dużych systemów. Pod koniec tej dekady powstało kilka nowych obiektowych języków programowania, w tym również C++, co wraz z pojawieniem się mocnych komputerów typu desktop przyczyniło się do szerszego zastosowania tych języków.

Dzisiaj prawie nikt nie kwestionuje korzyści płynących ze stosowania technik programowania obiektowego. Niemal każdy nowy język programowania to język obiektowy, co więcej — cechy obiektowe dodawane są do tradycyjnych języków programowania.

Czy programowanie obiektowe możliwe jest tylko w obiektowym języku programowania? Odpowiedź jest zaskakująca — nie, nie tylko. Programowanie obiektowe to po prostu technika, którą z większym lub mniejszym powodzeniem można zastosować w każdym języku programowania. Można pisać obiektowy kod w nieobektowym języku programowania i vice versa — nieobektowy (innymi słowy bardzo zły) kod w języku obiektowym. Programistą obiektowym nie zostaje się z powodu pisania w języku obiektowym, tak jak nie zostaje się mechanikiem z powodu zakupu zestawu kluczy. Obiektowy język programowania tylko ułatwia wyrażenie koncepcji obiektowych w tworzonym kodzie.

Co to jest obiekt?

Bardzo trudno podać zwięzłą definicję pojęcia obiekt. Co więcej, ankieta przeprowadzona wśród programistów i informatyków pokazałaby duże rozbieżności w rozumieniu tego terminu. Oto prosta i użyteczna definicja — obiekt to coś, czemu można nadać nazwę, np.: samochód, rachunek bankowy, tablica, przycisk na formularzu. Niektórzy nazywają obiektem reprezentację rzeczy istniejącej w świecie rzeczywistym. Może to i prawda, ale dosyć rzadko spotyka się na co dzień tablice lub listy.

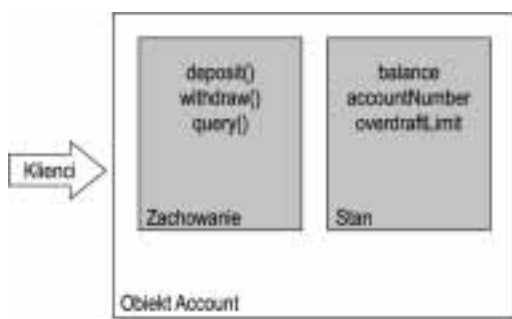
Programowanie obiektowe to styl programowania, którego stosowanie daje w efekcie program będący systemem obiektów. Program do obsługi banku może zawierać takie obiekty, jak bank, rachunek i transakcja. Program do obsługi ruchu drogowego może z kolei zawierać takie obiekty, jak droga, znak drogowy, pojazd. Obiekty te charakteryzują się pewnym zachowaniem. Wiadomo, że na rachunek można wpłacić jakąś kwotę, można wypłacić jakąś kwotę oraz sprawdzić saldo. Aby korzystać z rachunku, nie musimy wiedzieć jak on działa — gdzieś w tle są na pewno jakieś dane opisujące stan rachunku.

Dane te określają, w jaki sposób obiekt reaguje na zewnętrzne żądania. Obiekt rachunek nie pozwoli na wypłacenie kwoty większej niż saldo. Jeżeli na rachunku brak środków, żądanie wypłaty zostanie odrzucone. Kluczową kwestią jest to, że sam obiekt określa, co powinien zrobić, na podstawie danych opisujących jego stan, a dane te są zarządzane przez obiekt.

Obiekt to właśnie pewna całość, na którą składają się: zachowanie i dane opisujące stan obiektu. Na rysunku 2.1 pokazano obiekt *Account* (rachunek). Jak widać, klient ma kontakt tylko z tą częścią obiektu, która reprezentuje zachowanie. Celowo przedstawiono to w taki sposób, gdyż klient nie powinien mieć możliwości bezpośredniej zmiany stanu obiektu. W końcu bank nie byłby zadowolony, gdyby klient mógł bezpośrednio zmieniać salda swoich rachunków. Na wspomnianym rysunku *deposit()* to operacja wpłaty, *withdraw()* to operacja wypłaty, *query()* to zapytanie o saldo, *balance* to saldo, *accountNumber* to numer rachunku, a *overdraftLimit* to dopuszczalne saldo zadłużenia.

Rysunek 2.1.

Struktura obiektu



Generalnie obiekty można podzielić na trzy kategorie, w zależności od tego, czy najbardziej istotną cechą obiektu jest stan, zachowanie, czy tożsamość. Jeżeli najważniejszy jest stan, to taki obiekt nazywany jest *obiektem wartościowym* (ang. *value object*). Dobrymi przykładami takich obiektów są daty, napisy, waluty. Najistotniejszą cechą tych obiektów są dane przez nie przechowywane. Dwa obiekty typu *data* o tej samej wartości, na przykład 27 października, mogą być używane zamiennie, gdyż ich tożsamość nie jest istotna.

Jeżeli zachowanie jest ważniejsze od stanu i tożsamości, to taki obiekt nazywamy *obiektem usługowym* (ang. *service object*). Przypomnijmy sobie odprawę na dużym lotnisku — trzeba podejść do stanowiska odpraw, gdzie urzędnik sprawdzi nasz bilet i przydzieli nam miejsce w samolocie. Czy ma znaczenie, który urzędnik dokonuje naszej odprawy? Zazwyczaj nie. Dobrym przykładem obiektu usługowego może być obiekt sprawdzający numer karty kredytowej. Obiekt ten, po przekazaniu mu numeru karty kredytowej, zwraca odpowiedź stwierdzającą, czy numer jest poprawny czy też nie. Stan nie musi być pamiętany przez ten obiekt, więc każdy obiekt usługowy może wykonać tę usługę.

Najistotniejszą cechą trzeciego rodzaju obiektów jest jego tożsamość. Przykładem niech będzie wspomniany wcześniej obiekt *Account*. Jego stan (na przykład saldo) jest ważny, ważne jest również jego zachowanie (możliwość dokonywania wpłat i wypłat), ale absolutnie podstawowe znaczenie ma jego tożsamość (który to jest rachunek). Chcielibyśmy zapewne, by nasza wpłata trafiła na nasz rachunek. Taki obiekt nazywany jest *obiektem rzeczywistym* (ang. *entity object*). Tego rodzaju obiekty często reprezentują dane odczytane z bazy danych i identyfikowane jakimś kluczem.

Jakie znaczenie ma ta klasyfikacja? Pomaga w określeniu, co powinien zawierać kod klasy, gdyż pewne operacje mają sens tylko dla pewnych kategorii obiektów. Rozważmy kwestię, czy dopuszczalna jest operacja dokładnego kopiowania danego obiektu (jego klonowanie)? W przypadku obiektu wartościowego — jak najbardziej, gdyż tożsamość takiego obiektu nie ma znaczenia. Jego skopiowanie nie powinno zatem być źródłem problemów. Klonowanie oznacza powielenie stanu obiektu, a ponieważ obiekty usługowe nie mają stanu,

to ich kopiowanie nie ma większego sensu. Równie dobrze można utworzyć nowy egzemplarz. Sytuacja wygląda zupełnie inaczej, gdy chodzi o obiekty rzeczywiste, gdyż na ogół nie należy dopuszczać do tworzenia identycznych kopii takich obiektów. Fakt istnienia dwóch obiektów reprezentujących ten sam rachunek bankowy mógłby doprowadzić do katastrofy, gdyż każdy z tych obiektów mógłby mieć inną wartość salda.

Zapis klas i obiektów w kodzie

Przedstawimy teraz, w jaki sposób podstawowe pojęcia obiektowe zapisywane są w kodzie programu.

Jak już wspomniano, obiekt posiada stan i zachowanie. W obiektowych językach programowania zachowanie zapisywane jest w postaci funkcji nazywanych *metodami* (ang. *method*), a stan w postaci zmiennych nazywanych *polami* (ang. *field*). Klasa to definicja typu obiektów, na przykład *Account* (rachunek) lub *Car* (samochód), zawierająca zmienne i funkcje składające się na obiekt.

Poniżej pokazano definicję bardzo prostej klasy *Account* w języku Visual Basic 7.

```
Public Class Account
    Private balance As Double
    Private accountNo As Integer

    Public Function deposit(ByVal amount As Double) As Boolean
        balance = balance + amount
        Return True
    End Function

    Public Function withdraw(ByVal amount As Double) As Boolean
        If (balance-amount < 0) Then
            Return False
        End If
        balance = balance - amount
        Return True
    End Function

    Public Function query() As Double
        Return balance
    End Function
End Class
```

Do zrozumienia tego kodu nie jest konieczna dogłębna znajomość Visual Basica. W pierwszym wierszu rozpoczyna się definicja klasy *Account*, która kończy się tekstem *End Class*. W klasie zdefiniowane są dwie zmienne do przechowywania stanu obiektu. Pierwsza z nich — *balance* — to zmienna zmiennopozycyjna, która będzie przechowywała saldo. Druga natomiast — *accountNo* — to zmienna całkowita, która będzie przechowywała numer rachunku. Zmienne są zadeklarowane jako prywatne (służy do tego słowo *Private*), co oznacza, że nie można z nich korzystać na zewnątrz klasy *Account*.

Po deklaracjach zmiennych znajdują się definicje metod: *deposit* (wpląć), *withdraw* (wypłąć) i *query* (zapytaj). Metody te zadeklarowane są jako publiczne (służy do tego słowo *Public*), co oznacza, że można z nich korzystać na zewnątrz klasy. W przykładzie widać, w jaki sposób funkcje te operują na saldzie rachunku. Metoda *deposit* dodaje przekazaną kwotę do salda rachunku. Dla uproszczenia nie sprawdza się poprawności

przekazanej kwoty, ale łatwo takie sprawdzenie dodać. W metodzie `withdraw` sprawdza się, czy wystąpi debet. Jeżeli nie, to zmniejsza się saldo. Metoda `query` zwraca wartość salda. Używając tych trzech metod, klienci mogą manipulować obiektami `Account`, nie mogą natomiast bezpośrednio zmieniać wartości salda. Obiekt może sprawdzić, czy żądania zgłaszane przez klientów są dozwolone.

Po zdefiniowaniu klasy możemy z niej korzystać, tworząc obiekty. W przykładowym kodzie w VB zamieszczonym poniżej tworzy się obiekty klasy `Account` i wywołuje ich metody. Nie należy tu zwracać uwagi na składnię, ale skoncentrować się na interakcji z obiektami.

```
Dim myAccount As New Account
Dim yourAccount As New Account

myAccount.deposit(1000)           * wpłacić 1000 na mój rachunek
yourAccount.deposit(100)         * wpłacić 100 na twój rachunek

myAccount.withdraw(500)          * wypłacić 500 - OK
myAccount.query()                * wynikiem jest 500

yourAccount.withdraw(500)        * niepowodzenie!
```

Pierwsze dwa wiersze to żądanie utworzenia dwóch obiektów `Account` o nazwach `myAccount` (mój rachunek) i `yourAccount` (twój rachunek). Ponieważ wcześniej wystąpiła definicja klasy `Account`, to wiadomo, co to jest `Account` i jak utworzyć te obiekty.

W trzecim wierszu wpłaca się 1000 na pierwszy rachunek, a w czwartym — 100 na drugi rachunek. Proszę zwrócić uwagę na sposób zapisu wywołań metod. Wywołanie rozpoczyna się od nazwy obiektu, którego chcemy użyć; po niej występuje nazwa metody, którą obiekt powinien wykonać. Jest to sposób zapisu typowy dla obiektowych języków programowania. W dalszej części rozdziału pokazano, że sposób zapisu w C# jest bardzo podobny.

W kolejnym wierszu wykonuje się wypłatę kwoty 500 z obiektu `myAccount`. Operacja kończy się sukcesem, ponieważ saldo wynosi 1000. Następnie odczytywane jest saldo; wynikiem tej operacji jest bieżąca wartość salda. W ostatnim wierszu mamy próbę wykonania wypłaty kwoty 500 z drugiego rachunku. Próba ta kończy się niepowodzeniem, gdyż saldo rachunku wynosi 100. Innymi słowy, te dwa obiekty w różny sposób reagują na próbę wykonania tej samej operacji (wypłaty kwoty 500), gdyż stan tych obiektów jest różny.

W trakcie dokładniejszej analizy kodu metod `deposit`, `withdraw` i `query` może nasunąć się pytanie — skąd w treści metody wiadomo, którego obiektu ona dotyczy? W kodzie nie ma informacji o obiekcie, którego saldo ulega zmianie, a jednak wpłaty i wypłaty dotyczą właściwego rachunku. Otóż referencja do obiektu jest niejawnie przekazywana do każdej metody i ta referencja jest wykorzystywana podczas wszystkich odwołań do składowych klasy. Tak więc, gdy metoda `deposit` jest wywoływana z obiektu `myAccount`, to jej działanie można by zapisać w następujący sposób:

```
Public Function deposit(myAccount, ByVal amount As Double) As Boolean
    myAccount.balance = myAccount.balance + amount
    Return True
End Function
```

Oczywiście nie widać tego w kodzie, ale metoda zawsze „wie”, z którego obiektu została wywołana. W wielu językach obiektowych w metodzie można odwołać się do obiektu, z którego ta metoda została wywołana. W Visual Basicu służy do tego słowo kluczowe `Me`, a w C# i w C++ słowo `this`.

Te same zasady obowiązują oczywiście w innych językach programowania, co ilustruje zamieszczony poniżej kod klasy `Account` zapisany w języku C#. Pomijając drobne różnice w składni, zauważymy, że struktura kodu jest niemal identyczna w obu językach.

```
public class Account {
    private double balance;
    private int accountNo;

    public bool deposit(double amount) {
        balance = balance + amount;
        return true;
    }

    public bool withdraw(double amount) {
        if (balance-amount < 0) return false;

        balance = balance - amount;
        return true;
    }

    public double query() {
        return balance;
    }
}
```

Kod, w którym korzysta się z tej klasy, zapisany w C# jest również bardzo podobny do kodu w VB — inny jest tylko sposób tworzenia obiektów.

```
Account myAccount = new Account();
Account yourAccount = new Account();

myAccount.deposit(1000);           // wpłać 1000 na mój rachunek
yourAccount.deposit(100);         // wpłać 100 na twój rachunek

myAccount.withdraw(500);          // wypłać 500 - OK
myAccount.query();                // wynikiem jest 500

yourAccount.withdraw(500);        // niepowodzenie!
```

Dziedziczenie i polimorfizm

Dotychczas przedstawiono dwie ważne zasady programowania obiektowego: hermetyzację i ukrywanie danych. *Hermetyzacja* (ang. *encapsulation*) to połączenie danych i funkcji w pewną całość, zwaną obiektem, a *ukrywanie danych* (ang. *data hiding*) to ograniczenie dostępu do zmiennych przechowujących stan obiektu¹. Omówimy teraz dwie inne bardzo ważne cechy występujące w każdym „prawdziwym” języku obiektowym, a mianowicie *dziedziczenie* (ang. *inheritance*) i *polimorfizm* (ang. *polymorphism*).

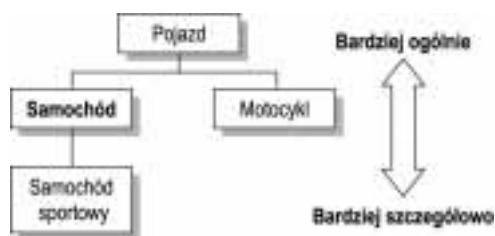
¹ W literaturze na ogół używa się bardziej ogólnego pojęcia, a mianowicie *ukrywanie informacji* (ang. *information hiding*). Jest to zasada, która mówi, że programista korzystający z jakiegoś składnika oprogramowania może go poprawnie zastosować bez znajomości jego budowy lub implementacji. Zobacz, na przykład: Subieta K.: „Słownik terminów z zakresu obiektowości”, Warszawa 1999. — *przyp. tłum.*

Obiekty świata rzeczywistego często są klasyfikowane i przydzielane do różnych typów czy też kategorii. Na przykład samochód sportowy to samochód, ale również pojazd. Można więc powiedzieć, że samochód sportowy należy do trzech typów — jest samochodem sportowym, jest samochodem i jest pojazdem. W zależności od okoliczności, możemy użyć każdego z tych typów, mówiąc o samochodzie sportowym. Gdybyśmy na przykład mieli policzyć wszystkie samochody na parkingu, to uwzględnilibyśmy również samochody sportowe, gdyż są samochodami. Taka umiejętność dokonywania hierarchicznej klasyfikacji obiektów jest bardzo naturalna dla ludzi oraz jest bardzo przydatna w programowaniu.

Dziedziczenie pozwala zapisać w kodzie związki „jest” (ang. *is-a relationship*), zachodzące między klasami. Na rysunku 2.2 przedstawiono prostą hierarchię dziedziczenia, która pokazuje, w jaki sposób różne typy pojazdów są ze sobą powiązane.

Rysunek 2.2.

Hierarchia dziedziczenia



W języku Visual Basic związki te zapisuje się w kodzie w sposób następujący:

```

Public Class Pojazd
    * kod klasy Pojazd
End Class

Public Class Samochod
    Inherits Pojazd
    * kod klasy Samochod
End Class

Public Class SamochodSportowy
    Inherits Samochod
    * kod klasy SamochodSportowy
End Class
  
```

Jak widać w przykładzie, do wyrażenia relacji dziedziczenia używane jest słowo kluczowe `Inherits`. Klasa taka jak `Pojazd` nazywana jest *klasą bazową* (ang. *base class*) lub *nadklasą* (ang. *superclass*), a klasy takie jak `Samochod` i `SamochodSportowy` nazywane są *klasami pochodnymi* (ang. *derived class*) lub *podklasami* (ang. *subclass*).

Widać również, że zakodowanie relacji dziedziczenia jest proste, ale prawdziwą sztuką w programowaniu obiektowym, tak samo zresztą jak w życiu, jest umiejętne zdefiniowanie relacji.

Jakie korzyści daje stosowanie dziedziczenia? Rozważmy następujący przykład. Świadek wypadku stwierdził, że wyprzedził go pojazd. Opis ten nie jest zbyt dokładny — mógł to być zarówno motocykl, samochód, jak i samochód sportowy. Gdyby powiedział, że był to samochód, to mógł mieć na myśli samochód lub samochód sportowy, ale nie motocykl — motocykl nie jest rodzajem samochodu. Podobnie można postąpić w kodzie

programu. Ponieważ samochód jest pojazdem, to obiekt samochód może wystąpić w kodzie wszędzie tam, gdzie zapisano, że ma wystąpić obiekt Pojazd. Ma to ogromne znaczenie. Można napisać program korzystający z klas Samochod i Motocyk1 i korzystać z tych klas wszędzie tam, gdzie wymagane są obiekty Pojazd. Gdy zajdzie taka potrzeba, można później dodać do programu inny rodzaj pojazdu, na przykład autobus. I znowu — ponieważ autobus jest pojazdem, może wystąpić wszędzie tam, gdzie wymagane są obiekty Pojazd. Oznacza to, że można dodawać nowe funkcje do programu bez konieczności wprowadzania zmian w istniejącym już kodzie. Jest to bardzo istotne podczas tworzenia dużych programów.

