helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

.NET Framework 2.0. Zaawansowane programowanie

Autor: Joe Duffy
Tłumaczenie: Paweł Dudziak, Bogdan Kamiński, Grzegorz Werner
ISBN: 978-83-246-0654-2
Tytuł oryginału: Professional .NET Framework 2.0
Format: B5, stron: 672
oprawa twarda
Przykłady na ftp: 78 kB

Przegląd funkcji i możliwości .NET Framework 2.0 oraz środowiska CLR 2.0 dla zaawansowanych

- Jakie możliwości oferuje platforma .NET Framework 2.0 i środowisko CLR 2.0?
- Jak szybko i łatwo pisać aplikacje dla systemu Windows?
- Jak zwiększyć swą produktywność?

Wraz z coraz bardziej rozbudowaną funkcjonalnością .NET Framework rośnie także jej popularność. Możliwość błyskawicznego tworzenia zaawansowanych aplikacji dla systemu Windows na bazie tej platformy oraz wspólnego środowiska uruchomieniowego CLR sprawia, że coraz większa rzesza programistów pragnie poznać te technologie i wykorzystać je do zwiększenia swej produktywności. Wersja 2.0 .NET Framework udostępnia większą liczbę wbudowanych kontrolek, nowe funkcje obsługi baz danych za pomocą ADO.NET, rozbudowane narzędzia do tworzenia witryn internetowych przy użyciu ASP.NET i wiele innych usprawnień znacznie ułatwiających programowanie.

„.NET Framework 2.0. Zaawansowane programowanie" to podręcznik dla programistów, którzy chcą szybko rozpocząć pracę z tą platformą. Dzięki tej książce poznasz mechanizmy działania .NET Framework i środowiska CLR, a także funkcje licznych bibliotek, zarówno tych podstawowych, jak i bardziej wyspecjalizowanych. Dowiesz się, jak przy użyciu tych technologii łatwo zapewniać bezpieczeństwo kodu, debugować oprogramowanie, obsługiwać transakcje, zapewniać współdziałanie aplikacji z kodem niezarządzanym i wykonywać wiele innych potrzebnych operacji.

- Funkcjonowanie środowiska CLR
- Struktura i mechanizmy wspólnego systemu typów (CTS)
- Działanie języka pośredniego (IL) i kompilacji JIT
- Obsługa operacji wejścia-wyjścia
- Tworzenie aplikacji międzynarodowych
- Zapewnianie bezpieczeństwa kodu
- Programowanie współbieżne przy użyciu wątków, domen i procesów
- Umożliwianie współdziałania z kodem niezarządzanym
- Debugowanie oprogramowania
- Stosowanie wyrażeń regularnych
- Programowanie dynamiczne z zastosowaniem metadanych i refleksji
- Obsługa transakcji

Zacznij korzystać z możliwości .NET Framework 2.0 i już dziś zwiększ swą produktywność

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

O autorze 11

Przedmowa 13

Część I Podstawowe informacje o CLR 21

Rozdział 1. Wprowadzenie 23

Historia platformy 23
Nadejście platformy .NET Framework 24
Przegląd technologii .NET Framework 25
Kluczowe udoskonalenia w wersji 2.0 26

Rozdział 2. Wspólny system typów 29

Wprowadzenie do systemów typów 30
Znaczenie bezpieczeństwa typologicznego 31
Statyczna i dynamiczna kontrola typów 33
Typy i obiekty 37
Unifikacja typów 37
Typy referencyjne i wartościowe 39
Dostępność i widoczność 47
Składowe typów 48
Podklasy i polimorfizm 73
Przestrzenie nazw: organizowanie typów 82
Typy specjalne 84
Generyki 94
Podstawy i terminologia 94
Ograniczenia 102
Lektura uzupełniająca 104
Książki poświęcone .NET Framework i CLR 104
Systemy typów i języki 104
Generyki i pokrewne technologie 105
Konkretne języki 105

Rozdział 3. Wewnątrz CLR 107
Intermediate Language (IL) 108
Przykład kodu IL: „Witaj, świecie!” 108
Asemblacja i dezasemblacja IL 110
Abstrakcyjna maszyna stosowa 110
Zestaw instrukcji 113
Wyjątki 127
Podstawy wyjątków 128
Szybkie zamknięcie 140
Wyjątki dwuprzebiegowe 140
Wydajność 142
Automatyczne zarządzanie pamięcią 144
Alokacja 144
Odśmiecanie 150
Finalizacja 153
Kompilacja just-in-time (JIT) 155
Przegląd procesu kompilacji 155
Wywoływanie metod 156
Obsługa architektury 64-bitowej 162
Lektura uzupełniająca 162

Rozdział 4. Podzespoły, wczytywanie i wdrażanie 165
Jednostki wdrażania, wykonywania i wielokrotnego użytku 166
Metadane podzespołu 168
Podzespoły współdzielone (Global Assembly Cache) 177
Podzespoły zaprzyjaźnione 178
Wczytywanie podzespołów 179
Proces wiązania, mapowania i wczytywania 179
Wczytywanie CLR 188
Statyczne wczytywanie podzespołów 189
Dynamiczne wczytywanie podzespołów 191
Przekazywanie typów 195
Generowanie obrazów natywnych (NGen) 197
Zarządzanie buforem (ngen.exe) 198
Adresy bazowe i poprawki 198
Wady i zalety 201
Lektura uzupełniająca 202

Część II Podstawowe biblioteki .NET Framework 203

Rozdział 5. Najważniejsze typy .NET 205
Typy podstawowe 205
Object 207
Liczby 214
Wartości logiczne 219
Łańcuchy 219
IntPtr 227
Daty i czas 227
Pomocnicze klasy BCL 231
Formatowanie 231
Analiza składniowa 235

Konwersja typów podstawowych ... 236
Budowanie łańcuchów ... 237
Odśmiecanie ... 238
Słabe referencje ... 240
Wywołania matematyczne ... 241
Najważniejsze wyjątki ... 244
Wyjątki systemowe ... 245
Inne standardowe wyjątki ... 247
Wyjątki niestandardowe ... 249
Lektura uzupełniająca ... 249

Rozdział 6. Tablice i kolekcje ... 251

Tablice ... 251
Tablice jednowymiarowe ... 252
Tablice wielowymiarowe ... 253
Obsługa tablic w BCL (System.Array) ... 256
Tablice stałe ... 261
Kolekcje ... 261
Kolekcje generyczne ... 262
Słabo typizowane kolekcje ... 283
Porównywalność ... 284
Funkcjonalne typy delegacyjne ... 289
Lektura uzupełniająca ... 291

Rozdział 7. Wejście-wyjście, pliki i sieć ... 293

Strumienie ... 294
Praca z klasą bazową ... 294
Klasy czytające i piszące ... 303
Pliki i katalogi ... 310
Inne implementacje strumieni ... 318
Urządzenia standardowe ... 320
Zapisywanie danych na standardowym wyjściu i standardowym wyjściu błędów ... 320
Czytanie ze standardowego wejścia ... 321
Sterowanie konsolą ... 321
Port szeregowy ... 322
Sieć ... 322
Gniazda ... 323
Informacje o sieci ... 331
Klasy do obsługi protokołów ... 332
Lektura uzupełniająca ... 340

Rozdział 8. Internacjonalizacja ... 343

Definicja internacjonalizacji ... 344
Obsługa platformy ... 344
Proces ... 346
Przykładowe scenariusze ... 348
Dostarczanie zlokalizowanej treści ... 348
Formatowanie regionalne ... 350
Kultura ... 351
Reprezentowanie kultur (CultureInfo) ... 352
Formatowanie ... 357

Zasoby ... 358
Tworzenie zasobów ... 358
Pakowanie i wdrażanie ... 360
Dostęp do zasobów ... 362
Kodowanie ... 363
Obsługa BCL ... 364
Problemy z domyślną kulturą ... 365
Manipulacja łańcuchami (ToString, Parse i TryParse) ... 365
Lektura uzupełniająca ... 369

Część III Zaawansowane usługi CLR 371

Rozdział 9. Bezpieczeństwo ... 373

Zabezpieczenia dostępu do kodu ... 374
Definiowanie zaufania ... 376
Uprawnienia ... 380
Zarządzanie polityką ... 385
Stosowanie zabezpieczeń ... 386
Zabezpieczenia oparte na tożsamości użytkowników ... 391
Tożsamość ... 392
Kontrola dostępu ... 393
Lektura uzupełniająca ... 396

Rozdział 10. Wątki, domeny i procesy ... 397

Wątki ... 400
Przydzielanie pracy wątkom należącym do puli ... 400
Jawne zarządzanie wątkami ... 402
Odizolowane dane wątku ... 411
Współdzielenie elementów pomiędzy wątkami ... 414
Częste problemy współbieżności ... 428
Zdarzenia ... 430
Model programowania asynchronicznego ... 433
Zaawansowane zagadnienia wątkowości ... 436
Domeny AppDomain ... 441
Tworzenie ... 441
Zwalnianie ... 442
Wczytywanie kodu do domeny AppDomain ... 442
Szeregowanie ... 443
Wczytywanie, zwalnianie i wyjątki ... 443
Izolacja domeny AppDomain ... 444
Procesy ... 447
Istniejące procesy ... 447
Tworzenie ... 449
Kończenie procesów ... 450
Lektura uzupełniająca ... 451

Rozdział 11. Interoperacyjność z kodem niezarządzanym ... 453

Wskaźniki, uchwyty i zasoby ... 454
Definicja interoperacyjności ... 454
Natywne wskaźniki w CTS (IntPtr) ... 455
Zarządzanie pamięcią i zasobami ... 458

Niezawodne zarządzanie zasobami (SafeHandle) ... 463
Powiadamianie GC o wykorzystaniu zasobów ... 467
Regiony ograniczonego wykonania ... 469
Interoperacyjność z COM ... 473
Krótka powtórka z COM ... 473
Interoperacyjność wsteczna ... 475
Interoperacyjność w przód ... 481
Praca z kodem niezarządzanym ... 483
Platform Invoke (P/Invoke) ... 484
Łączenie systemów typów ... 487
Lektura uzupełniająca ... 490

Część IV Zaawansowane biblioteki .NET Framework 491

Rozdział 12. Śledzenie i diagnostyka ... 493
Śledzenie ... 494
Dlaczego śledzenie, a nie wyjątki? ... 495
Architektura śledzenia ... 496
Korzystanie ze źródeł śledzenia ... 499
Słuchacze śledzenia ... 506
Konfiguracja ... 513
Lektura uzupełniająca ... 518

Rozdział 13. Wyrażenia regularne ... 519
Podstawowa składnia wyrażeń ... 520
Kilka przykładowych wyrażeń regularnych ... 521
Literały ... 524
Metaznaki ... 526
Obsługa wyrażeń regularnych w BCL ... 539
Wyrażenia ... 539
Wyrażenia kompilowane ... 548
Lektura uzupełniająca ... 551

Rozdział 14. Programowanie dynamiczne ... 553
API refleksji ... 554
API informacyjne ... 555
Odwzorowywanie tokenów i uchwytów ... 569
Atrybuty niestandardowe ... 573
Deklarowanie atrybutów niestandardowych ... 573
Dostęp do atrybutów niestandardowych ... 577
Delegacje ... 578
Wewnątrz delegacji ... 578
Delegacje asynchroniczne ... 585
Metody anonimowe (mechanizm językowy) ... 586
Emitowanie kodu i metadanych ... 588
Generowanie podzespołów ... 588
Lektura uzupełniająca ... 592

Rozdział 15. Transakcje 593
Model programowania transakcyjnego 595
Zasięgi transakcyjne 596
Zagnieżdżanie i kontrola przepływu 601
Integracja z Enterprise Services 605
Menedżery transakcji 607
Lektura uzupełniająca 609

Dodatki 611

Dodatek A Spis instrukcji IL 613

Skorowidz 635

2
Wspólny system typów

System typów to syntaktyczna metoda dowodzenia braku pewnych niepożądanych działań programu przez klasyfikowanie fraz według rodzaju wartości, które w sobie zawierają.

— Benjamin C. Pierce, *Types and Programming Languages*

Ostatecznie wszystkie programy są zbudowane z typów danych. U podstaw każdego języka leżą typy wbudowane, sposoby łączenia ich w celu utworzenia nowych typów oraz metody nadawania nowym typom nazw, aby można było ich używać tak samo jak typów wbudowanych.

— Jim Miller, *The Common Language Infrastructure Annotated Standard*

Środowisko ***Common Language Runtime*** (CLR) — mówiąc ściślej, każda implementacja specyfikacji ***Common Language Infrastructure*** (CLI) — wykonuje kod w ramach dobrze zdefiniowanego systemu typów nazywanego ***Common Type System*** (CTS). CTS stanowi część specyfikacji CLI standaryzowanej przez międzynarodowe organizacje normalizacyjne ECMA i ISO z udziałem przedstawicieli branży i środowisk akademickich. CTS definiuje zbiór struktur i usług, których mogą używać programy przeznaczone do wykonania przez CLR, w tym bogaty system typów umożliwiający tworzenie abstrakcji z wykorzystaniem zarówno typów wbudowanych, jak i zdefiniowanych przez użytkownika. Innymi słowy, CTS stanowi interfejs między programami zarządzanymi a samym środowiskiem uruchomieniowym.

Ponadto CTS wprowadza zbiór reguł i aksjomatów, które definiują weryfikowalne bezpieczeństwo typologiczne. Proces **weryfikacji** klasyfikuje kod na bezpieczny albo niebezpieczny typologicznie, przy czym ta pierwsza kategoria gwarantuje bezpieczne wykonanie kodu w ramach CLR. Wykonywanie bezpieczne typologicznie pozwala uniknąć uszkodzenia zawartości pamięci, do którego mogą doprowadzić nieweryfikowalne programy. CLR zezwala jednak na wykonywanie takich programów, zapewniając dużą elastyczność kosztem potencjalnego uszkodzenia danych i nieoczekiwanych błędów.

Zunifikowany system typów kontroluje dostęp do danych w pamięci, ich przetwarzanie i łączenie. Oferuje statyczne wykrywanie i eliminowanie niektórych klas błędów programistycznych, usystematyzowany sposób budowania i wielokrotnego używania abstrakcji,

wsparcie dla twórców kompilatorów w postaci bezpiecznego, abstrakcyjnego **wirtualnego systemu wykonawczego** (ang. *virtual execution system*, VES), a wreszcie mechanizm samoopisywania się programów z wykorzystaniem **metadanych**. Bezpieczeństwo typologiczne i metadane to dwie kluczowe cechy platformy, które zapewniają największe korzyści pod względem produktywności, bezpieczeństwa i niezawodności. Innymi ważnymi składnikami platformy są usługi uruchomieniowe, takie jak odśmiecanie (*Garbage Collection*, GC), oraz obszerny zbiór wywołań API oferowanych przez .NET Framework. Wszystkie te elementy zostaną dokładnie omówione w kolejnych rozdziałach.

Myślenie w kategoriach „czystego CTS" bywa trudne. Niemal wszyscy twórcy zarządzanych bibliotek i aplikacji pracują z konkretnym językiem, takim jak C#, VB, C++/CLI lub Python. Poszczególne języki oferują własne „spojrzenie" na system uruchomieniowy, abstrahując, ukrywając, a czasem nawet nadmiernie uwydatniając niektóre jego części. Wszystkie jednak są ostatecznie kompilowane do tego samego, podstawowego zbioru konstrukcji. Ta różnorodność jest jednym z powodów, dla których CLR jest tak znakomitym środowiskiem programowania i może obsługiwać tyle odmiennych języków. Z drugiej strony utrudnia to zrozumienie sposobu, w jaki zasady obowiązujące w różnych językach przekładają się na wspólny system typów. Niniejszy rozdział powinien to rozjaśnić.

Wprowadzenie do systemów typów

W tym rozdziale większość idiomów CTS prezentuję z wykorzystaniem C#, choć próbuję wskazywać obszary, w których występuje rozbieżność między semantyką języka a CTS. Ponieważ nie omówiłem jeszcze Common Intermediate Language (CIL) — języka, do którego kompilowane są wszystkie zarządzane programy (zostanie on opisany w rozdziale 3.) — posłużenie się językiem wyższego poziomu, takim jak C#, pozwoli efektywniej wyjaśnić najważniejsze pojęcia.

Dowodem na różnorodność języków obsługiwanych przez CTS mogą być poniższe cztery przykłady, każdy z publicznie dostępnym kompilatorem, który tworzy programy przeznaczone dla CLR: C#, C++/CLI, Python i F#:

- C# to (w dużej mierze) statycznie typizowany, imperatywny język w stylu C. Oferuje bardzo nieliczne funkcje, które wykraczają poza ramy weryfikowalnego bezpieczeństwa typologicznego CLR, i cechuje się bardzo wysokim stopniem obiektowości. C# zapewnia też interesujące mechanizmy języków funkcjonalnych, takie jak funkcje klasy pierwszej i blisko spokrewnione z nimi domknięcia, i nadal zmierza w tym kierunku, o czym świadczy wprowadzenie dedukcji typów oraz lambd w nowszych wersjach języka. Kiedy pisałem tę książkę, był to najpopularniejszy język programowania na platformie CLR.
- C++/CLI to implementacja języka C++ dostosowana do zbioru instrukcji CTS. Programiści tego języka często wykraczają poza ramy weryfikowalnego bezpieczeństwa typologicznego, bezpośrednio manipulując wskaźnikami i segmentami pamięci. Kompilator obsługuje jednak opcje, które pozwalają ograniczyć programy do weryfikowalnego podzbioru języka. Możliwość łączenia świata zarządzanego z niezarządzanym za pomocą C++ jest imponująca — język

ten pozwala rekompilować wiele istniejących programów niezarządzanych i wykonywać je pod kontrolą CLR, oczywiście z korzyściami w postaci GC oraz (w dużej mierze) weryfikowalnego IL.

- Python, tak jak C#, przetwarza dane w sposób obiektowy. Jednak w przeciwieństwie do C# — i bardzo podobnie jak Visual Basic — dedukuje wszystko, co możliwe, i do chwili uruchomienia programu odwleka wiele decyzji, które zwyczajowo podejmuje się w czasie kompilacji. Programiści tego języka nigdy nie pracują na „surowej" pamięci i zawsze operują w ramach weryfikowalnego bezpieczeństwa typologicznego. W tego rodzaju językach dynamicznych kluczowe znaczenie ma produktywność i łatwość programowania, dzięki którym nadają się one dobrze do pisania skryptów lub rozszerzeń istniejących programów. Pomimo to muszą one produkować kod, który uwzględnia typizację oraz inne kwestie związane z CLR gdzieś między kompilacją a wykonaniem programu. Niektórzy twierdzą, że przyszłość należy do języków dynamicznych. Na szczęście CLR obsługuje je równie dobrze jak każdy inny rodzaj języka.
- Wreszcie F# jest typizowanym językiem funkcjonalnym wywodzącym się z O'Caml (który z kolei wywodzi się z języka Standard ML). Oferuje dedukcję typów oraz mechanizmy interoperacyjności przypominające języki skryptowe. F# z całą pewnością eksponuje składnię bardzo odmienną od C#, VB czy Pythona; w istocie wielu programistów posługujących się na co dzień językami w stylu C początkowo może uznać ją za bardzo niekomfortową. F# zapewnia matematyczny sposób deklarowania typów oraz wiele innych użytecznych mechanizmów znanych przede wszystkim z języków funkcjonalnych, takich jak dopasowywanie wzorców. Jest to doskonały język do programowania naukowego i matematycznego.

Każdy z tych języków oferuje odmienny (czasem skrajnie różny) widok systemu typów, a wszystkie kompilują się do abstrakcji z tego samego systemu CTS oraz instrukcji z tego samego języka CIL. Biblioteki napisane w jednym języku można wykorzystać w drugim. Pojedynczy program może składać się z wielu części napisanych w różnych językach i połączonych w jeden zarządzany plik binarny. Zauważmy też, że idea weryfikacji pozwala dowieść bezpieczeństwa typologicznego, a jednocześnie w razie potrzeby ominąć całe sekcje CTS (jak w przypadku manipulowania wskaźnikami do surowej pamięci w C++). Oczywiście, istnieją ograniczenia, które można nałożyć na wykonywanie nieweryfikowalnego kodu. W dalszej części rozdziału wrócimy do tych ważnych zagadnień.

Znaczenie bezpieczeństwa typologicznego

Nie tak dawno temu kod niezarządzany i programowanie w C oraz C++ były faktycznie standardem w branży, a typy — jeśli obecne — stanowiły niewiele więcej niż sposób nadawania nazw przesunięciom w pamięci. Na przykład struktura C to w rzeczywistości duża sekwencja bitów z nazwami, które zapewniają precyzyjny dostęp do przesunięć od adresu bazowego (tzn. pól). Referencje do struktur mogły wskazywać niezgodne instancje, a danymi można było manipulować w zupełnie dowolny sposób. Trzeba przyznać, że C++ był krokiem we właściwym kierunku. Nie istniał jednak żaden system uruchomieniowy, który gwarantowałby, że dostęp do pamięci będzie odbywał się zgodnie z regułami systemu typów. We wszystkich językach niezarządzanych istniał jakiś sposób obejścia iluzorycznego bezpieczeństwa typologicznego.

Takie podejście do programowania okazało się podatne na błędy, co z czasem doprowadziło do ruchu w kierunku języków całkowicie bezpiecznych typologicznie. (Języki z ochroną pamięci były dostępne, jeszcze zanim pojawił się C. Na przykład LISP używa maszyny wirtualnej oraz środowiska z odśmiecaniem przypominającego CLR, ale pozostaje językiem niszowym wykorzystywanym do badań nad sztuczną inteligencją i innych zastosowań akademickich). Z czasem bezpieczne języki i kompilatory zyskiwały na popularności, a dzięki statycznemu wykrywaniu programiści byli powiadamiani o operacjach, które mogą doprowadzić do uszkodzenia danych w pamięci, jak na przykład rzutowanie w górę w C++. W innych językach, takich jak VB i Java, zastosowano pełne bezpieczeństwo typologiczne, aby zwiększyć produktywność programistów i niezawodność aplikacji. Jeśli nawet kompilator zezwalałby na rzutowanie w górę, środowisko uruchomieniowe wyłapałoby nielegalne rzutowania i obsłużyło je w kontrolowany sposób, na przykład zgłaszając wyjątek. CLR idzie w ślady tych języków.

Dowodzenie bezpieczeństwa typologicznego

Środowisko CLR jest odpowiedzialne za dowiedzenie bezpieczeństwa typologicznego kodu przed jego uruchomieniem. Szkodliwe **niezaufane** programy nie mogą obejść tych zabezpieczeń, a zatem nie mogą uszkodzić danych w pamięci. Gwarantuje to, że:

- Dostęp do pamięci odbywa się w dobrze znany i kontrolowany sposób z wykorzystaniem typizowanych referencji. Pamięć nie może zostać uszkodzona po prostu wskutek użycia referencji z błędnym przesunięciem pamięciowym, ponieważ spowodowałoby to zgłoszenie błędu przez weryfikator (a nie ślepe wykonanie żądania). Podobnie instancji typu nie można przypadkowo potraktować jako innego, zupełnie odrębnego typu.
- Wszystkie dostępy do pamięci muszą przechodzić przez system typów, co oznacza, że instrukcje nie mogą skłonić mechanizmu wykonawczego do przeprowadzenia operacji, która spowodowałaby błędny dostęp do pamięci w czasie działania programu. Przepełnienie bufora albo zaindeksowanie dowolnej lokacji pamięci po prostu nie jest możliwe (chyba że ktoś odkryje usterkę w CLR albo świadomie użyje **niezabezpieczonych**, a zatem nieweryfikowalnych konstrukcji).

Zauważmy, że powyższe uwagi dotyczą wyłącznie kodu weryfikowalnego. Korzystając z kodu nieweryfikowalnego, możemy konstruować programy, które hurtowo naruszają te ograniczenia. Oznacza to jednak, że bez zdefiniowania specjalnej polityki nie będzie można uruchomić tych programów w kontekście częściowego zaufania.

Istnieją też sytuacje, w których do wykonania nieprawidłowej operacji można skłonić mechanizmy współpracy z kodem niezarządzanym oferowane przez zaufaną bibliotekę. Wyobraźmy sobie, że zaufane, zarządzane wywołanie API z bibliotek Base Class Libraries (BCL) ślepo przyjmuje liczbę całkowitą i przekazuje ją do kodu niezarządzanego. Jeśli ów kod używa jej do wyznaczenia granic tablicy, napastnik mógłby celowo przekazać nieprawidłowy indeks, aby spowodować przepełnienie bufora. Weryfikacja jest omawiana w niniejszym rozdziale, natomiast częściowe zaufanie zostanie opisane w rozdziale 9. (poświęconym bezpieczeństwu). Twórcy biblioteki ponoszą pełną odpowiedzialność za to, aby ich produkt nie zawierał takich błędów.

Przykład kodu niebezpiecznego typologicznie (w C)

Rozważmy program C, który manipuluje danymi w niebezpieczny sposób, co zwykle prowadzi do tzw. **naruszenia zasad dostępu** do pamięci albo niewykrytego uszkodzenia danych. Naruszenie zasad dostępu występuje podczas przypadkowego zapisu do chronionej pamięci; zwykle jest to bardziej pożądane (i łatwiejsze do zdiagnozowania) niż ślepe nadpisywanie pamięci. Poniższy fragment kodu uszkadza stos, co może spowodować naruszenie przepływu sterowania i nadpisanie różnych danych — w tym adresu powrotnego bieżącej funkcji. Nie jest dobrze:

```
#include <stdlib.h>
#include <stdio.h>

void fill_buffer(char*, int, char);

int main()
{
    int x = 10;
    char buffer[16];
    /* ... */
    fill_buffer(buffer, 32, 'a');
    /* ... */
    printf("%d", x);
}

void fill_buffer(char* buffer, int size, char c)
{
    int i;
    for (i = 0; i < size; i++)
    {
        buffer[i] = c;
    }
}
```

Nasza główna funkcja umieszcza na stosie dwa elementy, liczbę całkowitą `x` oraz 16-znakową tablicę o nazwie `buffer`. Następnie przekazuje wskaźnik do bufora (który, jak pamiętamy, znajduje się na stosie), a odbiorcza funkcja `fill_buffer` używa parametrów `size` i `c` do wypełnienia bufora odpowiednim znakiem. Niestety, główna funkcja przekazała `32` zamiast `16`, co oznacza, że zapiszemy na stosie 32 elementy o rozmiarze typu `char`, o 16 więcej, niż powinniśmy. Rezultat może być katastrofalny. Sytuacja w pewnej mierze zależy od optymalizacji dokonanych przez kompilator — niewykluczone, że nadpiszemy tylko połowę wartości `x` — ale może być bardzo poważna, jeśli dojdzie do nadpisania adresu powrotnego. Dzieje się tak dlatego, że pozwoliliśmy na dostęp do „surowej” pamięci poza ramami prymitywnego systemu typów C.

Statyczna i dynamiczna kontrola typów

Systemy typów często dzieli się na **statyczne** i **dynamiczne**, choć w rzeczywistości różnią się także pod wieloma innymi względami. Tak czy owak, CTS oferuje mechanizmy obsługi obu rodzajów systemów, pozwalając projektantom języków na wybór sposobu, w jaki będzie eksponowane bazowe środowisko uruchomieniowe. Oba style mają zagorzałych zwolenników,

choć wielu programistów czuje się najbardziej komfortowo gdzieś pośrodku. Bez względu na język, w którym napisano program, CLR wykonuje kod w środowisku ze ścisłą kontrolą typów. Oznacza to, że język może unikać kwestii typizacji w czasie kompilacji, ale ostatecznie musi pracować z typologicznymi ograniczeniami weryfikowalnego kodu. Wszystko ma typ, nawet jeśli projektant języka postanowi, że użytkownicy nie będą tego świadomi.