Te same zasady obowiązują w innych językach programowania, co widać na poniższym przykładzie w języku C#.

```
class Pojazd {
    // kod klasy Pojazd
}

class Samochod : Pojazd {
    // kod klasy Samochod
}

class SamochodSportowy : Samochod {
    // kod klasy SamochodSportowy
}
```

Omówiliśmy już dziedziczenie, ale co z polimorfizmem? Jest to bardzo ważna i użyteczna cecha języków obiektowych. Słowo polimorfizm pochodzi z greki i oznacza „wiele kształtów”. Tutaj oznacza, że dany obiekt należy jednocześnie do wielu typów.

Założmy, że naszym zadaniem jest napisanie programu do obsługi grafiki. W programie zdefiniowaliśmy klasę Shape (figura), od której będą pochodziły wszystkie klasy opisujące inne figury. W każdej z tych klas powinna znajdować się metoda, której zadaniem będzie narysowanie tej właśnie figury. Definiujemy więc w każdej klasie metodę Draw (rysuj). Sygnatury tych metod są identyczne, ale treść w każdej klasie jest inna. Pseudokod w VB zamieszczony poniżej ilustruje przyjęte rozwiązanie².

```
Public Class Circle
    Inherits Shape
    Overrides Public Sub Draw()
        ' kod do rysowania okręgu
    End Sub
End Class

Public Class Square
    Inherits Shape
    Overrides Public Sub Draw()
        ' kod do rysowania kwadratu
    End Sub
End Class
```

² Uwaga: W klasie bazowej Shape metoda Draw musi być metodą wirtualną (przez użycie słowa kluczowego `Override`) lub czystą metodą wirtualną (przez użycie słowa kluczowego `MustOverride`). W drugim przypadku klasa Shape musi być klasą abstrakcyjną (przez użycie słowa kluczowego `MustInherit`). W przeciwnym wypadku zawsze wywołana byłaby metoda Draw z klasy Shape. Pojęcie „metoda wirtualna” wyjaśnione jest w dalszej części rozdziału — *przyj. tłum.*

Co daje takie rozwiązanie? Wiemy już, że Circle i Square to klasy pochodne od Shape. Ponadto w każdej klasie pochodnej od Shape występuje metoda Draw, która potrafi narysować odpowiednią figurę. Metodę pozwalającą narysować dowolną figurę można zwięźle zapisać następująco:

```
Public Sub DrawShape(ByRef s As Shape)
    s.Draw()
End Sub
```

Niezależnie od tego, jaką figurę przekaże się do metody DrawShape, wykonana zostanie właściwa metoda Draw. Ma to duże znaczenie dla konserwacji i rozwoju programu, gdyż można do hierarchii klas dodawać nowe figury i w dalszym ciągu korzystać z metody DrawShape, bez zmiany jej treści.

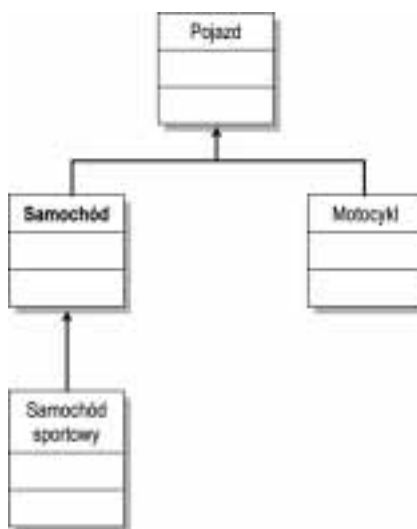
W wielu obiektowych językach programowania tak działające funkcje nazywane są *funkcjami wirtualnymi* (ang. *virtual function*), a mechanizm umożliwiający ich działanie w opisany powyżej sposób nazywany jest *późnym wiązaniem* (ang. *late binding*). Użyto tu słowa *późne*, gdyż dopiero w czasie działania programu okaże się, która metoda naprawdę zostanie wywołana w treści metody DrawShape. Jak się przekonamy, wszystkie języki platformy .NET pozwalają korzystać z funkcji wirtualnych.

Mała dygresja na temat UML

Osoby projektujące oprogramowanie obiektowe z pewnością zetkną się ze zuniifikowanym językiem do modelowania — UML (ang. *Unified Modeling Language*). W olbrzymim skrócie, UML to notacja do zapisu struktury i działania programów obiektowych. Znamość UML jest dziś nieodzowna do tworzenia oprogramowania obiektowego.

Zainteresowanych językiem UML odsyłamy do literatury. Dostępnych jest wiele dobrych książek na ten temat, jedną z nich jest „UML Distilled” autorstwa Martina Fowlera i Kendala Scotta (Addison Wesley, 1999). Rysunek 2.3 zawiera naszą hierarchię klas wyrażoną w notacji UML.

Rysunek 2.3.
Diagram klas
w notacji UML



W UML klasę reprezentuje prostokąt, którego najwyżej położona sekcja zawiera nazwę klasy. Pozostałe dwie sekcje zawierają pola i metody klasy. Strzałka prowadzi od klasy pochodnej do jej klasy bazowej.

Interfejsy

W tym krótkim wprowadzeniu do programowania obiektowego musimy jeszcze omówić pojęcie interfejsów. Są one powszechnie używane w klasach bibliotek .NET i z tego też powodu musimy wyjaśnić, czym one są i jak się z nich korzysta.

Przyjmijmy następującą uproszczoną definicję — jeśli obiekt to coś, czemu można nadać nazwę, to interfejs reprezentuje jakieś zachowanie, któremu można nadać nazwę. Przypuśćmy, że stworzymy oprogramowanie dla wydawnictwa, takiego jak Helion. Z pewnością w tym oprogramowaniu pojawi się wiele klas, reprezentujących rzeczy lub pojęcia pojawiające się w pracy wydawnictwa, takie jak: książki, katalogi, zamówienia, faktury. Ich wspólną cechą jest to, że są „drukowalne” (można je wydrukować), mimo że nie łączą ich relacje dziedziczenia. Interfejs jest sposobem zapisu w kodzie zachowania wspólnego dla kilku klas. Jest to jedna lub więcej funkcji, które klasa musi zaimplementować. Na przykład, interfejs Printable (drukowalna) może zawierać funkcję print (drukuj), a klasa może być uznana za „drukowalną”, jeżeli będzie implementować funkcję print.

W poniższym kodzie pokazano, w jaki sposób definiuje się i implementuje interfejsy w języku VB.

```
Interface Printable
    Sub print()
End Interface

Public Class Account
    Implements Printable

    Sub print() Implements Printable.print
        ' kod drukujący informacje o rachunku
    End Sub

    Private balance As Double
    Private accountNo As Integer

    ' pominięto resztę definicji klasy
End Class
```

Klasa Account jest teraz „drukowalna”, tak więc obiekt klasy Account może wystąpić wszędzie tam, gdzie wymagane są obiekty „drukowalne”.

Interfejsy powstają wówczas, gdy stwierdza się, że pewne zachowanie powinno być implementowane przez klasy, ale nie da się go tak samo zdefiniować dla wszystkich tych klas. Interfejs jest konstrukcją pozwalającą wyrazić myśl „jeśli chcemy wykonać to zadanie, to musimy zrobić to w następujący sposób: ...”.

Po omówieniu podstawowych pojęć programowania obiektowego przejdziemy teraz do informacji o .NET.

Klasy

Klasa jest podstawowym pojęciem w programowaniu obiektowym. Pokażemy teraz, w jaki sposób klasy są implementowane w CLR. Jak już wspomniano, platforma .NET jest dosyć nietypowa, gdyż IL, czyli jej język pośredni, to język obiektowy, natomiast w innych środowiskach języki pośrednie bardziej przypominają języki maszynowe.

Ponieważ w CLR zaimplementowano klasy i inne konstrukcje obiektowe, część z nich jest dostępna we wszystkich językach platformy .NET. Z drugiej strony języki programowania muszą implementować cechy obiektowe w sposób zgodny z CLR.

Części składowe klasy

Klasa w środowisku .NET zawiera następujące części składowe:

- ♦ metody,
- ♦ pola,
- ♦ właściwości,
- ♦ zdarzenia.

Należy pamiętać, że sposób ich implementacji (a nawet ich dostępność) zależy od używanego języka programowania.

Metody to funkcje realizujące „aktywność” klasy. Każda metoda ma nazwę i typ wyniku i może mieć parametry. Jeżeli typem wyniku jest `void`, to znaczy, że metoda nie zwraca wartości, co odpowiada konstrukcji `Sub` w VB. W klasie mogą wystąpić konstruktory, czyli metody wywoływane podczas tworzenia obiektu. Ich zadaniem jest zainicjalizowanie nowo utworzonego obiektu. W klasie może też wystąpić finalizator (ang. *finalizer*), czyli metoda, której zadaniem jest „posprzątanie” po obiekcie, gdy zajmowana przez niego pamięć jest zwalniana przez program odzyskiwania pamięci.

Pola przechowują dane należące do obiektu, którymi mogą być referencje do innych obiektów lub dane typów bezpośrednich. Ukrywanie danych składających się na stan obiektu przed innymi obiektami jest jedną z zasad programowania obiektowego. Obiekty w .NET mają *właściwości* (ang. *property*). Za ich pomocą klienci mogą korzystać ze stanu obiektu, nie mając do tego stanu bezpośredniego dostępu.

W poniższym kodzie pokazano, w jaki sposób definiuje się właściwości w języku VB³.

```
Property Color() As String
    Get
        Color = myColor
    End Get
    Set
        myColor = Value
    End Set
End Property
```

³ W tym przykładzie założono, że w klasie zdefiniowane jest pole o nazwie `myColor` przechowujące kolor danego obiektu — *przyj. tłum.*

Definicja właściwości przypomina definicję metody. Jej ciało zawiera część Get oraz może zawierać część Set. Część Get zwraca wartość właściwości, a część Set używana jest do jej ustawienia. Specjalna zmienna o nazwie `Value` oznacza wartość przekazaną podczas ustawiania właściwości. Mimo że właściwość implementowana jest jak metoda, to klient korzysta z niej jak z pola. Przykład:

```
MyObject.Color = "red"
```

Programiści znający VB natychmiast zauważą podobieństwo tej konstrukcji do konstrukcji Property Get i Property Set występujących w poprzednich wersjach VB. Należy podkreślić, że właściwości takie mogą być w .NET użyte w każdym języku programowania, również w C# i C++. Ponadto, z właściwości w klasie napisanej w jednym z języków platformy .NET można korzystać w dowolnym innym języku tej platformy.

Właściwości nie muszą li tylko być używane do obsługi stanu obiektu, co stanowi fundamentalną różnicę między właściwościami a polami. Można, na przykład, zdefiniować właściwość tylko do odczytu, która pobiera swoją wartość z bazy danych lub innego źródła.

.NET został zaprojektowany do tworzenia aplikacji z graficznym interfejsem użytkownika i aplikacji dla sieci WWW. Krytycznym elementem takich aplikacji są *zdarzenia* (ang. *event*) i ich obsługa. Z tego powodu Microsoft postanowił, że obsługa zdarzeń będzie częścią CLR. Klasa obsługująca zdarzenia może informować zainteresowanych odbiorców, że zdarzenie zaszło. Tak więc odbiorcy mogą być subskrybentami interesujących ich zdarzeń zdefiniowanych w klasie, która jest źródłem tych zdarzeń. Typowym przykładem jest przycisk umieszczony na formularzu. Przycisk może zgłaszać zdarzenie `Clicked` (przyciśnięto), a formularz może być subskrybentem tego zdarzenia. Ilekroć przycisk zostaje przyciśnięty, formularz zostanie o tym poinformowany. Zdarzenia to podstawowa technika przy tworzeniu graficznego interfejsu użytkownika, ale można z nich korzystać w innych celach.

Modyfikatory klas

W tabeli 2.1 zawarto modyfikatory, którymi mogą być oznaczone klasy w .NET. Sposób zapisu tych modyfikatorów jest różny w poszczególnych językach programowania. W tabeli 2.2 znajdują się modyfikatory, które mogą mieć składowe klas.

Tabela 2.1. Modyfikatory klas w środowisku .NET

Modyfikator	Opis
<code>sealed</code>	Oznacza klasę, która nie może być klasą bazową.
<code>implements</code>	Oznacza klasę implementującą jeden lub więcej interfejsów.
<code>abstract</code>	Oznacza klasę zawierającą jedną lub więcej metod abstrakcyjnych. Nie można tworzyć obiektów klas abstrakcyjnych.
<code>inherits</code>	Oznacza klasę pochodną innej klasy.
<code>public</code>	Oznacza klasę widoczną na zewnątrz podzespołu.
<code>internal</code>	Oznacza klasę niewidoczną na zewnątrz podzespołu.

Tabela 2.2. Modyfikatory składowych klas w środowisku .NET

Modyfikator	Opis
abstract	Metoda niemająca ciała jest metodą abstrakcyjną. Klasa zawierająca choć jedną metodę abstrakcyjną jest klasą abstrakcyjną. Klasa pochodna klasy abstrakcyjnej musi zawierać implementacje metod abstrakcyjnych.
private, public, protected, internal, protected or internal	Składowa oznaczona modyfikatorem <code>private</code> jest dostępna tylko w klasie, w której jest zdefiniowana. Składowa oznaczona modyfikatorem <code>public</code> jest powszechnie dostępna. Składowa oznaczona modyfikatorem <code>protected</code> dostępna jest w klasie, w której jest zdefiniowana i w jej podklasach ⁴ . Składowa oznaczona modyfikatorem <code>internal</code> jest dostępna w klasach zdefiniowanych w tym samym podzespole. Składowa oznaczona modyfikatorem <code>protected or internal</code> dostępna jest w podklasach i w klasach zdefiniowanych w tym samym podzespole.
sealed	Składowa oznaczona modyfikatorem <code>sealed</code> nie może być przesłonięta w podklasach.
override	Składowa oznaczona modyfikatorem <code>override</code> przesłania składową odziedziczoną po klasie bazowej.
static	Składowa oznaczona modyfikatorem <code>static</code> jest częścią składową klasy, a nie poszczególnych obiektów. Jest ona współdzielona przez wszystkie obiekty danej klasy. Z takiej składowej można korzystać nawet wtedy, gdy nie istnieje żaden obiekt tej klasy.
overloads	Modyfikator <code>overloads</code> używany jest do oznaczenia składowych o takich samych nazwach, ale różniących się listą parametrów.
virtual	Składowa oznaczona modyfikatorem <code>virtual</code> wywoływana jest za pomocą techniki późnego wiązania. Wybór wywoływanej metody następuje na podstawie typu obiektu, z którego metoda jest wywoływana, a nie na podstawie typu zmiennej referencyjnej, za pośrednictwem której sięga się do obiektu.

Typy referencyjne i bezpośrednie

W większości języków programowania zmienne typów prymitywnych, takich jak typ całkowity lub znakowy, tworzone są na stosie. Podczas przekazywania tych zmiennych, ich wartości są kopiowane. Natomiast zmienne obiektowe zazwyczaj tworzone są na stercie, a dostęp do nich jest możliwy przez referencje. To właśnie referencje, a nie same obiekty, są kopiowane podczas przekazywania zmiennych obiektowych.

Typy w CLR dzieli się na dwie kategorie.

1. *Typy bezpośrednie* (ang. *value type*) — typy pochodne od `System.ValueType`. Zmienne tych typów przekazywane są przez wartość. Przechowywane są równie efektywnie, jak zmienne prymitywne w innych językach programowania, ale zarazem są to obiekty zawierające metody. Należy pamiętać, że nowe typy bezpośrednie mogą być podtypami wyłącznie typu `System.ValueType`, nie innych typów bezpośrednich zdefiniowanych w przestrzeni nazw `System`.
2. *Typy referencyjne* (ang. *reference type*) — dobrze znane nam klasy, dostęp do obiektów tych typów (klas) jest możliwy przez referencje — stąd też ich nazwa. Definiując nowe typy powinniśmy rozważyć, w jaki sposób będziemy korzystać z obiektów tych typów i czy bardziej efektywne będzie przekazywanie ich przez wartość, czy też przez referencję. W zależności od wyniku tych rozważań, możemy zdefiniować je jako typ referencyjny lub bezpośredni.

⁴ Składowa taka nazywana jest *składową chronioną* (ang. *protected member*) — *przyp. tłum.*

Systemowe typy bezpośrednie, na przykład `System.Int32`, są dokładnymi odpowiednikami językowych typów prymitywnych. I tak, `Int32` jest odpowiednikiem typu `int` w języku C# oraz typu `Integer` w języku VB. Zawsze można użyć typu systemowego, jeżeli z jakiegoś powodu nie chcemy zastosować jego odpowiednika w danym języku programowania.

Typy bezpośrednie, takie jak `System.Int32`, zawsze przyprawiły projektantów obiektowych języków programowania o ból głowy. Kuszące jest posiadanie zunifikowanego systemu typów, w którym każdy element jest obiektem, ale takie rozwiązanie rodzi jeden istotny problem — brak wydajności. Gdyby bowiem każda liczba całkowita (lub znak) występowała w programie jako obiekt podlegający automatycznemu zarządzaniu pamięcią, a wykonanie każdej operacji (również dodanie dwóch liczb) oznaczałoby wywołanie metody, to programy pisane w takim języku byłyby bardzo niewydajne. Rozwiązanie alternatywne, w którym typy podstawowe obsługiwane są w specjalny sposób i traktowane jako ciągi bajtów przechowujących dane, a nie obiekty, również nie jest dobre. W tym rozwiązaniu operacje podstawowe, takie jak dodawanie liczb, są znacznie wydajniejsze, ale powstaje niespójny system typów.

W .NET zastosowano trzecie podejście, które łączy zalety obu wymienionych, a zarazem jest pozbawione ich wad. W .NET wartości typów bezpośrednich traktowane są jak obiekty tylko wówczas, gdy jest to konieczne. Tak więc, zadeklarowana zmienna całkowita zajmuje w pamięci kilka bajtów, dokładnie tak samo, jak w nieobiektowym języku programowania. Dodawanie dwóch zmiennych całkowitych wykonywane jest za pomocą zwykłych instrukcji arytmetycznych, a nie przez wywołanie metody. Jeżeli natomiast zmienna całkowita przekazywana jest do metody, której parametr jest obiektem, to zmienna ta zostanie automatycznie zamieniona w obiekt przez mechanizm zwany *opakowywaniem* (ang. *boxing*).

Opakowywanie wartości polega na utworzeniu obiektu przechowującego wartość i informacje o jej typie. Załóżmy, że napisano w C# metodę, której argumentem jest referencja do obiektu, ale wywołując ją przekazano zmienną całkowitą, jak pokazano to poniżej.

```
int n = 3;
foo(n);
...
public static void foo(object o) { ... }
```

Tak wyglądać będzie kod IL uzyskany w wyniku działania programu ILDASM:

```
.method public hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // Code size      14 (0xe)
    .maxstack 1
    .locals imit (int32 V_0)
    IL_0000: ldc.i4.3
    IL_0001: stloc.0
    IL_0002: ldc.oc.0
    IL_0003: box      [mscorlib]System.Int32
    IL_0008: call    void Class1::foo(object)
    IL_000d: ret
} // end of method Class1::Main
```

Pierwszy wyróżniony wiersz zawiera deklarację zmiennej całkowitej o nazwie *n*. Wiersze opatrzone etykietami IL_0002 i IL_0003 zawierają kod dokonujący opakowania zmiennej *n* w obiekt typu `System.Int32`, w następnym wierszu znajduje się wywołanie metody `foo`.

Mechanizm odwrotny, a więc wyłuskiwania wartości typu bezpośredniego z obiektu, nazywany jest *rozpakowywaniem* (ang. *unboxing*). Na ogół stosowane jest wtedy rzutowanie (ang. *cast*). Załóżmy, że treść metody `foo` wygląda następująco:

```
public static void foo(object o)
{
    int n = (int)o;
    ...
}
```

Wówczas wygenerowany kod IL będzie miał taką postać, jak pokazano poniżej. Widać wyraźnie, w jaki sposób wartość jest rozpakowywana i przypisywana zmiennej całkowitej.

```
.locals init (int32 V_0)
IL_0000: ldarg.0
IL_0001: unbox      [mscorlib]System.Int32
IL_0006: ldind.i4
IL_0007: stloc.0
```

Struktury

W .NET oprócz klas można korzystać również ze struktur. Ich nazwa pochodzi od słowa kluczowego `struct` w języku C, oznaczającego złożony (lub rekordowy) typ danych. Poniżej pokazano przykład definicji struktury opisującej punkt w języku VB.

```
Public Structure Point
    Private myX As Integer
    Private myY As Integer

    Property X() As Integer
        Get
            X = myX
        End Get
        Set
            myX = value
        End Set
    End Property

    Property Y() As Integer
        Get
            Y = myY
        End Get
        Set
            myY = value
        End Set
    End Property

    Sub New(ByVal a As Integer, ByVal b As Integer)
        myX = a
        myY = b
    End Sub
End Structure
```

Poniżej podano definicję tej struktury w języku C#. Można zauważyć, że konstrukcje językowe w VB i C# są bardzo podobne.

```
struct Point
{
    private int x,y;

    public int X
    {
        get { return x; }
        set { x = value; }
    }
    public int Y
    {
        get { return y; }
        set { y = value; }
    }

    public Point(int a, int b)
    {
        x = a; y = b;
    }
}
```

Struktury bardzo przypominają klasy, gdyż tak jak one mogą zawierać pola, metody, właściwości i zdarzenia. Jest jednak istotna różnica między nimi — struktury to typy bezpośrednie, a klasy to typy referencyjne. Ma to następujące konsekwencje:

- ◆ struktury tworzone są na stosie, a nie na sterwie,
- ◆ dostęp do nich jest bezpośredni, a nie za pośrednictwem referencji, tym samym nie podlegają automatycznemu zarządzaniu pamięcią,
- ◆ w wywołaniach metod struktury przekazywane są przez wartość.

Te różnice sprawiają, że reprezentowanie typów bezpośrednich za pomocą struktur jest bardziej efektywne aniżeli za pomocą klas.

Istnieje wiele innych różnic między strukturami a klasami, wymienimy trzy z nich. Po pierwsze, do struktur nie można zastosować mechanizmu dziedziczenia. Struktura nie może więc pochodzić od klasy lub struktury oraz nie może być klasą bazową. Może za to implementować interfejsy w taki sam sposób jak klasy.

Druga różnica dotyczy konstruktorów. W strukturze mogą wystąpić konstruktory, ale muszą one mieć parametry. Gdy w strukturze zostanie użyty konstruktor bezparametrowy, zgłoszony zostanie błąd kompilacji. Tak samo zostanie potraktowany w strukturze finalizator.

Kolejna różnica polega na tym, że składowe struktury nie mogą być inicjalizowane — przykładowy kod w C# podany poniżej jest błędny.

```
struct Point
{
    private int x=0, y=0;    // błąd!
    ...
}
```

Aby zainicjalizować strukturę, należy zdefiniować w niej konstruktor.

Dziedziczenie

W .NET dopuszczalne jest tylko *pojedyncze dziedziczenie* (ang. *single inheritance*), a więc takie, w którym klasa może mieć tylko jedną klasę bazową. W innych językach obiektowych, na przykład w C++, klasa może mieć wiele klas bazowych, co nazywane jest *dziedziczeniem wielokrotnym* (ang. *multiple inheritance*). W praktyce okazało się, że dziedziczenie wielokrotne może być źródłem pewnych problemów. Z tego też powodu w wielu językach obiektowych mechanizm ten nie występuje.

Pojedyncze dziedziczenie może być kłopotliwe dla programistów C++, gdyż w nadzorowanym kodzie w C++ w .NET klasy mogą mieć tylko jedną klasę bazową.

Często wielokrotnie dziedzicznie używane jest do wyrażenia wielu związków „jest”, a nie do dziedziczenia kodu po wielu klasach bazowych. W takiej sytuacji można zastosować interfejsy i za ich pomocą zamodelować związki, które miały być zapisane w postaci wielokrotnego dziedziczenia w C++.

Interfejsy

O interfejsach i ich roli w programowaniu obiektowym była już mowa. Interfejsy mają zasadnicze znaczenie w architekturze .NET. Można z nich korzystać we wszystkich dostępnych językach tej platformy.

Rozważmy przykład interfejsu `ICloneable` (klonowalny; taki, który można sklonować). W klasie bazowej `Object` znajduje się metoda `MemberwiseClone()`. Metoda ta tworzy dokładną kopię obiektu, tyle że jest to tak zwane *płytkie kopiowanie* (ang. *shallow copying*). Jeżeli obiekt zawiera zmienne będące referencjami do innych obiektów, to w trakcie płytkiego kopiowania skopiowane zostaną te zmienne, a nie obiekty przez nie wskazywane, jak pokazano to na rysunku 2.4. Kopiowany jest tylko obiekt, z którego wywołano tę metodę. Umieszczono ją w klasie `Object`, by taką operację można było zastosować do obiektu każdej klasy.

Rysunek 2.4.
*Kopiowanie płytkie
a kopiowanie
głębokie*



Co należy zrobić w sytuacji, gdy chcemy, żeby podczas kopiowania obiektu sklonowane zostały również te obiekty, do których się on odwołuje? Taki rodzaj kopiowania nazywany jest *kopiowaniem głębokim* (ang. *deep copying*). Ponieważ .NET nie zna struktury naszego obiektu, musimy sami zaimplementować taką operację. Innymi słowy, wiadomo co należy zrobić, ale nie wiadomo w jaki sposób. W tym celu zdefiniowano interfejs `ICloneable` zawierający jedną metodę `Clone()`. Jeżeli chcemy wykonywać głębokie kopie naszego obiektu, to jego klasa musi implementować interfejs `ICloneable`, a tym samym metodę `Clone()`. Poniżej podano definicję interfejsu `ICloneable` w języku Visual Basic.


```
Public Interface ICloneable
    Function Clone() As Object
End Interface
```

Natomiast definicja tego interfejsu w C# ma następującą postać:

```
public interface ICloneable {
    object Clone();
}
```

Zgodnie z powszechnie przyjętą umową, nazwy interfejsów rozpoczynają się od litery I. Łatwo wówczas stwierdzić, czy dana nazwa oznacza interfejs czy klasę. Jest to tylko umowa, a nie wymóg. Powyższa definicja interfejsu zawiera sygnaturę jednej bezparametrowej metody `Clone`, której wynik jest typu `Object`. Oczywiście definicja interfejsu powinna również zawierać opis wyjaśniający jego semantykę, tak aby było jasne, w jaki sposób dany interfejs powinien być zaimplementowany.

Delegacje

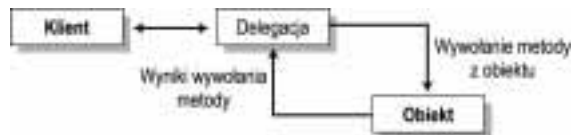
Delegacje to obiekty obsługiwane przez środowisko uruchomieniowe platformy .NET i służące do tego samego celu co wskaźniki do funkcji w C i C++.

Czym wobec tego są delegacje? Delegacja (ang. *delegate*) to obiekt wywołujący konkretną metodę z konkretnego obiektu. Przykład definicji delegacji:

```
Delegate Function MyDelegate(ByVal X As Integer, ByVal Y As Integer) _
    As Double
```

Tak zdefiniowana delegacja pozwala wywołać metodę, która ma dwa parametry typu `Integer` oraz zwraca wartość typu `Double`. Innymi słowy, sygnatura delegacji jest taka sama, jak sygnatura metody, którą można wywołać za pośrednictwem tej delegacji. W żargonie informatyki teoretycznej obiekt opakowujący wywołanie metody, jak pokazano to na rysunku 2.5, nazywany jest *funktorem* (ang. *functor*).

Rysunek 2.5.
Delegacja



Gdy chcemy użyć delegacji, musimy utworzyć obiekt delegacji i związać go z metodą, którą powinien wywoływać. Należy zwrócić uwagę, że to .NET zapewnia właściwe działanie delegacji. Zadanie programisty ogranicza się do utworzenia delegacji i wskazania metody, która ma być wywołana. W dalszej części rozdziału, w punkcie „Definiowanie i używanie delegacji” pokazano, w jaki sposób korzysta się z delegacji w programach.

Do czego używane są delegacje? Delegacja pozwala wywołać dowolną metodę — o sygnaturze zgodnej z sygnaturą delegacji — z dowolnego obiektu. Dla delegacji nie ma znaczenia, który to jest obiekt, a nawet jakiego jest on typu. Jest to bardzo przydatne podczas obsługi wywołań zwrotnych (ang. *callback*), gdy wiadomo, jak będzie wyglądało wywołanie metody, ale nie wiadomo, z którego obiektu zostanie ona wywołana. Dzięki temu delegacje świetnie nadają się do obsługi zdarzeń i do tego celu są głównie używane w .NET.

Można by odnieść wrażenie, że delegacje i interfejsy to podobne konstrukcje — i jest tak w istocie. Ale występują między nimi dwie istotne różnice. Po pierwsze, klient komunikuje się bezpośrednio z obiektem implementującym dany interfejs, podczas gdy delegacja jest pośrednikiem między klientem a obiektem. Po wtóre, delegacja reprezentuje jedną metodę, a definicja interfejsu może zawierać wiele powiązanych metod.

Standardowa klasa `Delegate` jest używana jako klasa bazowa dla delegacji pozwalających wywoływać w danej chwili tylko jedną metodę. Natomiast do definiowania delegacji pozwalających wywoływać więcej metod używana jest — jako klasa bazowa — klasa `MulticastDelegate`. Zastosowanie tego rodzaju delegacji wyjaśnione jest w kolejnym punkcie.

Zdarzenia

Środowisko .NET, w przeciwieństwie do innych środowisk, zaprojektowano głównie w celu tworzenia aplikacji z graficznym interfejsem użytkownika oraz aplikacji przeznaczonych dla sieci WWW. Z tego powodu .NET zawiera wiele cech ułatwiających tworzenie tego rodzaju aplikacji. Jedną z takich cech jest obsługa zdarzeń, pojęcie dobrze znane programistom VB.

W aplikacjach z graficznym interfejsem użytkownika zdarzenia pełnią rolę mechanizmu powiadamiania. Jeżeli zostanie naciśnięty przycisk znajdujący się na formularzu, to przycisk ten wygeneruje zdarzenie, by powiadomić formularz o tym, co zaszło. Podobnie, po wybraniu elementu z listy wyboru, wygeneruje ona zdarzenie w celu powiadomienia formularza. W praktyce wygenerowanie zdarzenia oznacza wywołanie metody z obiektu, który chce być informowany o zajściu tego zdarzenia — patrz rysunek 2.6.

Rysunek 2.6.

Generowanie zdarzeń



Do obsługi zdarzeń używa się w .NET delegacji. Za ich pomocą tworzy się połączenie między źródłem zdarzenia a obiektem, który chce być informowany o jego zajściu oraz ustanawia się mechanizm wywołania zwrotnego. Doświadczeni programiści obiektowi znający *wzorce projektowe* (ang. *design pattern*) rozpoznają z pewnością w zdarzeniach zastosowanie wzorca Obserwator (ang. *Observer*). Wzorec ten określa, jak zbudować mechanizm powiadamiania łączący obserwowany obiekt i jeden lub większą liczbę obserwatorów.

Na rysunku 2.7 pokazano, jak działa ten mechanizm. Obiekt, który będzie źródłem zdarzeń, definiuje publiczną delegację, do której dostęp mają inne klasy. Definiuje on również obiekt reprezentujący zdarzenie korzystające ze zdefiniowanej delegacji. Jak pamiętamy,

delegacja to obiekt wywołujący metodę z innego obiektu. Zdarzenie, w chwili wygenerowania, korzysta z delegacji będącej łączem do obiektów, które chcą być poinformowane o zajściu zdarzenia.

Rysunek 2.7.

*Mechanizm
zgłaszania zdarzeń
w .NET*



Klient musi zawierać metodę o sygnaturze zgodnej z sygnaturą delegacji zdefiniowanej przez obiekt będący źródłem zdarzeń (w przykładzie jest to przycisk). Referencję do tej metody klient przekazuje do obiektu reprezentującego zdarzenie. Obiekt ten umieszcza otrzymaną referencję na liście klientów. Zdarzenia korzystają z klasy `MulticastDelegate`, przechowującej listę metod, które mają być wywołane. Dzięki temu pojedynczy obiekt reprezentujący zdarzenie może powiadomić więcej niż jednego klienta.

O naciśnięciu przycisk informuje obiekt reprezentujący zdarzenie, który z kolei wywołuje każdą z metod przechowywanych na liście. W ten sposób wywołane zostaną wszystkie metody zarejestrowane przez klientów.

Metadane i atrybuty

Metadane to bardzo istotny składnik platformy .NET, o czym wspomniano już w rozdziale pierwszym. CLR czerpie z nich kluczowe informacje pozwalające załadować i wykonać kod. Większość metadanych w plikach wykonywalnych generowana jest automatycznie przez kompilator. Istnieje kilka standardowych elementów metadanych, które programista może dołączyć do klasy. Ale w CLR występuje mechanizm pozwalający programistom definiować własne elementy metadanych i dołączać je do klas. Te niestandardowe elementy metadanych nazywane są *atrybutami* (ang. *attribute*). We wszystkich językach platformy .NET można tworzyć atrybuty i odczytywać ich wartości.

Poniżej podano przykład definicji klasy w języku VB zawierającej atrybut.

```
<IsPlugin(" Foo Plugin", version=1.1)> Public Class Foo
...

```

W przykładzie przed słowem kluczowym `Public`, w nawiasie ostrokątnym podano atrybut o nazwie `IsPlugin`. W nawiasie okrągłym z kolei występują argumenty atrybutu. Atrybut może mieć parametry nazwane i pozycyjne, z tym że najpierw należy podać wartości parametrów pozycyjnych, a dopiero później parametry nazwane. W przykładzie występuje jeden parametr pozycyjny (" Foo Plugin") i jeden parametr nazwany (`version`).

Wyjątki

CLR zapewnia obsługę błędów wykonania za pomocą wyjątków. Ma to następujące konsekwencje:

- ♦ mechanizm wyjątków jest dostępny we wszystkich językach platformy .NET,
- ♦ wyjątki zgłaszane w kodzie napisanym w jednym języku mogą być obsługiwane w kodzie napisanym w innym języku.

Obsługa wyjątków jest dobrze znana z języków Java i C++. Osoby nieznające tych języków znajdą w kolejnych akapitach krótkie wyjaśnienie tego mechanizmu.

W wielu językach programowania fakt wystąpienia błędu w wywoływanej metodzie sygnalizowany jest przez zwrócenie specjalnej wartości lub też ustawienie jakiegoś znacznika. Niestety, nie ma mechanizmu wymuszającego obsłużenie błędu w kodzie wywołującym metodę. Dzieje się tak zwłaszcza w językach podobnych do C, w których wartość zwracana z metody może być zignorowana. Obsługa wyjątków (ang. *exception*) to mechanizm rozwiązujący problemy, które występują w tradycyjnych sposobach obsługi błędów, oparty na następujących założeniach:

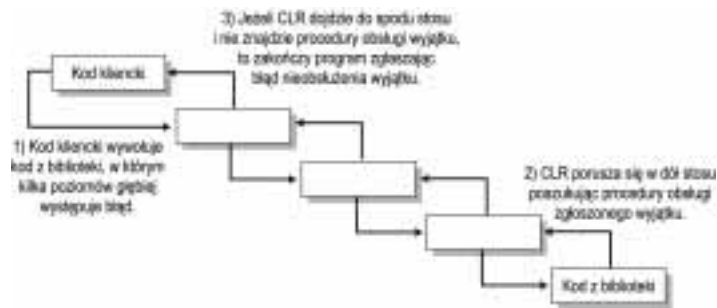
1. Zgłoszony wyjątek nie może zostać zignorowany. Wyjątek musi być obsługiwany w kodzie wywołującym metodę, w przeciwnym razie wykonanie programu zostanie przerwane.
2. Wyjątek nie musi być obsługiwany w metodzie, w której został zgłoszony, lecz w dowolnej metodzie znajdującej się poniżej tej metody na stosie wywołań. Jest to szczególnie użyteczne w przypadku bibliotek — metoda umieszczona w bibliotece może zgłosić wyjątek, który zostanie obsługiwany w kodzie klienta.
3. W pewnych sytuacjach wyjątek to jedyna możliwość zasygnalizowania błędu. Przykładem mogą być konstruktory nigdy nie zwracające wartości. Trudno wówczas inaczej zasygnalizować błąd powstały w trakcie konstruowania obiektu niż za pomocą wyjątków.

Na rysunku 2.8 pokazano, co dzieje się w momencie zgłoszenia wyjątku. Pierwszą czynnością jest przerwanie normalnego działania programu i przejęcie sterowania przez środowisko CLR. Sprawdza ono, czy w metodzie, w której zgłoszony został wyjątek, znajduje się procedura obsługi tego wyjątku (ang. *exception handler*). Jeżeli tak, wówczas normalne działanie programu jest wznowiane od tej procedury. W przeciwnym wypadku CLR sprawdza, czy wyjątek może być obsługiwany w metodzie wywołującej daną metodę — w poprzedniej metodzie znajdującej się na stosie wywołań. CLR kontynuuje poszukiwanie w dół stosu do momentu znalezienia właściwej procedury obsługującej zgłoszony wyjątek. Jeżeli CLR dojdzie do spodu stosu i nie znajdzie takiej procedury, zakończy wykonywanie programu z błędem „nieobsłużony wyjątek” (ang. *unhandled exception*).