Przyjrzymy się krótko pewnym różnicom między językami statycznymi i dynamicznymi, które są widoczne dla użytkownika. Większość omawianych tu zagadnień nie dotyczy wyłącznie CTS, ale może pomóc w zrozumieniu, co się dzieje w mechanizmie wykonawczym. Podczas pierwszej lektury niniejszego rozdziału Czytelnicy mogą pominąć te informacje, zwłaszcza jeśli zupełnie nie znają CLR.

Kluczowe różnice w strategiach typizacji

Typizacja statyczna próbuje dowieść bezpieczeństwa programu podczas kompilacji, tym samym eliminując całą kategorię błędów wykonania związanych z niedopasowaniem typów oraz naruszeniami zasad dostępu do pamięci. Programy C# są w znacznym stopniu typizowane statycznie, choć mechanizmy takie jak „brudne" rzutowanie w górę pozwalają rozluźnić statyczną kontrolę typów na rzecz dynamizmu. Innymi przykładami statycznie typizowanych języków są Java, Haskell, Standard ML i F#. C++ przypomina C# pod tym względem, że zasadniczo korzysta z typizacji statycznej, choć oferuje pewne mechanizmy, które mogą spowodować błędy w czasie wykonania, zwłaszcza w dziedzinie niebezpiecznych typologicznie manipulacji pamięcią, jak w przypadku C.

Niektórzy uważają, że typizacja statyczna wymusza bardziej rozwlekły i mniej eksperymentalny styl programowania. Programy są na przykład usiane deklaracjami typów, nawet w przypadkach, w których inteligentny kompilator mógłby je wydedukować. Korzyścią jest oczywiście wykrywanie większej liczby błędów w czasie kompilacji, ale w niektórych scenariuszach sztuczne ograniczenia zmuszają programistę do gry w przechytrzanie kompilatora. Języki dynamiczne obarczają środowisko uruchomieniowe odpowiedzialnością za wiele testów poprawności, które w językach statycznych są wykonywane w czasie kompilacji. Niektóre języki przyjmują skrajne podejście i rezygnują ze wszystkich testów, podczas gdy inne stosują mieszankę kontroli dynamicznej i statycznej. Do tej kategorii należą języki takie jak VB, Python, Common LISP, Scheme, Perl i Ruby.

Wiele osób mówi o programach typizowanych silnie lub słabo albo o programowaniu z wczesnym lub późnym wiązaniem. Niestety, terminologia ta rzadko bywa używana konsekwentnie. Ogólnie rzecz biorąc, typizacja silna oznacza, że podczas dostępu do pamięci programy muszą wchodzić w prawidłowe interakcje z systemem typów. Na podstawie tej definicji stwierdzamy, że CTS jest silnie typizowanym środowiskiem wykonawczym. Późne wiązanie to postać programowania dynamicznego, w których konkretny typ zostaje powiązany z docelową operacją dopiero w czasie wykonywania programu. Większość programów wiąże się z odpowiednim tokenem metadanych bezpośrednio w IL. Języki dynamiczne przeprowadzają to wiązanie bardzo późno, tzn. tuż przed ekspedycją (ang. *dispatch*) wywołania metody.

Jedna platforma, by wszystkimi rządzić

CLR obsługuje całe spektrum języków, od statycznych do dynamicznych i wszystko pomiędzy. Sama platforma .NET Framework oferuje całą bibliotekę do późno wiązanego programowania dynamicznego, określaną nazwą **refleksji** (szczegółowy opis można znaleźć w rozdziale 14.). Refleksja eksponuje cały CTS za pośrednictwem wywołań API z przestrzeni nazw `System.Reflection`, oferując funkcje, które ułatwiają twórcom kompilatorów implementowanie języków dynamicznych, a zwykłym programistom pozwalają na eksperymenty z programowaniem dynamicznym.

Przykłady obsługiwanych języków

Przyjrzyjmy się niektórym językom obsługiwanym przez CTS. Poniżej zamieszczono pięć krótkich programów, z których każdy wypisuje dziesiąty element ciągu Fibonacciego (jest to interesujący, dobrze znany algorytm; tutaj przedstawiono jego naiwną implementację). Dwa przykłady są napisane w językach typizowanych statycznie (C++ i F#), jeden w pośrednim (VB), a dwa w typizowanych dynamicznie (Python i Scheme, dialekt LISP-a). Rozbieżności, które widać na pierwszy rzut oka, mają charakter stylistyczny, ale podstawową różnicą jest to, czy IL emitowany przez poszczególne języki jest statyczny, czy też korzysta z dynamicznej kontroli typów i późnego wiązania. Niebawem wyjaśnię, co to oznacza.

C#

```
using System;

class Program
{
    static int Fibonacci(int x)
    {
        if (x <= 1)
            return 1;
        return Fibonacci(x - 1) + Fibonacci(x - 2);
    }

    public static void Main()
    {
        Console.WriteLine(Fibonacci(10));
    }
}
```

F#

```
let rec fibonacci x =
    match x with
        0 -> 1
    | 1 -> 1
    | n -> fibonacci(x - 1) + fibonacci(x - 2);;

fibonacci 10;;
```

VB

```
Option Explicit Off

Class Program
    Shared Function Fibonacci(x)
        If (x <= 1)
            Return 1
        End If

        Return Fibonacci(x - 1) + Fibonacci(x - 2)
    End Function

    Shared Sub Main()
        Console.WriteLine(Fibonacci(10))
    End Sub
End Class
```

Python

```
def fib(i):
    if i <= 1:
        return 1
    return fib(i-1) + fib(i-2)

print fib(10)
```

Scheme

```
(letrec ((fib (lambda (x)
                  (if (<= x 1)
                      1
                      (+ (fib (- x 1)) (fib (- x 2)))))))
    (fib 10))
```

Wszędzie nazwy typów!

Jak widać, tylko wersja C# wspomina, że pracujemy z 32-bitowymi wartościami `int`. Są to **statyczne adnotacje typów**, dzięki którym kompilator może dowieść bezpieczeństwa typologicznego programu. Z drugiej strony wiele języków statycznych, na przykład F#, używa techniki zwanej **dedukcją typów**, która pozwala uniknąć adnotacji, jeśli do ustalenia typów wystarczą literały. W tym przykładzie F# emituje kod IL, który pracuje z wartościami `int`, choć nie określiliśmy tego w kodzie źródłowym. Języki z dedukcją typów czasem wymagają adnotacji, kiedy nie da się wydedukować typu wyłącznie na podstawie użycia.

Język z dedukcją typów może łatwo ustalić, że jakaś zmienna `x` odnosi się do łańcucha, jeśli w programie pojawia się przypisanie `x = "Witaj, świecie"`. W tym nadmiernie uproszczonym przykładzie nie ma potrzeby deklarować typu zmiennej, a mimo to program pozostaje bezpieczny typologicznie. Funkcja `Fibonacci` w języku F# doskonale ilustruje sytuacje, w których dedukcja typów bywa pomocna. W bardziej skomplikowanych przypadkach — na przykład podczas przekazywania danych między granicami oddzielnie skompilowanych jednostek — sytuacja nie wygląda tak różowo.

Pozostałe języki emitują kod, który pracuje z typem `Object` — jak się niebawem przekonamy, jest to korzeń całej hierarchii typów — i przeprowadza ścisłe wiązanie w czasie wykonywania programu. W tym celu wywołuje własną bibliotekę uruchomieniową. Oczywiście, programy typizowane statycznie często są wydajniejsze od dynamicznych, po prostu dlatego, że mogą emitować „surowe" instrukcje IL, zamiast wywołać dodatkowe funkcje w bibliotekach późnego wiązania.

Dostępność kompilatorów

Niektórzy Czytelnicy zapewne zastanawiają się, czy mogą uruchomić powyższe przykłady w CLR. Dobra wiadomość jest taka, że — nie licząc zwykłego C — jest to możliwe! C#, VB i C++ wchodzą w skład dystrybucji .NET Framework 2.0 oraz Visual Studio 2005. F# można pobrać z witryny Microsoft Research pod adresem *http://research.microsoft.com/downloads*. Implementację Pythona dla CLR można pobrać pod adresem *http://workspaces.gotdotnet.com/ironpython*. Wreszcie implementacja Scheme używana podczas kursów na Northeastern University jest dostępna pod adresem *www.ccs.neu.edu/home/will/Larceny/CommonLarceny*.

Pełne omówienie systemów typów, różnic między nimi oraz zalet i wad różnych decyzji projektowych wykraczałoby poza ramy niniejszej książki. Są to jednak interesujące zagadnienia; dodatkowe materiały wymieniono w podrozdziale „Lektura uzupełniająca" na końcu niniejszego rozdziału.

Typy i obiekty

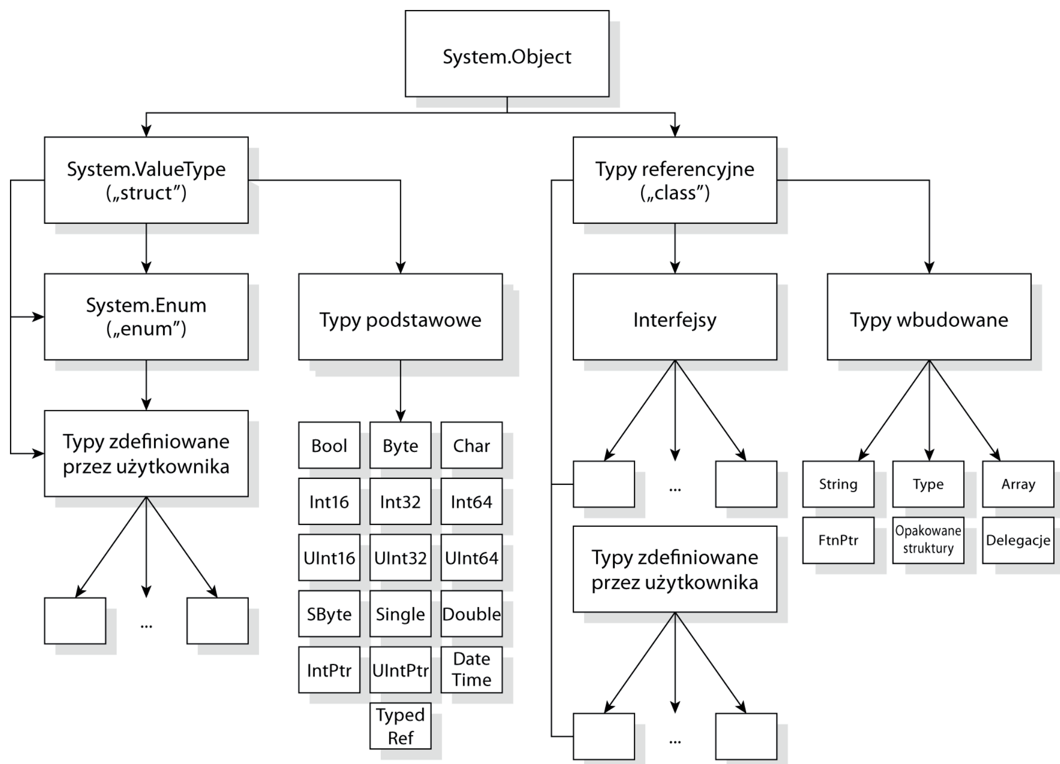
CTS używa abstrakcji wywodzących się ze środowisk programowania obiektowego, co wpływa zarówno na jednostki abstrakcji, jak i na zbiór instrukcji. Jak już wspomniano, ów system typów cechuje się dużą elastycznością i może pracować pod interfejsem niemal każdego języka. Oznacza to jednak, że kiedy mówimy o CTS, musimy posługiwać się kategoriami **klas** i **obiektów** reprezentujących dane i zahermetyzowane operacje.

Unifikacja typów

Wszystkie typy CTS mają wspólny typ bazowy stanowiący korzeń hierarchii: `System.Object`. Jak się niebawem przekonamy, unifikacja ta pozwala na bardzo elastyczne przekazywanie instancji typów w obrębie systemu. Oznacza to zarazem, że każdy typ dziedziczy wspólny zbiór składowych, na przykład metody do przekształcania instancji w reprezentację tekstową, do porównywania tożsamości instancji itd. W rezultacie każda instancja dowolnego typu może być „po prostu obiektem", co pozwala na implementowanie pewnych ogólnych funkcji. Jak się okazuje, jest to niezwykle użyteczne.

Hierarchia typów CTS dzieli się na dwa podstawowe drzewa: **typy referencyjne** i **typy wartościowe**. Typy referencyjne wywodzą się bezpośrednio z `System.Object`, a typy wartościowe — ze specjalnego typu CTS `System.ValueType` (który sam wywodzi się z `System.Object`).

Diagram hierarchii typów abstrakcyjnych oraz niektórych konkretnych typów wbudowanych przedstawiono na rysunku 2.1. Zawiera on kilka konstrukcji specjalnych, którym przyjrzymy się bliżej w dalszej części rozdziału, na przykład interfejsy i wyliczenia, które mają specjalny status w systemie typów.



Rysunek 2.1. *Hierarchia typów CTS*

Zauważmy, że kilka podstawowych typów danych jest wymienionych w gałęzi typów wartościowych. Znajduje się tu większość fundamentalnych typów, które programiści uznają za oczywiste:

- `System.Boolean` (lub `bool` w tekstowym IL) to typ, którego instancje mogą przybierać dwie wartości: `true` lub `false`, reprezentowane w IL odpowiednio przez `1` i `0`. Typ ten zajmuje w pamięci nie 1 bit, ale pełny bajt (8 bitów), dzięki czemu jest wyrównany z natywnymi granicami pamięciowymi, a operacje na nim są bardziej wydajne.
- `System.Char` (`char` w tekstowym IL) reprezentuje pojedynczy 2-bajtowy (16-bitowy) znak Unicode, na przykład „a", „5", „Æ", „á" i wiele, wiele innych.
- `System.SByte`, `Int16`, `Int32`, `Int64` (`int8`, `int16`, `int32` i `int64` w tekstowym IL) reprezentują odpowiednio 1-, 2-, 4- i 8-bajtową (8-, 16-, 32- i 64-bitową) liczbę całkowitą ze znakiem. „Ze znakiem" oznacza, że wartości mogą być dodatnie lub ujemne.

- `System.Byte`, `UInt16`, `UInt32`, `UInt64` (`unsigned int8`, `unsigned int16`, `unsigned int32` i `unsigned int64` w tekstowym IL) reprezentują odpowiednio 1-, 2-, 4- i 8-bajtową (8-, 16-, 32- i 64-bitową) liczbę całkowitą bez znaku. „Bez znaku" oczywiście oznacza, że nie używają one bitu do reprezentowania znaku, a zatem nie mogą przechowywać wartości ujemnych, ale dzięki dodatkowemu bitowi mogą reprezentować dwukrotnie więcej wartości dodatnich niż ich odpowiedniki ze znakiem.
- `System.Single` i `Double` (`float32` i `float64` w tekstowym IL) reprezentują standardowe 4- i 8-bajtowe (32- i 64-bitowe) liczby zmiennopozycyjne. Używa się ich do reprezentowania liczb z częścią całkowitą i ułamkową.
- `System.IntPtr` i `UIntPtr` (`native int` i `unsigned native int` w tekstowym IL) służą do reprezentowania maszynowych liczb całkowitych, odpowiednio ze znakiem i bez znaku. Najczęściej używa się ich do przechowywania wskaźników do pamięci. W systemach 32-bitowych składają się z 4 bajtów (32 bitów), a w systemach 64-bitowych — z 8 bajtów (64 bitów).
- `System.Void` (lub po prostu `void`) to specjalny typ danych, który reprezentuje brak typu. Używa się go tylko w sygnaturach składowych typu, a nie do określania typu lokacji w pamięci.

Z tych typów można konstruować inne abstrakcje hierarchii typów, na przykład:

- Tablice, czyli typizowane sekwencje elementów (na przykład `System.Int32[]`). Tablice zostaną omówione szczegółowo w rozdziale 6.
- Zarządzane i niezarządzane wskaźniki do typizowanych lokacji w pamięci (na przykład `System.Byte*` i `System.Byte&`).
- Bardziej zaawansowane struktury danych, zarówno w referencyjnej, jak i wartościowej hierarchii typów (na przykład `Struct Pair { int x; int y }`).

W rozdziale 5. podam więcej informacji o każdym z typów podstawowych, wyjaśnię definiowane przez nie metody oraz opiszę typy takie jak `Object.String` i `DateTime`, które nie zostały wymienione powyżej. Następnie omówię wyliczenia, interfejsy i delegacje.

Typy referencyjne i wartościowe

Jak już stwierdzono, typy CTS dzielą się na dwie podstawowe kategorie: **typy referencyjne** i **typy wartościowe**. Typy referencyjne często określa się mianem **klas**, a wartościowe — mianem **struktur**, co w dużej mierze jest produktem ubocznym słów kluczowych `class` i `struct`, za pomocą których definiuje się je w języku C#. Nie wspomniano jednak jeszcze, dlaczego istnieje takie rozróżnienie i co ono właściwie oznacza. Zajmiemy się tym w niniejszym punkcie.

Instancja typu referencyjnego, zwana **obiektem**, jest alokowana i zarządzana na odśmiecanej (*Garbage Collected*, GC) stercie, a jej współdzielenie oraz wszystkie odczyty i zapisy odbywają się przez referencję (tzn. za pośrednictwem wskaźnika). Instancja typu wartościowego, zwana **wartością**, jest natomiast alokowana jako sekwencja bitów, a jej położenie zależy od zasięgu, w którym jest zdefiniowana (na stosie wykonania, jeśli jest wartością lokalną, albo na stercie

GC, jeśli jest częścią struktury danych zaalokowanej na stercie). Wartości nie są zarządzane niezależnie przez GC, a w razie współdzielenia są kopiowane. Służą do reprezentowania typów podstawowych i skalarnych.

Aby zilustrować różnicę między współdzieleniem obiektu a współdzieleniem wartości, rozważmy następującą sytuację. Gdy wczytujemy pole zawierające referencję do obiektu, wczytujemy współdzieloną referencję do tego obiektu. Natomiast kiedy wczytujemy pole zawierające wartość, wczytujemy samą wartość, a nie referencję do niej. Dostęp do obiektu spowoduje wyłuskanie wskaźnika w celu odczytania danych ze współdzielonej pamięci, natomiast dostęp do wartości polega na bezpośredniej pracy z sekwencją bitów składającą się na tę wartość.

Niektóre sytuacje dyktują wybór jednej albo drugiej kategorii. Na przykład `System.String` jest typem referencyjnym, a `System.Int32` (tzn. `int32` w IL, `int` w C#) — wartościowym. Zdecydowano się na to nie bez powodu. Wyborem domyślnym zawsze powinna być klasa; jeśli jednak mamy małą strukturę danych z semantyką wartościową, użycie struktury często jest bardziej odpowiednie. Postaram się tu wyjaśnić różnice między obiema kategoriami oraz ich wady i zalety. Przedstawię też pojęcia interfejsów, typów wskaźników, opakowywania i odpakowywania oraz definiowania typów dopuszczających wartość pustą (ang. *nullability*).

Typy referencyjne (klasy)

Większość typów definiowanych przez użytkownika powinna mieć postać klas. Klasy wywodzą się bezpośrednio z `Object` albo z innych typów referencyjnych, co zapewnia większą elastyczność i ekspresywność w hierarchii typów. Wspomniano już, że wszystkie obiekty są alokowane i zarządzane przez GC na odśmiecanej stercie. Jak przekonamy się w rozdziale 3., oznacza to, że obiekt „żyje” tak długo, dopóki istnieje osiągalna referencja do niego, po czym GC może odzyskać i ponownie wykorzystać zajmowaną przez niego pamięć.

Referencje do obiektów mogą przyjmować specjalną wartość `null`, która zasadniczo oznacza, że referencja jest pusta. Innymi słowy, `null` może reprezentować nieobecność obiektu. Jeśli spróbujemy wykonać operację na referencji `null`, zwykle otrzymamy wyjątek `NullReferenceException`. Typy wartościowe nie obsługują takiego mechanizmu, choć w wersji 2.0 .NET Framework wprowadzono specjalny typ (opisywany niżej), który realizuje tę semantykę.

W C# nowy typ referencyjny można utworzyć za pomocą słowa kluczowego `class`, na przykład:

```
class Customer
{
    public string name;
    public string address;

    // Itd., itd., itd.
}
```

Klasa może zawierać każdą z jednostek abstrakcji omawianych w dalszej części rozdziału, w tym pola, metody, konstruktory, właściwości itd.

Typy wartościowe (struktury)

Typy wartościowe, znane też jako struktury, służą do reprezentowania prostych wartości. Każdy typ wartościowy wywodzi się niejawnie z klasy `System.ValueType` i jest automatycznie **pieczętowany** (co oznacza, że inne typy nie mogą się z niego wywodzić; omówimy to później). Instancje typów wartościowych są nazywane wartościami i alokowane na stosie wykonania (w przypadku instancji lokalnych) albo na stercie (w przypadku pól klas lub struktur, które same są polami klas (lub struktur…)). Typy wartościowe używane jako pola statyczne są zwykle alokowane na stercie GC, choć jest to szczegół implementacyjny. Pola statyczne o względnym adresie wirtualnym (ang. *Relative Virtual Address*, RVA) mogą być alokowane w specjalnych segmentach pamięci CLR, jak w przypadku typów skalarnych używanych jako pola statyczne.

Struktury narzucają mniejsze koszty pamięciowe i czasowe podczas pracy z lokalnym stosem, ale oszczędności te mogą szybko zostać zdominowane przez koszty kopiowania wartości, zwłaszcza jeśli wartość zajmuje zbyt wiele miejsca. Ogólnie rzecz biorąc, struktur należy używać do przechowywania niezmiennych danych o rozmiarze nieprzekraczającym 64 bajtów. Wkrótce wyjaśnię, jak można ustalić rozmiar struktury.

Czas życia wartości zależy od miejsca, w którym jej użyto. Jeśli została zaalokowana na stosie wykonania, jest dealokowana podczas usuwania ramki stosu. Dzieje się to podczas wyjścia z metody wskutek powrotu albo nieobsłużonego wyjątku. Porównajmy to ze stertą, czyli segmentem pamięci zarządzanym przez GC. Jeśli na przykład wartość jest instancyjnym polem klasy, to jest alokowana wewnątrz instancji obiektu na zarządzanej stercie i ma ten sam czas życia co instancja obiektu. Jeśli wartość jest instancyjnym polem struktury, to jest alokowana tam, gdzie została zaalokowana zawierająca ją struktura, a zatem ma ten sam czas życia co struktura.

W C# nowy typ wartościowy można utworzyć za pomocą słowa kluczowego `struct`, na przykład:

```
struct Point2d
{
    public int x;
    public int y;
}
```

Struktura może zasadniczo zawierać te same jednostki abstrakcji co klasa. Jednak typ wartościowy nie może definiować bezparametrowego konstruktora, co wynika ze sposobu, w jaki instancje wartości są tworzone przez środowisko uruchomieniowe (opisuję to niżej). Ponieważ inicjalizatory pól są w rzeczywistości kompilowane do postaci konstruktora domyślnego, nie można również określać domyślnych wartości pól struktur. Natomiast ze względu na to, że typy wartościowe wywodzą się niejawnie z `ValueType`, C# nie pozwala definiować typu bazowego, choć nadal można implementować interfejsy.

Wartości

Wartość to po prostu sekwencja bajtów pozbawiona wewnętrznego opisu, a referencja do wartości jest w rzeczywistości wskaźnikiem do pierwszego z tych bajtów. Podczas tworzenia wartości CLR „zeruje” bajty, ustawiając każde pole instancji na wartość domyślną. Tworzenie wartości odbywa się niejawnie w przypadku zmiennych lokalnych i pól typów.

Zerowanie wartości jest semantycznym odpowiednikiem ustawienia jej na `default(T)`, gdzie `T` jest typem docelowej wartości. Polega to po prostu na ustawieniu każdego bajta struktury na `0`, co daje wartość `0` dla liczb całkowitych, `0.0` dla liczb zmiennopozycyjnych, `false` dla wartości logicznych i `null` dla referencji. Wygląda to tak, jakby typ pokazany w poprzednim przykładzie był zdefiniowany w następujący sposób:

```
struct Point2d
{
    public int x;
    public int y;

    public Point2d()
    {
        x = default(int);
        y = default(int);
    }
}
```

Oczywiście, jest to tylko pojęciowy model tego, co zachodzi w rzeczywistości, ale może pomóc Czytelnikom zrozumieć proces tworzenia wartości. Instrukcja `default(T)` odpowiada wywołaniu bezargumentowego konstruktora. Na przykład instrukcje `Point2d p = default(Point2d)` oraz `Point2d p = new Point2d()` są kompilowane do takiego samego kodu IL.

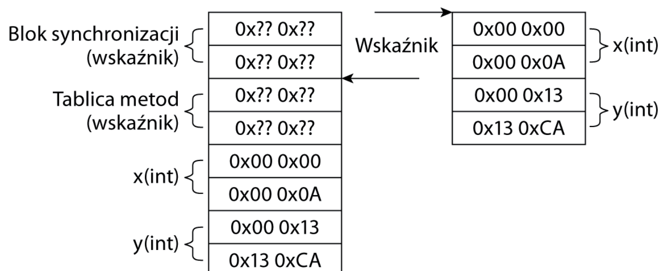
Układ w pamięci

Rozważmy krótko układ obiektów i wartości w pamięci. Powinno to zilustrować kilka fundamentalnych różnic między nimi. Przypuśćmy, że mamy klasę i strukturę, a każda z nich zawiera dwa pola typu `int`:

```
class SampleClass
{
    public int x;
    public int y;
}

struct SampleStruct
{
    public int x;
    public int y;
}
```

Wyglądają one bardzo podobnie, ale ich instancje znacznie się różnią. Przedstawiono to graficznie na rysunku 2.2 i opisano dokładnie poniżej.



Rysunek 2.2. *Układ obiektu i wartości w pamięci*

Na pierwszy rzut oka można stwierdzić, że wartość zajmuje mniej miejsca od obiektu.

Układ obiektu

Obiekt sam się opisuje. Referencja do niego ma rozmiar maszynowego wskaźnika — 32 bity w komputerach 32-bitowych, 64 w 64-bitowych — który wskazuje stertę GC. Celem wskaźnika jest w rzeczywistości inny wskaźnik, który odwołuje się do wewnętrznej struktury danych CLR zwanej **tablicą metod**. Tablica metod upraszcza ekspedycję wywołań metod i służy również do dynamicznego uzyskiwania typu obiektu. Poprzedzające ją **podwójne słowo** (wymyślne określenie 4 bajtów, czyli 32 bitów) tworzy tak zwany **blok synchronizacji** używany do przechowywania różnorodnych informacji, m.in. dotyczących blokowania, współpracy z COM i buforowania skrótów. Dalej następują rzeczywiste wartości składające się na obiekt.

Oznacza to, że każdy obiekt wnosi dodatkowy koszt w postaci mniej więcej poczwórnego słowa (8 bajtów, 64 bity). Dotyczy to oczywiście maszyn 32-bitowych; w komputerach 64-bitowych rozmiar jest nieco większy. Dokładna liczba bajtów zależy od implementacji i może się zwiększyć, jeśli program używa pewnych części środowiska uruchomieniowego. Na przykład blok synchronizacji wskazuje inne wewnętrzne struktury danych CLR, które w miarę używania obiektu mogą obrastać w dodatkowe informacje.

Układ wartości

Wartości nie opisują się same, lecz są po prostu zbiorem bajtów reprezentujących ich stan. Zauważmy, że wskaźnik odnosi się do pierwszego bajta wartości, bez angażowania bloku synchronizacji ani tablicy metod. Niektórzy Czytelnicy zapewne zastanawiają się, jak można kontrolować typy, skoro instancja wartości nie jest powiązana z żadnymi informacjami o typie. Oczywiście każdy typ wartościowy **ma** tablicę metod. Rozwiązanie polega na tym, że lokacja, w której zapisywana jest wartość, może przechowywać tylko wartości pewnego typu. Gwarantuje to weryfikator.