Sposób zapisu obsługi wyjątku jest mniej więcej taki sam w VB, C# i C++. We wszystkich tych językach występuje bowiem konstrukcja Try...Catch. W poniższym przykładzie pokazano obsługę wyjątków w VB.

Rysunek 2.8.

Obsługa wyjątków
w .NET



```
Imports System

Public Module modmain

Public Class Foo
    Public Sub Bar()
        Console.WriteLine("Wywołano metodę Bar z klasy Foo.")
    End Sub
End Class

Public Sub func(ByRef f As Foo)
    Try
        f.Bar()
    Catch
        Console.WriteLine("Wyjątek!")
    End Try
End Sub

Sub Main()
    Dim f As Foo
    func(f)
End Sub

End Module
```

W przykładzie zdefiniowano prostą klasę o nazwie `Foo`, zawierającą metodę `Bar()`, oraz funkcję `func`, której parametrem jest referencja do obiektu klasy `Foo`. Proszę zwrócić uwagę na treść funkcji `Main`, w której zdefiniowano zmienną `f` typu `Foo`. Zmienna ta jest przekazywana do funkcji `func`, mimo że nie utworzono obiektu. Jak można oczekiwać, w funkcji `func` podczas próby wywołania metody `Bar` generowany jest błąd wykonania, gdyż parametr `f` nie jest poprawną referencją do obiektu. Błąd ten powoduje zgłoszenie wyjątku. W czasie działania programu, na skutek zgłoszenia tego wyjątku, na konsoli zostanie wyświetlony komunikat *Wyjątek!*.

Kod, w którym w czasie działania mogą wystąpić błędy, umieszczamy wewnątrz bloku `Try`. Za tym blokiem występuje jedna lub więcej procedur obsługi wyjątków zapisywanych za pomocą instrukcji `Catch`. Zadaniem tych procedur jest obsłużenie poszczególnych wyjątków lub też poinformowanie, że wyjątki wystąpiły. Jeżeli w bloku `Try` zgłoszony zostanie wyjątek, sterowanie przejmuje CLR i rozpoczyna poszukiwanie procedury obsługującej ten wyjątek. Jeżeli znajdzie taką procedurę, to jest ona wykonywana i działanie programu jest wznowiane. W przeciwnym wypadku CLR sprawdza, czy taka procedura występuje w metodzie wywołującej (znajdującej się jeden poziom niżej na stosie).

Znamy już sposób obsługi wyjątku. Jest tylko jeden problem — wiemy, że jakiś wyjątek został zgłoszony, ale nie wiemy, co właściwie się stało. Na szczęście jest bardzo proste rozwiązanie — każdy wyjątek jest reprezentowany przez obiekt zawierający informacje o rodzaju błędu, który wystąpił oraz inne informacje ułatwiające zdiagnozowanie problemu.

Obiekty wyjątków są egzemplarzami podklas klasy `System.Exception`. W tabeli 2.3 zebrało najczęściej występujące podklasy systemowe dla wyjątków.

Tabela 2.3. Najczęściej używane klasy wyjątków w .NET

Klasa reprezentująca wyjątek	Opis
<code>SystemException</code>	Klasa bazowa dla wyjątków, które mogą być obsłużone (tzn. takich, które nie są krytyczne — nie powodują załamania się programu).
<code>ArgumentException</code>	Argument metody był niepoprawny.
<code>ArgumentNullException</code>	Argument o wartości <code>null</code> został przekazany do metody, dla której nie jest to poprawna wartość.
<code>ArgumentOutOfRangeException</code>	Wartość argumentu znajduje się poza zakresem dopuszczalnych wartości.
<code>ArithmeticException</code>	Wystąpił nadmiar lub niedomiar arytmetyczny ⁵ .
<code>ArrayTypeMismatchException</code>	Próba umieszczenia w tablicy obiektu o niewłaściwym typie.
<code>BadImageFormatException</code>	Niepoprawny format pliku DLL lub EXE.
<code>DivideByZeroException</code>	Próba dzielenia przez zero.
<code>DllNotFoundException</code>	Nie znaleziono wymaganego pliku DLL.
<code>FormatException</code>	Niepoprawny format argumentu.
<code>IndexOutOfRangeException</code>	Przekroczenie zakresu indeksów tablicy.
<code>InvalidCastException</code>	Próba rzutowania na niewłaściwą klasę.
<code>InvalidOperationException</code>	Metoda została wywołana w niewłaściwym momencie.
<code>MethodAccessException</code>	Niepoprawna próba dostępu do prywatnej lub chronionej składowej.
<code>MissingMemberException</code>	Próba dostępu do niewłaściwej wersji pliku DLL ⁶ .
<code>NotFiniteNumberException</code>	Obiekt nie reprezentuje poprawnej liczby.
<code>NotSupportedException</code>	Wywołano metodę niezaimplementowaną w klasie.
<code>NullReferenceException</code>	Odwołanie do obiektu przez zmienną referencyjną o wartości <code>null</code> .
<code>OutOfMemoryException</code>	Brak pamięci uniemożliwia kontynuowanie działania programu.
<code>PlatformNotSupportedException</code>	Próba odwołania do funkcji niedostępnej na danej platformie.
<code>StackOverflowException</code>	Wystąpiło przepełnienie stosu.

W instrukcji `Catch` może wystąpić deklaracja typu wyjątku obsługiwanego przez tę instrukcję. Ta konstrukcja oraz sposób użycia więcej niż jednej instrukcji `Catch` zostały pokazane w poniższym przykładzie.

⁵ Uwaga: Klasa ta jest klasą bazową dla kilku innych, bardziej szczegółowo opisujących błędy, na przykład wymienionych w tabeli: `DivideByZeroException` i `NotFiniteNumberException` — *przyj. tłum.*

⁶ Ten wyjątek jest zgłaszany przy próbie odwołania się do nieistniejącej składowej. Sytuacja taka może wystąpić, gdy po kompilacji składowa klasy zdefiniowanej w zewnętrznym podzespole zostanie usunięta lub przemianowana — *przyj. tłum.*

```
Public Sub func(ByRef f As Foo)
    Try
        f.Bar()
    Catch ae As NullReferenceException
        Console.WriteLine("Wystąpił wyjątek NullReferenceException: {0}", ae)
    Catch ex As Exception
        Console.WriteLine("Wystąpił wyjątek: {0}", ex)
    End Try
End Sub
```

Jak widać, pierwsza instrukcja `Catch` pozwala przechwycić wyjątki o typie `NullReferenceException`, a druga — wszystkie pozostałe wyjątki pochodzące od klasy `Exception`. W czasie działania programu wykonywana jest pierwsza napotkana instrukcja `Catch`, która odpowiada zgłoszonemu wyjątkowi. W podanym powyżej przykładzie zgłaszany jest wyjątek `NullReferenceException`, tak więc wykonana zostanie pierwsza instrukcja `Catch`. W rezultacie na konsoli zostanie wydrukowany następujący komunikat:

```
Wystąpił wyjątek NullReferenceException: System.NullReferenceException:
Object reference not set to an instance of an object.
at modmain.func(Foo& f)
```

Wydruk obiektu reprezentującego wyjątek zawiera wydruk komunikatu oraz zawartości stosu, co umożliwia zlokalizowanie miejsca zgłoszenia wyjątku.

Blok `Try` może również zawierać instrukcję `Finally`, która jest wykonywana zawsze przed wyjściem z metody, niezależnie od tego, czy wyjątek wystąpił czy też nie. Przykład:

```
Public Sub func(ByRef f As Foo)
    Try
        f.Bar()
    Catch ae As NullReferenceException
        Console.WriteLine("Wystąpił wyjątek NullReferenceException: {0}", ae)
        ' Exit here...
    Exit Sub
    Catch ex As Exception
        Console.WriteLine("Wystąpił wyjątek Exception: {0}", ex)
    Finally
        Console.WriteLine("Wykonanie instrukcji Finally...")
    End Try
End Sub
```

W instrukcji `Catch` obsługującej wyjątek `NullReferenceException` po wypisaniu komunikatu znajduje się instrukcja `Exit Sub` powodująca wyjście z metody, ale nim ono nastąpi, wykonywana jest instrukcja `Finally`. Jest ona bardzo użyteczna, jeżeli przed wyjściem z metody należy koniecznie wykonać jakieś czynności, na przykład zamknąć plik lub odświeżyć zawartość tabel bazodanowych. Gdyby nie instrukcja `Finally`, trzeba by było zamieścić wiele wierszy skomplikowanego kodu.

Refleksja i klasa `Type`

W .NET można odczytać informacje o podzespolo załadowanym do pamięci. Informacje te obejmują:

- ◆ listę klas zdefiniowanych w module,

- ♦ listę metod zdefiniowanych w klasie,
- ♦ nazwy i typy właściwości oraz pól,
- ♦ sygnatury metod.

Zgodnie z oczekiwaniami, informacje te zawarte są w metadanych powiązanych z podzespołami i klasami. Proces odczytywania tych informacji nazywany jest *refleksją* (ang. *reflection*).

Refleksja jest zaimplementowana za pomocą przestrzeni nazw `System.Reflection` i klasy `System.Type`, jest integralną częścią modelu programowania w środowisku .NET.

Refleksja, oprócz prostego odczytywania metadanych, pozwala również dynamicznie tworzyć obiekt danego typu i wywoływać metody z tego obiektu. Jest ona mechanizmem używanym przez VB do realizacji późnego wiązania (ang. *late binding*), którego przykład pokazano w poniższym fragmencie kodu.

```
* Deklaracja zmiennej referencyjnej o ogólnym typie Object
Dim obj As Object
* Utworzenie obiektu o typie Test
obj = New Test()

* Wywołanie z tego obiektu metody Foo
obj.Foo
```

Zadeklarowana została zmienna o ogólnym typie `Object` i za jej pomocą odwołujemy się do utworzonego obiektu. Z tego powodu kompilator VB podczas kompilacji nie ma informacji o typie obiektu, do którego się odwołujemy, a tym samym kompilator nie może sprawdzić, czy wywołanie metody `Foo()` jest dopuszczalne czy też nie. W czasie wykonywania programu środowisko uruchomieniowe CLR sprawdza — używając do tego refleksji — czy w obiekcie wskazywanym przez zmienną `obj` zdefiniowana jest metoda `Foo` oraz jakie ma argumenty.

Poniżej zamieszczono nieznacznie zmodyfikowany kod IL, otrzymany po kompilacji naszego programu, wyświetlony przez deassembler `ildasm`. Można zobaczyć, jak zapisano późne wiązanie.

```
IL_0000: newobj instance void Module1/Test::.ctor()
IL_0005: stloc.0
IL_0006: ldloc.0
IL_0007: ldnull
IL_0008: ldstr "Foo"
IL_000d: ldc.i4.0
IL_000e: newarr [mscorlib]System.Object
IL_0013: ldnull
IL_0014: ldnull
IL_0015: call void [Microsoft.VisualBasic]
        ↳ Microsoft.VisualBasic.CompilerServices.LateBinding::LateCall
        ↳ object, class [mscorlib]System.Type, ...)
IL_001a: ret
```

W przykładzie widać, że przed wywołaniem funkcji pomocniczej `LateCallNoByRef` łaadowany jest napis zawierający nazwę wywoływanej funkcji `Foo`. Jednym z argumentów

podawanych przy wywołaniu funkcji `LateCallNoByRef` jest obiekt `System.Type` zawierający informacje o obiekcie, z którego wywoływana jest metoda `Foo`.

Główne znaczenie dla realizacji refleksji ma klasa `System.Type`. Za pomocą refleksji można otrzymać obiekt tej klasy dla każdego załadowanego typu. Metody, pola i właściwości obiektu klasy `Type` pozwalają odczytać wszystkie informacje o typie reprezentowanym przez ten obiekt, a nawet tworzyć obiekty tego typu.

W VB można otrzymać obiekt klasy `Type` poprzez użycie operatora `GetType`, jak to pokazano poniżej.

```
* Pobranie obiektu klasy Type dla typu Integer
Dim tf As Type = GetType(Integer)
```

```
* Pobranie obiektu klasy Type dla tablicy o elementach Integer
Dim tf1 As Type = GetType(Integer())
```

Po pobraniu obiektu klasy `Type`, można z niego odczytać informacje o typie przez niego reprezentowanym. Metoda `GetMembers()` zwraca tablicę obiektów `MemberInfo` opisujących składowe dane tego typu. Zamiast niej można użyć bardziej wyspecjalizowanych metod: `GetMethods()`, `GetFields()` i `GetProperties()`, zwracających informacje o metodach, polach i właściwościach. Każda z tych metod zwraca tablicę obiektów odpowiedniego typu (`MethodInfo`, `FieldInfo`, `PropertyInfo`).

Gotowe rozwiązania

Definiowanie klas

Do zdefiniowania klasy w VB używane jest słowo kluczowe `Class`.

```
Class Foo
...
End Class
```

Definicja klasy może zawierać pola (dane), metody, właściwości i zdarzenia.

Przeciążanie i przesłanianie metod

Pojęcia: *przeciążanie* (ang. *overloading*) i *przesłanianie* (ang. *overriding*) często stosowane są zamiennie, mimo że są to dwa zupełnie różne pojęcia. Jeżeli w danej klasie występują dwie (lub więcej) metody o takich samych nazwach, lecz różniące się listą argumentów, mówimy, że metody te są przeciążone.

W języku VB definicja metody przeciążonej musi zawierać słowo kluczowe `Overloads`.

```
Class Foo
  Public Overloads Sub Bar()
  End Sub

  Public Overloads Sub Bar(ByVal n As Integer)
```

```
End Sub  
End Class
```

Zdefiniowane powyżej metody `Bar` różnią się listą argumentów, są więc poprawnie przeciążone.

W `C#` i w `C++` definicje metod przeciążonych nie muszą zawierać specjalnych słów kluczowych, muszą tylko różnić się listą argumentów.

Przesłanianie ma związek z dziedziczeniem i zostanie omówione w dalszej części rozdziału, w punkcie „Przesłanianie metod”.

Definiowanie pól i metod należących do klasy

W .NET można zdefiniować składowe należące do klasy, a nie do poszczególnych obiektów tej klasy. Rozważmy na przykład klasę reprezentującą rachunek bankowy. Każdy obiekt klasy `Account` (rachunek bankowy) ma saldo należące tylko do tego obiektu. W klasie `Account` może również wystąpić składowa `InterestRate` (stopa procentowa) przechowująca wartość stopy procentowej wszystkich rachunków. Wartość ta jest wspólna dla wszystkich rachunków, nie różni się dla poszczególnych obiektów tej klasy — z tego też powodu można powiedzieć, że zmienna przechowująca wartość stopy procentowej należy do klasy `Account`. Składowe należące do klasy nazywane są statycznymi (ang. *static*) w `C#` i w `C++`, a współdzielonymi (ang. *shared*) w `VB`. Nie tylko pola mogą być statyczne, ale również metody i właściwości.

W poniższym przykładzie pokazano, w jaki sposób definiuje się składowe współdzielone w `VB`.

```
Public Class Account  
    Private Shared InterestRate As Double  
    ...  
    Public Shared Function GetInterestRate() As Double  
        GetInterestRate = InterestRate  
    End Function  
End Class
```

Zarówno zmienna `InterestRate`, jak i metoda `GetInterestRate`, zwracająca wartość tej zmiennej, są współdzielone przez wszystkie obiekty klasy `Account`. Ponieważ metoda należy do klasy, możemy ją wywołać poprzedzając nazwę metody nazwą klasy, a nie obiektu.

```
d = Account.GetInterestRate
```

Można to odczytać następująco: „pobierz wartość stopy procentowej z klasy `Account`”. Ponieważ składowe współdzielone należą do klasy (nie są wywoływane z jakiegoś obiektu tej klasy) — nie jest w nich dostępna referencja `this` czy też `Me`.

Definiowanie struktur

W .NET struktury są podobne do klas, ale różni je jedna istotna cecha. Klasy to typy referencyjne, podczas gdy struktury to typy bezpośrednie. Struktury mogą zawierać takie

same składowe jak klasy (metody, pola, właściwości i zdarzenia), ale przy zachowaniu pewnych ograniczeń, wymienionych już w części „Teoria”.

W języku Visual Basic struktury definiuje się za pomocą słowa kluczowego Structure, jak to pokazano poniżej.

```
Public Structure Point
    Private myX As Integer
    Private myY As Integer

    Property X() As Integer
        Get
            X = myX
        End Get
        Set
            myX = Value
        End Set
    End Property

    ...

    * Konstruktor służyący do inicjalizacji struktury Point
    Sub new(ByVal anX As Integer, ByVal aY As Integer)
        myX = anX
        myY = aY
    End Sub
End Structure
```

Natomiast w C# struktury definiuje się za pomocą słowa kluczowego struct.

```
struct Point {
    int myX, myY;

    public int X {
        get {
            return myX;
        }
        set {
            myX = value;
        }
    }
}

...

// Konstruktor służyący do inicjalizacji struktury Point
public Point(int anX, int aY) {
    myX = anX;
    myY = aY;
}
}
```

W nadzorowanym kodzie C++ definicja struktury wygląda nieco inaczej, a mianowicie używa się słowa kluczowego `__value`. W standardowym języku C++ jedyną różnicą między strukturami a klasami jest domyślna widoczność składowych. Definicja klasy lub struktury w nadzorowanym kodzie C++, zawierająca słowo kluczowe `__value` oznacza strukturę właściwą dla .NET (typ bezpośredni). Klasy i struktury zdefiniowane za pomocą

słowa kluczowego `__value` podlegają tym samym ograniczeniom co typy bezpośrednie. Przykład definicji struktury właściwej dla .NET:

```
__value struct Point {  
    int myX, myY;  
};
```

Konstruktory i destruktory w VB

Do inicjalizacji i finalizacji obiektów w języku VB używa się metod `Sub New()` i `Sub Finalize()`. Gdy tworzy się egzemplarz klasy, wywoływana jest metoda `Sub New()`. Może mieć argumenty i być przeciążona. W poniższym przykładzie do nowo tworzonego obiektu typu `Car` (samochód) przekazywana jest marka samochodu.

```
Class Car  
    Public Sub New(ByRef make As String)  
        ...  
    End Sub  
End Class  
  
Sub Main()  
    Dim c As New Car("Ford")  
End Sub
```

Metoda `Sub New` ma w przykładzie jeden parametr typu `String`. Podczas tworzenia obiektu klasy `Car` w metodzie `Main` przekazywany jest napis zawierający markę samochodu. Po zdefiniowaniu drugiego konstruktora mającego dwa parametry (marka i model), metoda `New` będzie przeciążona.

```
Class Car  
    Public Sub New(ByRef make As String)  
        ...  
    End Sub  
  
    Public Sub New(ByRef make As String, ByRef model As String)  
        ...  
    End Sub  
End Class  
Sub Main()  
    Dim c As New Car("Ford")  
    Dim c2 As New Car("Ford", "Orion")  
End Sub
```

Kompilator na podstawie liczby argumentów jest w stanie określić, który konstruktor ma być wywołany.

Należy pamiętać, że konstruktor wywoływany jest zawsze w momencie tworzenia nowego egzemplarza obiektu. Jeżeli więc nie zdefiniowano w klasie metody `Sub New()`, kompilator wygeneruje pusty konstruktor dla tej klasy.

Klasa może również zawierać metodę `Sub Finalize()` pełniącą rolę finalizatora, czyli metody wywoływanej wówczas, gdy obiekt jest usuwany z pamięci przez program zarządzający pamięcią dynamiczną. Pamiętajmy, że nie wiadomo, kiedy i czy w ogóle obiekt zostanie usunięty z pamięci. Z tego powodu nie należy umieszczać w metodzie `Finalize`

kodu, który powinien być wywołany w jakimś określonym momencie czy też który powinien być w ogóle wykonany.

Ponadto, ponieważ metoda `Sub Finalize()` jest bezargumentowa z definicji, nie może być przeciążona.

Sprzątanie po obiektach .NET

Wszystkie obiekty .NET, niezależnie od tego, w jakim języku zostały zaimplementowane, dziedziczą po klasie bazowej `System.Object` metodę `Finalize`. Metoda ta jest wywoływana, gdy program zarządzający pamięcią dynamiczną „postanowi” ostatecznie usunąć obiekt z pamięci. Z tego powodu można by przypuszczać, że ta metoda to dobre miejsce do posprzątania po obiekcie i zwolnienia zajmowanych przez niego zasobów.

Jest jeden problem — finalizacja w .NET jest niedeterministyczna. Innymi słowy, nie wiadomo kiedy finalizator zostanie wykonany. Co więcej, program dynamicznie zarządzający pamięcią może nie zostać użyty do usunięcia obiektów podczas kończenia aplikacji. W takiej sytuacji finalizator nie zostanie w ogóle wywołany. Oznacza to, że operacje „sprzątające” po obiekcie, które powinny być wykonane w jakimś określonym momencie — na przykład zapisanie rekordów w tabeli bazodanowej — nie powinny być wykonywane w finalizatorze.

Zalecanym rozwiązaniem jest zaimplementowanie interfejsu `IDisposable` zawierającego jedną metodę `Dispose()`, którą klienci obiektu powinni wywołać, gdy kończą korzystanie z obiektu. W metodzie tej obiekt powinien po sobie „posprzątać” i zaznaczyć, że nie należy już z niego korzystać. W praktyce oznacza to, że należy ustawić znacznik, który powinien być badany w każdej metodzie obiektu. Jeżeli jest ustawiony, metoda powinna zakończyć się niepowodzeniem i zgłosić błąd. Jeżeli obiekt po sobie „posprzątał”, nie ma znaczenia, kiedy i czy w ogóle zostanie usunięty z pamięci.

Korzystanie z dziedziczenia

Do zdefiniowania relacji dziedziczenia między dwiema klasami używa się w VB słowa kluczowego `Inherits`.

```
Class Car
  Inherits Vehicle
  ...
End Class
```

Instrukcja `Inherits` może pojawić się tylko w klasie i musi wystąpić w pierwszym wierszu kodu definicji tej klasy (wcześniej mogą wystąpić tylko wiersze puste oraz zawierające tylko komentarz). Bezpośrednio po słowie `Inherits` występuje nazwa klasy bazowej. Zgodnie z modelem obiektowym .NET, może wystąpić tylko jedna klasa bazowa. W kodzie klasy można odwołać się do klasy bazowej za pomocą słowa kluczowego `MyBase`.

Jeżeli w klasie zdefiniowano konstruktor, w pierwszym jego wierszu musi znajdować się wywołanie konstruktora klasy bazowej, czyli `MyBase.New`, jak to pokazano poniżej.

```
Class Vehicle
```

```
Public Sub New(ByRef make As String)
    ...
End Sub
End Class

Class Car
    Inherits Vehicle

    Public Overloads Sub New(ByRef make As String)
        MyBase.New(make)
    End Sub
End Class
```

Przesłanie metod

Klasa pochodna dziedziczy po klasie bazowej metody w niej zdefiniowane.

W takiej sytuacji możliwe są cztery scenariusze.

1. W klasie pochodnej po prostu korzysta się z odziedziczonej metody.
2. W klasie pochodnej zdefiniowana jest metoda przesłaniająca odziedziczoną metodę w celu zmiany odziedziczonego zachowania. Metoda przesłaniająca musi mieć taką samą sygnaturę, jak metoda w klasie bazowej.
3. Klasa pochodna musi przesłonić odziedziczoną metodę, gdyż metoda ta w klasie bazowej nie ma ciała.
4. Klasie pochodnej nie wolno przesłonić odziedziczonej metody.

Do obsługi trzech ostatnich scenariuszy wprowadzono w Visual Basicu trzy nowe słowa kluczowe.

1. Słowo kluczowe `Overridable`, używane do oznaczenia metody, która może, ale nie musi być przesłonięta w klasie pochodnej.
2. Słowo kluczowe `MustOverride`, używane do oznaczenia metody nie mającej ciała w klasie bazowej, a tym samym metody, która musi być przesłonięta w klasie pochodnej. Klasa zawierająca metodę oznaczoną słowem kluczowym `MustOverride` jest klasą abstrakcyjną. Tego rodzaju klasy omówiono w dalszej części rozdziału.
3. Słowo kluczowe `NotOverridable`, używane do oznaczenia metody, która nie może być przesłonięta w klasie pochodnej. Jest to zarazem sytuacja domyślna, zachodząca wówczas, gdy w definicji metody nie podano żadnego z tych trzech słów kluczowych.

Jeżeli chcemy przesłonić odziedziczoną metodę w klasie pochodnej, metoda ta musi być w klasie bazowej oznaczona słowem `Overridable` lub `MustOverride`, a w klasie pochodnej — słowem kluczowym `Overrides`.

```
Public MustInherit Class Shape
    Public MustOverride Sub Draw()
End Class
```

```
Public Class Square
    Inherits Shape
    Public Overrides Sub Draw()
        ' kod do rysowania kwadratu
    End Sub
End Class
```

W języku C# w klasie na szczycie hierarchii dziedziczenia do oznaczenia metody wirtualnej używa się słowa kluczowego `virtual`. Natomiast metody w klasach pochodnych muszą mieć taką samą sygnaturę i być oznaczone słowem kluczowym `override` lub `new`.

```
public abstract class Shape {
    public virtual void Draw() {}
}

public class Square : Shape {
    public override void Draw() {
        // kod do rysowania kwadratu
    }
}
```

Słowo kluczowe `override` użyte w definicji metody `Draw` w klasie `Square` wskazuje, że metoda ta przesłania metodę `Draw` zdefiniowaną w klasie `Shape`. Załóżmy, że zmienna referencyjna klasy `Shape` wskazuje na obiekt klasy `Square`. Jeżeli za pośrednictwem tej zmiennej wywołano by metodę `Draw`, wykonana zostanie właściwa wersja tej metody, czyli ta zdefiniowana w klasie `Square`. Gdyby natomiast w definicji metody `Draw` zamiast `override` wystąpiło słowo kluczowe `new`, znaczyłoby, że metoda z klasy `Square` nie przesłania metody z klasy `Shape`, mimo iż obie metody mają takie same sygnatury. W sytuacji, gdy w klasie pochodnej występuje metoda o takiej samej sygnaturze, jak w klasie bazowej, definicja metody w klasie pochodnej musi zawierać słowo kluczowe `override` lub `new`. W przeciwnym wypadku kompilator wygeneruje ostrzeżenie.

Definiowanie klas abstrakcyjnych

Klasę nazywamy abstrakcyjną (ang. *abstract class*), jeżeli nie można tworzyć egzemplarzy tej klasy. Może ona być użyta jedynie jako klasa bazowa. Nie oznacza to wszakże, że klasa abstrakcyjna nie może zawierać kodu. W wielu przypadkach klasy abstrakcyjne zawierają kod używany przez klasy pochodne.

Klasy abstrakcyjne definiuje się w VB za pomocą słowa kluczowego `MustInherit`.

```
Public MustInherit Class Shape
    ...
End Class
```

Nie można tworzyć obiektów klasy `Shape`, ale można zdefiniować klasy pochodne, a następnie odwoływać się do obiektów klas pochodnych za pomocą zmiennych referencyjnych o typie `Shape`. Klasa abstrakcyjna nie musi zawierać metod abstrakcyjnych zdefiniowanych za pomocą słowa kluczowego `MustOverride`, ale często tak się właśnie dzieje.

W języku C# klasy abstrakcyjne definiujemy używając słowa kluczowego `abstract`.

```
public abstract class Shape {
```

```
    ...  
}
```

Natomiast w kodzie nadzorowanym C++ klasy abstrakcyjne definiuje się za pomocą słowa kluczowego `__abstract`. Użycie tego słowa uniemożliwia tworzenie egzemplarzy takiej klasy nawet wówczas, gdy wszystkie składowe mają ciała, w przeciwieństwie do klas abstrakcyjnych w tradycyjnym C++. Przykład:

```
__abstract __gc class MyBaseClass {  
    // kod klasy...  
};
```

Należy zauważyć, że słowo kluczowe `__abstract` może być użyte zarówno w definicji nadzorowanej, jak i nienadzorowanej klasy lub struktury.

Definiowanie zapieczętowanych klas i metod

Klasa zapieczętowana (ang. *sealed class*) jest przeciwieństwem klasy abstrakcyjnej w tym sensie, że klasa zapieczętowana nie może być klasą bazową, a klasa abstrakcyjna może być użyta tylko jako klasa bazowa.

Klasy zapieczętowane definiuje się w VB za pomocą słowa kluczowego `NotInheritable`.

```
Public NotInheritable Class MySealedClass  
    ...  
End Class
```

Natomiast w kodzie nadzorowanym C++ do tego samego celu służy słowo kluczowe `__sealed`.

```
__sealed __gc class MySealedClass {  
    // kod klasy...  
};
```

Oczywiście w definicji klasy może wystąpić albo `__abstract`, albo `__sealed`, ale nie obydwa słowa kluczowe jednocześnie.

Metoda zapieczętowana to metoda, która nie może być przesłonięta w klasie pochodnej. W VB metody publiczne są domyślnie zapieczętowane. Słowo kluczowe `NotOverridable` może być użyte w definicji metody lub właściwości do podkreślenia tego faktu.

Do tego samego celu w kodzie nadzorowanym C++ stosowane jest słowo kluczowe `__sealed`, a w C# — słowo `sealed`.

Definiowanie właściwości

Nowy sposób definiowania właściwości wprowadzony w .NET zastąpił w VB konstrukcje: `Property Get` i `Property Set`. Przykład definicji właściwości w nowym VB:

```
Property Color() As String  
    Get
```



```
        Color = myColor
    End Get
    Set
        myColor = Value
    End Set
End Property
```

Procedura `Get` służy do pobrania wartości właściwości, a procedura `Set` — do jej ustawienia. W przykładzie wartość właściwości przechowywana jest w zmiennej `myColor`. Specjalna zmienna `Value` przechowuje wartość przekazaną do procedury `Set`. Typ właściwości określa typ zmiennej `Value`.

Można zdefiniować właściwość tylko do odczytu. W tym celu w jej definicji należy podać kwalifikator `ReadOnly` oraz pominąć procedurę `Set`.

```
ReadOnly Property Color() As String
    Get
        Color = myColor
    End Get
End Property
```

Definicja właściwości w języku C# wygląda niemal identycznie.

```
public string Color {
    get {
        return myColor;
    }
    set {
        myColor = Value;
    }
}
```

W C# właściwość tylko do odczytu lub tylko do zapisu możemy zdefiniować pomijając, odpowiednio, procedurę `set` lub procedurę `get`.

W celu zdefiniowania właściwości w kodzie nadzorowanym C++, należy użyć dwóch metod opatrzonych słowem kluczowym `__property`, nazwy tych metod powinny rozpoczynać się od `get` i `set`.

```
__gc class test
{
    int prop;
public:
    __property int get_Prop() { return prop; }
    __property void set_Prop(int m) { prop=m; }
};
```

Kompilator po napotkaniu w kodzie źródłowym deklaracji `get_Prop()` i `set_Prop()` generuje w kodzie wynikowym pseudopole o nazwie `Prop`. W klasie nie może istnieć zwykłe pole o takiej nazwie — jak widać w przykładzie, wystarczy, że nazwy właściwości i pola różnią się wielkością liter. Podobnie jak w C#, w celu zdefiniowania właściwości tylko do odczytu lub tylko do zapisu wystarczy pominąć odpowiednią metodę.

Definiowanie interfejsów

Interfejs w .NET może zawierać wirtualne metody, właściwości i zdarzenia. Interfejs definiowany w języku C# może ponadto zawierać indeksatory (ang. *indexer*).

Do zdefiniowania interfejsu w VB służy słowo kluczowe `Interface`.

```
Interface IAnimal
    Sub MakeNoise()
    ReadOnly Property Name() As String
End Interface
```

Jak widać, definicje wchodzące w skład interfejsu zawierają sygnatury, nie zawierają natomiast ciał. Zgodnie z powszechnie przyjętą umową, nazwy interfejsów w .NET rozpoczynają się od litery I. Ciało interfejsu zawiera sygnatury metod, które muszą być zaimplementowane w klasie implementującej dany interfejs. Definicje metod w interfejsie nie mogą zawierać modyfikatorów; zakłada się, że te metody są publiczne.

Jak widać w przykładzie, w VB7 wprowadzono nowy sposób definiowania właściwości, zastępujący znane z poprzednich wersji języka metody: `Property Get` i `Property Set`. Celem tej zmiany było nie tylko ulepszenie składni (poprawiające jej czytelność), ale również dostosowanie jej do składni C# i innych języków platformy .NET. Modyfikator `ReadOnly` wskazuje, że właściwość `Name` może być odczytywana, ale nie ustawiana.



W poprzednich wersjach języka VB interfejsy były sztucznym tworem, gdyż definiowało się je za pomocą klas, w których metody nie posiadały ciał. Visual Basic .NET zawiera stosowne konstrukcje do definiowania i implementowania interfejsów.

Interfejsy w języku C# definiuje się w bardzo podobny sposób. Poniżej zamieszczono definicję interfejsu `IAnimal` w języku C#.

```
interface IAnimal {
    void MakeNoise();
    string Name { get; }
}
```

Implementowanie interfejsów

Implementowanie interfejsów bardzo przypomina dziedziczenie, z tą różnicą, że zamiast słowa kluczowego `Inherits` używa się słowa `Implements`.

```
Public Class Dog
    Implements IAnimal

    Sub MakeNoise() Implements IAnimal.MakeNoise
        Console.WriteLine("Hau!")
    End Sub

    ReadOnly Property Name() As String Implements IAnimal.Name
        Get
            Name = "dog"
        End Get
    End Property
End Class
```

W powyższym przykładzie należy zwrócić uwagę na dwie rzeczy. Po pierwsze: słowo kluczowe `Implements` informuje kompilator, że klasa implementuje podany interfejs. Wówczas kompilator sprawdza, czy klasa implementuje wszystkie składowe występujące w interfejsie. Klasa może implementować wiele interfejsów. W takiej sytuacji po słowie `Implements` podaje się listę nazw interfejsów, rozdzielonych przecinkami.

Po drugie, należy wskazać, która metoda lub właściwość w klasie jest implementacją metody lub właściwości zdefiniowanej w interfejsie. Do tego celu służy słowo kluczowe `Implements`. Przykład:

```
Sub MakeNoise() Implements IAnimal.MakeNoise
```

Powyzsza definicja informuje kompilator, że metoda `MakeNoise()` jest implementacją metody `MakeNoise` z interfejsu `IAnimal`. Jakie ma to konsekwencje? Po pierwsze — można zaimplementować dwa interfejsy zawierające metody o takich samych nazwach i nie pojawią się problemy z rozstrzygnięciem, o którą metodę chodzi.

Po drugie — nazwy metod i właściwości mogą się różnić od nazw zdefiniowanych w implementowanym interfejsie. W większości przypadków nazwy te powinny być takie same (w kolejnym punkcie, „Korzystanie z obiektu za pośrednictwem interfejsu”, wyjaśniono dlaczego), ale nie musi tak być. Można by na przykład zaimplementować w klasie `Dog` metodę `MakeNoise` w sposób następujący:

```
Sub DogMakeNoise() Implements IAnimal.MakeNoise
```

Implementowanie interfejsów w `C#` bardzo przypomina dziedziczenie. Po nazwie klasy występuje dwukropki, a po nim nazwa interfejsu lub lista nazw interfejsów rozdzielonych przecinkami (gdy klasa implementuje więcej niż jeden interfejs). Przykład:

```
public class Dog : IAnimal {  
    public void MakeNoise() {  
        Console.WriteLine("Hau!");  
    }  
  
    public string Name {  
        get { return "dog"; }  
    }  
}
```

Korzystanie z obiektu za pośrednictwem interfejsu

Korzystanie za pośrednictwem interfejsu z obiektu implementującego ten interfejs jest w `VB` bardzo proste.

```
* Utworzenie obiektu klasy Dog i wywołanie metody MakeNoise  
Dim d1 As New Dog()  
d1.MakeNoise()  
  
* Pies jest zwierzęciem  
Dim a1 As IAnimal  
a1 = d1  
a1.MakeNoise()
```

Po utworzeniu obiektu klasy Dog można, zgodnie z oczekiwaniami, wywołać z niego bezpośrednio metodę MakeNoise. W celu skorzystania z obiektu Dog za pośrednictwem interfejsu IAnimal, należy zdefiniować zmienną referencyjną o typie IAnimal. Za pośrednictwem tej zmiennej można używać każdego obiektu implementującego interfejs IAnimal, niezależnie od tego, czy jest on klasy Dog, Cat, czy Platypus.

W czasie wykonywania instrukcji a1 = d1 następuje sprawdzenie, czy można odwołać się do obiektu d1 za pośrednictwem interfejsu IAnimal. Ponieważ klasa Dog implementuje interfejs IAnimal, sprawdzenie kończy się powodzeniem i zmiennej a1 przypisywana jest referencja do interfejsu implementowanego przez obiekt d1. Gdyby obiekt d1 był obiektem takiej klasy, w której nie zaimplementowano interfejsu IAnimal (na przykład klasy Car), wystąpiłby błąd. Po przypisaniu zmiennej a1 referencji do interfejsu IAnimal, można za jej pośrednictwem wywoływać metody i właściwości zdefiniowane w tym interfejsie.