Na przykład ciało metody może mieć pewną liczbę lokalnych slotów, z których każdy przechowuje tylko wartości konkretnego typu; podobnie pola mają precyzyjnie określony typ. Rozmiar lokacji pamięci zajmowanej przez wartość jest zawsze znany statycznie. Na przykład pokazana wyżej struktura `SampleStruct` zajmuje 64 bity pamięci, ponieważ składa się z dwóch 32-bitowych liczb całkowitych. Zauważmy, że nie ma tu żadnych kosztów dodatkowych — dostajemy dokładnie to, co widzimy. Różni się to od typów referencyjnych, które potrzebują dodatkowej przestrzeni do przechowywania informacji o typach. Jeśli struktura nie jest właściwie wyrównana, CLR odpowiednio ją uzupełnia; dzieje się tak w przypadku struktur, które nie są wyrównane do granicy słowa.

> Układ wartości można kontrolować za pomocą specjalnych wskazówek dla CLR. Zagadnienie to zostanie wyjaśnione poniżej, kiedy będzie mowa o polach.

Ponieważ wartości są po prostu zbiorem bajtów reprezentujących dane instancji, wartość nie może przybierać specjalnej wartości `null`. Innymi słowy, `0` jest znaczącą wartością wszystkich typów wartościowych. Typ `Nullable<T>` zapewnia obsługę typów dopuszczających wartość pustą; omówimy to niebawem.

Ustalanie rozmiaru typu

Rozmiar typu można ustalić w C# za pomocą operatora `sizeof(T)`, który zwraca rozmiar typu `T`. W IL realizuje to instrukcja `sizeof`:

```
Console.WriteLine(sizeof(SampleStruct));
```

W przypadku typów podstawowych instrukcja `sizeof` nie jest wykonywana, a zamiast tego w pliku źródłowym osadzana jest stała liczba, ponieważ ich rozmiary są niezależne od implementacji. W przypadku wszystkich innych typów wykonanie tej instrukcji wymaga prawa do wykonywania niezabezpieczonego kodu.

Unifikacja obiektów i wartości

Jak widzieliśmy, obiekty i wartości są traktowane inaczej przez środowisko uruchomieniowe. Są reprezentowane odmiennie: obiekty zajmują więcej miejsca ze względu na ekspedycję metod wirtualnych oraz badanie tożsamości typów, a wartości są surowymi sekwencjami bajtów. Zdarzają się sytuacje, w których ta różnica może spowodować rozbieżność między fizyczną reprezentacją a tym, co chce osiągnąć programista, na przykład:

- Przechowywanie wartości w referencji typu `Object` — jako zmiennej lokalnej, pola lub argumentu — nie będzie działać prawidłowo. Referencja oczekuje, że wskazywane przez nią podwójne słowo będzie wskaźnikiem do tablicy metod obiektu.
- Wywoływanie metod, które zostały zdefiniowane w typie innym niż dana wartość, wymaga zdefiniowania wskaźnika `this` zgodnie z definicją pierwotnej metody. Wartość pochodnego typu wartościowego nie wystarczy.
- Wywoływanie metod wirtualnych na wartości wymagałoby tablicy metod wirtualnych (co zostanie opisane w punkcie poświęconym metodom wirtualnym). Wartość nie ma tablicy metod wirtualnych, a zatem prawidłowa ekspedycja wywołania nie byłaby możliwa.
- Podobnie jak w przypadku metod wirtualnych, wywoływanie metod interfejsu wymaga obecności mapy interfejsu. Jest ona dostępna tylko za pośrednictwem tablicy metod obiektu. Wartości jej nie mają.

Aby rozwiązać powyższe cztery problemy, musimy utworzyć pomost między obiektami a wartościami.

Opakowywanie i odpakowywanie

Tutaj na scenę wkracza **opakowywanie** (ang. *boxing*) i **odpakowywanie** (ang. *unboxing*). Opakowywanie wartości przekształca ją w obiekt poprzez skopiowanie jej do obiektopodobnej struktury na stercie GC. Struktura ta ma tablicę metod i ogólnie wygląda jak obiekt, co zapewnia zgodność z typem `Object` oraz prawidłową ekspedycję metod wirtualnych i metod interfejsu. Odpakowanie typu wartościowego zapewnia dostęp do surowej wartości, z reguły kopiowanej na stos wywołującego i potrzebnej do zapisania wartości z powrotem w slocie, którego typ jest zgodny z bazową wartością.

Niektóre języki, w tym C# i VB, automatycznie opakowują i odpakowują wartości. Na przykład kompilator C# wykrywa przypisanie `int` na `object` w poniższym programie:

```
int x = 10;
object y = x;
int z = (int)y;
```

Kompilator reaguje na to automatycznym wstawieniem instrukcji IL `box`, kiedy zmiennej `y` jest przypisywana wartość `x`, oraz instrukcji `unbox`, kiedy zmiennej `z` jest przypisywana wartość `y`:

```
ldc.i4.s 10
stloc.0
ldloc.0
box [mscorlib]System.Int32
stloc.1
ldloc.1
unbox.any [mscorlib]System.Int32
stloc.2
```

Kod wczytuje stałą `10` i zapisuje ją w lokalnym slocie 0; następnie wczytuje wartość `10` na stos i opakowuje ją, po czym zapisuje ją w lokalnym slocie 1; wreszcie wczytuje opakowaną wartość `10` z powrotem na stos, odpakowuje ją do `int` i zapisuje w lokalnym slocie 2. Czytelnicy prawdopodobnie zauważyli, że IL używa instrukcji `unbox.any`. Różnica między `unbox` a `unbox.any` jest wyjaśniona w rozdziale 3., choć jest to szczegół implementacyjny.

Unifikacja null

Nowy typ `System.Nullable<T>` został dodany do BCL w wersji 2.0, aby zapewnić semantykę `null` dla typów wartościowych. Jest on obsługiwany przez samo środowisko uruchomieniowe. (`Nullable<T>` to typ generyczny. Czytelnicy, którzy nie znają składni i przeznaczenia generyków, powinni najpierw o nich poczytać w punkcie zamieszczonym pod koniec niniejszego rozdziału; składnia będzie wówczas znacznie bardziej przystępna. Warto jednak wrócić w to miejsce — `Nullable<T>` to bardzo użyteczna nowość).

Parametr `T` typu `Nullable<T>` jest ograniczony do struktur. Sam typ oferuje dwie właściwości:

```
namespace System
{
    struct Nullable<T> where T : struct
    {
        public Nullable(T value);
        public bool HasValue { get; }
        public T Value { get; }
    }
}
```

Semantyka tego typu jest taka, że jeśli właściwość `HasValue` jest równa `false`, instancja reprezentuje semantyczną wartość `null`. W przeciwnym razie wartość reprezentuje bazowe pole `Value`. C# oferuje obsługującą to składnię, na przykład pierwsze dwa i drugie dwa wiersze w poniższym przykładzie są równoważne:

```
Nullable<int> x1 = null;
Nullable<int> x2 = new Nullable<int>();
Nullable<int> y1 = 55;
Nullable<int> y1 = new Nullable<int>(55);
```

Co więcej, w C# nazwa typu `T?` jest aliasem `Nullable<T>`, więc powyższy przykład można zapisać następująco:

```
int? x1 = null;
int? x2 = new int?();
int? y1 = 55;
int? y1 = new int?(55);
```

Jest to tylko syntaktyczny lukier. W IL ten kod zostaje przekształcony w konstrukcję `Nullable<T>` oraz instrukcje dostępu do właściwości.

C# przeciąża też testy pustości dla typów `Nullable<T>`, aby ich użycie było bardziej intuicyjne. Oznacza to, że warunek `x == null` — gdzie `x` jest typu `Nullable<T>` — jest spełniony, kiedy właściwość `HasValue` jest równa `false`. Aby zapewnić to samo działanie, kiedy typ `Nullable<T>` jest opakowany — przekształcony w reprezentację na stercie GC — środowisko uruchomieniowe przekształca wartości `Nullable<T>` o właściwości `HasValue == false` w rzeczywiste referencje `null`. Zauważmy, że to pierwsze jest mechanizmem czysto językowym, a to drugie — wewnętrzną cechą traktowania typów przez środowisko uruchomieniowe.

Aby to zilustrować, rozważmy poniższy kod:

```
int? x = null;
Console.WriteLine(x == null);
object y = x; // opakowuje x, zmieniając je w null
Console.WriteLine(y == null);
```

Zgodnie z oczekiwaniami obie instrukcje `WriteLine` wypisują `"True"`. Dzieje się tak jednak tylko dlatego, że zarówno język, jak i środowisko uruchomieniowe zapewniają specjalną obsługę typu `Nullable<T>`.

Zauważmy też, że kiedy wartość `Nullable<T>` jest opakowana, a jej właściwość `HasValue == true`, operacja opakowywania polega na wyodrębnieniu wartości, opakowaniu jej i pozostawieniu na stosie. Wyjaśnię to na poniższym przykładzie:

```
int? x = 10;
Console.WriteLine(x.GetType());
```

Ten fragment wypisuje łańcuch `"System.Int32"`, a nie `"System.Nullable`1<System.Int32>"`, jak można by oczekiwać. Powodem jest to, że aby wywołać metodę `GetType` na instancji typu, wartość musi zostać opakowana. Ma to związek ze sposobem wywoływania metod, mianowicie w celu wywołania metody instancyjnej na typie wartościowym instancję trzeba najpierw opakować. Jest to konieczne dlatego, że odziedziczony kod „nie wie", jak dokładnie należy pracować z typem pochodnym (napisano go jeszcze przed powstaniem tego typu!), a zatem typ musi zostać przekształcony w obiekt. Opakowanie instancji `Nullable<T>`, w której `HasValue == true`, powoduje umieszczenie na stosie opakowanej wartości `int`, a nie `Nullable<T>`, i dopiero na tej wartości wywoływana jest metoda.

Dostępność i widoczność

Zanim zaczniemy szczegółowo zgłębiać każdy z dostępnych typów, omówmy krótko reguły **widoczności** i **dostępności** typów oraz składowych. Widoczność określa, czy typ jest **eksportowany** na zewnątrz podzespołu — jednostki pakowania i wielokrotnego użytku zarządzanych plików binarnych. Dostępność określa natomiast, jaki kod wewnątrz podzespołu może uzyskać dostęp do typu albo konkretnej składowej. W obu przypadkach możemy ograniczyć części systemu, które „widzą” typ albo składową.

Widoczność typów jest określana przez kompilator i w dużej mierze zależna od reguł dostępności. Ogólnie rzecz biorąc, jeśli typ używa **publicznych** lub **rodzinnych** deklaracji dostępności, to staje się widoczny dla innych podzespołów. Wszystkie widoczne typy są oznaczone jako takie w manifeście podzespołu, co zostanie dokładniej opisane w rozdziale 4. Choć reguły te nie są precyzyjne, to sprawdzają się w większości sytuacji. Dalszą dyskusję ograniczymy do modyfikatorów dostępności.

Przez „widzenie” rozumiemy to, że środowisko uruchomieniowe wymusza, aby wszystkie odwołania do typów i składowych odbywały się zgodnie z opisanymi niżej zasadami. Gwarantuje to zachowanie hermetyzacji danych oraz możliwość kontrolowania pewnych niezmienników przez sam typ. Co więcej, IDE Visual Studio ukrywa takie składowe, a kompilator C# sprawdza zasady dostępu, aby programiści przypadkowo nie wprowadzili błędnych odwołań, które zawiodą w czasie wykonywania programu.

Dostępność składowych jest zdefiniowana po części przez leksykalne reguły zasięgu w danym języku programowania, a po części przez modyfikatory dostępności dołączone do samej składowej. Poniżej przedstawiono prawidłowe modyfikatory dostępności. Zauważmy, że większość języków — w tym C# — obsługuje tylko niektóre z nich:

- **Dostęp publiczny** — typ lub składowa są dostępne dla dowolnego kodu, wewnętrznego lub zewnętrznego w stosunku do podzespołu, bez względu na typ. Jest to wskazywane przez słowo kluczowe `public` języka C#.
- **Dostęp prywatny** — dotyczy tylko składowych (i typów zagnieżdżonych, które są specjalną postacią składowej). Oznacza to, że dostęp do składowej ma wyłącznie kod należący do typu, w którym jest ona zdefiniowana. Jest to wskazywane przez słowo kluczowe `private` języka C#. W większości języków składowe są domyślnie prywatne, chyba że programista zdecyduje inaczej.
- **Dostęp rodzinny (chroniony)** — dotyczy tylko składowych i oznacza, że dostęp do składowej ma tylko typ, w którym jest ona zdefiniowana, oraz klasy pochodne (a także pochodne tych klas itd.). Jest to wskazywane przez słowo kluczowe `protected` języka C#.
- **Dostęp podzespołowy** — typ lub składowa są dostępne tylko w podzespole, w którym zostały zaimplementowane. Jest to często domyślne ustawienie typów. W języku C# jest to określane przez słowo kluczowe `internal`.
- **Dostęp rodzinny (chroniony) lub podzespołowy** — dostęp do składowej ma typ, w którym jest ona zdefiniowana, podklasy tego typu oraz dowolny kod wewnątrz tego samego podzespołu, czyli elementy spełniające określone wyżej kryteria dostępu rodzinnego lub podzespołowego. Jest to wskazywane przez słowa kluczowe `protected internal` języka C#.

- **Dostęp rodzinny (chroniony) i podzespołowy** — dostęp do składowej ma tylko typ, w którym jest ona zdefiniowana, oraz podklasy tego typu znajdujące się w tym samym podzespole, czyli elementy spełniające zarówno kryteria dostępu rodzinnego, jak i podzespołowego. C# nie obsługuje tego poziomu dostępności.

Zagnieżdżanie

Część leksykalna opisanych wyżej reguł dostępności staje się istotna dopiero podczas pracy z zagnieżdżonymi definicjami typów. Jest to oczywiście zależne od języka, ponieważ w CLR nie występuje pojęcie zasięgu leksykalnego (pominąwszy reguły dostępu do pól z poziomu metod oraz wczytywania zmiennych lokalnych, argumentów i innych rzeczy związanych z ramką aktywacji metody). CLR pozwala jednak językom na tworzenie zagnieżdżonych typów klasy pierwszej.

Na przykład C# umożliwia zagnieżdżenie typu wewnątrz innego typu (itd.):

```
internal class Outer
{
    private static int state;

    internal class Inner
    {
        void Foo() { state++; }
    }
}
```

Aby uzyskać dostęp do klasy `Inner` na zewnątrz `Outer`, należy użyć kwalifikowanej nazwy `Outer.Inner`. Typy wewnętrzne mają tę samą widoczność co zawierający je typ, a reguły dostępności są takie same jak w przypadku każdej innej składowej. Język może oczywiście oferować zasady przesłaniania, ale większość tego nie robi. Dostępność można również określić ręcznie, jak w przypadku wewnętrznej klasy `Inner` oznaczonej jako `internal`.

Wewnętrzny typ `Inner` ma dostęp do wszystkich składowych typu zewnętrznego, nawet prywatnych, tak jak każda zwykła składowa `Outer`. Co więcej, ma dostęp do dowolnych składowych `family` (`protected`) w hierarchii typów swojej zewnętrznej klasy.

Składowe typów

Typ może mieć dowolną liczbę **składowych**. Składowe te tworzą interfejs i implementację typu, obejmując dane i realizowane operacje. Składowymi są konstruktory, metody, pola, właściwości i zdarzenia. W tym punkcie omówimy je szczegółowo, wspominając także o pewnych ogólnych zagadnieniach.

Istnieją dwa typy składowych: **instancyjne** i **statyczne**. Dostęp do składowych instancyjnych uzyskuje się za pośrednictwem instancji typu, a do składowych statycznych — za pośrednictwem samego typu, a nie jego instancji. Składowe statyczne są zasadniczo składowymi typu, ponieważ pojęciowo należą do samego typu. Składowe instancyjne mają dostęp do dowolnych składowych statycznych lub instancyjnych osiągalnych leksykalnie (według reguł zasięgu danego języka), natomiast składowe statyczne mają dostęp tylko do innych składowych statycznych tego samego typu.

Pola

Pole to nazwana zmienna, która wskazuje typizowany slot danych przechowywany w instancji typu. Pola definiują dane związane z instancją. Pola statyczne są pogrupowane według typu i domeny aplikacji (w przybliżeniu odpowiadającej procesowi; więcej informacji można znaleźć w rozdziale 10.). Nazwy pól w danym typie muszą być niepowtarzalne, choć typy pochodne mogą je przedefiniowywać tak, aby wskazywały odmienne lokacje. Rozmiar typu wartościowego jest w przybliżeniu równy sumie rozmiarów wszystkich jego pól (może na to wpływać uzupełnianie, które gwarantuje, że instancje są wyrównane do granicy słowa maszynowego). Obiekty są podobne, choć (jak opisano wyżej) wnoszą pewne koszty dodatkowe.

Na przykład poniższy typ określa zbiór pól, jedno statyczne i pięć instancyjnych:

```
class FieldExample
{
    private static int idCounter;

    protected int id;
    public string name;
    public int x;
    public int y;
    private System.DateTime createDate;
}
```

Oto odpowiednik tego typu w tekstowym IL:

```
.class private auto ansi beforefieldinit FieldExample
    extends [mscorlib]System.Object
{
  .field private static int32 idCounter
  .field family int32 id
  .field public string name
  .field public int32 x
  .field public int32 y
  .field private valuetype [mscorlib]System.DateTime createDate
}
```

Informacje w składowej statycznej możemy zapisywać lub odczytywać za pomocą metod statycznych albo instancyjnych, a informacje w składowych instancyjnych — za pomocą metod instancyjnych `FieldExample`.

Rozmiar instancji `FieldExample` jest sumą rozmiarów jej pól: `id` (4 bajty), `name` (4 bajty w komputerze 32-bitowym), `x` (4 bajty), `y` (4 bajty) oraz `createDate` (pole typu `DateTime` liczące 8 bajtów). Łączny rozmiar to 24 bajty. Zauważmy, że pole `name` jest zarządzaną referencją i zwiększa się w komputerach 64-bitowych, zatem typ `FieldExample` miałby w nich 28 bajtów. Ponadto dodatkowe koszty sprawiłyby, że obiekt `FieldExample` na stercie GC liczyłby przynajmniej 32 bajty (w komputerze 32-bitowym), a prawdopodobnie jeszcze więcej.

Pola tylko do odczytu

Pole może być oznaczone jako `initonly` w IL (`readonly` w C#), co wskazuje, że po pełnym utworzeniu instancji jego wartość nie może zostać zmieniona. Składowe statyczne przeznaczone tylko do odczytu mogą zostać ustawione wyłącznie podczas inicjalizacji typu, którą

na przykład w C# można wygodnie przeprowadzić za pomocą inicjalizatora zmiennej, a później nie mogą być modyfikowane.

```
class ReadOnlyFieldExample
{
    private readonly static int staticData; // Możemy tu ustawić to pole
    static ReadOnlyFieldExample()
    {
        // Pole staticData możemy również ustawić tutaj
    }

    private readonly int instanceData; // Możemy tu ustawić to pole
    public ReadOnlyFieldExample()
    {
        // Pole instanceData możemy również ustawić
        // w jednym z konstruktorów ReadOnlyFieldExample
    }
}
```

Nie należy jednak mylić funkcji tylko do odczytu z niezmiennością. Jeśli na przykład pole przeznaczone tylko do odczytu odwołuje się do zmiennej struktury danych, zawartość tej struktury można zmieniać. „Tylko do odczytu" oznacza po prostu, że nie można zaktualizować referencji tak, aby wskazywała nową instancję struktury danych. Oto przykład:

```
class ReadOnlyFieldBadExample
{
    public static readonly int[] ConstantNumbers = new int[] {
        0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    };
}

// Jakiś szkodliwy kod może nadal zmieniać liczby w tablicy;
// nie może tylko ustawić zmiennej ConstantNumbers na nową tablicę!
ReadOnlyFieldBadExample.ConstantNumbers[0] = 9;
ReadOnlyFieldBadExample.ConstantNumbers[1] = 8;
// ...
```

Ci, którzy mają dostęp do tablicy `ConstantNumbers` (wszyscy, którzy mają dostęp do `ReadOnlyFieldBadExample`, ponieważ pole jest oznaczone jako `public`), mogą zmieniać jej elementy. Bywa to dość zaskakujące, kiedy pozornie niezmienne wartości w kodzie zostają zmodyfikowane przez użytkowników, co prowadzi do dziwnych błędów (które nie zostały wychwycone podczas testów).

Pola stałe

Można również tworzyć pola **stałe** lub **literalne**. Są one oznaczone jako `static literal` w IL i definiowane z wykorzystaniem modyfikatora `const` w C#. Stałe muszą być inicjalizowane w kodzie programu, na przykład przez użycie wplecionego inicjalizatora pola w C#, i mogą zawierać tylko wartości podstawowe takie jak liczby całkowite, liczby zmiennopozycyjne, literały łańcuchowe itd. Można oczywiście używać prostych wyrażeń matematycznych, pod warunkiem że kompilator będzie mógł zredukować je do stałej wartości podczas kompilacji programu.

```
class UsefulConstants
{
    public const double PI = 3.1415926535897931;

    // Poniższe wyrażenie jest na tyle proste, że kompilator może zredukować je do 4:
    public const int TwoPlusTwo = 2 + 2;
}
```

Zauważmy, że C# emituje stałe jako wartości statyczne, czyli związane z typem. Ma to sens, ponieważ wartość nie może zależeć od instancji. Bądź co bądź, jest to stała.

Użycie pola stałego pozwala na efektywne reprezentowanie wartości stałych, ale może prowadzić do subtelnych problemów związanych z kontrolą wersji. Typowy kompilator w razie wykrycia pola `const` osadza w kodzie literalną wartość stałej. Właśnie tak postępuje kompilator C#. Jeśli stała pochodzi z innego podzespołu, zmiana jej wartości w tamtym podzespole może spowodować problemy, mianowicie skompilowany uprzednio kod IL będzie nadal używał starej wartości. Trzeba będzie go ponownie skompilować, aby korzystał z nowej wartości stałej. Oczywiście, problem ten nie występuje w przypadku wewnętrznych stałych danego podzespołu.

Kontrolowanie układu pól struktur

W dotychczasowej dyskusji przyjmowaliśmy nieco naiwny model układu struktur; dotyczy to nawet punktu poświęconego właśnie temu zagadnieniu! Choć model ten jest prawidłowy w 99 procentach przypadków, istnieje mechanizm, który pozwala przedefiniować domyślny układ struktury. W szczególności można zmieniać kolejność i przesunięcia pól w typach wartościowych. Mechanizm ten zwykle wykorzystuje się do współpracy z kodem niezarządzanym (rozdział 11.), ale bywa również przydatny w zastosowaniach zaawansowanych, na przykład w przypadku unii albo pakowania bitów.

W C# układ jest kontrolowany przez atrybut `System.Runtime.InteropServices.StructLayoutAttribute`. Istnieją trzy podstawowe tryby wskazywane przez wyliczeniową wartość `LayoutKind` przekazywaną do konstruktora atrybutu. Są one kompilowane do odpowiednich słów kluczowych IL (wymienionych poniżej):

- `LayoutKind.Automatic` — ta opcja zezwala CLR na dowolne rozmieszczenie pól typu wartościowego, co jest wskazywane przez słowo kluczowe `autolayout` języka IL i umożliwia optymalizację wyrównania pól. Gdybyśmy na przykład mieli trzy pola, 1-bajtowe, 2-bajtowe i znów 1-bajtowe, struktura danych prawdopodobnie zostałaby ułożona tak, aby pole 2-bajtowe znajdowało się na granicy słowa. Jest to opcja domyślna. Uniemożliwia ona jednak szeregowanie wartości przez granicę z kodem niezarządzanym, ponieważ układ jest nieprzewidywalny.
- `LayoutKind.Sequential` — w tym trybie wszystkie pola są rozmieszczone dokładnie tak, jak określono to w kodzie IL. Jest to wskazywane przez słowo kluczowe `layoutsequential` języka IL. Choć nie pozwala to środowisku uruchomieniowemu na optymalizowanie układu, to gwarantuje przewidywalną kolejność pól podczas współpracy z kodem niezarządzanym.
- `LayoutKind.Explicit` — ta ostatnia opcja obarcza autora struktury pełną odpowiedzialnością za jej układ i jest wskazywana przez słowo kluczowe `explicitlayout` języka IL. Do każdego pola struktury trzeba dołączyć atrybut

FieldOffsetAttribute, który określa, gdzie ma występować pole w ogólnym układzie struktury. Opcja ta wymaga dużej ostrożności, ponieważ przypadkowo można utworzyć nakładające się pola, ale umożliwia ona realizację pewnych zaawansowanych scenariuszy. Przykład przedstawię poniżej.

Podczas definiowania układu można dostarczyć trzech dodatkowych informacji. Pierwszą jest bezwzględny rozmiar (Size, w bajtach) całej struktury. Musi on być równy sumie rozmiarów pól w określonym układzie lub większy od niej. Można to wykorzystać na przykład do poszerzenia rozmiaru struktury danych w celu jej wyrównania albo wtedy, kiedy kod niezarządzany zapisuje jakieś informacje w bajtach poza zarządzanymi danymi struktury. Następnie można określić sposób wypełniania struktury (Pack, również w bajtach). Ustawienie domyślne to 8 bajtów, co oznacza, że struktura danych licząca mniej niż 8 bajtów zostanie uzupełniona tak, aby zajmowała 8 bajtów. Wreszcie opcja CharSet jest używana wyłącznie do współpracy z kodem niezarządzanym, więc nie będziemy jej tutaj omawiać.

Przykład jawnego definiowania układu struktury

Poniższy przykład demonstruje zastosowanie układu jawnego, w którym pola nie są rozmieszczone sekwencyjnie, a ponadto niektóre dane celowo nałożono na siebie, aby utworzyć unię w stylu C:

```
using System;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Explicit)]
struct MyUnion
{
    [FieldOffset(0)] string someText;
    [FieldOffset(7)] byte additionalData;
    [FieldOffset(6)] byte unionTag; // 0 = a, 1 = b
    [FieldOffset(4)] short unionA;
    [FieldOffset(4)] byte unionB1;
    [FieldOffset(5)] byte unionB2;
}
```

Pierwsze cztery bajty są zajęte przez zarządzaną referencję someText. Następnie celowo rozpoczynamy pola unionA i unionB1 po czwartym bajcie; unionA jest typu short i zajmuje 2 bajty, podczas gdy unionB1 zajmuje tylko 1 bajt. Następnie pakujemy dodatkowy bajt po polu unionB1, które nakłada się na drugi bajt pola unionA. Dalej mamy pojedynczy bajt, który określa typ unii; jeśli unionTag jest równy 0, prawidłowe dane zawiera pole unionA; jeśli jest równy 1, prawidłowe dane zawierają pola unionB1 i unionB2. Wreszcie w ostatnim (pojedynczym) bajcie zapisujemy pewne dodatkowe dane (additionalData).

Metody

Funkcje pojawiły się w językach programowania już dawno temu, stając się pierwszym sposobem na abstrahowanie i wielokrotne używanie bloków kodu, których działanie zależy od danych wejściowych. Metoda to po prostu funkcja w żargonie obiektowym. Każdy fragment wykonywalnego kodu użytkownika w CTS jest metodą — czy jest to tradycyjna metoda, czy konstruktor, czy też kod pobierający lub ustawiający właściwość. W tym punkcie wyjaśnię, z czego składają się metody i jak są wywoływane.