W poprzednich wersjach VB do przypisywania referencji stosowane było słowo kluczowe Set. W Visual Basic .NET nie jest to już konieczne.

Jak wspomniano w poprzednim punkcie, metoda implementująca nie musi mieć takiej samej nazwy, jak metoda z interfejsu. Jeśli na przykład metoda DogMakeNoise w klasie Dog zdefiniowana była następująco:

```
Sub DogMakeNoise() Implements IAnimal.MakeNoise
```

to podany wyżej kod, wywołujący tę metodę, należałoby zapisać tak:

```
Dim d1 As New Dog()
d1.DogMakeNoise()

' Pies jest zwierzęciem
Dim a1 As IAnimal
a1 = d1
a1.MakeNoise()
```

Widać już, na czym polega problem. Gdy korzystamy bezpośrednio z obiektu Dog, metodę powinniśmy wywołać podając nazwę DogMakeNoise. Odwołując się natomiast do tego obiektu za pośrednictwem interfejsu IAnimal, musimy podać nazwę MakeNoise. Konieczność użycia dwóch różnych nazw w celu wywołania tej samej metody jest źródłem niepożądanego zamieszania. Z tego też powodu zaleca się, by metody implementujące miały takie same nazwy, jak metody implementowane zdefiniowane w interfejsie.

Jeszcze jedna kwestia w VB jest istotna, a mianowicie jak sprawdzić, czy klasa Dog implementuje interfejs IAnimal? Do tego celu służy słowo kluczowe TypeOf. Za jego pomocą można sprawdzić, jakiego typu jest obiekt, ale może być użyte również do interfejsów, jak to pokazano poniżej.

```
' Sprawdzenie, czy obiekt jest klasy Dog
If TypeOf d1 Is Dog Then
    Console.WriteLine("Obiekt jest klasy Dog")
End If

' Sprawdzenie, czy obiekt implementuje interfejs IAnimal
If TypeOf d1 Is IAnimal Then
    Console.WriteLine("Obiekt implementuje IAnimal")
End If
```

Z obiektów implementujących interfejs korzysta się w C# w bardzo podobny sposób.

```
// Utworzenie obiektu klasy Dog i wywołanie metody MakeNoise
Dog d = new Dog();
d.MakeNoise();

// Pies jest zwierzęciem
IAnimal a;
a = (IAnimal)d;
a.MakeNoise();
```

Najważniejszy wiersz, w którym zmiennej `a` przypisywana jest zmienna `d`, został wyróżniony. Ponieważ zmienne `a` oraz `d` są zmiennymi referencyjnymi określonych typów, w trakcie wykonywania tej instrukcji następuje sprawdzenie, czy zmienna `a` może wskazywać na ten sam obiekt, na który wskazuje zmienna `d`. Sprawdzenie to zakończy się powodzeniem tylko wtedy, gdy obiekt implementuje interfejs `IAnimal`.

W sytuacji, gdy chcemy sprawdzić, czy obiekt implementuje na przykład interfejs `IAnimal`, należy zastosować operator `is`.

```
if (someObjectReference is IAnimal) {
    IAnimal ia = (IAnimal) someObjectReference;
    ia.MakeNoise();
}
else
    Console.WriteLine("Obiekt nie implementuje IAnimal...");
```

Zadaniem tego operatora jest sprawdzenie, czy obiekt wskazywany przez zmienną referencyjną implementuje podany interfejs. Jeżeli tak, wynikiem będzie wartość `true` i można dokonać rzutowania wartości zmiennej referencyjnej na typ `IAnimal`, a następnie użyć tej zmiennej.

Inny sposób wykonania tego sprawdzenia i rzutowania wartości zmiennej polega na użyciu operatora `as`.

```
IAnimal ia = someObjectReference as IAnimal;

if (ia != null)
    ia.MakeNoise();
else
    Console.WriteLine("Obiekt nie implementuje IAnimal...");
```

Operator `as` dokonuje sprawdzenia i rzutowania wartości zmiennej. Jeżeli zmienna nie jest określonego typu, wartością wyrażenia z operatorem `as` jest `null`.

Definiowanie i używanie delegacji

Z delegacji można korzystać we wszystkich językach platformy .NET. W pierwszym przykładzie pokazemy stosowanie delegacji w VB, następnie w C# i na końcu w C++.

W przykładach będziemy korzystać z klasy, której obiekt może informować swoich klientów, że wydarzyło się coś interesującego. Ten prosty model pozwoli nam pokazać, w jaki sposób delegacje mogą być zastosowane w realizacji stylu programowania przypominającego obsługę zdarzeń. Zaczniemy od klasy `EventSource`, której obiekt może być użyty do powiadamiania klientów, gdy coś się wydarzy.

```
Imports System.Collections

Public Class EventSource
    Delegate Sub Notify(ByRef s As String)
    Dim al As New ArrayList()

    Public Sub AddMe(ByRef n As Notify)
        al.Add(n)
    End Sub

    Public Sub TellAll()
        Dim ie As IEnumerator
        ie = al.GetEnumerator()

        While (ie.MoveNext)
            Dim n As Notify
            n = CType(ie.Current, Notify)
            n("Komunikat")
        End While
    End Sub
End Class
```

W wyróżnionym wierszu znajduje się definicja delegacji o nazwie `Notify`, która ma jeden parametr typu `String` i nie zwraca żadnej wartości. Klienci, którzy chcą być powiadamiani, wywołują metodę `AddMe`, przekazując jako argument referencję do metody, która będzie wywoływana przez delegację. Referencja ta jest zapamiętywana w tablicy dynamicznej będącej obiektem klasy `ArrayList`. Ta klasa to jeden ze standardowych typów kolekcji zdefiniowanych w przestrzeni nazw `System.Collections`.

W metodzie `TellAll` tworzony jest enumerator `ie`, który pozwala przejść przez elementy obiektu `ArrayList`. W pętli wywoływana jest metoda `MoveNext`, wykonująca przejście do następnego elementu i metoda `Current`, zwracająca bieżący element. Wynik metody `Current` jest typu `Object`. Za pomocą funkcji `CType` jest on rzutowany na typ `Notify` i przypisywany zmiennej `n`. Następnie wywoływana jest metoda wskazywana przez zmienną `n`, a jako argument przekazywany jest napis *Komunikat*.

Wywołanie metody wskazywanej przez zmienną `n` to zwrotne wywołanie metody realizowanej przez klienta. Obiekt klasy `EventSource` nie „wie”, jakiego typu jest klient, „wie” tylko, że klient przekazał delegację, która pełni rolę pośrednika między obiektem a klientem. Jak widać, wywołanie delegacji wygląda tak samo, jak wywołanie zwykłej funkcji. W nawiasach po nazwie delegacji występują argumenty, które zostaną przekazane do metody wskazywanej przez delegację.

W jaki sposób klienci korzystają z klasy `EventSource`? Klient musi zawierać metodę, którą zwrotnie będzie wywoływała delegacja. Metoda ta musi mieć sygnaturę taką, jak podano w definicji delegacji. W naszym przykładzie musi mieć jeden argument o typie `String` i nie może zwracać żadnej wartości.

```
Public Class EventClient
    Public Sub CallbackFunction(ByRef s As String)
        Console.WriteLine(s)
    End Sub
End Class
```

Jak zdefiniować współpracę klienta i klasy EventSource? Poniżej pokazano kod, który to realizuje.

```
Sub Main()
    ' Deklaracja klienta i źródła
    Dim es as New EventSource()
    Dim ec as New EventClient()

    ' Zainicjalizowanie delegacji, która będzie wywoływała metodę CallbackFunction
    Dim en as EventSource.Notify
    en = New EventSource.Notify(AddressOf ec.CallbackFunction)

    ' Przekazanie delegacji do źródła
    es.AddMe(en)

    ' Użycie delegacji
    es.TellAll()
End Sub
```

Po utworzeniu obiektu klienta można utworzyć delegację, do której przekazywany jest adres funkcji wywoływanej zwrótnie. Do tego celu używane jest słowo kluczowe AddressOf. W rezultacie wywołanie delegacji spowoduje wywołanie metody CallbackFunction z obiektu ec.

Utworzona i zainicjalizowana delegacja musi być przekazana do obiektu es klasy EventSource, do czego używana jest metoda AddMe. Obiekt es dodaje otrzymaną delegację do swojej listy. Wreszcie wywoływana jest metoda TellAll z obiektu es, co powoduje zwrótnie wywołanie metod zarejestrowanych klientów.

A oto definicja klasy EventSource w C#:

```
public class EventSource
{
    public delegate void Notify(String s);

    ArrayList al;

    public EventSource() {
        al = new ArrayList();
    }

    public void AddMe(Notify n)
    {
        al.Add(n);
    }

    public void TellAll()
    {
        IEnumerator ie = al.GetEnumerator();
        while (ie.MoveNext()) {
            Notify n = (Notify)ie.Current;
            n("Komunikat");
        }
    }
}
```

Jak widać, kody w C# i w VB wyglądają niemal identycznie. W klasie zapisanej w C# również korzysta się z kolekcji `ArrayList` w celu przechowania referencji do delegacji oraz z enumeratora do przejścia przez elementy tej kolekcji.

W klasie `EventClass` nie ma żadnych niespodzianek. Zawiera ona jedynie definicję metody, która będzie zwrótnie wywołana przez delegację.

```
public class EventClient
{
    public void CallBackFunction(string s)
    {
        Console.WriteLine("Przekazany argument to: " + s);
    }
}
```

Po zdefiniowaniu tej metody można utworzyć i wywołać delegację.

```
public class EventClient
{
    public void CallBackFunction(string s)
    {
        Console.WriteLine("Przekazany argument to: " + s);
    }

    public static int Main(string[] s)
    {
        EventClient ec = new EventClient();

        EventSource.Notify en = new EventSource.Notify(ec.CallBackFunction);

        EventSource e = new EventSource();
        e.AddMe(en);
        e.TellAll();

        return 0;
    }
}
```

Pierwszą czynnością wykonywaną w programie jest utworzenie obiektu klasy `EventClient`. Najważniejsze rzeczy — jeśli chodzi o delegację — dzieją się w drugim wierszu. Tam tworzona jest nowa delegacja `en` oraz kojarzona z metodą `CallBackFunction` z obiektu `ec`. Wykonanie delegacji `Notify` spowoduje zwrótnie wywołanie metody z obiektu `ec`.

Delegacje mogą być również stosowane w kodzie nadzorowanym C++, jak to pokazano w poniższym przykładzie.

```
#using <mscorlib.dll>
#include <tchar.h>
using namespace System;

__delegate void Notify(String* s);

public __gc class EventClient
{
public:
    void CallBackFunction(String* s)
```



```
{
    Console.WriteLine(s);
}
};

int _tmain()
{
    EventClient* pec = new EventClient;

    Notify* pn = new Notify(pec, &EventClient::CallbackFunction);

    pn->Invoke("Komunikat");

    return 0;
}
```

Do zdefiniowania delegacji w kodzie nadzorowanym C++ używane jest słowo kluczowe `__delegate`. Delegacja ma taką samą sygnaturę, jak w poprzednich przykładach, dlatego też przekazuje się do niej wskaźnik do obiektu klasy `System.String`.



W kodzie nadzorowanym C++ dostęp do obiektów nadzorowanych typów jest możliwy tylko za pośrednictwem wskaźników.

`EventClient` to klasa nadzorowana, zawierająca jedną funkcję używaną do wywołania zwrótnego. W programie głównym (funkcja `main`) tworzona jest delegacja, do której przekazywany jest adres obiektu klasy `EventClient` i wskaźnik do metody, która zostanie wywołana. Po zainicjalizowaniu delegacji, wywoływana jest z niej metoda `Invoke`, co powoduje, że napis „Komunikat” zostanie przekazany do funkcji `CallbackFunction` w obiekcie klasy `EventClient`.

Definiowanie i używanie zdarzeń

Zdarzenia to standardowy mechanizm asynchronicznego powiadamiania. Klasa będąca źródłem zdarzeń publikuje informacje o generowanych przez nią zdarzeniach w stosownych momentach. Obiekty będące klientami zawierają definicje metod, które mają być zwrótnie wywołane. Klienci rejestrują te metody w obiekcie reprezentującym zdarzenie. Obiekt wywoła zwrótnie zarejestrowane metody w momencie wygenerowania zdarzenia.

Zdarzenia są często wykorzystywane w programach z graficznym interfejsem użytkownika, gdzie służą do realizacji komunikacji między składowymi tegoż interfejsu. W .NET ich zastosowanie jest znacznie szersze. Bazują na delegacjach, tak więc niniejszy opis bardzo przypomina opis delegacji z poprzedniego punktu.

We wcześniejszych wersjach VB do obsługi zdarzeń używany był mechanizm `WithEvents`. W VB .NET mechanizm ten jest nadal dostępny, ale oprócz niego pojawił się nowy mechanizm obsługi zdarzeń, który wykorzystuje delegacje. Zastosowanie tego mechanizmu sprawia, że obsługa zdarzeń w VB i w C# wygląda bardzo podobnie, mimo że szczegóły działania nowego mechanizmu są bardziej ukryte w VB. Poniżej przedstawiono przykład znany z poprzedniego punktu, ale zmieniony tak, by można było korzystać ze zdarzeń. Wyróżniono najbardziej istotne wiersze.

```
Imports System

Module Module1
  Sub Main()
    Dim es As New EventSource()
    Dim ob As New EventClient(es)

    es.Notify("Pierwsze wywołanie")
  End Sub

  ' Klasa będąca źródłem zdarzeń
  Public Class EventSource
    Public Event MyEvent(ByVal msg As String)

    Public Sub Notify(ByVal msg As String)
      RaiseEvent MyEvent(msg)
    End Sub
  End Class

  ' Klasa będąca klientem
  Public Class EventClient
    Private src As EventSource

    Public Sub New(ByRef es As EventSource)
      src = es
      AddHandler src.MyEvent, AddressOf Me.GotNotified
    End Sub

    Private Sub GotNotified(ByVal msg As String)
      Console.WriteLine("Przekazany argument to " + msg + " ")
    End Sub
  End Class
End Module
```

W klasie `EventSource`, która będzie generowała zdarzenia, zadeklarowany jest obiekt `Event`, przekazujący obiekt klasy `String` klientom. W celu umożliwienia generowania zdarzeń, w przykładzie zdefiniowano metodę `Notify()`, która generuje zdarzenia korzystając z instrukcji `RaiseEvent`. Po słowie kluczowym `RaiseEvent` występuje nazwa zdarzenia oraz lista argumentów wymaganych przez zdarzenie.

Jedną ze składowych klasy reprezentującej klienta jest referencja do obiektu klasy `EventSource`. W konstruktorze ustawia się tę składową na przekazaną referencję. Następnie za pomocą instrukcji `AddHandler` metoda obsługująca zdarzenie (`GotNotified`) jest kojarzona ze zdarzeniem. W VB zdefiniowana jest również instrukcja `RemoveHandler`, za pomocą której obiekt może odłączyć się od źródła zdarzeń, jeżeli nie chce już być informowany o ich występowaniu.

Przejdźmy teraz do obsługi zdarzeń w C#. Jak się przekonamy, obsługa zdarzeń w C# jest bardziej skomplikowana niż w VB.

```
public static int Main(string[] args) {
  // Utworzenie źródła zdarzeń
  EventSource es = new EventSource();

  // Utworzenie klientów korzystających ze źródła zdarzeń
  EventClient one = new EventClient(es);
  EventClient two = new EventClient(es);
}
```

```
// Wywołanie metody powiadamiającej klientów
es.Notify("Zaszło zdarzenie...");

return 0;
}
```

Najpierw tworzony jest obiekt będący źródłem zdarzeń. W klasie tego obiektu zdefiniowana jest delegacja i zdarzenie, z których korzystają klienci. Następnie tworzone są dwa obiekty klienckie, do których przekazywana jest referencja do źródła zdarzeń. W obiekcie reprezentującym zdarzenie klienci zarejestrują funkcje, które mają być zwrotnie wywołane, po czym będą oczekiwać na to wywołanie. Wreszcie wywoływana jest metoda powiadamiająca klientów. Ponieważ zarejestrowało się dwóch klientów, na konsoli powinny być wypisane dwa komunikaty.

W klasie `EventSource` występują trzy składowe: definicja delegacji, definicja obiektu i metoda `Notify()`.

```
public class EventSource {
    public delegate void MyEvent(object sender, EventArgs ei);
    public event MyEvent OnMyEvent;

    public void Notify(string msg) {
        if (OnMyEvent != null)
            OnMyEvent(this, new EventArgs(msg));
    }
}
```

W przypadku ogólnym delegacja może mieć dowolną sygnaturę, ale jeśli ma współpracować ze zdarzeniami, musi mieć dwa parametry, a typem wyniku musi być `void`. Pierwszym parametrem jest wówczas referencja do obiektu generującego zdarzenie, a drugim referencja do obiektu przechowującego informacje o tym zdarzeniu.

Każde zdarzenie może przesłać opisujące je informacje do klienta. W tym celu należy zdefiniować klasę pochodną od klasy `EventArgs`, która będzie zawierać pola, właściwości i metody niezbędne do przesłania informacji o zdarzeniu. Obiekt tej klasy będzie przekazany do klienta, gdy zajdzie zdarzenie. W naszym przykładzie do klienta przekazywany jest tylko napis, więc klasa ta jest bardzo prosta.

```
public class EventArgs {
    public readonly string msg;

    public EventArgs(string msg) {
        this.msg = msg;
    }
}
```

Pole `msg` zostało zdefiniowane jako pole tylko do odczytu za pomocą słowa kluczowego `readonly`. Taki sposób definiowania pola pozwala jednokrotnie nadać wartość polu (podczas inicjalizacji), po czym jego wartość nie może być zmieniona.

W definicji klasy `EventSource` zdarzenie `OnMyEvent` zostało zadeklarowane oraz skojarzone z delegacją `MyEvent`. Oznacza to, że każdy klient mający metodę o sygnaturze zgodnej z sygnaturą delegacji może zarejestrować tę metodę w źródle zdarzeń, co pokazano w następnym fragmencie kodu.

Ostatnią składową klasy `EventSource` jest metoda `Notify`. Zdarzenie `OnMyEvent` będzie różne od `null`, jeżeli co najmniej jeden klient zarejestruje się jako odbiorca tego zdarzenia. W takiej sytuacji w metodzie `Notify` za pośrednictwem delegacji nastąpi powiadomienie klientów.