Metoda przyjmuje dowolną liczbę **parametrów formalnych** (od tego miejsca nazywanych po prostu **parametrami**) zasadniczo nazywanych zmiennymi. Są one dostępne w **ciele** metody i dostarczane przez wywołujący kod za pośrednictwem **argumentów rzeczywistych** (dalej nazywanych po prostu **argumentami**). Metody mają także pojedynczy wyjściowy **parametr zwrotny**, który jest opcjonalny; jeśli metoda nie zwraca żadnej wartości, określa się to za pomocą typu `void`. Parametr zwrotny pozwala przekazać informacje z powrotem do wywołującego kodu po zakończeniu działania metody. CTS obsługuje też parametry przekazywane przez referencję, które pozwalają wywołującemu i wywoływanemu na współdzielenie tych samych danych. Kilka akapitów niżej wyjaśnię, jak wykorzystać wszystkie te funkcje.

Poniższy kod definiuje metodę `Add`, która przyjmuje dwa parametry i zwraca wartość:

```
class Calculator
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

Metodę tę można wywołać z kodu użytkownika, na przykład w następujący sposób:

```
Calculator calc = new Calculator();
int sum = calc.Add(3, 5);
```

Wygenerowany kod IL będzie zawierał wywołanie metody `Add` obiektu `calc`. Wartości `3` i `5` są przekazywane jako argumenty odpowiadające parametrom, odpowiednio `x` i `y` w pojedynczej **ramce aktywacji** metody `Add`. Metoda następnie wykorzystuje dynamiczne wartości `3` i `5` z ramki aktywacji, dodając je do siebie i zwracając wynik. Ramka jest następnie zdejmowana, a do kodu wywołującego przekazywana jest wartość 8; w tym przypadku kod zapisuje ją w lokalnej zmiennej sum.

Metody, tak jak wszystkie inne składowe, mogą być **instancyjne** lub **statyczne**. Metody instancyjne mają dostęp do specjalnego wskaźnika `this` (nazywanego `this` w C#, a `Me` w VB) odnoszącego się do instancji, na której wywołano metodę. Za jego pomocą mogą uzyskać dostęp do publicznych, chronionych i prywatnych składowych instancyjnych typu (oraz publicznych i chronionych składowych w hierarchii typu). Rozważmy na przykład klasę z instancyjną składową `state`:

```
class Example
{
    int state;
    public void Foo()
    {
        state++;
        this.state++;
    }
}
```

Zauważmy, że C# dedukuje kwalifikację `this` w odwołaniu do pola `state`, ponieważ wykrywa, że w zasięgu znajduje się zmienna instancyjna. C# obsługuje też jawną kwalifikację za pomocą słowa kluczowego `this`. W obu przypadkach w wygenerowanym IL zostanie to zakodowane przez wczytanie zerowego argumentu ramki aktywacji. Jest to konwencja przekazywania wskaźnika `this` od wywołującego do metody instancyjnej. Wszystkie rzeczywiste argumenty zaczynają się od pozycji pierwszej.

Metody statyczne nie mają natomiast wskaźnika `this`. Mają dostęp do prywatnych składowych statycznych typu, ale nie otrzymują aktywnej instancji, która pozwoliłaby im na dostęp do składowych instancyjnych. Argumenty metod statycznych zaczynają się od pozycji zerowej. Jest to całkowicie niewidoczne na poziomie języka C#, ale na poziomie IL trzeba uwzględniać tę różnicę.

Zmienne lokalne

Ramka aktywacji metody może zawierać dwa rodzaje danych lokalnych: argumenty i **zmienne lokalne**. Są one przechowywane na fizycznym stosie; są przydzielane w momencie wywołania metody i usuwane, kiedy metoda kończy działanie (wskutek powrotu albo nieobsłużonego błędu). Omówiliśmy już użycie argumentów. Argumenty to lokacje, do których wywołujący kopiuje dane przekazywane wywoływanemu. Później zbadamy różne style przekazywania argumentów, które — w zależności od scenariusza — mogą obejmować kopiowanie wartości albo przekazywanie referencji do lokalnych struktur danych wywołującego.

Metoda może też używać zmiennych lokalnych. Są one również alokowane na stosie, ale całkowicie niewidoczne dla wywołujących; stanowią szczegół implementacyjny metody. Podczas przydzielania miejsca na stosie zmienne lokalne są inicjalizowane przez zerowanie (co daje `0` dla wartości skalarnych, `0.0` dla wartości zmiennopozycyjnych, `false` dla wartości logicznych i `null` dla referencji), a każda z nich otrzymuje koncepcyjny slot. Sloty są typizowane, aby CLR mogło przydzielić odpowiednią ilość miejsca, a weryfikator sprawdzić, czy zmienne lokalne są używane w sposób bezpieczny typologicznie.

Oto przykład:

```
class Example
{
    public void Foo()
    {
        int x = 0;
        double y = 5.5;
        string s = "Witaj, świecie";
        // ...
    }
}
```

Ten kod definiuje trzy zmienne lokalne, `x`, `y` i `z`, które zostają ponumerowane podczas emisji kodu IL. Większość kompilatorów nadałaby im numery odpowiednio: 0, 1 i 2, ale kompilatory mogą przeprowadzać optymalizacje zmieniające tę kolejność. Oczywiście, w języku C# dodatkowe zmienne lokalne mogą być zdefiniowane w dalszej części metody (inaczej niż w C, w którym wszystkie zmienne muszą zostać wprowadzone na początku funkcji), ale jest to cecha specyficzna dla języka; każda taka zmienna zwykle otrzymuje własny slot (choć kompilatory mogą wielokrotnie wykorzystywać te same sloty).

Przeciążanie

Typ może zawierać wiele metod o takiej samej nazwie. Metody te muszą się jednak różnić w jakiś znaczący sposób; określa się to mianem **przeciążania**. Metody można przeciążać przez definiowanie wyróżniających je parametrów. Różnice typów zwrotnych nie wystarczają do przeciążenia metody.

Przeciążanie pozwala tworzyć metody, które działają podobnie, ale przyjmują różne typy parametrów, co często zwiększa wygodę korzystania z klasy. Ponadto umożliwia określanie domyślnych wartości argumentów, na przykład przez napisanie wersji, które po prostu wywołują inne przeciążone wersje z odpowiednimi wartościami parametrów.

Rozważmy na przykład przeciążoną metodę `Bar` w poniższym typie:

```
class Foo
{
    internal void Bar() { Bar(10); /* wartość "domyślna" */ }
    internal void Bar(int x) { /* ... */ }
    internal void Bar(int x, int y) { /* ... */ }
    internal void Bar(object o) { /* ... */ }
}
```

Kompilatory (a w przypadku języków dynamicznych również programy do późnego wiązania) zajmują się rozwikływaniem przeciążeń, procesem, który polega na dopasowywaniu argumentów wywołującego do odpowiedniej wersji metody. Jest to zasadniczo wyszukiwanie wersji najlepiej pasującej do danego zbioru argumentów. Ogólnie rzecz biorąc, kompilatory wybierają najbardziej specyficzne dopasowanie. Jeśli dopasowanie jest niejednoznaczne albo nie uda się znaleźć pasującej wersji metody, kompilator zwykle zgłasza ostrzeżenie lub błąd.

Jeśli na przykład mamy dwie metody:

```
void Foo(object o);
void Foo(string s);
```

to poniższy kod zostanie dowiązany do metody `Foo(string)`, ponieważ jest ona bardziej specyficzna:

```
Foo("Witaj, wiązanie!");
```

Oczywiście, istnieją bardziej złożone przykłady, które demonstrują szczegóły procesu wiązania. Wystarczy jednak kilka eksperymentów, aby rozwiać ewentualne wątpliwości dotyczące zasad wiązania w danym języku. CTS nie narzuca żadnych reguł dotyczących rozwikływania przeciążeń, więc mogą one być różne w poszczególnych językach.

Styl przekazywania argumentów

Sposób przekazywania argumentów do metody jest często źródłem nieporozumień. CTS używa dwóch podstawowych stylów, **przekazywania przez wartość** i **przekazywania przez referencję**, często określanych skrótowo jako ***byval*** i ***byref*** (od *pass-by-value* i *pass-by-reference*). Różnica bywa trudna do zrozumienia, zwłaszcza ze względu na rozbieżności między typami wartościowymi i referencyjnymi.

Przekazywanie przez nazwę i przez wartość

Przekazywanie przez wartość określa się również mianem **przekazywania przez kopiowanie**. Polega to na utworzeniu kopii argumentu i przekazaniu jej do wywoływanej metody. Jeśli metoda ta zmodyfikuje wartość argumentu, wywołujący nigdy nie „zobaczy" wyniku tej modyfikacji. W przypadku referencji do obiektu wartością jest adres lokacji w pamięci. W związku z tym wiele osób uważa, że przekazywanie referencji do obiektu jest przekazywaniem przez

referencję, a to nieprawda. Tworzymy kopię referencji i przekazujemy ją. Rzeczywiście, kopia wskazuje ten sam adres, ale metoda przyjmująca argument nie może zmienić pierwotnej referencji tak, aby wskazywała ona inny obiekt.

Przekazywanie przez referencję, jak można było oczekiwać, dostarcza wywoływanemu referencji do tej samej lokacji pamięci, którą wskazywał argument wywołującego. Dzięki temu w ciele metody można uzyskać dostęp do współdzielonej referencji, a nawet zmienić ją tak, aby wskazywała inną lokację. Jeśli wywołujący przekaże referencję do jednej ze swoich danych, to zobaczy jej aktualizacje dokonane przez wywoływaną metodę. Weryfikowalność gwarantuje, że referencja do danych nie będzie istniała dłużej niż same dane. C# nie używa domyślnie przekazywania przez referencję; musi to być zadeklarowane jawnie przez metodę i przez wywołującego. W VB tryby przekazywania określa się za pomocą słów kluczowych `ByVal` i `ByRef`.

Oto przykładowa metoda z argumentem *byref*:

```
static void Swap<T>(ref T x, ref T y)
{
    T t = x;
    x = y;
    y = t;
}
```

Dostęp do wartości przekazanej przez referencję — nawet jeśli jest to typ wartościowy — wymaga dodatkowego wyłuskania w celu pobrania danych wskazywanych przez referencję. W powyższym przykładzie wartości `x` i `y` to adresy argumentów wywołującego, a nie same argumenty. Wynikiem wykonania metody `Swap` jest przestawienie wartości wskazywanych przez argumenty. Nie jest to możliwe w przypadku przekazywania przez wartość, bo wówczas modyfikacja `x` i `y` przestawiłaby wartości lokalnych kopii, a nie współdzielonych danych. Modyfikacje nie byłyby widoczne dla wywołującego i zostałyby całkowicie utracone.

Aby wywołać tę metodę w C#, użytkownik musi jawnie określić, że podaje argument *byref*:

```
void Baz()
{
    string x = "Foo";
    string y = "Bar";

    // Wyświetlamy pierwotne wartości x i y:
    Console.WriteLine("x:{0}, y:{1}", x, y);

    // Przestawiamy wartości...
    Swap<string>(ref x, ref y);

    // Teraz wyświetlamy zmodyfikowane wartości x i y:
    Console.WriteLine("x:{0}, y:{1}", x, y);
}
```

Wykonanie tego kodu pokazuje, że przed wywołaniem `Swap` zmienna `x` odnosi się do łańcucha `"Foo"`, a zmienna `y` do łańcucha `"Bar"`. Po wykonaniu `Swap` zmienna `x` odnosi się do `"Bar"`, a `y` do `"Foo"`. Do zilustrowania efektu użyto typu `String`, ponieważ łańcuchy są niezmienne; skoro funkcja `Swap` nie mogła zmodyfikować ich zawartości, to musiała zmienić wartości wskaźników `x` i `y` w lokalnych slotach adresowych funkcji `Baz`.

W IL skutkuje to użyciem instrukcji **wczytania adresu** zamiast typowego **wczytania wartości**. Różnica zostanie lepiej wyjaśniona w rozdziale 3., w którym szczegółowo zbadamy te instrukcje. Na razie wystarczy wiedzieć, że przekazywanie wartości `x` w powyższym przykładzie polega na wczytaniu adresu lokalnej zmiennej `x` za pomocą instrukcji `ldloca`.

Parametry wyjściowe (mechanizm językowy)

Domyślnie wszystkie parametry są parametrami wejściowymi. Oznacza to, że ich wartości są przekazywane metodom przez wywołujących, ale metody nie mogą przekazać zmian z powrotem do wywołujących. Wiemy już, że sprawy wyglądają inaczej w przypadku przekazywania przez referencję. Wartość zwrotną metody można potraktować jak specjalny rodzaj parametru, mianowicie taki, który przekazuje wartość od wywoływanego do wywołującego, ale nie przyjmuje danych wejściowych w momencie wywołania. Parametry zwrotne podlegają istotnemu ograniczeniu: może istnieć tylko jeden.

Język C# pozwala tworzyć dodatkowe **parametry wyjściowe** metody, co jest po prostu szczególną postacią przekazywania przez referencję. Różnica pomiędzy zwykłym parametrem *byref* a parametrem wyjściowym polega na tym, że C# dopuszcza przekazywanie referencji do lokacji pamięci niezainicjalizowanej przez program użytkownika, nie zezwala wywoływanej metodzie na odczytywanie tego parametru, dopóki ona sama nie przypisze mu wartości, i gwarantuje, że wywoływana metoda zapisze coś w parametrze, zanim normalnie powróci.

Rozważmy poniższy przykład:

```
static void Next3(int a, out int x, out int y, out int z)
{
    x = a + 1;
    y = a + 2;
    z = a + 3;
}
```

Metoda ta przypisuje trzy kolejne liczby całkowite następujące po wejściowym parametrze `a` parametrom wyjściowym `x`, `y` i `z`. Podobnie jak w przypadku parametrów przekazywanych przez referencję, wywołujący musi jawnie określić, że przekazuje parametry wyjściowe (`out`):

```
int a, b, c;
Next3(0, out a, out b, out c);
```

Również w tym przypadku kompilator C# generuje kod IL, który po prostu używa instrukcji **wczytania adresu**.

Zarządzane wskaźniki

W C# przekazywanie przez referencję oraz parametry wejściowe działają, opierając się na zarządzanych wskaźnikach. Zarządzany wskaźnik wskazuje wnętrze instancji obiektu, w przeciwieństwie do zwykłego wskaźnika, który wskazuje początek rekordu obiektu. Mówimy, że typem zarządzanego wskaźnika jest `T&`, gdzie `T` jest typem wskazywanego typu danych. Zarządzane wskaźniki pozwalają odwoływać się na przykład do pola instancji, argumentu albo danych na stosie wykonania bez troszczenia się o to, gdzie te dane mogą zostać przeniesione. GC aktualizuje zarządzane wskaźniki podczas przenoszenia instancji, a weryfikacja gwarantuje, że wskaźnik zarządzany nie będzie istniał dłużej niż wskazywane przez niego dane.

Podczas przekazywania argumentu w C# z wykorzystaniem słów kluczowych `ref` lub `out` niejawnie tworzymy nowy zarządzany wskaźnik, który jest przekazywany do metody przez jedną z instrukcji wczytywania adresu, tzn. `ldflda`, `ldarga`, `ldloca` lub `ldelema`. Dostęp do danych poprzez wskaźnik zarządzany prowadzi do dodatkowego poziomu pośredniości, ponieważ mechanizm wykonawczy musi wyłuskać zarządzany wskaźnik, aby dostać się do bazowej wartości lub referencji.

Metody o zmiennej liczbie argumentów

Od czasów C i jego funkcji `printf` języki programowania obsługują funkcje o zmiennej liczbie argumentów. Dzięki temu funkcje mogą przyjmować argumenty, których liczba nie jest znana w czasie kompilacji programu. Kod wywołujący takie funkcje może przekazać do nich nieograniczony zbiór argumentów. W ciele metody używa się specjalnych instrukcji, które pozwalają na wyodrębnienie takich argumentów i pracę z nimi.

CTS obsługuje zmienną liczbę argumentów bezpośrednio na poziomie IL, z wykorzystaniem modyfikatora metody `vararg`. Takie metody używają instrukcji `arglist`, aby uzyskać instancję `System.ArgIterator`, która umożliwia sekwencyjne przetworzenie listy argumentów. Sygnatura metody w ogóle nie określa typów argumentów.

Jednakże wiele języków (w tym C#) używa całkowicie innej konwencji przekazywania zmiennej liczby argumentów. Języki te reprezentują zmienną część argumentów za pomocą tablicy i oznaczają metodę atrybutem niestandardowym `System.ParamArrayAttribute`. Konwencja wywoływania jest taka, że wywołujący umieszcza wszystkie argumenty w tablicy, a metoda po prostu operuje na tej tablicy. Rozważmy poniższy przykład w C#, w którym wykorzystano słowo kluczowe `params`:

```
static void PrintOut(params object[] data)
{
    foreach (object o in data)
        Console.WriteLine(o);
}

PrintOut("Witaj", 5, new DateTime(1999, 10, 10));
```

C# przekształci to wywołanie `PrintOut` w kod IL, który konstruuje tablicę, umieszcza w niej łańcuch `"Witaj"`, liczbę całkowitą `5` oraz obiekt `DateTime` reprezentujący 10 października 1999 roku, po czym przekazuje ją jako argument. Funkcja `PrintOut` operuje na `data` tak jak na każdej innej tablicy.

Metody i podklasy

Tworzenie podtypów i polimorfizm zostaną omówione dokładniej w dalszej części rozdziału. Zakładam jednak, że Czytelnicy znają podstawowe zasady programowania obiektowego, więc tutaj omówię metody wirtualne, przesłanianie i nowe sloty. Do celów niniejszej dyskusji wystarczy wiedzieć, że typ może być podklasą innego typu, co tworzy specjalną relację pomiędzy nimi. Podklasa dziedziczy nieprywatne zmienne typu bazowego, w tym metody. Wyjaśnię tu, w jaki sposób dziedziczone są te metody.

Metody wirtualne i przeciążanie

Domyślnie podklasy dziedziczą zarówno interfejsy, jak i implementacje nieprywatnych metod swoich klas bazowych (aż do najwyższego poziomu hierarchii). Metoda może jednak zostać oznaczona jako wirtualna (`virtual`), co oznacza, że dalsze typy pochodne mogą **przesłaniać** działanie typu bazowego. Przesłonięcie metody wirtualnej polega po prostu na zastąpieniu implementacji odziedziczonej po typie bazowym.

W niektórych językach, na przykład w Javie, metody są domyślnie traktowane jako wirtualne. W C# jest inaczej; trzeba jawnie zadeklarować wirtualność metody za pomocą słowa kluczowego `virtual`.

```
class Base
{
    public virtual void Foo()
    {
        Console.WriteLine("Base::Foo");
    }
}

class Derived : Base
{
    public override void Foo()
    {
        Console.WriteLine("Derived::Foo");
    }
}
```

Klasa `Derived` przesłania metodę `Foo`, oferując jej własną implementację. Kiedy program wywołuje metodę wirtualną `Base::Foo`, właściwa implementacja jest wybierana na podstawie instancji, na której wykonano wywołanie. Rozważmy następujący przykład:

```
// Konstruujemy kilka instancji:
Base base = new Base();
Derived derived = new Derived();
Base derivedTypedAsBase = new Derived();

// Teraz wywołujemy Foo na każdej z nich:
base.Foo();
derived.Foo();
derivedTypedAsBase.Foo();
```

Powyższy fragment kodu wypisuje na konsoli następujące wyniki:

```
Base::Foo
Derived::Foo
Derived::Foo
```

Aby dowieść, że nie jest to żaden trik, rozważmy poniższą metodę:

```
public void DoSomething(Base b) { b.Foo(); }

// W zupełnie innym podzespole...
DoSomething(new Base());
DoSomething(new Derived());
```

Kod ten wypisuje następujące wyniki:

```
Base::Foo
Derived::Foo
```

Ekspedycja metod wirtualnych polega na wykorzystaniu referencji do obiektu w celu uzyskania dostępu do tablicy metod instancji (układ obiektów został omówiony wcześniej w tym rozdziale; Czytelnicy, którzy go nie pamiętają, powinni wrócić do odpowiedniego punktu). Tablica metod przechowuje zbiór slotów, z których każdy jest wskaźnikiem do pewnego fragmentu kodu. Podczas ekspedycji metod wirtualnych najpierw zostaje wyłuskany wskaźnik do obiektu, a następnie wskaźnik do odpowiedniego slotu metody wirtualnej w tablicy metod; wartość przechowywana w tym slocie jest używana jako docelowy adres wywołania metody. W rozdziale 3. można znaleźć więcej informacji o ekspedycji metod wirtualnych, zwłaszcza o roli kompilatora JIT w tym procesie.

Nowe sloty

Metoda może być również oznaczona jako `newslot` (zamiast `override`), co wskazuje, że nie przesłania ona implementacji typu bazowego, ale wprowadza zupełnie nową implementację. Modyfikatora `newslot` używa się, kiedy nie istnieje implementacja typu bazowego, aby zapobiec problemom związanym z kontrolą wersji.

W C# deklaruje się to za pomocą słowa kluczowego `new`. Wyjaśnię to, opierając się na poprzednim przykładzie:

```
class Base
{
    public virtual void Foo()
    {
        Console.WriteLine("Base::Foo");
    }
}

class Derived : Base
{
    public new void Foo()
    {
        Console.WriteLine("Derived::Foo");
    }
}
```

Działanie przykładowego kodu nieznacznie się zmienia:

```
// Konstruujemy kilka instancji:
Base base = new Base();
Derived derived = new Derived();
Base derivedTypedAsBase = new Derived();

// Teraz wywołujemy Foo na każdej z nich:
base.Foo();
derived.Foo();
derivedTypedAsBase.Foo();
```

Powyższy kod wypisuje na konsoli:

```
Base::Foo
Derived::Foo
Base::Foo
```

Dwa pierwsze wywołania metody działają zgodnie z oczekiwaniami. Istnieje jednak pewna subtelna różnica. W pierwotnym przykładzie metod wirtualnych i przeciążania wygenerowany kod IL zawierał instrukcje `callvirt` odnoszące się do metody `Base::Foo`. W tym przypadku zawiera pojedynczą instrukcję `callvirt` dotyczącą `Base::Foo`, po której następuje zwykła instrukcja `call` dotycząca `Derived::Foo`, zupełnie innej metody.

Dalej następuje niespodzianka: trzecie wywołanie daje wynik `"Base::Foo"`. Dzieje się tak dlatego, że wywołanie metody przez referencję typu `Base`, która wskazuje instancję `Derived`, spowoduje wyemitowanie wirtualnego wywołania metody `Base::Foo`. W czasie wykonania programu metoda wirtualna ustali, że wersja metody `Foo` z klasy `Base` nie została przesłonięta; `Derived::Foo` jest reprezentowana jako zupełnie odrębna metoda z własnym slotem w tablicy metod. Dlatego wywoływana jest wersja metody zdefiniowana w klasie `Base`.

Procedury obsługi wyjątków

Każda metoda może definiować zbiór bloków kodu do obsługi wyjątków. Jest on używany do tłumaczenia bloków `try/catch/finally` na IL, ponieważ binarna reprezentacja IL nie rozpoznaje bloków; jest to wyłącznie abstrakcja językowa. Blok obsługi wyjątku określa położenie swojej klauzuli `catch` w kodzie IL za pomocą przesunięć instrukcji, które są następnie odczytywane przez podsystem wyjątków środowiska uruchomieniowego. Szczegółowe omówienie obsługi wyjątków, w tym tłumaczenia procedur obsługi na kod IL, można znaleźć w rozdziale 3.

Konstruktory i konkretyzacja

Przed użyciem typu trzeba utworzyć jego instancję, co określa się mianem **konkretyzacji**. Instancja jest zwykle tworzona przez wywołanie **konstruktora** za pomocą instrukcji `newobj`; zadaniem konstruktora jest zainicjalizowanie obiektu przed udostępnieniem go dalszemu kodowi. Typy wartościowe obsługują niejawną inicjalizację, która polega na prostym zerowaniu ich bitów, na przykład z wykorzystaniem instrukcji IL `initobj`, na przypisaniu wartości `0` albo na wykorzystaniu domyślnego sposobu inicjalizacji zmiennych lokalnych i pól. Typ może mieć dowolną liczbę konstruktorów, czyli specjalnych metod o typie zwrotnym `void`, przeciążanych tak jak wszystkie inne metody (tzn. przez parametry).

Kiedy CLR alokuje nowy obiekt, najpierw przydziela mu miejsce na stercie GC (proces ten zostanie opisany dokładniej w rozdziale 3.). Następnie wywołuje konstruktor, przekazując mu w ramce aktywacji argumenty instrukcji `newobj`. Kod konstruktora jest odpowiedzialny za to, aby po zakończeniu jego działania typ był w pełni zainicjalizowany i nadawał się do użytku. Przejście do innych stanów może wymagać dodatkowych operacji, ale jeśli konstruktor pomyślnie kończy działanie, instancja powinna być zdatna do użytku i nieuszkodzona.

Konstruktory w reprezentacji IL noszą nazwę `.ctor`, choć na przykład C# korzysta z prostszej składni: nazwa typu jest używana tak, jakby była nazwą metody, z pominięciem specyfikacji typu zwrotnego (składnia zapożyczona z C++). Na przykład poniższy fragment kodu zawiera konstruktor, który inicjalizuje pewien stan instancji:

```
class Customer
{
    static int idCounter;

    private int myId;
    private string name;

    public Customer(string name)
    {
        if (name == null)
            throw new ArgumentNullException("name");
        this.myId = ++idCounter;
        this.name = name;
    }
    // ...
}
```

W skompilowanym kodzie IL konstruktor ma postać metody `instance void .ctor(string)`.

Klasa `Customer` oferuje tylko jeden konstruktor, który przyjmuje argument `string` reprezentujący nazwę instancji. Ustawia on pole `id`, opierając się na statycznym liczniku `idCounter`, a następnie zapisuje wartość argumentu `name` w polu instancyjnym `name`. Klasa implementuje niejawny niezmiennik: w pełni zainicjalizowani klienci mają niezerowy identyfikator (`id`) i niepustą nazwę (`name`).

Konstruktory domyślne (mechanizm językowy)

Języki C# i VB emitują **konstruktor domyślny** — nieprzyjmujący żadnych parametrów — w przypadku typów referencyjnych, które nie mają konstruktora zdefiniowanego przez programistę. Dzięki temu użytkownicy typu mogą tworzyć nowe instancje z wykorzystaniem składni `new Typ()`. Oznacza to, że poniższa deklaracja typu C# powoduje utworzenie konstruktora domyślnego w skompilowanym IL:

```
class CtorExample {}
```

Poniższy kod jest strukturalnie równoważny w IL:

```
class CtorExample
{
    public CtorExample() {}
}
```

Kiedy napiszemy jawny konstruktor, kompilator przestanie automatycznie dodawać konstruktor domyślny. Na przykład kod IL poniższego programu C# nie zawiera konstruktora domyślnego. Jedynym sposobem skonstruowania instancji jest użycie konstruktora, który przyjmuje liczbę całkowitą:

```
class CtorExample
{
    private int value;

    public CtorExample(int value)
    {
        this.value = value;
    }
}
```

Zadaniem konstruktora jest po prostu zainicjalizowanie typu i przygotowanie go do dalszego użytku. W konstruktorach należy unikać kosztownych operacji, takich jak blokujące się we-wy, i generalnie wszystkiego, co mogłoby zaskoczyć użytkowników typu. W takich sytuacjach lepiej zastosować wielostanowy obiekt, który oferuje jawną operację we-wy w celu pobrania danych z zewnątrz.

Konstruowanie wartości

Ponieważ typy wartościowe są alokowane „w miejscu", ich domyślna inicjalizacja polega na wyzerowaniu wszystkich bitów bez wywoływania konstruktora. Jest to bardzo wydajne, ponieważ nie wymaga wywoływania metody podczas konkretyzacji każdej wartości. Z tej przyczyny C# nie zezwala na definiowanie bezparametrowych konstruktorów struktur, dzięki czemu nie można ich pomylić z instrukcją `new TypWartościowy()`, która jest kompilowana do kodu IL tworzącego nowy, wyzerowany blok bitów. W C# nie można zagwarantować, że tworzenie typu wartościowego spowoduje wywołanie jednego z konstruktorów, choć na pierwszy rzut oka nie jest to oczywiste.

Łańcuchy konstruktorów (mechanizm językowy)

Typ pochodny może wywoływać konstruktor typu bazowego. W rzeczywistości jest to wymagane w języku C#, aby klasy bazowe mogły prawidłowo konstruować stan. Bez tego wykonywanie kodu na typie pochodnym mogłoby prowadzić do nieoczekiwanych wyników. To **łączenie łańcuchowe** odbywa się w C# przed wykonaniem konstruktora typu pochodnego.

Jeśli nie określimy jawnie, które z przeciążeń konstruktora typu bazowego powinno być wywołane, kompilator C# automatycznie wykorzysta jego konstruktor domyślny. Jeśli konstruktor domyślny nie istnieje, C# zgłosi błąd kompilacji. Rozważmy na przykład typ `CtorExampleDerived`, którego typem bazowym jest zdefiniowana wcześniej klasa `CtorExample`:

```
class CtorExampleDerived : CtorExample
{
    private DateTime initializedDate;
    public CtorExampleDerived(int value) : base(value * 2)
    {
        initializedDate = DateTime.Now;
    }
}
```

Zauważmy, że konstruktor ten wywołuje konstruktor klasy bazowej, wykorzystując specjalną składnię C#. Wygląda to jak wywołanie metody — gdzie `base` jest funkcją konstruktora — i rzeczywiście odpowiedni konstruktor jest wybierany w zależności od tego, ile parametrów jest przekazywanych do `base`, dokładnie tak samo jak w przypadku wybierania przeciążonej

wersji metody. Konstruktor bazowy jest wykonywany do końca, zanim pole `initializedDate` zostanie ustawione przez konstruktor klasy `CtorExampleDerived`. Gdybyśmy nie określili docelowego konstruktora, kompilator C# zgłosiłby błąd, ponieważ klasa `CtorExample` (pokazana wyżej wersja, która przyjmuje liczbę całkowitą) nie ma bezparametrowego konstruktora.

Można również łączyć przeciążone konstruktory jednego typu, podobnie jak w przypadku przeciążania metod, na przykład po to, aby zdefiniować wartości domyślne i uniknąć powielania kodu. Składnia jest bardzo podobna jak w powyższym przykładzie, z tym że zamiast `base` używa się `this`:

```
class CtorExampleDerived : CtorExample
{
    private bool wasDefaultCalled;
    private DateTime initializedDate;

    public CtorExampleDerived() : this(0)
    {
        wasDefaultCalled = true;
    }

    public CtorExampleDerived(int value) : base(value * 2)
    {
        initializedDate = DateTime.Now;
    }
}
```

Ten kod używa konstruktora domyślnego w celu określenia domyślnej wartości pola `value`. Oczywiście, oba konstruktory nie są ograniczone tylko do przekazywania wywołań innemu konstruktorowi. Konstruktor domyślny może wykonywać dodatkowe operacje, jak ustawianie pola `wasDefaultCalled` w powyższym przykładzie.

Inicjalizacja pól (mechanizm językowy)

Jeśli inicjalizujemy „w miejscu" pola instancyjne, kod tej inicjalizacji jest kopiowany przez kompilator C# do wszystkich konstruktorów, które nie odwołują się do innego konstruktora typu z wykorzystaniem słowa kluczowego `this`. Rozważmy na przykład poniższy typ:

```
class FieldInitExample
{
    int x = 5;
    int y;

    public FieldInitExample() : this(5)
    {
    }

    public FieldInitExample(int y)
    {
        this.y = y;
    }
}
```

Wiersz `int x = 5` zostanie skopiowany do pierwszych kilku instrukcji konstruktora, który przyjmuje argument w postaci liczby całkowitej. Nie zostanie skopiowany do konstruktora domyślnego, ponieważ odwołuje się on do innej przeciążonej wersji, co prowadziłoby do dwukrotnego zapisywania tych samych danych w polu `x`.

Wynikowy kod IL wygląda tak, jakbyśmy napisali:

```
class FieldInitExample
{
    int x;
    int y;

    public FieldInitExample() : this(5)
    {
    }

    public FieldInitExample(int y)
    {
        this.y = y;
        this.x = 5;
    }
}
```

Inicjalizacja pól statycznych odbywa się niemal identycznie, z tym zastrzeżeniem, że zamiast konstruktorów instancyjnych używane są statyczne konstruktory klasy. Typ `CtorExample` można by napisać w sposób pokazany niżej z zachowaniem tej samej semantyki:

```
class CtorExample
{
     static DateTime classLoadTimestamp = DateTime.Now;
}
```

Inicjalizatory typów

Istnieje też inny rodzaj konstruktora, nazywany **konstruktorem typu**. Często określa się go mianem **konstruktora statycznego** lub **konstruktora klasy**, czasem również **inicjalizatora typu**; pozwala on zainicjalizować statyczne informacje związane z klasą. W kodzie IL konstruktory te noszą nazwę `.cctor`, a w C# zapisuje się je za pomocą specjalnej składni:

```
class CctorExample
{
    static DateTime classLoadTimestamp;

    static CctorExample()
    {
        classLoadTimestamp = DateTime.Now;
    }
}
```

Zauważmy, że ten fragment kodu jest równoważny z poniższym:

```
class CctorExample
{
    static DateTime classLoadTimestamp = DateTime.Now;
}
```

Kompilator C# przekształca ten drugi w pierwszy. Ten sam typ może zawierać konstruktor jawny, a także kombinację inicjalizatorów statycznych pól. Podobnie jak w przypadku omówionej wcześniej inicjalizacji pól instancyjnych, kompilator poprzedzi treść konstruktora typu inicjalizacjami wszystkich pól.

Inicjalizatory typów napisane w C# mają semantykę `beforefieldinit`. Oznacza to, że nie wiadomo dokładnie, kiedy się uruchomią, ale zawsze zostaną wykonane do momentu, w którym nastąpi pierwszy dostęp do danego typu. W tym przypadku dostęp oznacza odczytanie lub zapisanie pola statycznego albo instancyjnego bądź też wywołanie statycznej lub instancyjnej metody typu. Dzięki temu CLR może wybrać optymalny moment uruchomienia inicjalizatora, opóźniając go lub przyspieszając na podstawie heurystyki. Natomiast zasadą domyślną jest uruchamianie inicjalizatorów dokładnie w tym momencie, w którym następuje dostęp.

Konstruktory typów są wykonywane w kontekście wątku, który spowodował ich uruchomienie. Środowisko uruchomieniowe gwarantuje, że okrężne zależności między konstruktorami statycznymi nie doprowadzą do nieskończonych pętli. Można jawnie wymusić przeprowadzenie inicjalizacji określonego typu za pomocą metody `System.Runtime.CompilerServices.RuntimeHelpers.RunClassConstructor`.

Nieobsłużone wyjątki w konstruktorach

Nieobsłużony wyjątek w konstruktorze może wystąpić wskutek zgłoszenia wyjątku przez sam kod konstruktora. Czasem jest to nieuniknione, choć zawsze niepożądane. Jeśli na przykład nie możemy w pełni zainicjalizować obiektu ze względu na niespójne lub uszkodzone informacje o stanie, nie mamy innego wyjścia, jak zgłosić wyjątek. Jedyną inną możliwością byłoby udawać, że wszystko jest w porządku, co mogłoby doprowadzić do błędów podczas dalszego wykonywania programu. Lepiej jest zatrzymać program natychmiast po wykryciu błędu.

> Wyjątki całkowicie nieobsługiwane przez CLR opisano w rozdziale 3. Istnieje też wywołanie API, dzięki któremu kod może zawieść natychmiast, na przykład w razie wykrycia poważnego uszkodzenia stanu. To również zostanie omówione w następnym rozdziale.

Wspominam o tym dlatego, że kiedy konstruktor zgłasza wyjątek, to wywołujący nie ma referencji do obiektu. Rozważmy sekwencję kodu IL odpowiadającą instrukcji C# `Foo f = new Foo()`:

```
newobj instance void Foo::.ctor()
stloc.0
```

Instrukcja `newobj` wywołuje konstruktor i pozostawia wynik na stosie wykonania. Instrukcja `stloc.0` zapisuje go w lokalnym slocie na stosie wykonania. Jeśli jednak wskutek wykonania instrukcji `newobj` zostanie zgłoszony wyjątek (ze względu na brak pamięci na alokację obiektu albo z powodu nieobsłużonego wyjątku zgłoszonego przez konstruktor), wynik nie zostanie umieszczony na stosie, a instrukcja `stloc.0` nie zostanie wykonana, ponieważ uaktywni się mechanizm obsługi wyjątków CLR. W rezultacie zmienna `f` będzie równa `null`.

Może to doprowadzić do nieoczekiwanego działania programu. Rozważmy typ, który przydziela zasoby w swoim konstruktorze i może zgłosić wyjątek. Zwykle takie typy implementują interfejs `IDisposable` — opisywany szczegółowo w rozdziale 5. — aby umożliwić deterministyczne odzyskiwanie zasobów. Kod mógłby wyglądać tak:

```
using (Foo f = new Foo())
{
    // Robimy coś interesującego...
}
```

Instrukcja `using` sprawia, że pod koniec bloku jest wywoływana metoda `Dispose` zmiennej umieszczonej w nawiasie, zarówno w przypadku zwykłego wyjścia, jak i wyjątku zgłoszonego w ciele bloku. Jeśli jednak wyjątek wystąpi podczas wywoływania konstruktora `Foo`, zmiennej `f` nie zostanie przypisana wartość! Metoda `Dispose` nie zostanie zatem wykonana i pozostanie liczyć na to, że zasoby zostaną zwolnione później przez finalizator `f`. Wyjątki, finalizację i zarządzanie zasobami omówimy szczegółowo w rozdziale 3.

Właściwości

Właściwości to w dużej mierze syntaktyczny lukier. Nie są one obsługiwane w specjalny sposób przez CTS, ale mają wsparcie na poziomie bibliotek oraz języka i stanowią bardzo wygodny sposób definiowania metod służących do pobierania i ustawiania stanu instancji. Ich szczególny status jest wskazywany przez słowo kluczowe `specialname` w kodzie IL. Czasem określa się je mianem **metod pobierających** i **ustawiających**. Zamiast bezpośrednio eksponować pola przed wywołującymi, można zahermetyzować je w niewielkich metodach; metody te mogą wykonywać na przykład opóźnioną inicjalizację albo jakieś inne wymagane operacje.

Język C# oferuje jawną obsługę właściwości. Rozważmy przykładową klasę, która eksponuje dwa pola publiczne:

```
class PropertyExample
{
    public int X;
    public int Y;
}
```

Klasa ta działa doskonale, ale narusza jedną z fundamentalnych zasad programowania obiektowego: hermetyzację. Byłoby lepiej, gdyby dostęp do pól był kontrolowany przez metody. W językach takich jak Java w tym celu zwyczajowo pisze się metody takie jak `int getX()`, `void setX(int x)` itp. W C# nie trzeba jednak polegać na konwencjach, na przykład:

```
class PropertyExample
{
    private int x;
    private int y;

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

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

Zapewnia to wywołującym możliwość odczytywania i ustawiania pól X i Y, a jednocześnie pozwala typowi PropertyExample na sterowanie dostępem do tych pól. Każdy dostęp do właściwości wymaga przejścia przez metody get i set. Jeśli przyjrzymy się wygenerowanemu kodowi IL, zauważymy metody get_X, set_X, get_Y i set_Y; szczegóły te są abstrahowane przez kompilator C#.

Nie musimy pisać zarówno metody pobierającej, jak i ustawiającej — możemy zdefiniować tylko jedną z nich. Jeśli na przykład napiszemy tylko metodę pobierającą, utworzymy właściwość przeznaczoną tylko do odczytu. Wspomnijmy jeszcze o dość oczywistej sprawie: dostępność pól, do których odwołują się właściwości, powinna być bardziej ograniczona niż dostępność samych właściwości. Jeśli zdefiniujemy właściwości, ale kod kliencki nadal będzie miał bezpośredni dostęp do pól, to zrezygnujemy z podstawowej zalety właściwości: hermetyzacji.

Właściwości indeksujące

Można utworzyć specjalną odmianę właściwości nazywaną właściwością indeksującą. Rozważmy poniższy przykład:

```
class Customer
{
    private Dictionary<int, Order33> orders = new Dictionary<int, Order33>();

    public Order this[int id]
    {
        get { return orders[id]; }
        set { orders[id] = value; }
    }
}
```

Jest to zasadniczo domyślna właściwość typu, co wskazuje atrybut System.Reflection.DefaultMemberAttribute w kodzie IL. Ten rodzaj właściwości pozwala wywołującym na dostęp do obiektu z wykorzystaniem specjalnej składni. Na przykład w C# wywołujący mogą stosować składnię tablicową:

```
Customer c = /* ... */;
Order o = c[10010];
```

Właściwości indeksujące wyróżniają się również tym, że mogą być przeciążane przez parametry. Mogą też przyjmować wiele parametrów wejściowych.

Dostępność w trybie mieszanym (mechanizm językowy)

Właściwości składają się z dwóch bloków kodu: metody pobierającej i ustawiającej. Od wersji 2.0 C# bloki te mogą mieć odmienne poziomy dostępności. Oczywiście, nie jest to zmiana w CTS — poprzednio można było zapisać to w kodzie IL — ale stanowi ulepszenie samego języka C#. Często na przykład pisze się publiczną metodę pobierającą i chronioną ustawiającą. Pozwala na to następująca składnia:

```
class PropertyExample
{
    private int x;
    private int y;
```

```
    public int X
    {
        get { return x; }
        protected set { x = value; }
    }

    public int Y
    {
        get { return y; }
        protected set { y = value; }
    }
}
```

Zauważmy, że deklaracja właściwości nadal ma dostępność domyślną, podczas gdy metoda ustawiająca ma bardziej ograniczoną dostępność zdefiniowaną dokładnie w tym miejscu, w którym piszemy `set`.

Zdarzenia

Typy CLR mogą zawierać **zdarzenia**. Podobnie jak właściwości, zdarzenia są raczej pewną konwencją i wzorcem niż cechą samego CTS. W związku z tym one również są oznaczone tokenem `specialname`. Kod kliencki może subskrybować zdarzenia, aby otrzymywać powiadomienia o interesujących zmianach stanu typu. Na przykład w Windows Forms zdarzenia pozwalają programom reagować na działania użytkownika, takie jak kliknięcie przycisku, zamknięcie okna, wpisanie znaku w polu tekstowym itd. Oczywiście, można ich również używać do komunikacji między obiektami.

W C# definiowanie zdarzeń jest bardzo proste. Słowo kluczowe `event` deklaruje składową klasy jako zdarzenie, na przykład:

```
using System;

class Foo : IDisposable
{
    public event EventHandler OnInitialized;
    public event EventHandler OnDisposed;

    public void Init()
    {
        // Inicjalizujemy stan...
        EventHandler onInit = OnInitialized;
        if (onInit != null)
            onInit(this, new EventArgs());
    }

    public void Dispose()
    {
        // Zwalniamy stan...
        EventHandler onDisp = OnDisposed;
        if (onDisp != null)
            onDisp(this, new EventArgs());
    }
}
```

Typ Foo eksponuje dwa zdarzenia, OnInitialized oraz OnDisposed, które mogą być subskrybowane przez kod kliencki. Każde z nich jest typu EventHandler; jest to typ delegacyjny z przestrzeni nazw System. Delegacje są kluczowym elementem CTS i zostaną opisane w dalszej części rozdziału. W tym przykładzie zaakceptowaliśmy domyślną implementację subskrypcji i usuwania zdarzeń, choć C# oferuje również bardziej zaawansowaną składnię, która pozwala na pisanie własnej implementacji. Wystąpienie zdarzeń sygnalizujemy przez wywołanie odpowiednich delegacji. Jeśli nikt nie subskrybuje zdarzeń, delegacja ma wartość null, więc sprawdzamy to przed próbą wywołania (aby uniknąć wyjątku NullReferenceException).

Kod kliencki subskrybuje zdarzenia w następujący sposób:

```
using (Foo f = new Foo35())
{
    f.OnInitialized += delegate { Console.WriteLine("inicjalizacja"); };
    f.OnDisposed += delegate { Console.WriteLine("usuwanie"); };
    // ...
    f.Init();
    // ...
}
```

Zdarzenia są tu subskrybowane z wykorzystaniem składni C#. Zauważmy, że kodem reagującym na zdarzenie jest delegacja anonimowa. Mechanizm ten pozwala na rozszerzanie interfejsów API przez kod użytkownika.

Reprezentacja zdarzeń w kodzie IL nie jest tak prosta, jak sugerowałby kod C#. Są one kompilowane do metod add_OnXxx oraz remove_OnXxx, z których każda przyjmuje instancję EventHandler. Lista subskrypcji zdarzeń jest przechowywana w obiekcie Delegate, który oferuje metody takie jak Combine i Remove pozwalające na manipulowanie zdarzeniami.

Przeciążanie operatorów

Gdyby przyszło wskazać mechanizm, który w ostatnich latach był przyczyną największej liczby sporów i nieporozumień wśród projektantów oprogramowania, byłoby to z pewnością **przeciążanie operatorów**. Każdy doskonale zna operatory oferowane przez jego ulubiony język. Operatory unarne, binarne i przekształcające traktujemy jako coś oczywistego. Zawsze zakładamy na przykład, że 1 + 1 równa się 2. A jeśli użytkownik mógłby przedefiniować te operatory na użytek swoich własnych typów?

Właśnie do tego służy przeciążanie operatorów. Doskonałym przykładem dobrego wykorzystania tego mechanizmu jest klasa System.Decimal. Większość użytkowników po prostu nie wie, że wartość dziesiętna nie ma w CTS tak wysokiego statusu jak na przykład Int32. Różnice są zamaskowane przez odpowiednie zastosowanie przeciążonych operatorów.

Autorzy typów danych mogą przedefiniowywać wiele różnych operatorów, zasadniczo stanowiących zbiór „zaczepów” do rozszerzania języka. Przeciążanie operatora na użytek jakiegoś typu polega na zdefiniowaniu statycznej metody o specjalnej nazwie i sygnaturze. Rozważmy poniższy typ C#, który przeciąża operator dodawania:

```
class StatefulInt
{
    private int value;
    private State state;

    public StatefulInt(int value, State state)
    {
        this.value = value;
        this.state = state;
    }

    public static StatefulInt operator+(StatefulInt i1, StatefulInt i2)
    {
        return new StatefulInt(i1.value + i2.value,
            i1.state.Combine(i2.state));
    }

    public static StatefulInt operator+(StatefulInt i1, int i2)
    {
        return new StatefulInt(i1.value + i2, i1.state);
    }
}
```

W rezultacie użytkownicy mogą dodawać dwie wartości `StatefulInt` albo wartość `int` do `StatefulInt`:

```
StatefulInt i1 = new StatefulInt(10, new State());
StatefulInt i2 = new StatefulInt(30, new State());
StatefulInt i3 = i1 + i2; // w wynikowym obiekcie value == 40
StatefulInt i4 = i2 + 50; // w wynikowym obiekcie value == 80
```

W tabeli na następnej stronie opisano specjalne wewnętrzne nazwy używane w IL oraz odpowiednie operatory w C#, VB oraz C++/CLI.

W zależności od typu operatora — unarny, binarny lub konwersji — sygnatura musi mieć odpowiednią liczbę argumentów. Typ zwrotny zależy od samego operatora. Więcej informacji o przeciążaniu konkretnych operatorów można znaleźć w dokumentacji SDK.

Koercja

Koercja to proces kopiowania wartości skalarnej miedzy instancjami dwóch różnych typów. Zachodzi ona, kiedy cel przypisania jest innego typu niż wartość przypisywana (znajdująca się po prawej stronie przypisania). Rozważmy na przykład przypisywanie 32-bitowej wartości zmiennopozycyjnej do 64-bitowej zmiennej podwójnej precyzji. Mówimy o koercji poszerzającej, kiedy cel ma większą pojemność pamięciową, jak w przypadku opisanej wyżej konwersji z 32 na 64 bity, a o koercji zawężającej, kiedy cel ma mniejszą pojemność. Koercje poszerzające są obsługiwane niejawnie przez środowisko uruchomieniowe, podczas gdy zawężające mogą spowodować utratę informacji, a zatem muszą być jawnie określone.

<table>
<tr><th>Specjalna nazwa</th><th>Typ</th><th>C#</th><th>VB</th><th>C++/CLI</th></tr>
<tr><td>op_Decrement</td><td>Unarny</td><td>--</td><td>Brak</td><td>--</td></tr>
<tr><td>op_Increment</td><td>Unarny</td><td>++</td><td>Brak</td><td>++</td></tr>
<tr><td>op_UnaryNegation</td><td>Unarny</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>op_UnaryPlus</td><td>Unarny</td><td>+</td><td>+</td><td>+</td></tr>
<tr><td>op_LogicalNot</td><td>Unarny</td><td>!</td><td>Not</td><td>!</td></tr>
<tr><td>op_AddressOf</td><td>Unarny</td><td>&</td><td>AddressOf</td><td>&</td></tr>
<tr><td>op_OnesComplement</td><td>Unarny</td><td>~</td><td>Not</td><td>~</td></tr>
<tr><td>op_PointerDereference</td><td>Unarny</td><td>* (niebezpieczna)</td><td>Brak</td><td>*</td></tr>
<tr><td>op_Addition</td><td>Binarny</td><td>+</td><td>+</td><td>+</td></tr>
<tr><td>op_Subtraction</td><td>Binarny</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>op_Multiply</td><td>Binarny</td><td>*</td><td>*</td><td>*</td></tr>
<tr><td>op_Division</td><td>Binarny</td><td>/</td><td>/</td><td>/</td></tr>
<tr><td>op_Modulus</td><td>Binarny</td><td>%</td><td>Mod</td><td>%</td></tr>
<tr><td>op_ExclusiveOr</td><td>Binarny</td><td>^</td><td>Xor</td><td>^</td></tr>
<tr><td>op_BitwiseAnd</td><td>Binarny</td><td>&</td><td>And</td><td>&</td></tr>
<tr><td>op_BitwiseOr</td><td>Binarny</td><td>|</td><td>Or</td><td>|</td></tr>
<tr><td>op_LogicalAnd</td><td>Binarny</td><td>&&</td><td>And</td><td>&&</td></tr>
<tr><td>op_LogicalOr</td><td>Binarny</td><td>||</td><td>Or</td><td>||</td></tr>
<tr><td>op_Assign</td><td>Binarny</td><td>=</td><td>=</td><td>=</td></tr>
<tr><td>op_LeftShift</td><td>Binarny</td><td><<</td><td><<</td><td><<</td></tr>
<tr><td>op_RightShift</td><td>Binarny</td><td>>></td><td>>></td><td>>></td></tr>
<tr><td>op_Equality</td><td>Binarny</td><td>==</td><td>=</td><td>==</td></tr>
<tr><td>op_Inequality</td><td>Binarny</td><td>!=</td><td><></td><td>!=</td></tr>
<tr><td>op_GreaterThan</td><td>Binarny</td><td>></td><td>></td><td>></td></tr>
<tr><td>op_GreaterThanOrEqual</td><td>Binarny</td><td>>=</td><td>>=</td><td>>=</td></tr>
<tr><td>op_LessThan</td><td>Binarny</td><td><</td><td><</td><td><</td></tr>
<tr><td>op_LessThanOrEqual</td><td>Binarny</td><td><=</td><td><=</td><td><=</td></tr>
<tr><td>op_MemberSelection</td><td>Binarny</td><td>.</td><td>.</td><td>. lub -></td></tr>
<tr><td>op_PointerToMemberSelection</td><td>Binarny</td><td>Brak</td><td>Brak</td><td>.* lub -></td></tr>
<tr><td>op_MultiplicationAssignment</td><td>Binarny</td><td>*=</td><td>*=</td><td>*=</td></tr>
<tr><td>op_SubtractionAssignment</td><td>Binarny</td><td>-=</td><td>-=</td><td>-=</td></tr>
<tr><td>op_ExclusiveOrAssignment</td><td>Binarny</td><td>^=</td><td>Brak</td><td>^=</td></tr>
<tr><td>op_LeftShiftAssignment</td><td>Binarny</td><td><<=</td><td><<=</td><td><<=</td></tr>
</table>

Specjalna nazwa	Typ	C#	VB	C++/CLI
op_RightShiftAssignment	Binarny	>>=	>>=	>>=
op_ModulusAssignment	Binarny	%=	Brak	%=
op_AdditionAssignment	Binarny	+=	+=	+=
op_BitwiseAndAssignment	Binarny	&=	Brak	&=
op_BitwiseOrAssignment	Binarny	|=	Brak	|=
op_Comma	Binarny	,	Brak	,
op_DivisionAssignment	Binarny	/=	/=	/=
op_Implicit	Konwersja	Brak (niejawna)	Brak (niejawna)	Brak (niejawna)
op_Explicit	Konwersja	CInt, CDbl, ..., CTyp	(typ)	(typ)

Podklasy i polimorfizm

Typ może wywodzić się z innego typu, co tworzy specjalną relację między nimi. Mówimy, że B jest **podklasą** A, jeśli B wywodzi się z A. W terminologii IL B **rozszerza** A. W tym przykładzie A jest **typem bazowym** B (zwanym też typem nadrzędnym lub macierzystym). W CTS typy mogą mieć tylko jeden typ bezpośrednio nadrzędny; innymi słowy, **dziedziczenie wielokrotne** nie jest obsługiwane. Interfejsy, które zostaną omówione w dalszej części rozdziału, obsługują wielokrotne **dziedziczenie interfejsów**, ale różni się to nieco od tego, co zwykło się uważać za dziedziczenie wielokrotne. Ponieważ struktury definiowane przez użytkownika nie mogą się wywodzić z dowolnego typu (niejawnie wywodzą się z System.ValueType i są domyślnie zapieczętowane), dalej będę pisał po prostu o podklasach, rezygnując z bardziej precyzyjnej terminologii.

Relację podtypu można wyrazić w C# w następujący sposób:

```
class A {}
class B : A {}
class C : A {}
```

W tym przykładzie zarówno B, jak i C są podklasami A. Oczywiście, możemy utworzyć kolejne podklasy A, a nawet podklasy B i C. W IL wygląda to następująco:

```
.class private auto ansi beforefieldinit A extends [mscorlib]System.Object {}
.class private auto ansi beforefieldinit B extends A {}
.class private auto ansi beforefieldinit C extends A {}
```

Kiedy typ B jest podklasą A, mówimy, że B jest **polimorficzny** względem A. Oznacza to, że ponieważ podklasa A dziedziczy wszystkie widoczne publicznie cechy swojego typu bazowego, instancję B można traktować dokładnie tak samo jak instancję A bez naruszania bezpieczeństwa typologicznego. Twierdzenie odwrotne oczywiście nie jest prawdziwe. Niebawem omówimy dziedziczenie bardziej szczegółowo.

Style dziedziczenia

W CTS dostępne są dwa podstawowe style **dziedziczenia**: dziedziczenie **implementacji** i **interfejsu**. Zastosowanie jednego z nich nie wpływa na możliwość manipulowania typami w sposób bezpieczny z punktu widzenia polimorfizmu. Podstawową różnicą jest sposób, w jaki podklasy otrzymują kopie składowych typu bazowego. Poniżej wyjaśnię, co to oznacza. Obsługiwane jest też **dziedziczenie prywatnych interfejsów**, ale nie **dziedziczenie prywatnych implementacji**.

Dziedziczenie implementacji

Jest to domyślny styl dziedziczenia w CTS używany przy tworzeniu podklas innego typu. Dziedziczenie implementacji oznacza, że podklasa otrzymuje kopie wszystkich nieprywatnych składowych typu bazowego, w tym implementacji metod. Zatem zgodnie z oczekiwaniami metody zdefiniowane w typie bazowym można natychmiast wywoływać na podklasie. Jeśli metody są **wirtualne**, podklasa może je **przesłonić**, aby dostarczyć własnej implementacji, zamiast zachowywać istniejące wersje zdefiniowane w typie bazowym. Metody wirtualne i przesłanianie zostaną omówione wkrótce.