Poniżej przedstawiono definicję klasy klienckiej.

```
public class EventClient {
    EventSource src;
    EventSource.MyEvent evt;

    public EventClient(EventSource src) {
        this.src = src;
        evt = new EventSource.MyEvent(EventHasHappened);
        src.OnMyEvent += evt;
    }

    public void EventHasHappened(object sender, EventArgs ei) {
        Console.WriteLine("Przekazany argument to " + ei.Message);
    }
}
```

Na początku zadeklarowano zmienną referencyjną `src` o typie `EventSource` oraz zmienną referencyjną `evt` wskazującą na delegację `MyEvent`, którą zdefiniowano w klasie `EventSource`. Argumentem konstruktora jest referencja do obiektu `EventSource`, która zapamiętywana jest w zmiennej `src`. Następnie tworzona jest delegacja, z którą kojarzona jest metoda `EventHasHappened`. W kolejnym wierszu konstruktora utworzona delegacja jest dodawana, za pomocą operatora `+=`, do listy przechowywanej w zdarzeniu. Operatory `+=` i `-=` mogą być stosowane tylko w przypadku delegacji wielozakresowych (ang. *multicast delegate*). Za pomocą pierwszego z nich klient się rejestruje, drugi służy do wyrejestrowania klienta.

W klasie `EventClient` zdefiniowana jest również metoda `EventHasHappened`, która ma sygnaturę zgodną z sygnaturą delegacji `MyEvent`. Referencja do tej metody jest przekazywana do zdarzenia, dzięki czemu metoda ta zostanie wywołana, gdy obiekt klasy `EventSource` wygeneruje zdarzenie.

Tyle tytułem wyjaśnień. Po skompilowaniu i uruchomieniu, wypisane zostaną na konsoli dwa napisy, gdyż zwrotnie wywołane będą metody z dwóch obiektów.

W ostatnim przykładzie pokazano, w jaki sposób wyrejestrować obiekt, gdy nie chcemy już, by był powiadamiany o zajściu zdarzenia. Najprościej możemy to zrobić dodając metodę `Dispose` do klasy `EventClient`, w której za pomocą operatora `-=` obiekt usuwa swoją delegację z listy przechowywanej przez zdarzenie.

```
public class EventClient : IDisposable {
    EventSource src;
    EventSource.MyEvent evt;

    public EventClient(EventSource src) {
        this.src = src;
        evt = new EventSource.MyEvent(EventHasHappened);
        src.OnMyEvent += evt;
    }
}
```

```

public void Dispose() {
    src.OnMyEvent -= evt;
}

public void EventHasHappened(object sender, EventArgs ei) {
    Console.WriteLine("Przekazany argument to " + ei.msg + "");
}
}

```

Metodę `Dispose` można wywołać następująco:

```

public static int Main(string[] args) {
    // Utworzenie źródła zdarzeń
    EventSource es = new EventSource();

    // Utworzenie klientów korzystających ze źródła zdarzeń
    EventClient one = new EventClient(es);
    EventClient two = new EventClient(es);

    // Wywołanie metody powiadamiającej klienty
    es.Notify("Zaszło zdarzenie...");

    // Pierwszy obiekt wyrejestrowuje się
    one.Dispose();

    // Powtórne wywołanie metody powiadamiającej
    es.Notify("Zaszło drugie zdarzenie...");
    return 0;
}

```

Komunikat o drugim zdarzeniu zostanie wypisany tylko raz, gdyż w momencie wygenerowania tego zdarzenia zarejestrowany jest już tylko jeden obiekt. Wydawać by się mogło, że kod dokonujący wyrejestrowania można umieścić w finalizatorze. Nie jest to jednak dobry pomysł, gdyż obiekt będzie powiadamiany o zdarzeniach do chwili usunięcia z pamięci, a jak wiemy, może się to nigdy nie zdarzyć. W takiej sytuacji nieużywany obiekt będzie powiadamiany o występowaniu zdarzeń, co wywrze negatywny wpływ na efektywność programu.

Przedstawimy teraz, w jaki sposób zdarzenia obsługiwane są w C++. Microsoft zdefiniował w Visual Studio .NET tak zwany zuniifikowany model zdarzeń (ang. *Unified Event Model*). Programiści korzystający z tego modelu mogą zapisać obsługę zdarzenia w zwykłym C++, w ATL i w kodzie nadzorowanym C++ za pomocą tych samych konstrukcji językowych. Konstrukcje te wykorzystują podstawowy mechanizm obsługi zdarzeń platformy .NET. Tak więc, łączenie klientów i źródeł zdarzeń oraz obsługa stanu tych źródeł są realizowane przez system automatycznie. Sposób użycia tego mechanizmu został zilustrowany w poniższym przykładzie, jak zazwyczaj najistotniejsze wiersze zostały wyróżnione.

```

#using <mcorlib.dll>
#include <tchar.h>
using namespace System;

[event_source(managed)]
public __gc class EventSource

```

```

{
public:
    __event void MyEvent(String* msg);

    void Notify(String* msg)
    {
        __raise MyEvent(msg);
    }
};

[event_receiver(managed)]
public __gc class EventClient : public IDisposable
{
    EventSource* pes;

public:
    EventClient(EventSource* pes)
    {
        this->pes = pes;
        __hook(&EventSource::MyEvent, pes, EventClient::GotNotified);
    }

    void GotNotified(String* msg)
    {
        Console::WriteLine("Przekazany argument to '{0}'", msg);
    }

    void Dispose()
    {
        __unhook(&EventSource::MyEvent, pes, EventClient::GotNotified);
    }
};

int _tmain()
{
    // Utworzenie źródła zdarzeń i dwóch klientów
    EventSource* pes = new EventSource();
    EventClient* pec1 = new EventClient(pes);
    EventClient* pec2 = new EventClient(pes);

    // Wywołanie metody powiadamiającej klientów
    pes->Notify("Zaszło zdarzenie");

    // Drugi obiekt wyrejestrowuje się
    pec2->Dispose();
    // Powtórne wywołanie metody powiadamiającej
    pes->Notify("Zaszło drugie zdarzenie");
    return 0;
}

```

Klasa, która ma być źródłem zdarzeń, musi być oznaczona atrybutem `event_source`. Ponadto w definicji klasy należy wskazać, czy generowane będą zdarzenia zwykłego C++, zdarzenia COM, czy też nadzorowanego C++. W przykładzie korzystamy z kodu nadzorowanego, dlatego też użyto argumentu `managed`. W klasie, która będzie źródłem zdarzeń, należy zdefiniować jedno lub więcej zdarzeń, do czego służy słowo kluczowe `__event`. W przykładzie tworzona jest delegacja, z której korzystać mogą inne obiekty.

W kodzie klasy `EventSource` wyróżniono również wiersz, w którym generowane jest zdarzenie. Do tego celu służy słowo kluczowe `__raise`, po którym występuje nazwa zdarzenia i argumenty wymagane przez zdarzenie. Jak widać, `__raise` to dokładny odpowiednik słowa kluczowego `RaiseEvent` występującego w VB.

Klasa, która będzie korzystała ze zdarzeń, musi być oznaczona atrybutem `event_receiver`. W powyższym przykładzie atrybut ten ma argument `managed`, co oznacza, że wykorzystywane będą zdarzenia nadzorowanego C++. Za pomocą słowa kluczowego `__hook` klasa odbierająca zdarzenia rejestruje metodę obsługującą zdarzenie. Po słowie `__hook` występują: adres zdarzenia, które ma być obsługiwane, adres obiektu będącego źródłem zdarzeń i adres funkcji obsługującej zdarzenie. Klasa może się wyrejestrować za pomocą funkcji `__unhook`, co pokazano w metodzie `EventClient::Dispose()`.

W metodzie `main` przedstawiono, jak połączyć ze sobą producentów i konsumentów zdarzeń. Tworzony jest tam obiekt klasy `EventSource` i dwa obiekty klasy `EventClient`, które rejestrują się w obiekcie generującym zdarzenia. Następnie wywoływana jest metoda powiadamiająca klientów o zajściu zdarzenia. W kolejnym kroku jeden z klientów wyrejestrowuje się. W wyniku powtórnego wywołania metody generującej zdarzenia, tylko jeden klient zostanie powiadomiony.

Jak dołączyć atrybuty do klas i składowych?

Większość atrybutów używanych w .NET jest niewidoczna dla programistów. Atrybuty te są tworzone przez kompilator, który umieszcza je w metadanych w pliku wykonywalnym, a korzysta z nich CLR. Istnieją jednak w .NET standardowe atrybuty, które mogą być stosowane przez programistów. Większość tych atrybutów umożliwia współpracę .NET z COM. Tabela 2.4 zawiera standardowe atrybuty występujące w .NET.

Tabela 2.4. Standardowe atrybuty platformy .NET

Atrybut	Przeznaczenie atrybutu
<code>AttributeUsage</code>	Stosowany do atrybutów — wskazuje, do których elementów klasy atrybut może być zastosowany.
<code>Conditional</code>	Stosowany do metod — umożliwia warunkowe dołączenie metody do klasy.
<code>Obsolete</code>	Stosowany do składowych — wskazuje, że składowa jest przestarzała. Kompilator wygeneruje ostrzeżenie lub zgłosi błąd, jeżeli składowa ta zostanie użyta.
<code>Guid</code>	Stosowany do klas i interfejsów — umożliwia podanie GUID dla klasy lub interfejsu współpracującego z COM.
<code>In</code>	Stosowany do parametrów — wskazuje, że dany parametr to parametr [in] w znaczeniu używanym w COM.
<code>Out</code>	Stosowany do parametrów — wskazuje, że dany parametr to parametr [out] w znaczeniu używanym w COM.
<code>Serializable</code>	Stosowany do klas i struktur — wskazuje, że klasa lub struktura podlega serializacji.
<code>Nonserialized</code>	Stosowany do pól klasy podlegającej serializacji — wskazuje, że pole nie podlega serializacji.

W poniższym przykładzie pokazano sposób użycia atrybutu `conditional` w języku C#.

```
public class Foo {
    [conditional(DEBUG)]
    void SomeMethod() {
        ...
    }
}
```

Atrybuty podaje się w nawiasie kwadratowym bezpośrednio przed elementem, do którego się odnoszą. W podanym przykładzie atrybut `conditional` odnosi się do metody `SomeMethod()` i oznacza, że metoda ta ma być skompilowana tylko wówczas, gdy symbol `DEBUG` jest zdefiniowany. Do danego elementu może odnosić się więcej niż jeden atrybut. W takiej sytuacji wymieniamy je kolejno, rozdzielając przecinkami.

Nazwy wszystkich atrybutów kończą się słowem `Attribute`, co ma zapobiec konfliktom nazw, ale można to słowo pominąć. Na przykład, w celu odwołania się do atrybutu `ConditionalAttribute` można podać nazwę `Conditional` lub `ConditionalAttribute`.

Atrybuty mogą mieć argumenty *pozycyjne* (ang. *positional*) i *nazwane* (ang. *named*). Jak sugerują te określenia, argumenty pozycyjne identyfikowane są pozycją na liście argumentów, a argumenty nazwane zapisuje się w postaci „nazwa argumentu = wartość argumentu”. Przykład:

```
[test("abc", 123, name="fred")]
```

Powyższy atrybut ma dwa argumenty pozycyjne i jeden argument nazwany. Argumenty nazwane mogą wystąpić tylko na końcu listy argumentów, stosowane są dla elementów opcjonalnych.

W języku Visual Basic atrybuty podaje się w nawiasie ostrokątnym bezpośrednio przed elementem, do którego się odnoszą. Przykładem niech będzie atrybut `WebMethod`, który wskazuje, że metoda jest usługą Web Service.

```
<WebMethod(> Public Function GetName() As String
    ' Miejsce na kod metody
End Function
```

Z kolei w języku C++, tak jak w C#, atrybuty podaje się w nawiasie kwadratowym, co pokazano w poniższym przykładzie. Taka notacja stosowana była również w języku IDL do zapisu atrybutów COM.

```
[AnAttribute] __gc class Foo
{
    [conditional(DEBUG)]
    void SomeMethod() {
        ...
    }
};
```

Jak definiuje się atrybuty użytkownika?

Atrybuty użytkownika (ang. *custom attribute*) mogą być stosowane zarówno w C++, C#, jak i w VB. Należy pamiętać, że atrybuty są reprezentowane przez klasy, więc mogą mieć swoje pola i składowe.

Atrybuty mogą mieć parametry, które dzielą się na dwie kategorie. Parametry pozycyjne identyfikowane są pozycją na liście argumentów, natomiast parametry nazwane identyfikowane są nazwami. Należy pamiętać, że parametry pozycyjne muszą wystąpić przed parametrami nazwanymi. Parametry pozycyjne definiuje się w konstruktorach klasy reprezentującej atrybut, a parametry nazwane — poprzez właściwości takiej klasy.

Klasa w VB reprezentująca atrybut użytkownika musi być pochodną klasy System.Attribute oraz musi zawierać atrybut AttributeUsage wskazujący, gdzie atrybut użytkownika może wystąpić. Atrybut AttributeUsage ma jeden parametr, którym może być wartość typu wyliczeniowego AttributeTargets. Wartości tego typu opisano w tabeli 2.5.

Tabela 2.5. Wartości typu AttributeTargets wskazujące, gdzie może być użyty atrybut użytkownika

Wartość	Opis
All	Atrybut może być zastosowany wszędzie.
Assembly	Atrybut może być zastosowany do podzespołu.
Class	Atrybut może być zastosowany do klasy.
Constructor	Atrybut może być zastosowany do konstruktora.
Delegate	Atrybut może być zastosowany do delegacji.
Enum	Atrybut może być zastosowany do typu wyliczeniowego.
Event	Atrybut może być zastosowany do zdarzenia.
Field	Atrybut może być zastosowany do pola.
Interface	Atrybut może być zastosowany do interfejsu.
Method	Atrybut może być zastosowany do metody.
Module	Atrybut może być zastosowany do modułu.
Parameter	Atrybut może być zastosowany do parametru.
Property	Atrybut może być zastosowany do właściwości.
ReturnValue	Atrybut może być zastosowany do wartości zwracanej z metody.
Struct	Atrybut może być zastosowany do typu bezpośredniego.

W przykładzie pokazanym poniżej zdefiniowano atrybut o nazwie Author, zawierający jeden parametr pozycyjny (name) i jeden parametr nazwany (ModDate). Atrybut ten może być zastosowany do klas.

```
Imports System

<AttributeUsage(AttributeTargets.Class)> Public Class Author
    Inherits Attribute
    private authorName As String
    private lastModDate As String

    Public Sub New(ByVal name As String)
        authorName = name
    End Sub

    Public ReadOnly Property Name() As String
        Get
            Name = authorName
        End Get
    End Property
End Class
```

```

End Property

Public Property ModDate() As String
    Get
        ModDate = lastModDate
    End Get
    Set
        lastModDate = Value
    End Set
End Property
End Class

```

Jak widać, atrybut `Author` jest klasą pochodną klasy `System.Attribute`, a do zdefiniowania parametrów użyto konstruktora i właściwości. Zdefiniowany atrybut może być zastosowany do klasy w VB w sposób następujący:

```

<Author("Julian")> Public Class Fred
    ...
End Class

```

W języku C++ do zdefiniowania atrybutu użytkownika stosowane są klasy nadzorowane i struktury opatrzone atrybutem `attribute`. Nie muszą one być podklasami klasy `System.Attribute`, co pokazano poniżej.

```

[attribute(target)]
public __gc class Author
{
    ...
};

```

Jeżeli atrybut ten ma być stosowany w innych podzespółach, to klasa tego atrybutu musi być publiczna. Wartością argumentu `target` musi być wartość typu wyliczeniowego `System.AttributeTargets`, który przedstawiono już w tabeli 2.5. Do zdefiniowania parametrów pozycyjnych używane są konstruktory klasy reprezentującej atrybut, a do zdefiniowania parametrów nazwanych — pola i właściwości. Poniżej pokazano definicję atrybutu `Author` w języku C++.

```

[attribute(Class)]
public __gc class Author
{
    String *authorName;           // zmienna do przechowania wartości parametru
    // pozycyjnego
    String *lastModDate;         // zmienna do przechowania wartości parametru
    // nazwanego
public:
    __property String* get_Name () { return authorName; }

    __property String* get_ModDate () { return lastModDate; }
    __property void set_ModDate (String* date) { lastModDate = date; }

    Author(String* name) { authorName = name; }
};

```

Atrybut ten może być zastosowany w sposób następujący:

```

[Author("Julian", ModDate="21/12/00")]
public __gc class Foo
{
};

```

Ponieważ `ModDate` jest parametrem nazwanym, jego wartość musi być poprzedzona nazwą parametru i musi on wystąpić na końcu listy argumentów.

Atrybuty w C# definiuje się podobnie, ale istnieją pewne różnice.

```
[AttributeUsage(AttributeTargets.Class)]
public class Author : System.Attribute
{
    private string authorName; // zmienna do przechowania
                                // wartości parametru pozycyjnego
    private string lastModDate; // zmienna do przechowania
                                // wartości parametru nazwanego

    public string Name {
        get { return authorName; }
    }

    public string ModDate {
        get { return lastModDate; }
        set { lastModDate = value; }
    }

    public Author(string name) { authorName = name; }
};
```

Po pierwsze, klasa reprezentująca atrybut musi być podklasą klasy `System.Attribute`, podczas gdy w C++ atrybuty nie wymagają tego dziedziczenia. Po drugie, do wskazania, gdzie atrybut może być użyty, w C# stosowany jest atrybut `AttributeUsage`. Pomijając te różnice stwierdzimy, że kod w C# ma budowę bardzo zbliżoną do kodu w C++. W obu językach konstruktory stosowane są do definiowania parametrów pozycyjnych, a właściwości — parametrów nazwanych.

Jak odczytać wartość atrybutu?

Większość atrybutów tworzona jest na wyłączny użytek CLR. W pewnych sytuacjach zachodzi jednak konieczność sprawdzenia, czy dany element ma określony atrybut i jakie są argumenty tego atrybutu.

Do tego celu używana jest klasa `System.Type`. Reprezentuje ona definicje typów klas, interfejsów, tablic, typów bezpośrednich i wyliczeniowych. Obiekty tej klasy zawierają mnóstwo informacji o typach i ich właściwościach, ale w niniejszym punkcie skupimy się na odczytywaniu informacji o atrybutach.



`System.Type` to odpowiednik RTTI (ang. *Run-Time Type Information*) w języku C++.)

W poniższym przykładzie pokazano, jak odczytać informacje o atrybucie w języku VB. Skorzystano z atrybutu `Author` zdefiniowanego w poprzednim punkcie.

```
<Author("Julian", ModDate:="21/12/00")> Public Class Foo
...
End Class
```

W celu sprawdzenia, czy klasa `Foo` ma atrybut `Author`, należy pobrać obiekt klasy `System.Type` dla typu `Foo`, a następnie odczytać z niego informacje o atrybutach użytkownika.

```
Imports System
...
' Definicja obiektu klasy Type
Dim tf As Type

' Utworzenie obiektu, z którego odczytana będzie informacja o typie
Dim aa As New Foo()
' Odczytanie informacji o typie
tf = aa.GetType

Dim obj, atts() As Object
atts = tf.GetCustomAttributes(True)

For Each obj In atts
  If (TypeOf obj Is Author) Then
    Console.WriteLine("Atrybut Author, wartością argumentu name jest {0}", _
      (CType(obj, Author)).Name)
  End If
Next
```

W przykładzie tworzony jest obiekt `aa` klasy `Foo`, która jak każda klasa w .NET, jest podklasą klasy `Object`, a tym samym dziedziczy po niej metodę `GetType`. Za pomocą tej metody pobierany jest obiekt klasy `System.Type`, zawierający informacje o typie obiektu `aa`. Następnie z otrzymanego obiektu wywoływana jest metoda `GetCustomAttributes`, której wynikiem jest tablica referencji do obiektów klasy `Object` reprezentujących atrybuty użytkownika. Parametrem tej metody jest wartość `Boolean` określająca, czy należy przejść całe drzewo dziedziczenia w celu pobrania atrybutów. W naszym przykładzie nie ma znaczenia, jaka wartość zostanie podana⁷.

Następnie przeglądane są kolejne elementy tablicy `atts` w celu sprawdzenia, czy któryś z nich nie wskazuje na obiekt typu `Author`. Jeżeli taki element zostanie znaleziony, wówczas po dokonaniu rzutowania na typ `Author`, można ze wskazanego obiektu odczytać wartości argumentów.

Bardzo podobnie można to zapisać w języku C#.

```
using System;
...
Type tf = typeof(Foo);

object[] atts = tf.GetCustomAttributes(true);
foreach(object o in atts) {
  if(o.GetType().Equals(typeof(Author)))
    Console.WriteLine("W klasie Foo występuje atrybut Author");
}
```

Pierwszą czynnością jest pobranie za pomocą operatora `typeof` obiektu klasy `Type` z informacjami o klasie `Foo`. Następnie za pomocą metody `GetCustomAttributes` pobierana jest tablica referencji do obiektów reprezentujących atrybuty użytkownika. Ponieważ tab-

⁷ Klasa `Foo` jest podklasą tylko klasy `Object`, więc nie dziedziczy żadnych atrybutów użytkownika — *przyt. tłum.*

lica ta zawiera referencje do obiektów klasy `Object`, należy sprawdzić, czy któryś z elementów wskazuje na obiekt klasy `Author`. W tym celu używana jest metoda `GetType`. Wywoływana jest z obiektu w przeciwieństwie do operatora `typeof`, który stosowany jest do klasy. Po stwierdzeniu, że klasa ma atrybut `Author`, można odczytać argumenty obiektu.

Na koniec pokażemy, w jaki sposób można zapisać ten przykład w kodzie nadzorowanym C++. Wymaga to oczywiście więcej zabiegów niż w C#, ale i tak jest stosunkowo proste.

```
#using <mscorlib.dll>;
using namespace System;
...
Foo* f = new Foo();
Type* pt = f->GetType();

Object* patts[] = pt->GetCustomAttributes(true);

for (int i=0; i<patts->Length; i++)
{
    Console::WriteLine("Atrybut numer {0} to {1}", __box(i),
        patts[i]->GetType()->get_Name());

    Type* pa = Type::GetType("Author");
    if (patts[i]->GetType()->Equals(pa))
        Console::WriteLine("Obiekt ma atrybut Author");
}
```

Po pierwsze, pobierany jest typ obiektu `f` klasy nadzorowanej `Foo`, a następnie tablica atrybutów użytkownika. W trakcie przeglądania tablicy, dla każdego atrybutu w niej zawartego pobiera się obiekt `Type`, który jest porównywany za pomocą metody `Equals()` z obiektem `Type` atrybutu `Author`. Proszę zwrócić uwagę na użycie słowa kluczowego `__box` do konwersji zmiennej `i` typu `int` na typ `System.Int32`, który jest wymagany przez metodę `Console.WriteLine`.

Jak obsłużyć wyjątek?

Wyjątki są obsługiwane niemal tak samo we wszystkich językach platformy .NET, a mianowicie za pomocą konstrukcji `Try...Catch...Finally`. Konstrukcja ta ma następującą składnię:

```
Try
    * Kod, w którym może wystąpić błąd
Catch ex As SomeExceptionType
    * Obsługa wyjątków typu SomeExceptionType
Catch ex2 As SomeOtherExceptionType
    * Obsługa wyjątków typu SomeOtherExceptionType
Catch
    * Obsługa wszystkich pozostałych wyjątków
Finally
    * Kod, który zawsze jest wykonywany
End Try
```

Kod, w którym może wystąpić błąd, umieszczany jest wewnątrz bloku Try. Blok ten zawiera również instrukcje Catch deklarujące wyjątki przez nie obsługiwane. Musi w nim wystąpić co najmniej jedna instrukcja Catch.⁸ Instrukcje Catch nie mogą wystąpić poza blokiem Try.

Wyjątki reprezentowane są przez obiekty. W instrukcji Catch może znajdować się informacja, jaką klasę wyjątków ta instrukcja obsługuje, na przykład:

```
Catch ex As SomeExceptionType
```

Obiekty reprezentujące wyjątki muszą być obiektami klasy pochodnej klasy System.Exception. Po przechwyceniu wyjątku można za pomocą składowych tej klasy odczytać informacje o zgłoszonym wyjątku, a mianowicie:

- ♦ wartością właściwości StackTrace jest napis reprezentujący zawartość stosu w momencie zgłoszenia wyjątku,
- ♦ wartością właściwości Message jest komunikat opisujący wyjątek,
- ♦ wartością właściwości Source jest nazwa aplikacji lub obiektu, w których zgłoszony został wyjątek,
- ♦ wartością właściwości TargetSite jest nazwa metody, w której zgłoszony został wyjątek.

Przykład zapisany w VB:

```
Try
    f.bar()
Catch ae As NullReferenceException
    Console.WriteLine("Stos wywołań ma postać: {0}", ae.StackTrace)
    Console.WriteLine("Komunikat o wyjątku: {0}", ae.Message)
    Console.WriteLine("Źródło: {0}", ae.Source)
    Console.WriteLine("Metoda: {0}", ae.TargetSite)
End Try
```

Do przechwycenia wszystkich wyjątków można użyć klasy Exception, gdyż są one obiektami klas pochodnych Exception. Jeżeli korzysta się z kilku instrukcji Catch, należy uważać, by wcześniejsza nie obsługiwała wyjątku bardziej ogólnego. Przykład:

```
Try
    ' Kod, w którym może wystąpić błąd
Catch ex As Exception
    ' Obsługa wszystkich wyjątków...
Catch ex2 As SomeOtherExceptionType
    ' Ten kod nigdy nie będzie wykonany...
End Try
```

Ponieważ wszystkie wyjątki są również wyjątkami klasy Exception, zostaną przechwycone przez pierwszą instrukcję Catch. Kompilator C# wygeneruje ostrzeżenie, ale kompilator VB nie.

Bloki Try...Catch można w sobie zagłębiać, należy jedynie pamiętać o poprawnym zamknięciu bloków za pomocą End Try. W praktyce rzadko się korzysta z zagłębiania, gdyż kod jest wtedy nieczytelny.

⁸ Jeżeli występuje instrukcja Finally, nie musi być instrukcji Catch — *przypr. tłum.*

Jak zgłosić wyjątek?

We wszystkich językach platformy .NET wyjątki zgłaszane są za pomocą instrukcji `Throw`. Poniżej pokazano typowe zastosowanie instrukcji `Throw` w programie zapisanym w VB. W ten sam sposób postępuje się w C# i w C++.

```
Public Sub someMethod(ByRef o As Object)
    ' Sprawdzenie, czy przekazano Nothing
    If o = Nothing Then
        Throw New ArgumentException()
    End If
End Sub
```

Argumentem instrukcji `Throw` musi być referencja do obiektu klasy pochodzącej od klasy `System.Exception`. Może to być jedna ze standardowych klas systemowych, jak w powyższym przykładzie, lub też klasa opracowana przez programistę. Konstruktory tych klas mogą mieć parametry, co pozwala przekazać dane do procedury obsługi wyjątku.

W instrukcji `Catch` można powtórnie zgłosić ten sam wyjątek. CLR rozpoczyna wtedy poszukiwanie procedury obsługującej ten wyjątek, począwszy od poprzedniego poziomu na stosie. Przykład:

```
Try
    someFunc(obj)
Catch e As ArgumentException
    Console.WriteLine("Przechwycono wyjątek: {0}", e)
    Throw
End Try
```

Jeżeli jest to konieczne, to w instrukcji `Catch` za pomocą `Throw` można zgłosić zupełnie inny wyjątek.

Jak otrzymać obiekt klasy `Type` z informacjami o typie?

Obiekty klasy `System.Type` przechowują informacje o typach. Podczas tworzenia obiektu klasy `Type` środowisko CLR, za pomocą refleksji, odczytuje informacje o interesującej nas klasie. Do tego celu wykorzystywane są metadane. Utworzony obiekt klasy `Type` zawiera wszystkie informacje o typie, jego polach, właściwościach i metodach.

Zastosowanie klasy `Type` umożliwia tworzenie bardzo ciekawych i wyrafinowanych programów, ale najczęściej obiekty klasy `Type` używane są jako argumenty w wywołaniach metod z biblioteki. Na przykład, korzystając z klasy `System.Array` możemy zbudować tablicę o elementach danego typu, ale wymagany do tego jest obiekt klasy `Type` z informacjami o tym typie. W VB informacje te uzyskuje się przez zastosowanie operatora `GetType`, jak to pokazano poniżej.

```
Dim arr As Array = Array.CreateInstance(GetType(Integer), 10)
```

W przykładzie tworzona jest tablica zawierająca 10 elementów typu `Integer`.

W C# do pobrania informacji o typie można użyć operatora `typeof`.

```
Type t = typeof(int);
```

Jak odczytać informacje o typie?

Po otrzymaniu obiektu `Type` można z niego odczytać informacje o interesującym nas typie, jak pokazano to poniżej w kodzie VB.

```
* Pobranie obiektu Type z informacjami o typie Test
Dim t1 As Type = GetType(Test)

* Odczytanie nazwy typu
Console.WriteLine("Nazwa typu to: {0}", t1.Name)
Console.WriteLine("Nazwa modułu to: {0}", t1.Module)

* Inne informacje...
Console.WriteLine("Jest to klasa: {0}", t1.IsClass)
Console.WriteLine("Jest to typ bezpośredni: {0}", t1.IsValueType)
```

Dla przykładowej klasy `Test` program wypisał następujące informacje:

```
Nazwa typu to: Test
Nazwa modułu to: VBReflect.exe

Jest to klasa: True
Jest to typ wartościowy: False
```

Pierwszy wiersz informuje, że klasa ma nazwę `Test`, natomiast drugi wiersz, że klasa ta znajduje się w pliku `VBReflect.exe`. Klasa `Type` zawiera ponad 30 właściwości, za pomocą których można odczytać informacje o typie. Część z nich jest zupełnie ezoteryczna, te najbardziej użyteczne zebrano w tabeli 2.6.

Tabela 2.6. Właściwości – informacjami o typie zdefiniowane w klasie `System.Type`

Właściwość	Opis
<code>IsAbstract</code>	Wartością jest <code>True</code> , jeżeli klasa jest klasą abstrakcyjną.
<code>IsArray</code>	Wartością jest <code>True</code> , jeżeli typ jest tablicą.
<code>IsClass</code>	Wartością jest <code>True</code> , jeżeli typ jest klasą.
<code>IsInterface</code>	Wartością jest <code>True</code> , jeżeli typ jest interfejsem.
<code>IsPublic</code>	Wartością jest <code>True</code> , jeżeli typ jest publiczny.
<code>IsNotPublic</code>	Wartością jest <code>True</code> , jeżeli typ nie jest publiczny.
<code>IsSealed</code>	Wartością jest <code>True</code> , jeżeli typ jest zapieczętowany.
<code>IsSerializable</code>	Wartością jest <code>True</code> , jeżeli typ jest serializowalny.
<code>IsValueType</code>	Wartością jest <code>True</code> , jeżeli typ jest typem bezpośrednim.

Więcej informacji o typie możemy odczytać korzystając z elementów przestrzeni nazw `System.Reflection`. Zawiera ona wiele struktur pomocnych w odczytywaniu informacji o metodach, właściwościach i polach wchodzących w skład klasy.

Przestrzeń nazw `System.Reflection` może być użyta na dwa sposoby. Po pierwsze, można użyć kwalifikowanych nazw klas z tej przestrzeni, na przykład:

```
Dim mi() As System.Reflection.MethodInfo
```


Można oczywiście korzystać z nazw kwalifikowanych, ale wymaga to pisania sporej ilości kodu. W miarę zagłębiania się w .NET nazwy przestrzeni nazw stają się coraz dłuższe, tak więc ten sposób jest niepraktyczny.

Drugi sposób użycia przestrzeni nazw System.Reflection to zastosowanie w języku VB instrukcji Imports (w C# jest to instrukcja using, a w C++ — #using). Jest to informacja dla kompilatora, że podczas ustalania, do których typów odnoszą się występujące w programie nazwy, powinien uwzględnić nazwy występujące w przestrzeniach nazw z instrukcji Imports. Nie trzeba wówczas stosować nazw kwalifikowanych.

```
Imports System.Reflection
...
Dim mi() As MethodInfo
```

Możemy teraz w następujący sposób wypisać informacje o metodach danej klasy:

```
* Pobranie obiektu Type dla typu Test
Dim t1 As Type = GetType(Test)

* Pobranie obiektów MethodInfo
Dim mi() As MethodInfo = t1.GetMethods
Dim m As MethodInfo

For Each m In mi
    Console.WriteLine("Metoda: {0}", m)
Next
```

W deklaracji tablicy mi o elementach typu MethodInfo tablica ta jest wypełniana za pomocą metody GetMethods. Następnie tablica jest przeglądana i wypisywane są informacje o kolejnych metodach. Składowa ToString zaimplementowana w klasie MethodInfo zwraca sygnaturę metody. Po uruchomieniu program ten wypisze następujące informacje:

```
Metoda: Int32 GetHashCode()
Metoda: Boolean Equals(System.Object)
Metoda: System.String ToString()
Metoda: Void Foo()
Metoda: System.Type GetType()
```

Na wydruku znalazły się wszystkie metody klasy, również te odziedziczone po klasie Object. Poniższy kod umieści dodatkowo na wydruku informacje o tym, czy metoda jest publiczna i czy jest wirtualna:

```
For Each m In mi
    Console.WriteLine("Metoda: {0}, publiczna: {1}, wirtualna: {2}", _
        m, m.IsPublic, m.IsVirtual)
Next
```

Teraz wydruk wygląda następująco:

```
Metoda: Int32 GetHashCode(), publiczna: True, wirtualna: True
Metoda: Boolean Equals(System.Object) , publiczna: True, wirtualna: True
Metoda: System.String ToString(), publiczna: True, wirtualna: True
Metoda: Void Foo(), publiczna: True, wirtualna: False
Metoda: System.Type GetType(), publiczna: True, wirtualna: False
```

W podobny sposób można pobrać informacje o polach i właściwościach klas oraz o parametrach metod.

Dynamiczne tworzenie obiektów

Dynamiczne tworzenie obiektów polega na tworzeniu egzemplarza klasy, której nazwa znana jest dopiero w czasie działania programu. Przykładem może być przeglądarka plików odczytująca w czasie działania programu, jaka klasa, i z jakiego podzespołu, ma być użyta do wyświetlenia pliku o określonym rozszerzeniu. Przeglądarka ta musi tworzyć obiekty klas, których nazwy odczytuje dopiero w czasie wykonania.

Jeżeli klasa, z której należy skorzystać nie jest załadowana, to należy załadować podzespół, w którym ta klasa się znajduje. Służy do tego metoda `Load` klasy `Assembly`. Po załadowaniu podzespołu można pobrać obiekt `Type` dla klasy, której chcemy użyć — w tym celu należy wywołać metodę `GetType` z obiektu klasy `Assembly`. Przykład kodu w VB:

```
* Załadowanie podzespołu o nazwie MyAssembly
Dim ass As Assembly = Assembly.Load("MyAssembly")
```

```
* Pobranie obiektu Type dla klasy, której chcemy użyć
Dim tp As Type = ass.GetType("MyClass")
```

Mając obiekt `Type` reprezentujący klasę, której chcemy użyć, możemy dynamicznie utworzyć egzemplarz tej klasy. Do tego celu służy klasa `Activator`, będąca częścią przestrzeni nazw `System`. Klasa ta zawiera metody pozwalające dynamicznie tworzyć obiekty w sposób pokazany poniżej.

```
* Utworzenie obiektu za pomocą klasy Activator
Dim obj As Object = Activator.CreateInstance(tp)
```

Wynikiem metody `CreateInstance` jest referencja do obiektu typu `Object`, gdyż metoda ta może tworzyć obiekty wszystkich typów.

Składowa `InvokeMember` zdefiniowana w klasie `Type` umożliwia wywołanie metody z dynamicznie utworzonych obiektów przez podanie nazwy tej metody w postaci napisu. Składowa ta jest odpowiednikiem metody `Invoke` z interfejsu `IDispatch`, znanych z COM i Automatykacji (ang. *Automation*). `InvokeMember` pozwala również pobrać i ustawić wartości właściwości, ale w przykładzie pokażemy tylko, jak za jej pomocą wywołać metodę.

```
tp.InvokeMember("Foo", _
    BindingFlags.Default Or BindingFlags.InvokeMethod, _
    Nothing, obj, New Object({}))
```

Pierwszym argumentem jest napis zawierający nazwę metody, którą chcemy wywołać — w przykładzie jest to „Foo”. Drugi argument określa sposób działania składowej `InvokeMember`. Jest to wartość bardzo obszernego typu wyliczeniowego `BindingFlags` — w przykładzie użyto dwóch wartości z tego typu. Pierwsza podana wartość, `BindingFlags.Default`, oznacza, że składowa powinna zastosować domyślne dla danego języka programowania reguły kojarzenia nazw (ang. *binding*). Druga podana wartość, `BindingFlags.InvokeMethod`, oznacza, że wywołana ma być metoda, a nie na przykład właściwość. Trzeci argument najczęściej ma wartość `Nothing` (a w C# `null`). Argument ten określa obiekt definiujący reguły kojarzenia nazw (ang. *binder object*). Czwarty argument to referencja do obiektu, z którego ma być wywołana metoda. Ostatnim argumentem jest tablica referencji do argumentów wymaganych przez wywoływaną metodę. W przykładzie metoda nie ma argumentów, dlatego też przekazywana jest pusta tablica.