Przykładem tworzenia podklas są poniższe typy A i B:

```
class A
{
    public void Foo()
    {
        Console.WriteLine("A::Foo");
    }
}

class B : A
{
    public void Bar()
    {
        Console.WriteLine("B::Bar");
    }
}
```

Klasa A wywodzi się z B — o czym świadczy składnia <Podklasa> : <TypBazowy> — dziedzicząc publiczną metodę Foo i rozszerzając A o nową metodę Bar. Rezultat jest taki, że A ma pojedynczą metodę Foo, a B dwie metody Foo i Bar. Wywołanie metody Foo na instancji B „po prostu działa” i daje ten sam wynik co wywołanie Foo na instancji A (wypisuje "A::Foo") — twórca klasy B nie musi robić nic specjalnego, aby uzyskać taki efekt. Wynika to z tego, że podklasa dziedziczy zarówno interfejs, jak i implementację metody Foo zdefiniowanej w typie bazowym.

Dziedziczenie interfejsu

Podklasa dziedzicząca interfejs otrzymuje w swojej przestrzeni publicznej jedynie sygnatury API nieprywatnych składowych typu bazowego. Oznacza to, że odziedziczone metody nie mają implementacji, tylko sygnatury. Podklasa zwykle musi „ręcznie” zaimplementować te składowe. CTS obsługuje dziedziczenie interfejsu w dwóch przypadkach: **klas abstrakcyjnych** i **interfejsów**. Są one omówione poniżej.

Klasy abstrakcyjne

Klasa abstrakcyjna to klasa, której nie można konkretyzować. Stanowi ona klasę bazową, z której wywodzą się inne typy. Jeśli klasa nie jest abstrakcyjna, to nazywamy ją **konkretną** (choć jest to cecha domyślna, więc zwykle się o niej nie wspomina). Poszczególne metody klasy abstrakcyjnej mogą również być oznaczone jako abstrakcyjne, co znaczy, że nie dostarczają implementacji. Właściwości nie mogą być oznaczone jako abstrakcyjne. Metody abstrakcyjne są znane większości programistów C++ pod nazwą metod **czysto wirtualnych**. Klasa abstrakcyjna nie musi zawierać metod abstrakcyjnych, ale typ z metodami abstrakcyjnymi musi być abstrakcyjny.

Na przykład każda z czterech poniższych klas może być oznaczona jako abstrakcyjna. Tylko przykłady 2. i 4. **muszą** być oznaczone jako abstrakcyjne, ponieważ zawierają abstrakcyjne składowe:

```
// Abstrakcyjna, nie zawiera składowych
abstract class AbstractType1 {}

// Abstrakcyjna, zawiera tylko abstrakcyjne składowe
abstract class AbstractType2
{
    public abstract void Foo();
    public abstract void Bar();
}

// Abstrakcyjna, zawiera tylko nieabstrakcyjne składowe
abstract class AbstractType3
{
    public void Foo()
    {
        Console.WriteLine("AbstractType3::Foo");
    }
    public void Bar()
    {
        Console.WriteLine("AbstractType3::Bar");
    }
}

// Abstrakcyjna, zawiera mieszankę składowych abstrakcyjnych i nieabstrakcyjnych
abstract class AbstractType4
{
    public void Foo()
    {
        Console.WriteLine("AbstractType4::Foo");
    }
    public abstract void Bar();
}
```

Kiedy typ wywodzi się z klasy abstrakcyjnej, to dziedziczy wszystkie składowe typu bazowego, tak jak w przypadku zwykłych klas. Jeśli jakaś metoda dostarcza implementacji, to owa implementacja również jest dziedziczona. Jednak w przypadku metod oznaczonych jako abstrakcyjne podklasa musi albo dostarczyć własnej implementacji każdej z nich, albo zadeklarować, że sama jest klasą abstrakcyjną.

Rozważmy na przykład klasę wywodzącą się z pokazanego wyżej typu `AbstractType4`:

```
class ConcreteType : AbstractType4
{
    public override void Bar()
    {
        // Musimy tu dostarczyć implementacji, a w przeciwnym razie
        // oznaczyć również klasę ConcreteType jako abstrakcyjną.
        Console.WriteLine("ConcreteType::Bar");
    }
}
```

Typy abstrakcyjne są oznaczone tokenem metadanych `abstract` w wygenerowanym kodzie IL, a metody abstrakcyjne są oznaczone niejawnie jako `abstract` i `virtual`. Zatem klasa pochodna działa tak, jakby przesłaniała każdą zwykłą metodę wirtualną w celu dostarczenia implementacji. Podobnie jak w przypadku każdej innej metody wirtualnej, przesłonięcie nie może zmieniać widoczności metody.

Interfejsy

Interfejs to specjalny typ, który nie zawiera implementacji metod, ale może być używany przez inne typy w celu zadeklarowania obsługi jakiegoś zbioru publicznych wywołań API. Na przykład poniższy interfejs definiuje metodę i trzy właściwości:

```
interface ICollection : IEnumerable
{
    void CopyTo(Array array, int index);
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
}
```

> Interfejsom zwykle nadaje się nazwy zaczynające się od wielkiej litery `I`; zwyczaj ten wywodzi się z ery COM. Wszystkie interfejsy COM — również według konwencji — miały nazwy zaczynające się na `I`.

Podobnie jak w przypadku abstrakcyjnych metod abstrakcyjnej klasy, nie określa się implementacji składowych. W przeciwieństwie do klasy abstrakcyjnej interfejs **nie może** zawierać żadnej implementacji. Zauważmy też, że interfejs może wywodzić się z innego interfejsu. W takim przypadku dziedziczy wszystkie składowe interfejsu bazowego, co oznacza, że implementator musi podać konkretne wersje składowych zarówno interfejsu bazowego, jak i pochodnego.

Kiedy typ **implementuje** interfejs, to musi zapewnić obsługę całego interfejsu. Można jednak użyć klasy abstrakcyjnej i uniknąć implementowania niektórych składowych, oznaczając je jako abstrakcyjne. Następnie można odwoływać się do tego typu i uzyskiwać do niego dostęp za pośrednictwem zmiennych typizowanych jako dany interfejs. Jest to czyste dziedziczenie interfejsu, ponieważ nie są w to zaangażowane żadne implementacje. Rozważmy poniższy prosty interfejs i jego przykładową implementację:

```
interface IComparable
{
    int CompareTo(object obj);
}

struct Int32 : IComparable
{
    private int value;
    public int CompareTo(object obj)
    {
        if (!(obj is Int32))
            throw new ArgumentException();
        int num = ((Int32)obj).value;
        if (this.value < num)
            return -1;
        else if (this.value > num)
            return 1;
        return 0;
    }
    // ...
}
```

Przy takiej definicji instancji `Int32` można używać wszędzie tam, gdzie oczekiwany jest interfejs `IComparable`, na przykład jako argument metody, lokalną zmienną lub pole itd. To samo dotyczy typów bazowych, z których wywodzi się interfejs. Na przykład interfejs `ICollection` implementuje `IEnumerable`; zatem każdy typ, który implementuje `ICollection`, może być traktowany jako `ICollection` albo `IEnumerable`. Ponieważ `Int32` jest strukturą, musi zostać opakowany, zanim będzie można go przekazać jako `IComparable`.

Warto wspomnieć o kilku innych interesujących rzeczach:

To, że typ implementuje interfejs, jest częścią jego publicznego kontraktu. Oznacza to, że implementowane metody interfejsu również muszą być publiczne. Jedynym wyjątkiem jest prywatna implementacja interfejsu (opisywana poniżej), w której metoda jest nadal dostępna przez wywołania interfejsu, ale w rzeczywistości pozostaje prywatna.

Programista może oznaczyć implementację metody jako wirtualną lub jako finalną (to drugie ustawienie jest domyślne). Zauważmy, że ze względu na **reimplementację interfejsów** (omówioną niżej) nie można zapobiec tworzeniu nowych slotów metod tego samego interfejsu przez dalsze podklasy.

Ekspedycja metod interfejsu

Wywołanie interfejsu z grubsza przypomina wywołanie metody wirtualnej. Choć IL wygląda tak samo — tzn. wywołanie interfejsu jest emitowane jako instrukcja `callvirt` — w rzeczywistości ekspedycja metody wymaga dodatkowej warstwy pośredniości. Jeśli przyjrzymy się kodowi maszynowemu tworzonemu przez JIT, zobaczymy przeszukiwanie mapy interfejsu (dołączonej do tablicy metod) przeprowadzane w celu skorelowania implementacji interfejsu z odpowiednim slotem w tablicy metod. W większości przypadków te dodatkowe koszty nie stanowią problemu.

Wspomniano już — a dotyczy to również niniejszej dyskusji — że wywoływanie wirtualnej metody na typie wartościowym wymaga uprzedniego opakowania wartości. Wywołania ograniczone (nowa funkcja w wersji 2.0) w niektórych sytuacjach pozwalają środowisku uruchomieniowemu zoptymalizować ten proces. Funkcja ta zostanie opisana dokładniej w rozdziale 3.

Dziedziczenie wielokrotne

CTS nie pozwala, aby typ dziedziczył po więcej niż jednej klasie bazowej. Jest to decyzja podjęta na wczesnym etapie projektowania CLR i .NET Framework, odbiegająca od praktyk stosowanych w niektórych językach programowania, szczególnie w C++. Dziedziczenie wielokrotne jest powszechnie uważane za złą praktykę programowania obiektowego, choć możliwość traktowania instancji jednego typu jako kilku innych polimorficznie zgodnych typów ma niepodważalne zalety.

Wielokrotne dziedziczenie interfejsów jest kompromisem. Pozwala na dziedziczenie różnych interfejsów bez problemów związanych z wielokrotnym dziedziczeniem implementacji. Na przykład poniższy kod deklaruje, że typ `Implementer` implementuje zarówno interfejs `IFoo`, jak i `IBar`:

```
interface IFoo
{
    void Foo();
}

interface IBar
{
    void Bar();
}

class Implementer : IFoo, IBar
{
    public void Foo()
    {
        Console.WriteLine("Implementer::Foo");
    }

    public void Bar()
    {
        Console.WriteLine("Implementer::Bar");
    }
}
```

Dzięki temu instancję `Implementer` można traktować jak `IFoo` lub jak `IBar`.

Prywatne dziedziczenie interfejsów

Niektóre języki zezwalają na dziedziczenie prywatne. W C# jeden typ może dziedziczyć po drugim, nie będąc z nim zgodnym polimorficznie. Jest to po prostu wygodny sposób ponownego wykorzystania implementacji. W CTS nie można prywatnie dziedziczyć implementacji, ale można stosować prywatne dziedziczenie interfejsów.

Prywatne dziedziczenie interfejsu to po prostu sposób ukrywania metod w publicznym interfejsie typu. Metody są kompilowane jako prywatne, ale w rzeczywistości pozostają dostępne poprzez mapę interfejsu. Innymi słowy, mogą być wywoływane tylko za pośrednictwem referencji typizowanej jako interfejs, w którym zdefiniowana jest metoda. Najłatwiej wyjaśnić to na przykładzie:

```
class PrivateImplementer : IFoo
{
    void IFoo.Foo()
    {
        Console.WriteLine("PrivateImplementer::IFoo.Foo");
    }
}
```

W tym przypadku `PrivateImplementer` jest publicznie znany jako typ implementujący `IFoo`. Jego instancję można więc traktować polimorficznie jak instancję `IFoo`, ale nie można wywoływać na nim metody `Foo`, jeśli nie jest rzeczywiście traktowany jak `IFoo`. Demonstruje to poniższy kod:

```
PrivateImplementer p = new PrivateImplementer();
p.Foo(); // Ten wiersz się nie skompiluje
IFoo f = p;
f.Foo();
```

Implementacja prywatna może dotyczyć tylko niektórych metod interfejsu. Gdyby `PrivateImplementer` implementował interfejs `IFooBar`, mógłby zaimplementować metodę `Foo` prywatnie, a `Bar` publicznie, z wykorzystaniem zwykłej składni.

W praktyce implementacji prywatnej nie używa się zbyt często. Biblioteka `System.Collections.Generic` wykorzystuje to podejście, aby potajemnie zaimplementować wszystkie tradycyjne, słabo typizowane interfejsy `System.Collections`. Dzięki temu zgodność wstecz po prostu działa, na przykład można przekazać instancję `List<T>` do metody, która oczekuje `IList`. W tym konkretnym przykładzie zaśmiecanie nowych, ściśle typizowanych interfejsów API byłoby niefortunne (współpraca ze słabo typizowanym kodem wymaga znacznej liczby metod).

Dostępność w dziedziczeniu prywatnym

W dziedziczeniu prywatnym występuje pewien ukryty problem: implementowane metody są generowane jako prywatne. Można uzyskać do nich dostęp przez mapę interfejsu, ale są one niedostępne dla podklas. Na przykład popularną praktyką jest reimplementacja interfejsu i połączenie jej z implementacją bazową. Jednak w przypadku implementacji prywatnych jest to niemożliwe.

Przypuśćmy, że chcemy utworzyć typ `PrivateExtender`, który przedefiniowuje metodę `Foo`, ale nadal wywołuje jej wersję zdefiniowaną w typie bazowym:

```
class PrivateExtender : PrivateImplementer, IFoo
{
    void IFoo.Foo()
    {
        base.Foo(); // Ten wiersz się nie skompiluje
        Console.WriteLine("PrivateExtender::IFoo.Foo");
    }
}
```

Zwykle osiągnęlibyśmy to za pomocą słowa kluczowego `base` języka C#, ale w tym przypadku kompilator zgłosi błąd, ponieważ `Foo` jest metodą prywatną typu `PrivateImplementer`. Wydawałoby się, że w takiej sytuacji można posłużyć się składnią `((IFoo)base).Foo()`, ale takie wywołanie również się nie skompiluje (nie ma ono zresztą żadnej reprezentacji w IL). Moglibyśmy napisać `((IFoo)this).Foo()`, ale wówczas wpadlibyśmy w nieskończoną pętlę (ponieważ powoduje to wirtualne wywołanie kopii należącej do `PrivateExtender` — tej samej metody, która wykonuje wywołanie). Po prostu nie da się napisać kodu, który by to robił!

Reimplementacja interfejsu

Ogólnie rzecz biorąc, można zapobiec przedefiniowywaniu metody przez podklasy, oznaczając ją jako `final` (albo po prostu nie oznaczając jej jako `virtual`). Podklasa zawsze może oznaczyć swoją metodę jako `newslot`, aby dostarczyć nową metodę, która odpowiada sygnaturze istniejącej. Kiedy jednak następuje wirtualne wywołanie wersji należącej do typu bazowego, nowa definicja nie zostanie wykonana. Dzieje się tak dlatego, że tablica wirtualna odróżnia obie metody — innymi słowy, tworzy nowy slot dla każdej z nich.

Jednakże w przypadku interfejsów istnieje tylko jedna mapa na każdy interfejs danego typu. Oznacza to, że wywołanie poprzez mapę interfejsu jest zawsze kierowane do najbardziej pochodnej wersji, bez względu na to, czy podklasy zdefiniowały swoje implementacje jako finalne, czy też jako wirtualne. Może to być zaskakujące.

Rozważmy na przykład poniższy typ bazowy i jego podklasę:

```
class FooBase : IFoo
{
    public void Foo()
    {
        Console.WriteLine("FooBase::Foo");
    }
    public void Bar()
    {
        Console.WriteLine("FooBase::Bar");
    }
}

class FooDerived : FooBase, IFoo
{
    public new void Foo()
    {
        Console.WriteLine("FooDerived::Foo");
    }
    public new void Bar()
    {
        Console.WriteLine("FooDerived::Bar");
    }
}
```

Jeśli napiszemy kod, który wywołuje na różne sposoby metody `Foo` i `Bar`, niektóre wyniki mogą nas zaskoczyć. Gdy spojrzymy na nie w całości, będą miały sens. Mimo to wiele osób dziwi się, że typ może całkowicie przedefiniować implementację interfejsu określoną w typie bazowym, choć pierwotna implementacja jest chroniona.

```
FooDerived d = new FooDerived();
FooBase b = d;
IFoo i = d;

b.Foo(); // Wyświetla "FooBase::Foo"
b.Bar(); // Wyświetla "FooBase::Bar"

d.Foo(); // Wyświetla "FooDerived::Foo"
d.Bar(); // Wyświetla "FooDerived::Bar"

i.Foo(); // Wyświetla "FooDerived::Foo"
```

Wybór między klasą abstrakcyjną a interfejsem

Klasy abstrakcyjne i interfejsy pełnią podobne funkcje, ale mają różne wady i zalety. Klasy abstrakcyjne mogą oferować nie tylko interfejs, lecz również implementację, co znacznie upraszcza kontrolę wersji. Należy z nich więc korzystać w większości sytuacji, choć w niektórych przypadkach użycie interfejsów również będzie miało sens.

Jako przykład problemów z kontrolą wersji rozważmy następującą sytuację: opublikowaliśmy klasę abstrakcyjną i interfejs z dwiema metodami, `void A()` i `void B()`. Zasadniczo jesteśmy na nie skazani, tzn. nie możemy się ich pozbyć, nie naruszając typów, które wywodzą się z naszej klasy albo implementują nasz interfejs. Klasy abstrakcyjne można jednak z czasem rozszerzać. Gdybyśmy chcieli na przykład dodać nową metodę `void C()`, moglibyśmy zdefiniować ją w klasie abstrakcyjnej z pewną domyślną implementacją. W podobny sposób moglibyśmy dodać pomocnicze, przeciążone wersje metody. W przypadku interfejsów jest to po prostu niemożliwe.

Z drugiej strony klasy abstrakcyjne przejmują hierarchię typów klas pochodnych. Klasa może implementować interfejs, a mimo to nadal zachować jakąś sensowną hierarchię typów. W przypadku klas abstrakcyjnych jest inaczej. Co więcej, interfejsy pozwalają stosować dziedziczenie wielokrotne, a klasy abstrakcyjne — nie.

Pieczętowanie typów i metod

Typ może być oznaczony jako zapieczętowany, co znaczy, że nie można wywodzić z niego dalszych typów. W C# sygnalizuje się to za pomocą słowa kluczowego `sealed`:

```
sealed class Foo {}
```

Nie można utworzyć podklasy typu `Foo`. Wszystkie niestandardowe typy wartościowe są niejawnie zapieczętowane wskutek zasady, że nie można dziedziczyć po typach wartościowych zdefiniowanych przez użytkownika.

Zauważmy też, że typ zarówno zapieczętowany, jak i abstrakcyjny z definicji nie może mieć konkretnej instancji. Uważa się go zatem za **typ statyczny**, czyli taki, który powinien mieć wyłącznie składowe statyczne. Słowo kluczowe języka C# dla klas statycznych używa właśnie takiego wzorca w IL, tzn. poniższe dwa wiersze kodu są semantycznie równoważne:

```
static class Foo { /* ... */ }
abstract sealed class Foo { /* ...*/ }
```

Kompilator C# zapobiega przypadkowemu dodaniu składowych instancyjnych do klasy statycznej, tym samym unikając tworzenia całkowicie niedostępnych składowych.

Poszczególne metody wirtualne również mogą być zapieczętowane, co wskazuje, że ich dalsze przesłanianie jest nielegalne. Dalsze typy pochodne mogą jednak nadal ukryć metodę przez tworzenie nowych slotów, na przykład:

```
class Base
{
    protected virtual void Bar() { /*...*/ }
}

class Derived : Base
{
    protected override sealed void Bar() { /*...*/ }
}
```

W języku C# do pieczętowania metod używa się słowa kluczowego `sealed`. W powyższym przykładzie oznacza to, że podklasy `Base` nie mogą przesłaniać metody `Bar`.

Sprawdzanie typów podczas wykonywania programu

CLR oferuje różne sposoby przeprowadzania dynamicznej kontroli typów w celu zweryfikowania polimorficznej zgodności między instancją a typem. Dysponując obiektem `o` i typem `T`, możemy użyć instrukcji IL `castclass` oraz `isinst`, aby sprawdzić, czy `o` jest typu `T` albo czy jego typ implementuje `T` (jeśli `T` jest interfejsem) lub jest podtypem `T`, co oznacza, że można bezpiecznie traktować go jak `T`. W języku C# instrukcje te są eksponowane jako rzutowanie oraz słowa kluczowe `is` i `as`:

```
object o = /*...*/;
string s1 = (string)o; // Rzutowanie używa instrukcji castclass
string s2 = o as string; // Słowo kluczowe as używa instrukcji isinst
if (o is string) { // Słowo kluczowe is używa instrukcji isinst
    // ...
}
```

W tym przykładzie rzutowanie używa instrukcji `castclass`, aby dynamicznie sprawdzić, czy instancja `o` jest typu `System.String`. Jeśli nie jest, środowisko uruchomieniowe zgłasza wyjątek `CastClassException`. Słowa kluczowe `as` i `is` używają instrukcji `isinst`. Instrukcja ta bardzo przypomina `castclass`, ale nie generuje wyjątku; jeśli typ nie przejdzie pomyślnie testu, instrukcja zwraca 0. W przypadku słowa kluczowego `as` oznacza to, że jeśli instancja nie jest prawidłowego typu, wynik będzie równy `null`; w przypadku słowa kluczowego `is` ten sam warunek daje wynik `false`.

Przestrzenie nazw: organizowanie typów

Wszystkie dobre środowiska programistyczne mają system tworzenia modułów i pakietów. Pominąwszy podzespoły i moduły, które stanowią fizyczne jednostki dystrybucji i wielokrotnego użytku — podzespoły zostaną opisane szczegółowo w rozdziale 4. — mechanizmem pakowania logicznego są **przestrzenie nazw**. Nazwę typu połączoną z jego przestrzenią

nazw określa się mianem **nazwy w pełni kwalifikowanej**. Wszystkie typy składające się na .NET Framework mają przestrzenie nazw, zwykle zaczynające się od System, choć niektóre typy specyficzne dla produktu zaczynają się od Microsoft. W CTS pojęcie przestrzeni nazw nie występuje. Wszystkie typy i referencje do nich są emitowane z wykorzystaniem w pełni kwalifikowanych nazw.

Przestrzenie nazw mają naturę hierarchiczną. Odwołujemy się do nich według kolejności, zatem na przykład korzeń hierarchii jest poziomem pierwszym, położony pod nim zbiór nazw — drugim itd. Rozważmy na przykład w pełni kwalifikowaną nazwę System.Collections.Generic.List<T>. System to poziom pierwszy, Collections to poziom drugi, Generic to poziom trzeci, a List<T> to nazwa typu. Pod względem technologicznym przestrzenie nazw nie mają nic wspólnego z podzespołami; typy należące do tej samej przestrzeni nazw mogą być umieszczone w wielu różnych podzespołach. Większość programistów stara się jednak zachować relację 1:1 między przestrzeniami nazw a podzespołami, ponieważ ułatwia to wyszukiwanie typów. Kiedy użytkownik szuka określonego typu należącego do określonej przestrzeni nazw, nie musi przeglądać wielu podzespołów, bo od razu wie, gdzie go znajdzie.

Definiowanie przestrzeni nazw

Aby przypisać typ do przestrzeni nazw w języku C#, wystarczy umieścić jego deklarację w bloku namespace:

```
namespace MyCompany.FooProject
{
    public class Foo { /* ... */ }
    public struct Bar { /* ... */ }

    namespace SubFeature
    {
        public class Baz { /* ... */ }
    }
}
```

Zauważmy, że mamy nadrzędną przestrzeń nazw MyCompany.FooProject, w której znajdują się klasy Foo i Bar. Ich w pełni kwalifikowane nazwy to odpowiednio MyCompany.FooProject.Foo oraz MyCompany.FooProject.Bar. Dalej mamy zagnieżdżoną przestrzeń nazw SubFeature. Nie musi ona znajdować się leksykalnie wewnątrz nadrzędnej przestrzeni nazw; moglibyśmy zapisać ją w innym miejscu albo w innym pliku jako MyCompany.FooProject.SubFeature. Zawiera ona pojedynczy typ Baz, którego w pełni kwalifikowaną nazwą jest MyCompany.FooProject.SubFeature.Baz.

Określanie przynależności do przestrzeni nazw

Różne języki rozwikłują odwołania do typów w przestrzeniach nazw na różne sposoby. Na przykład w C# za pomocą słowa kluczowego using (w VB — Import) można zadeklarować, że pewien zasięg używa przestrzeni nazw, aby uzyskać dostęp do typów bez potrzeby wpisywania w pełni kwalifikowanych nazw. Bez tego wszystkie pliki źródłowe trzeba by było pisać w następujący sposób:

```
class Foo
{
    void Bar()
    {
        System.Collections.Generic.List<int> list =
            new System.Collections.Generic.List<int>();
        // ...
    }
}
```

Na szczęście dzięki słowu kluczowemu `using` można oszczędzić na pisaniu:

```
using System.Collections.Generic

class Foo
{
    void Bar()
    {
        List<int> list = new List<int>();
        // ...
    }
}
```

Czasem w zaimportowanej przestrzeni nazw znajdują się typy o kolidujących nazwach. W takich przypadkach trzeba uciec się do stosowania w pełni kwalifikowanych nazw. Aby złagodzić ten problem, C# pozwala zdefiniować alias przestrzeni nazw. Przypuśćmy, że zaimportowaliśmy przestrzeń nazw `Microsoft.Internal.CrazyCollections`, w której zdefiniowany jest typ `List<T>`. Spowodowałoby to konflikt z przestrzenią nazw `System.Collections.Generic`, który trzeba jakoś rozwiązać:

```
using SysColGen = System.Collections.Generic;
using Microsoft.Internal.CrazyCollections;

class Foo
{
    void Bar()
    {
        SysColGen.List<int> list = new SysColGen.List<int>();
        // ...
    }
}
```

Typy specjalne

Na rysunku 2.1 zamieszczonym wcześniej w tym rozdziale pokazano kilka typów specjalnych wchodzących w skład hierarchii typów CTS. Są to delegacje, atrybuty niestandardowe i wyliczenia. Każdy z nich oferuje rozszerzone mechanizmy systemu typów do pisania kodu zarządzanego.

Delegacje

Delegacje to specjalne typy CTS, które reprezentują ściśle typizowane sygnatury metod. Typy delegacyjne wywodzą się ze specjalnego typu `System.Delegate`, który z kolei wywodzi się z `System.ValueType`. Delegacja może być utworzona i skonstruowana na dowolnej

kombinacji metody i instancji, w której sygnatura metody odpowiada sygnaturze delegacji. Delegacje można również tworzyć na metodach statycznych, a w takich przypadkach instancja jest niepotrzebna. W CLR instancja delegacji jest odmianą ściśle typizowanego wskaźnika do funkcji.

Większość języków oferuje składnię upraszczającą tworzenie i konkretyzację delegacji. Na przykład w C# nowy typ delegacyjny tworzy się za pomocą słowa kluczowego `delegate`:

```
public delegate void MyDelegate(int x, int y);
```

Instrukcja ta tworzy nowy typ delegacyjny o nazwie `MyDelegate`, który można konstruować na metodach mających typ zwrotny `void` i przyjmujących dwa argumenty typu `int`. W VB składnia jest podobna. Ukrywa ona wiele złożonych operacji, które musi wykonać kompilator w celu wygenerowania rzeczywistych delegacji, dzięki czemu użytkownicy języka mają ułatwione zadanie.

Naszą delegację można następnie skonstruować na docelowej metodzie, przekazywać, a wreszcie wywołać. W języku C# wygląda to jak zwykłe wywołanie funkcji:

```
class Foo
{
     void PrintPair(int a, int b)
     {
         Console.WriteLine("a = {0}", a);
         Console.WriteLine("b = {0}", b);
     }

     void CreateAndInvoke()
     {
         // Odpowiednik instrukcji new MyDelegate(this.PrintPair):
         MyDelegate del = PrintPair;
         del(10, 20);
     }
}
```

Metoda `CreateAndInvoke` konstruuje nową instancję `MyDelegate` na metodzie `PrintPair` z celem w postaci bieżącego wskaźnika `this`, po czym wywołuje ją.

Obsługa w CTS

Emitowany kod IL pokazuje złożoność obsługi delegacji w systemie typów:

```
struct MyDelegate : System.MulticastDelegate
{
    public MyDelegate(object target, IntPtr methodPtr);

    private object target;
    private IntPtr methodPtr;

    public internal void Invoke(int x, int y);
    public internal System.IAsyncResult BeginInvoke(int x, int y,
        System.IAsyncCallback callback, object state);
    public internal void EndInvoke(System.IAsyncResult result);
}
```

Konstruktor służy do formowania delegacji na docelowym obiekcie i wskaźniku do funkcji. Metody Invoke, BeginInvoke oraz EndInvoke implementują procedurę wywoływania delegacji i są oznaczone jako internal (tzn. runtime w IL), aby wskazać, że implementację zapewnia CLR; w IL ich ciało jest puste. Metoda Invoke wykonuje wywołanie synchroniczne, a BeginInvoke oraz EndInvoke realizują wzorzec znany z modelu programowania asynchronicznego (opisywanego dokładniej w rozdziale 10.), aby wykonać asynchroniczne wywołanie metody.

Zauważmy najpierw, że typ MyDelegate narusza jedną z omówionych wyżej zasad, mianowicie tę, że struktury nie mogą wywodzić się z typów innych niż ValueType. CTS zapewnia specjalną obsługę delegacji, więc jest to dozwolone. Zwróćmy też uwagę, że MyDelegate wywodzi się z MulticastDelegate; typ ten jest wspólną klasą bazową dla wszystkich delegacji tworzonych w C# i obsługuje delegacje, które mają wiele celów. W punkcie poświęconym zdarzeniom wyjaśnię, do czego to się przydaje. MulticastDelegate definiuje też wiele dodatkowych metod, które chwilowo zignorujemy i które pominięto w zamieszczonej wyżej deklaracji. Więcej informacji o tych metodach można znaleźć w rozdziale 14.

Aby uformować delegację na docelowym obiekcie, IL używa konstruktora MyDelegate. Konstruktor przyjmuje parametr wejściowy typu object, którym jest wskaźnik this przekazywany w momencie wywoływania metody (albo null w przypadku metod statycznych), oraz IntPtr reprezentujący zarządzany wskaźnik do metody CLR. Jest to wskaźnik do funkcji w stylu C, który wskazuje kod skompilowany przez JIT w środowisku uruchomieniowym. Sygnatura Invoke odpowiada sygnaturze zdefiniowanej delegacji, a CLR zapewnia zarówno statycznie, jak i dynamicznie, że wskaźnik do funkcji zawarty w delegacji pasuje do tej sygnatury.

Pokazany wyżej kod CreateAndInvoke emituje poniższą sekwencję IL:

```
ldarg.0    // wczytuje wskaźnik this na użytek metody CreateAndInvoke
ldftn      void Foo::PrintPair(int32, int32)
newobj     instance void MyDelegate::.ctor(object, native int)
ldc.i4.s   10
ldc.i4.s   20
callvirt   instance void MyDelegate::Invoke(int32, int32)
ret
```

Zauważmy, że kod wczytuje wskaźnik do metody PrintPair za pomocą instrukcji ldftn, a następnie używa metody Invoke zdefiniowanej w MyDelegate, aby wywołać docelową metodę. Powoduje to pośrednie wywołanie metody PrintPair z argumentami w postaci wartości 10 i 20.

Delegacje zostaną opisane znacznie bardziej szczegółowo w rozdziale 14.

Kowariancja i kontrawariancja

Reguły wiązania delegacji, które podałem wcześniej, są nieco uproszczone. Stwierdziłem, że wartość zwrotna i typy parametrów docelowej metody muszą dokładnie odpowiadać delegacji, aby można było uformować instancję delegacji na tej metodzie. Z technicznego punktu widzenia nie jest to do końca prawdą. CLR 2.0 dopuszcza tzw. delegacje **kowariancyjne** i **kontrawariancyjne** (choć wersje 1.x na to nie pozwalały). Terminy te są dobrze zakorzenione w informatyce i określają pewne postaci polimorfizmu w systemie typów. Kowariancja oznacza, że można podstawić bardziej pochodny typ w miejscu, w którym oczekiwany jest mniej pochodny typ. Kontrawariancja jest odwrotnością kowariancji — oznacza, że można podstawić mniej pochodny typ w miejscu, w którym oczekiwany jest bardziej pochodny typ.

Wejście kowariancyjne jest dozwolone w kategoriach tego, co użytkownik może dostarczyć metodzie. Jeśli metoda oczekuje klasy `Bazowa`, a ktoś dostarczy jej instancję klasy `Pochodna` (wywodzącej się z `Bazowa`), środowisko uruchomieniowe na to zezwoli. Jest to standardowa postać obiektowego polimorfizmu. Podobnie wyjście może być kontrawariancyjne w tym sensie, że jeśli wywołujący oczekuje mniej pochodnego typu, można mu dostarczyć instancję typu bardziej pochodnego.

> Zagadnienia ko- i kontrawariancji są dość skomplikowane. W większości literatury można przeczytać, że prawidłowe jest kontrawariancyjne wejście i kowariancyjne wyjście. Jest to dokładna odwrotność tego, co napisałem powyżej! W literaturze stwierdzenie to odnosi się zazwyczaj do możliwości przesłaniania metody za pomocą ko- lub kontrawariancyjnej sygnatury, a w takim kontekście jest prawdziwe. Klasy pochodne mogą bezpiecznie rozluźnić wymagania typizacji dotyczące wejścia i zaostrzyć wymagania dotyczące wyjścia, jeśli jest to wskazane. Wywołanie metody przez wersję bazową nadal będzie bezpieczne typologicznie. CLR nie obsługuje natywnie ko- i kontrawariancji w taki sposób.
>
> W przypadku delegacji przyglądamy się jednak temu samemu problemowi z innej perspektywy: stwierdzamy po prostu, że ktokolwiek dokonuje wywołania przez delegację, może podlegać bardziej specyficznym wymaganiom dotyczącym wejścia i może oczekiwać mniej specyficznego wyjścia. Jeśli weźmiemy pod uwagę, że wywoływanie funkcji za pośrednictwem delegacji przypomina wywoływanie za pośrednictwem sygnatury klasy bazowej, jest to ta sama zasada.

Rozważmy na przykład poniższe definicje:

```
class A { /*...*/ }
class B : A { /*...*/ }
class C : B { /*...*/ }

B Foo(B b) { return b; }
```

Gdybyśmy chcieli uformować delegację na metodzie `Foo`, to do dokładnego dopasowania potrzebowaliśmy delegacji, która zwraca `B` i przyjmuje `B` jako parametr wejściowy. Wyglądałaby ona tak jak `MyDelegate1` poniżej:

```
delegate B MyDelegate1(B b);
delegate B MyDelegate2(C c);
delegate A MyDelegate3(B b);
delegate A MyDelegate4(C c);
```

Możemy jednak użyć kowariancji wejścia, aby użytkownicy wywołujący metodę za pośrednictwem delegacji musieli podawać bardziej specyficzny typ niż `B`. Ilustruje to powyższa delegacja `MyDelegate2`. Możemy też użyć kontrawariancji, aby ukryć fakt, że metoda `Foo` zwraca `B`, i stworzyć pozór, że zwraca `A`, jak w przypadku delegacji `MyDelegate3`. Możemy wreszcie użyć ko- i kontrawariancji jednocześnie, jak pokazano na przykładzie `MyDelegate4`. Wszystkie te sygnatury delegacji zostaną powiązane z metodą `Foo`.

Delegacje anonimowe (mechanizm językowy)

Jest to funkcja języka C# 2.0, a nie samego CLR. Delegacje anonimowe są jednak tak użyteczne i powszechne, że zasługują na specjalną wzmiankę. Sprawiają one, że przekazywanie wskaźników do metod jako argumentów innych metod jest bardzo łatwe, więc czasem lepiej jest napisać odpowiedni blok kodu „w miejscu”, zamiast definiować oddzielną metodę. Jest to jednak wyłącznie lukier syntaktyczny.

Rozważmy metodę, która przyjmuje delegację i kilkakrotnie ją wywołuje:

```
delegate int IntIntDelegate(int x);

static void TransformUpTo(IntIntDelegate d, int max)
{
    for (int i = 0; i <= max; i++)
        Console.WriteLine(d(i));
}
```

Gdybyśmy chcieli przekazać metodzie `TransformUpTo` funkcję, która oblicza pierwiastek kwadratowy argumentu wejściowego, musielibyśmy najpierw napisać oddzielną metodę, na której formowalibyśmy delegację. W .NET 2.0 możemy osiągnąć to samo dzięki delegacji anonimowej:

```
TransformUpTo(delegate(int x) { return x * x; }, 10);
```

Kompilator C# generuje w podzespole anonimową metodę, która implementuje operację zdefiniowaną w nawiasie klamrowym. Kompilator potrafi wywnioskować, że funkcja zwraca liczbę całkowitą (ponieważ jest używana w kontekście, w którym oczekiwana jest funkcja zwracająca liczbę całkowitą), a typy parametrów są podane jawnie w nawiasie okrągłym za słowem kluczowym `delegate`.

Nie poświęcimy zbyt wiele czasu temu mechanizmowi, ale warto wiedzieć, że jest on bardzo złożony i niezwykle elastyczny. Delegacja może **przechwytywać** zmienne, które są widoczne leksykalnie. Kompilator wykonuje mnóstwo pracy, aby działało to prawidłowo; aby to docenić, można przyjrzeć się generowanemu kodowi IL. Rozważmy poniższy przykład:

```
delegate void FooBar();

void Foo()
{
    int i = 0;
    Bar(delegate { i++; });
    Console.WriteLine(i);
}

void Bar(FooBar d)
{
    d(); d(); d();
}
```

Zapewne nikogo nie zdziwi, że wynikiem wywołania `Foo` jest `3`. W metodzie `Foo` deklarowana jest lokalna zmienna `i`, początkowo ustawiona na `0`. Następnie tworzymy anonimową delegację, której wywołanie powoduje zwiększenie `i` o `1`. Przekazujemy tę delegację do funkcji `Bar`, która stosuje ją trzykrotnie.

To, co robi kompilator C#, jest pomysłowe i imponujące (a jednocześnie odrobinę przerażające). Wykrywa on, że uzyskujemy dostęp do lokalnej zmiennej z poziomu delegacji, i przekształca ją w obiekt zaalokowany na stercie. Typ tego obiektu jest generowany przez kompilator i nigdy nie jest widoczny w kodzie. Następnie kompilator „magicznie" sprawia, że referencje do `i` w lokalnym zasięgu odnoszą się do tego samego obiektu co w delegacji.

To bardzo miło ze strony kompilatora, ale Czytelnicy przywiązujący dużą wagę do optymalizacji kodu mogą niepokoić się o wydajność tego mechanizmu. To, co wygląda na lokalny

dostęp do zmiennej, w rzeczywistości wymaga alokacji obiektu i przynajmniej dwóch poziomów pośredniości. Obawy nie są do końca nieuzasadnione, ale rzadko opłaca się przejmować tego rodzaju mikrodostrajaniem wydajności.

Atrybuty niestandardowe

W tym rozdziale widzieliśmy już różne słowa kluczowe IL, za pomocą których kompilatory modyfikują działanie niektórych typów i składowych. Są to **atrybuty pseudoniestandardowe**, które są serializowane w metadanych w stałym miejscu oraz ze stałym rozmiarem i zwykle mają własne, efektywne sloty pamięciowe w strukturach danych CLR. CLR rozpoznaje te atrybuty i odpowiednio na nie reaguje, w zależności od udokumentowanego kontraktu danego atrybutu.

Użytkownicy mogą jednak również dołączać **atrybuty niestandardowe** do typów danych CLR. W tym celu należy utworzyć nowy typ wywodzący się z `System.Attribute` i podać zbiór pól oraz właściwości, które atrybut będzie zawierał po konkretyzacji w czasie działania programu. Użytkownik atrybutu będzie mógł następnie podłączyć jego instancję do podzespołu, modułu, typu lub składowej. Atrybut i informacje o jego konkretyzacji są serializowane w metadanych podzespołu i mogą zostać odtworzone podczas wykonywania programu. Komponenty użytkownika mogą sprawdzać obecność tych atrybutów i odpowiednio reagować, podobnie jak środowisko uruchomieniowe robi to w przypadku atrybutów pseudoniestandardowych.

Oto przykładowy typ atrybutu w C#:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
class MyAttribute : System.Attribute
{
    private string myString;
    private int mySize;

    public MyAttribute(string myName)
    {
        this.myString = myName;
        this.mySize = 8; // wartość domyślna
    }

    public string MyName { get { /* ... */ } }
    public int MySize { get { /* ... */ } set { /* ... */ } }
}
```

Użytkownik będzie mógł dołączyć ten atrybut do klasy lub metody (dwóch legalnych celów określonych przez `AttributeUsage`, który sam jest atrybutem). W C# robi się to za pomocą składni `[Atrybut]`, a w VB —`<Atrybut>`, na przykład:

```
[MyAttribute("MyFoo")]
class Foo
{
    [MyAttribute("MyFoo::MyBar", MySize = 16)]
    public void Bar()
    {
    }
}
```

Zauważmy, że składnia C# umożliwia wywołanie konstruktora i dostarczenie zbioru wartości. Typ System.Reflection.MemberInfo i wywodzące się z niego podtypy (Type, MethodInfo, PropertyInfo, FieldInfo itd.) umożliwiają komponentom odczytanie atrybutów komponentu systemu typów z wykorzystaniem metody GetCustomAttributes. Poniższy fragment kodu odczytuje atrybuty Foo i Foo.Bar:

```
Type fooType = typeof(Foo);
object[] myAttrOnType = fooType.GetCustomAttributes(
    typeof(MyAttribute), false);
if (myAttrOnType.Length > 0)
{
    // Robimy coś specjalnego... typ ma atrybut MyAttribute
}

MethodInfo fooBarMethod = fooType.GetMethod("Bar");
foreach (object attr in fooBarMethod.GetCustomAttributes(false))
{
    if (attr.GetType() == typeof(MyAttribute))
    {
        // Ma atrybut MyAttribute, zróbmy coś z tym
    }
}
```

To był bardzo szybki przegląd atrybutów niestandardowych. W rozdziale 14. można znaleźć więcej informacji o pokazanych wyżej wywołaniach API, wewnętrznym działaniu atrybutów niestandardowych (na przykład o formacie ich przechowywania) oraz ich roli w programowaniu dynamicznym.

Wyliczenia

Wyliczenie to specjalny typ, który odwzorowuje zbiór nazw na wartości liczbowe. Korzystanie z wyliczeń jest alternatywą dla osadzania stałych w kodzie i zapewnia wyższy poziom nominalnego bezpieczeństwa typologicznego. Typy wyliczeniowe wyglądają w metadanych podobnie jak zwykłe typy, ale stosują się do ścisłych reguł zdefiniowanych w CTS. Na przykład definiowanie metod lub konstruktorów w typach wyliczeniowych jest niedozwolone, podobnie jak implementowanie interfejsów, i mogą one mieć tylko pojedyncze pole reprezentujące wartość. Reguły te istnieją po to, aby wyliczenia były wydajne i aby języki mogły traktować je w pewien specyficzny sposób. Na szczęście większość języków (w tym C#) oferuje składnię, która abstrahuje te reguły.

Typ wyliczeniowy wywodzi się z System.Enum, który z kolei wywodzi się z System.ValueType. Każdy opiera się na konkretnym typie podstawowym, jednym spośród Boolean, Char, Byte, Int16, Int32, Int64, SByte, UInt16, UInt32, UInt64, IntPtr, UIntPtr, Single i Double. W większości języków domyślnie używa się Int32; jest on dobrym kompromisem między zajmowaną pamięcią a możliwością przyszłego rozszerzania wyliczenia o nowe wartości.

Instancja danego wyliczenia zawiera pojedyncze pole reprezentujące wartość. Wyliczenia są typami wartościowymi, więc instancja wyliczenia jest zasadniczo równoważna wartości typu podstawowego, z tym że odwołujemy się do niej według nazwy typu wyliczeniowego, a przekształcenia w tę i z powrotem są dość łatwe.

W C# w celu utworzenia nowego wyliczenia wystarczy napisać:

```
enum Color : byte
{
    Czerwony,
    Zielony,
    Niebieski
}
```

Część określająca typ wyliczenia jako `byte` jest opcjonalna. Kompilator C# przekształca tę definicję w metadane, które są zgodne z powyższymi regułami:

```
.class private auto ansi sealed Color
    extends [mscorlib]System.Enum
{
    .field public specialname rtspecialname uint8 value__
    .field public static literal valuetype Color Czerwony = uint8(0x00)
    .field public static literal valuetype Color Zielony = uint8(0x01)
    .field public static literal valuetype Color Niebieski = uint8(0x02)
}
```

Wyliczenia `Color` można następnie użyć w programie. Wyobraźmy sobie, że pytamy użytkownika o jego ulubiony kolor. Możemy użyć wyliczenia do zaprezentowania listy, przeanalizowania danych wejściowych, a następnie do przekazywania wybranego koloru do różnych procedur:

```
class ColorPick
{
    static void Main()
    {
        Color favorite = SelectFavoriteColor();
        RespondToFavoriteColor(favorite);
    }

    static Color SelectFavoriteColor()
    {
        Color favorite = (Color)(0xff);

        // Program działa w pętli, dopóki nie zostanie wybrany prawidłowy kolor
        do
        {
            // Wyświetlamy monit i listę prawidłowych kolorów
            Console.Write("Wpisz swój ulubiony kolor (");
            foreach (string name in Enum.GetNames(typeof(Color)))
                Console.Write(" {0} ", name);
            Console.Write("): ");

            string input = Console.In.ReadLine();
            try
            {
                favorite = (Color)Enum.Parse(typeof(Color), input, true);
            }
            catch (ArgumentException)
            {
                // Dane wpisane przez użytkownika nie pasują do nazw w wyliczeniu
                Console.WriteLine("Błędny kolor, wybierz jeszcze raz!");
                Console.WriteLine();
```

```
            }
        }
        while (favorite == (Color)(0xFF));

        return favorite;
    }

    static void RespondToFavoriteColor(Color c)
    {
        // Zauważmy, że w C# można stosować instrukcję switch na wyliczeniu
        switch (c)
        {
            case Color.Czerwony:
                // Robimy coś dla osób, które lubią kolor czerwony
                // ...
                break;
            case Color.Zielony:
                // Robimy coś dla osób, które lubią kolor zielony
                // ...
                break;
            case Color.Niebieski:
                // Robimy coś dla osób, które lubią kolor niebieski
                // ...
                break;
            default:
                // Wykryto nieznany kolor.
                // Jest to prawdopodobnie błąd, ale musimy go obsłużyć!
                break;
        }
    }
}
```

Zauważmy pomocnicze metody statyczne takie jak `GetNames` i `Parse` należące do samego typu `Enum`. Klasę `Enum` omówimy dokładniej w dalszej części rozdziału.

Wyliczenia znacznikowe

Najczęściej wartości wyliczenia wzajemnie się wykluczają, co oznacza, że dana instancja wyliczenia może mieć tylko jedną z możliwych wartości. Czasem jednak pojedyncza instancja musi reprezentować kombinację wartości. Typ wyliczeniowy może być oznaczony atrybutem niestandardowym `System.FlagsAttribute`, który sprawia, że języki dopuszczają łączenie i wyodrębnianie poszczególnych wartości z pojedynczej instancji. Rozważmy na przykład zbiór uprawnień do operacji na pliku:

```
[Flags]
enum FileAccess
{
    Read = 1,
    Write = 2,
    ReadWrite = 3;
}
```

Pliki często otwiera się jednocześnie do odczytu (`Read`) i zapisu (`Write`). Użycie wyliczenia znacznikowego pozwala wyrazić tę ideę. Zauważmy, że liczbowe wartości `Read` i `Write` są potęgami 2, począwszy od 1. Dlaczego? Otóż łączenie lub wyodrębnianie wartości

z pojedynczej instancji odbywa się z wykorzystaniem bitowych operacji AND lub OR. Kolejne elementy wyliczenia zajmowałyby wartości 4, 8, 16, 32 itd. Kompilator C# nie numeruje ich automatycznie — trzeba zrobić to ręcznie, bo w przeciwnym razie zostanie zastosowany sekwencyjny sposób numerowania, który nie działałby prawidłowo z operacjami bitowymi.

Zauważmy, że w celu zasygnalizowania możliwości zapisu i odczytu dwie niezależne wartości są logicznie sumowane: 1 | 2 to 3, stąd pomocnicza wartość ReadWrite. Dzięki niej w poniższym przykładzie druga instrukcja jest równoważna pierwszej, ale bardziej czytelna:

```
FileAccess rw1 = FileAccess.Read | FileAccess.Write; // wartość 3
FileAccess rw2 = FileAccess.ReadWrite; // wartość 3
```

Wyliczeń znacznikowych oczywiście nie można testować pod kątem równości, aby sprawdzić, czy instancja (z kombinacją wartości) zawiera konkretną wartość. Jeśli na przykład sprawdzamy, czy instancja FileAccess zezwala na odczyt, moglibyśmy napisać:

```
FileAccess fa = /* ... */;
if (fa == FileAccess.Read)
    // mamy zezwolenie na odczyt
else
    // nie mamy dostępu
```

Niestety, gdyby wyliczenie fa miało wartość FileAccess.ReadWrite, test zawiódłby i nie uzyskalibyśmy dostępu do pliku. W większości przypadków byłoby to niewłaściwe. Musimy zatem użyć bitowej operacji AND i wyodrębnić poszczególne wartości z instancji wyliczenia:

```
FileAccess fa = /* ... */;
if ((fa & FileAccess.Read) != 0)
    // mamy zezwolenie na odczyt
else
    // nie mamy dostępu
```

Ten test daje prawidłowy wynik, tzn. zezwala na dostęp, kiedy wartością wyliczenia jest FileAccess.ReadWrite.

Ograniczenia bezpieczeństwa typologicznego

Wspomniałem już kilkakrotnie, że wyliczenia zapewniają bezpieczeństwo typologiczne tam, gdzie dawniej istniało ryzyko, że użytkownik poda wartość spoza dopuszczalnego zakresu. Jest to jednak nieco mylące. Wyliczenia opierają się na prostych, podstawowych typach danych, więc można sfabrykować instancję, która zawiera wartość nieodpowiadającą żadnej nazwie. W przykładzie zamieszczonym na początku niniejszego punktu — z wyliczeniem Color o wartościach Czerwony (0), Zielony (1) i Niebieski (2) — możemy utworzyć instancję, która ma wartość 3!

```
Color MakeFakeColor()
{
    return (Color)3;
}
```

Nie ma sposobu, aby temu zapobiec. Jedyne, co można zrobić, to zachować ostrożność w razie przyjmowania instancji wyliczenia z niezaufanego źródła. Jeśli na przykład napisaliśmy publiczne wywołanie API, które przyjmuje wyliczenie jako parametr wejściowy, musimy

zawsze sprawdzać, czy jego wartość mieści się w dozwolonym zakresie. Aby to ułatwić, typ `System.Enum` definiuje statyczną metodę pomocniczą `IsDefined`. Zwraca ona wartość logiczną, która wskazuje, czy dana wartość należy do zdefiniowanego zakresu określonego wyliczenia:

```
void AcceptColorInput(Color c)
{
    // Sprawdzamy, czy wejściowa wartość wyliczenia jest prawidłowa
    if (!Enum.IsDefined(typeof(Color), c))
        throw new ArgumentOutOfRangeException("c");

    // Zweryfikowaliśmy dane wejściowe, możemy z nimi pracować
    // ...
}
```

Jeśli ktoś celowo wywoła metodę `AcceptColorInput` z nieprawidłową instancją wyliczenia `Color`, metoda zgłosi wyjątek, zamiast zawieść w nieoczekiwany sposób:

```
AcceptColorInput(MakeFakeColor());
```

Metoda ta nie sprawdza się zbyt dobrze w przypadku wyliczeń znacznikowych. Jeśli instancja takiego wyliczenia reprezentuje kombinację wartości, to sama nie jest prawidłowym wyborem spośród zakresu wartości wyliczenia. Zwykle jest to jednak do przyjęcia. Prawdopodobnie kod nie będzie wykonywał instrukcji `switch` ani sprawdzał dokładnej równości. Zamiast tego zawiodą wszystkie testy bitów, prowadząc do gałęzi programu, która zapoczątkuje kontrolowane zgłoszenie błędu.

Inne metody pomocnicze

Typ `System.Enum` ma kilka przydatnych metod do pracy z wyliczeniami, zwłaszcza w zakresie analizy składniowej, weryfikacji oraz generowania list prawidłowych nazw lub wartości. W wersji 2.0 większość metod nadal nie ma wersji generycznych, choć niewątpliwie miałoby to sens; zamiast tego przyjmują one typ wyliczenia jako parametr `Type`. Większość tych metod przedstawiono już w powyższych fragmentach kodu, a pozostałe nietrudno opanować samemu.

Generyki

W CLR 2.0 nowa funkcja nazywana genetycznością pozwala programistom stosować w pisanym przez nich kodzie **polimorfizm parametryczny**. Termin ten oznacza, że można pisać kod operujący na instancjach, których typ nie jest znany podczas kompilowania, a użytkownicy mogą później wykorzystać go do pracy z konkretnymi typami. Funkcja ta jest zbliżona do szablonów C++ — i ma zwodniczo podobną składnię — choć nie jest tak elastyczna, a zarazem niewygodna i problematyczna.

Podstawy i terminologia

Podstawowym założeniem generyczności jest to, że typy (klasy, struktury, delegacje) lub metody można definiować w kategoriach **formalnych parametrów typu**, które w definicjach mogą zostać zastąpione rzeczywistymi typami. Liczba parametrów akceptowanych

przez typ lub metodę określana jest mianem **arności typu**. Typ o arności 1 lub więcej jest nazywany **typem generycznym**, a metoda o arności 1 lub więcej — **metodą generyczną**. Użytkownik generycznego typu lub metody musi dostarczyć **rzeczywiste argumenty typu**, co określa się mianem **konkretyzacji typu**. Zauważmy, że przypomina to konkretyzowanie instancji obiektu za pomocą konstruktora.

Typ może na przykład przyjmować pojedynczy parametr o nazwie T i używać go w swojej definicji:

```
class Foo<T>
{
    T data;

    public Foo(T data)
    {
        this.data = data;
    }

    public U Convert<U>(MyConverter<T,U> converter)
    {
        return converter(data);
    }
}
```

Typ Converter to delegacja zdefiniowana w przestrzeni nazw System, której sygnatura wygląda następująco:

```
public delegate TOutput Converter<TInput, TOutput>(TInput input);
```

Typ Foo ma pojedynczy argument (T) i używa go do zdefiniowania instancyjnego pola data. Jego arność wynosi 1. Typ data nie będzie znany, dopóki wywołujący nie skonstruuje typu Foo<T>, podstawiając argument typu w miejsce T. Argumentem tym może być int, string albo dowolny inny typ, w tym typy zdefiniowane przez użytkownika. Z tej przyczyny nie da się zrobić zbyt wiele z czymś, co ma typ T. Statycznie można wykonywać na nim tylko operacje zdefiniowane w klasie Object, ponieważ jest to wspólna baza dla wszystkich typów, co oczywiście oznacza, że można przekazywać go do oczekujących tego metod, a także przeprowadzić na nim dynamiczną introspekcję z wykorzystaniem mechanizmu refleksji (refleksja zostanie opisana w rozdziale 14.).

Zauważmy też, że T użyto również w definicji parametru konstruktora oraz w innym generycznym typie Converter<T, U> przyjmowanym jako parametr wejściowy metody Convert. Ilustruje to przekazywanie instancji T w sposób bezpieczny typologicznie, mimo że ich rzeczywiste wartości będą znane dopiero w czasie działania programu. Mamy wreszcie metodę Convert, która przyjmuje swój własny parametr typu o nazwie U. W swojej definicji ma ona dostęp zarówno do T, jak i do U i przyjmuje parametr Convert<T, U>. Zwróćmy uwagę, że jej typ zwrotny to U.

Konkretyzacja

Użytkownik typu Foo<T> może napisać następujący kod:

```
Foo<int> f = new Foo<int>(2005);
string s = f.Convert<string>(delegate(int i) { return i.ToString(); });
Console.WriteLine(s);
```

Konkretyzuje to typ Foo<T> z argumentem typu int. Od tego momentu możemy wyobrazić sobie, że w powyższej definicji klasy każde wystąpienie T zostaje zastąpione przez int. Innymi słowy, pole data jest teraz typu int, jego konstruktor przyjmuje int, a metoda Convert przyjmuje Converter<int, U>, gdzie typ U jest nadal nieznany. Foo<int> to typ skonkretyzowany typu generycznego Foo<T>; jest **w pełni skonstruowany**, ponieważ dostarczyliśmy argumenty odpowiadające wszystkim parametrom. Następnie tworzymy instancję tego typu, przekazując 2005 jako argument konstruktora.

Dalej kod konkretyzuje metodę Convert<U> z argumentem typu string. I w tym przypadku możemy w wyobraźni zamienić każde wystąpienie U na string w definicji i ciele metody Convert. Innymi słowy, zwraca ona string, a jako parametr przyjmuje Convert<int, string>. Następnie przekazujemy anonimową delegację, która przekształca int w string przez wywołanie metody ToString. W rezultacie s zawiera łańcuch "2005", który wypisujemy na konsoli. Oczywiście, w każdej instancji Foo<T> można podać inną wartość T, a w każdym wywołaniu metody Convert<U> — inną wartość U.

Obsługa generyków w innych językach

Oczywiście, powyższe przykłady skupiały się na języku C#; na początku rozdziału zaznaczyłem, że tak będzie. Ale generyki to mechanizm oplatający cały system typów, a zarówno VB, jak i C++/CLI obsługują typy i metody generyczne. Powyższą klasę można by zapisać w VB z wykorzystaniem składni Of T:

```
Class Foo(Of T)
    Private data As T

    Public Sub Foo(data As T)
        Me.data = data
    End Sub

    Public Function Convert(Of U)(converter As Converter(Of T, U)) As U
        Return converter(Me.data)
    End Function
End Class
```

Tekstowy IL reprezentuje generyki z wykorzystaniem własnej składni. Część poniższego kodu może być niezrozumiała, ponieważ nie omówiliśmy jeszcze IL (jest to temat następnego rozdziału). Pomimo to warto zwrócić uwagę na syntaktyczne różnice w sposobie reprezentowania parametrów typu oraz wykorzystywania ich w definicjach typów i metod:

```
.class auto ansi nested private beforefieldinit Foo`1<T>
    extends [mscorlib]System.Object
{
    .field private !T data
    .method public hydebysig specialname rtspecialname
            instance void .ctor(!T data) cil managed
    {
        .maxstack  8
        ldarg.0
        call instance void [mscorlib]System.Object::.ctor()
        ldarg.0
        ldarg.1
```

```
        stfld !0 class Foo`1<!T>::data
        ret
    }

    .method public hydebysig instance !!U
        Convert<U>(class [mscorlib]System.Converter`2<!T, !!U> convert)
        cil managed
    {
        .maxstack  2
        .locals init ([0] !!U $0000)
        ldarg.1
        ldarg.2
        ldfld !0 class Foo`1<!T>::data
        callvirt instance !1 class
            [mscorlib]System.Converter`2<!T,!!U>::Invoke(!0)
        stloc.0
        ldloc.0
        ret
    }
}
```

Zauważmy, że sam typ nosi nazwę `Foo`1<T>`. Jego arność jest reprezentowana przez `` `1 ``, a w jego definicji `!T` oznacza parametr `T`. Metoda jest podobna, choć nie ma znacznika arności, a do swoich parametrów odwołuje się z wykorzystaniem podwójnego wykrzyknika, na przykład `!!U`.

Generyki początkowo mogą wydawać się skomplikowane, ale kto je opanuje, będzie dysponował bardzo zaawansowanym narzędziem. Najlepiej zacząć do przeanalizowania przykładu.

Przykład: kolekcje

Kolekcje to kanoniczny przykład, który ilustruje zalety generyków. Zważywszy na dostępność bardzo dojrzałej standardowej biblioteki szablonów (ang. *Standard Template Library*, STL) C++, nie ma w tym nic dziwnego: oferuje ona wiele doskonałych interfejsów API, które przekonają każdego sceptyka. Przyczyna jest oczywista: kolekcje bez generyków to utrapienie; kolekcje z generykami są eleganckie, a praca z nimi wydaje się naturalna.

W wersji 1.x platformy Framework większość programistów używała typu `System.Collections.ArrayList` do przechowywania kolekcji obiektów lub wartości. Typ ten zawiera m.in. metody do dodawania, lokalizowania, usuwania i wyliczania elementów. Jeśli przyjrzymy się publicznej części definicji typu `ArrayList`, znajdziemy kilka metod, które operują na elementach typu `System.Object`, na przykład:

```
public class ArrayList : IList, ICollection, IEnumerable, ICloneable
{
    public virtual int Add(object value);
    public virtual bool Contains(object value);
    public object[] ToArray();
    public object this[int index] { get; set; }
    // I tak dalej...
}
```

Elementy `ArrayList` są typu `object`, aby lista mogła przechowywać dowolne obiekty lub wartości. Inne kolekcje, takie jak `Stack` i `Queue`, również działają w ten sposób. Wystarczy jednak chwilę popracować z tymi typami, aby dostrzec ich wady.

Brak bezpieczeństwa typologicznego

Po pierwsze i najważniejsze, większość kolekcji wcale nie jest przeznaczona do przechowywania instancji zupełnie dowolnych typów. Zwykle używa się listy klientów, listy łańcuchów albo listy jakichś elementów o wspólnym typie bazowym. Kolekcja rzadko jest zbiorem przypadkowych elementów, z którym pracuje się wyłącznie za pośrednictwem interfejsu `object`. Oznacza to, że każdy obiekt pobrany z kolekcji wymaga rzutowania:

```
ArrayList listOfStrings = new ArrayList();
listOfStrings.Add("jakiś łańcuch");
// ...
string contents = (string)listOfStrings[0]; // konieczne rzutowanie
```

To jest jednak najmniejszy problem.

Pamiętajmy, że instancja `ArrayList` nie dostarcza żadnych informacji o naturze swoich elementów. Na każdej liście można zapisać zupełnie dowolny element. Problem pojawia się dopiero podczas pobierania elementów z listy. Rozważmy poniższy kod:

```
ArrayList listOfStrings = new ArrayList();
listOfStrings.Add("jeden");
listOfStrings.Add(2); // Ups!
listOfStrings.Add("trzy");
```

Ten fragment kompiluje się bez problemów, choć zamiast łańcucha `"dwa"` przypadkowo dodaliśmy do listy liczbę całkowitą `2`. Nic nie wskazuje, że zamierzamy dodawać do listy tylko łańcuchy (może z wyjątkiem nazwy zmiennej, która oczywiście może zostać zmieniona), a już z pewnością nic tego nie wymusza. Gdzieś dalej ktoś może napisać następujący kod:

```
foreach (string s in listOfStrings)
    // Zrobić coś z łańcuchem s...
```

W tym momencie program zgłosi wyjątek `CastClassException` z wiersza `foreach`. Dlaczego? Ponieważ na liście znajduje się wartość `int`, której oczywiście nie można rzutować na `string`.

Czy nie byłoby dobrze, gdybyśmy mogli ograniczyć listę wyłącznie do elementów typu `string`? Wiele osób rozwiązuje ten problem, opracowując własne kolekcje, pisząc ściśle typizowane metody (na przykład `Add`, `Remove` itd.), które działają tylko z prawidłowym typem, i przesłaniając lub ukrywając metody operujące na typie `object`, aby przeprowadzać dynamiczną kontrolę typów. Określa się to mianem **ściśle typizowanych kolekcji**.

`System.Collections.Specialized.StringCollection` to ściśle typizowana kolekcja dla typu `string`. Gdyby w powyższym przykładzie użyto `StringCollection` zamiast `ArrayList`, kod nie skompilowałby się, gdybyśmy spróbowali dodać do kolekcji wartość `int`. Nawet to podejście ma jednak wady. Jeśli uzyskujemy dostęp do metod za pośrednictwem interfejsu `IList` — obsługiwanego przez `StringCollection` i typizowanego jako `object` — kompilator niczego nie zauważy. Dopiero metoda `Add` wykryje niezgodny typ i zgłosi wyjątek w czasie wykonania programu. Trzeba przyznać, że jest to lepsze niż wyjątek zgłoszony podczas pobierania elementów z listy, ale nadal dalekie od doskonałości.

Koszty opakowywania i odpakowywania

Klasa `ArrayList` powoduje również bardziej subtelne problemy. Najważniejszym z nich jest to, że tworzenie list typów wartościowych wymaga opakowania wartości przed umieszczeniem ich na liście oraz odpakowania ich podczas pobierania. W przypadku długich list opakowywanie i odpakowywanie mogą łatwo zdominować całą operację. Rozważmy poniższy przykład; generuje on listę 1 000 000 przypadkowych liczb całkowitych, a następnie oblicza ich sumę:

```
ArrayList listOfInts = GenerateRandomInts(1000000);
long sum = 0;
foreach (int x in listOfInts)
    sum += x;
// ...
```

Kiedy profilowałem ten kod, okazało się, że opakowywanie i odpakowywanie zajmują 74% czasu wykonania programu! Koszty te można wyeliminować przez utworzenie własnej, ściśle typizowanej kolekcji wartości `int` (przy założeniu, że dostęp do elementów uzyskujemy bezpośrednio za pomocą metod kolekcji, a nie na przykład za pośrednictwem interfejsu `IList`).

Rozwiązanie: generyki

Tworzenie i konserwowanie własnych kolekcji jest czasochłonne i ma niewiele wspólnego z logiką aplikacji, a przy tym jest tak powszechne, że często w jednej aplikacji funkcjonuje wiele tak zwanych **ściśle typizowanych kolekcji**. Spójrzmy prawdzie w oczy: nie ma w tym nic przyjemnego.

Rozwiązaniem tego problemu jest nowy, generyczny typ kolekcji `System.Collections.Generic.List<T>`, który ma arność równą 1 i parametr typu `T`. Argument typu, określany podczas konkretyzacji, reprezentuje typ instancji, które będą przechowywane na liście. Gdybyśmy potrzebowali listy łańcuchów, moglibyśmy to łatwo wyrazić, określając typ zmiennej jako `List<string>`. Podobnie gdybyśmy napisali `List<int>`, mielibyśmy gwarancję, że lista będzie przechowywać tylko wartości `int`, i uniknęlibyśmy kosztów opakowywania oraz odpakowywania, ponieważ CLR wygenerowałoby kod operujący bezpośrednio na wartościach `int`. Zauważmy też, że możemy utworzyć listę przechowującą zbiór wartości, które są polimorficznie zgodne z argumentem typu. Gdybyśmy na przykład mieli hierarchię typów, w której `A` jest klasą bazową, a `B` i `C` wywodzą się z `A`, lista `List<A>` mogłaby przechowywać elementy typu `A`, `B` i `C`.

Definicja typu `List<T>` przypomina `ArrayList`, ale używa `T` zamiast `object`, na przykład:

```
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>,
    IList, ICollection, IEnumerable
{
    public virtual void Add(T item);
    public virtual bool Contains(T item);
    public T[] ToArray();
    public T this[int index] { get; set; }
    // I tak dalej...
}
```

Teraz pierwotny program można zmienić w następujący sposób:

```
List<string> listOfStrings = new List<string>();
listOfStrings.Add("jeden");
listOfStrings.Add(2); // Tutaj kompilator zgłosi błąd
listOfStrings.Add("trzy");
```

Teraz kompilator nie pozwoli dodać nieprawidłowego elementu do listy `listOfStrings`, a instrukcja `foreach` na pewno nie spowoduje wyjątku `CastClassException` podczas pobierania elementów z listy. Oczywiście, z generykami wiąże się wiele dodatkowych zagadnień, których część zostanie omówiona poniżej. To samo dotyczy kolekcji, których szczegółowy opis znajduje się w rozdziale 6.

Konstruowanie: od typów otwartych do zamkniętych

W pierwszych akapitach niniejszego podrozdziału wyjaśniłem krótko pojęcie konkretyzowania typów i metod generycznych. Nie opisałem jednak możliwych sposobów konkretyzacji. Typ generyczny, który nie otrzymał żadnych argumentów odpowiadających parametrom typu, nazywamy **typem otwartym** — ponieważ jest „otwarty” na przyjmowanie kolejnych argumentów — natomiast typ, który otrzymał wszystkie swoje argumenty, nazywamy **typem skonstruowanym** (czasem również **typem zamkniętym**). Typ może znajdować się gdzieś pomiędzy otwartym a skonstruowanym, a wówczas nazywamy go **otwartym typem skonstruowanym**. Można tworzyć wyłącznie instancje typu skonstruowanego, który otrzymał wszystkie argumenty typu, a nie typu otwartego ani otwartego skonstruowanego.

Sprawdźmy, skąd się biorą otwarte typy skonstruowane. Nie wspomniano jeszcze, że klasa wywodząca się z typu generycznego może określać 1 albo więcej generycznych parametrów swojej klasy bazowej. Rozważmy poniższy typ generyczny o arności 3:

```
class MyBaseType<A, B, C> {}
```

Oczywiście, w celu skonkretyzowania klasy `MyBaseType` kod kliencki musiałby podać argumenty `A`, `B` i `C`, tworząc typ skonstruowany. Ale podklasa `MyBaseType` może określić dowolną liczbę argumentów typu, od 0 do 3, na przykład:

```
class MyDerivedType1<A, B, C> : MyBaseType<A, B, C> {}
class MyDerivedType2<A, B> : MyBaseType<A, B, int> {}
class MyDerivedType3<B> : MyBaseType<string, B, int> {}
class MyDerivedType4 : MyBaseType<string, object, int> {}
```

Klasa `MyDerivedType1` nie określa żadnych argumentów typu, więc jest typem otwartym. `MyDerivedType4` jest typem skonstruowanym, a dwa pozostałe są otwartymi typami skonstruowanymi. Mają przynajmniej jeden argument typu, ale potrzebują przynajmniej jeszcze jednego, zanim staną się w pełni skonstruowane.

Metody również określa się mianem otwartych lub zamkniętych, ale nie mogą one przyjmować postaci otwartej skonstruowanej. Metoda generyczna jest albo w pełni skonstruowana, albo nie; nie może być gdzieś pośrodku. Nie można na przykład przesłonić wirtualnej metody generycznej i podać argumentów generycznych.

Przechowywanie typów generycznych: pola statyczne i typy wewnętrzne

Dane należące do typu — w tym pola statyczne i typy wewnętrzne — są unikatowe dla każdej instancji typu generycznego. Przypuśćmy, że mamy następujący typ:

```
class Foo<T>
{
    public static int staticData;
}
```

Każda unikatowa instancja `Foo<T>` będzie miała własną kopię `staticData`. Innymi słowy, `Foo<int>.staticData` to zupełnie inna lokacja niż `Foo<string>.staticData` itd. Gdyby typ pola `staticData` był określony jako `T`, byłoby jasne dlaczego.

Podobnie każda konkretyzacja typu generycznego powoduje utworzenie unikatowych typów wewnętrznych:

```
class Foo60<T>
{
    enum MyEnum
    {
        One, Two, Three
    }
}
```

Okazuje się, że `Foo<int>.MyEnum` oraz `Foo<string>.MyEnum` to dwa zupełnie różne (i niezgodne) typy! Nie powinno to dziwić, a jednak bywa zaskakujące.

Kilka ostrzeżeń

Zanim zaczniemy stosować generyki w swoich aplikacjach, powinniśmy rozważyć ich wpływ na użyteczność i łatwość konserwacji programu. Oto kilka ogólnych spraw, o których warto pamiętać:

- Wielu użytkowników ma kłopoty ze składnią generyków. Czytelnicy, którzy zrozumieli powyższe przykłady bez ponownego czytania niektórych punktów, prawdopodobnie mieli już do czynienia z bliskimi kuzynami generyków, takimi jak szablony C# albo system typów generycznych w językach Eiffel lub Java. Większość osób musi poświęcić sporo czasu, zanim opanuje składnię generyków. Jest bardzo prawdopodobne, że jeszcze przez wiele lat znaczna część programistów .NET Framework nie będzie miała wiedzy o generykach ani nie będzie stosowała ich w aplikacjach produkcyjnych.
- Duży wpływ na czytelność programu ma nazewnictwo generycznych parametrów typu. Wiele typów używa tradycyjnej konwencji nazewniczej z pojedynczą literą, zaczynając od `T` i wykorzystując kolejne litery alfabetu na oznaczenie dodatkowych parametrów. Takiej konwencji użyto na przykład w klasie `List<T>`. Jeśli jednak parametr nie jest oczywisty, zastosowanie bardziej opisowej nazwy — na przykład `System.EventArgs<TEventArgs>` — może znacznie zwiększyć czytelność programu. Konwencja nakazuje rozpoczynać nazwę parametru typu od przedrostka `T`.

- Trudno pracuje się z typami i metodami generycznymi o wysokiej arności. Niektóre języki (na przykład C#) dedukują generyczne argumenty typu na podstawie zwykłych argumentów. Nieco ułatwia to zadanie programisty, ale — ogólnie rzecz biorąc — lepiej jest tego unikać. Bardzo łatwo jest zapomnieć, w jakiej kolejności występują typy, co powoduje problemy podczas pisania programu, ale jeszcze gorsze podczas jego konserwacji.

Trzeba też rozważyć kwestie wydajności. Widzieliśmy już, że kiedy generyki są używane w sytuacjach wymagających opakowywania i odpakowywania wartości, to mogą zwiększyć wydajność programu. Trzeba jednak ponieść pewne koszty związane z rozmiarem wygenerowanego kodu (tzn. zestawu roboczego) wynikającym z dużej liczby unikatowych konkretyzacji pojedynczego typu generycznego, zwłaszcza w przypadku argumentów typu wartościowego. Powodem jest to, że do pracy z różnymi argumentami potrzebny jest wyspecjalizowany kod. Zostanie to wyjaśnione dokładniej w opisie kompilatora JIT w rozdziale 3.

Ograniczenia

Dotychczas mówiliśmy o generykach bez wprowadzenia pojęcia ograniczeń. Ograniczenia są jednak niezwykle użyteczne, gdyż pozwalają dopuścić tylko stosowanie parametrów typu spełniających określone kryteria, a w definicjach typów i metod wykonywać takie operacje, które są (statycznie) bezpieczne typologicznie przy określonym ograniczeniu. Można przyjmować założenia dotyczące wartości argumentu typu, a środowisko uruchomieniowe zagwarantuje, że będą one spełnione. Gdyby nie ograniczenia, wszystkie składowe oznaczone parametrem typu trzeba by traktować jak obiekty, a więc przekazywać tylko do metod oznaczonych tym samym parametrem typu.

Ograniczanie typu

Istnieją dwa sposoby ograniczania parametrów typu. Pierwszym jest zdefiniowanie, że parametr typu musi być polimorficznie zgodny z określonym typem, czyli wywodzić się ze wspólnego typu bazowego (lub nim być) albo implementować określony interfejs. Parametr typu bez żadnych ograniczeń można uważać za niejawnie ograniczony do `System.Object`. Ograniczenie pozwala zamiast tego wybrać dowolną niezapieczętowaną klasę bazową lub interfejs. C# ułatwia to dzięki specjalnej składni:

```
class Foo<T> where T : IComparable<T>
{
    public void Bar(T x, T y)
    {
        int comparison = x.CompareTo(y);
        // ...
    }
}
```

Klauzula `where T : <typ>` określa typ ograniczenia, w tym przypadku deklarując, że `T` musi być typem implementującym interfejs `IComparable<T>`. Zauważmy, że w definicji typu możemy teraz wywoływać operacje `IComparable<T>` na instancjach oznaczonych jako `T`. Dotyczyłoby to również składowych klasy, gdybyśmy ograniczyli nasz typ nie do interfejsu, lecz do klasy bazowej. Tę samą składnię możemy stosować w metodach generycznych:

```
class Foo
{
    public void Bar<T>(T x, T y) where T : IComparable<T>
    {
        int comparison = x.CompareTo(y);
        // ...
    }
}
```

Przykłady te w istocie pokazują jeszcze jedną zaletę generyków — mianowicie to, że parametr typu jest w zasięgu samego ograniczenia, co pozwala definiować ograniczenie w kategoriach typu, który będzie znany dopiero podczas wykonywania programu. Innymi słowy, ograniczenie wspomina `T` w tym sensie, że wymaga, aby `T` implementował `IComparable<T>`. Może to być dezorientujące dla nowicjuszy, ale bywa niezwykle użyteczne. Można oczywiście używać również zwykłych typów bazowych i interfejsów:

```
class Foo<T> where T : IEnumerable {}
class Foo<T> where T : Exception {}
// I tak dalej...
```

Również te ograniczenia można stosować zarówno do typów, jak i metod generycznych.

Specjalne ograniczenia egzekwowane przez środowisko uruchomieniowe

Drugim sposobem ograniczania argumentów typu jest użycie jednego ze specjalnych ograniczeń oferowanych przez CLR. Istnieją trzy takie ograniczenia. Dwa wskazują, że argument musi być typu referencyjnego lub wartościowego (`class` i `struct`), i używają znanej już składni, z tym że słowo kluczowe `class` lub `struct` zastępuje nazwę typu:

```
class OnlyRefTypes<T> where T : class {}
class OnlyValTypes<T> where T : struct {}
```

Warto podkreślić, że zarówno ograniczenie `class`, jak i `struct` celowo wyklucza specjalny typ `System.Nullable<T>`. Powodem jest to, że w środowisku uruchomieniowym `Nullable` znajduje się gdzieś pomiędzy typem referencyjnym a wartościowym, a projektanci CLR uznali, że żaden z nich nie byłby odpowiedni. Zatem ograniczony w ten sposób parametr typu nie może przyjmować argumentu `Nullable<T>` podczas konstrukcji.

Można wreszcie ograniczyć parametr typu do takich argumentów, które mają konstruktor domyślny. Dzięki temu generyczny kod może tworzyć ich instancje z wykorzystaniem konstruktora domyślnego, na przykład:

```
class Foo
{
    public void Bar<T>() where T : new()
    {
        T t = new T(); // Jest to możliwe tylko ze względu na klauzulę T : new()
        // ...
    }
}
```

Emitowany kod IL używa wywołania API `Activator.CreateInstance`, aby wygenerować instancję `T`, wiążąc ją z konstruktorem domyślnym w czasie wykonywania programu. Wywołanie to jest również używane do konkretyzacji opartej na refleksji i na COM.

Wykorzystuje ono dynamiczne informacje dostępne w wewnętrznych strukturach danych CLR, aby automatycznie skonstruować nową instancję. Jest to zwykle niewidoczne dla programisty, choć jeśli konstruktor zgłosi wyjątek, to na stosie wywołań będzie można zobaczyć wywołanie `CreateInstance`.

Lektura uzupełniająca

Warto przeczytać wymienione niżej książki.

Książki poświęcone .NET Framework i CLR

Poniższe książki dotyczą .NET Framework i (lub) CLR i opisują dokładniej pojęcia zaprezentowane w niniejszym rozdziale.

Essential .NET, Volume 1. The Common Language Runtime, Don Box, Chris Sells, ISBN 0-201-73411-7, Addison-Wesley, 2003.

Common Language Infrastructure Annotated Standard, James S. Miller, Susann Ragsdale, ISBN 0-321-15493-2, Addison-Wesley, 2004.

Systemy typów i języki

Poniższe materiały powinien przeczytać każdy, kto chce dokładniej poznać tajniki systemów typów oraz projektowania języków programowania. Zawierają zarówno informacje podstawowe, jak i najbardziej zaawansowane tematy.

Structure and Interpretation of Computer Programs, wydanie drugie, Harold Abelson, Gerald Jay Sussman, ISBN 0-262-01153-0, MIT Press, 1996.

Types and Programming Languages, Benjamin C. Pierce, ISBN 0-262-16209-1, MIT Press, 2002.

Concepts of Programming Languages, wydanie siódme, Robert W. Sebesta, ISBN 0-321-33025-0, Addison-Wesley, 2005.

Essentials of Programming Languages, wydanie drugie, Daniel P. Friedman, Mitchell Wand, Christopher T. Haynes, ISBN 0-262-06217-8, MIT Press, 2001.

Concepts, Techniques, and Models of Computer Programming, Peter Van Roy, Seif Haridi, ISBN 0-262-22069-5, MIT Press, 2005.

Static Typing Where Possible, Dynamic Typing When Needed: The End of Cold War Between Programming Languages, Erik Meijer, Peter Drayton, *http://pico.vub.ac.be/~wdmeuter/RDL04/papers/Meijer.pdf*, 2005.

Generyki i pokrewne technologie

Dostępnych jest kilka książek, które szczegółowo omawiają generyki i polimorfizm parametryczny (na przykład szablony C++). Warto przeczytać którąś z nich, aby uświadomić sobie pełny potencjał generyków.

Professional .NET 2.0 Generics, Tod Golding, ISBN 0-764-55988-5, Wrox, 2005.

C++ Templates: The Complete Guide, David Vandevoorde, Nicolai M. Josuttis, ISBN 0-201-73484-2, Addison-Wesley, 2002.

Generative Programming: Methods, Tools, and Applications, Krzysztof Czarnecki, Ulrich Eisenecker, ISBN 0-201-30977-7, Addison-Wesley, 2000.

Konkretne języki

W tym rozdziale przedstawiono ideę CLR jako środowiska uruchomieniowego obsługującego różne języki i podano przykłady konkretnych języków. Choć niniejsza książka skupia się przede wszystkim na C#, wiele języków oferuje unikatowe funkcje i „widok na świat" poprzez swoją składnię oraz sposób łączenia typów danych. Wymienione niżej pozycje pozwalają zapoznać się z konkretnymi językami.

Professional C# 2005, Christian Nagel, Bill Evjen, Jay Glynn, Morgan Skinner, Karli Watson, Allen Jones, ISBN 0-764-57534-1, Wrox, 2005.

The C# Programming Language, Anders Hejlsberg, Scott Wiltamuth, Peter Golde, ISBN 0-321-15491-6, Addison-Wesley, 2003.

Professional VB 2005, Bill Evjen, Billy Hollis, Rockford Lhotka, Tim McCarthy, Rama Ramachandran, Bill Shelden, Kent Sharkey, ISBN 0-764-57536-8, Wrox, 2005.

The C Programming Language, 2nd Edition, Brian Kernighan, Dennis M. Ritchie, ISBN 0-131-10362-8, Prentice Hall, 1988.

The C++ Programming Language, Special 3rd Edition, Bjarne Stroustrup, ISBN 0-201-70073-5, Addison-Wesley, 2000.

The Design and Evolution of C++, Bjarne Stroustrup, ISBN 0-201-54330-3, Addison-Wesley, 1994.

Dive Into Python, Mark Pilgrim, ISBN 1-590-59356-1, Apress, 2004.

Practical Common Lisp, Peter Seibel, ISBN 1-590-59239-5, Apress, 2005.

Common LISP: The Language, Guy Steele, ISBN 1-555-58041-6, Digital Press, 1984.

The Scheme Programming Language, wydanie trzecie, R. Kent Dybvig, ISBN 0-262-54148-3, MIT Press, 2003.

Haskell: The Craft of Functional Programming, wydanie drugie, Simon Thompson, ISBN 0-201-34275-8, Addison-Wesley, 1999.