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 CLR. Księga eksperta

Autor: Kevin Burton
Tłumaczenie: Marcin Jędrysiak
ISBN: 83-7197-780-8
Tytuł oryginału: .NET Common Language Runtime Unleashed
Format: B5, stron: 868

„.NET CLR. Księga eksperta" to całościowe opracowanie poświęcone głównemu składnikowi platformy .NET Microsoftu, jakim jest Common Language Runtime. CLR pozwala na korzystanie z dowolnych języków programowania zgodnych z .NET udostępniając im wspólne usługi. Niezależnie od tego, czy uruchamiasz kod napisany w VB, C#, zarządzanym C++, JScripcie czy też w dowolnym innym obsługiwanym przez platformę .NET języku — wykorzystywane są funkcje i usługi CLR.

Tematy poruszone w książce obejmują m.in.:

- Typy .NET i Common Type System (CTS)
- Strukturę i układ metadanych podzespołu .NET
- Współpracę COM i COM+ z komponentami .NET
- Integrację z bibliotekami Win32 DLL poprzez P/Invoke
- Zarządzanie pamięcią i zasobami w CLR
- Zarządzanie i użycie wątków w środowisku .NET
- Tworzenie wydajnych aplikacji do obsługi sieci równorzędnych
- Tworzenie rozproszonych aplikacji
- Interakcje aplikacji poprzez zdarzenia i delegaty
- Obsługę błędów w .NET przy użyciu wyjątków
- Tworzenie i obsługę bezpiecznych aplikacji dzięki zabezpieczeniom .NET
- Dynamiczne uzyskiwanie informacji o typach poprzez refleksję
- Użycie narzędzi do globalizacji i lokalizacji
- Usuwanie błędów aplikacji .NET
- Profilowanie aplikacji .NET
- Omówienie składni C# i najważniejszych kwestii związanych z tym językiem
- Przegląd bibliotek struktury .NET
- Tworzenie własnego hosta CLR
- Porównanie CLR i JVM

Jeżeli chcesz zajmować się pisaniem aplikacji opartych na .NET, „.NET CLR. Księga eksperta" stanowić będzie cenne uzupełnienie Twojej wiedzy i pozwoli na lepsze zrozumienie zasad, na których oparta jest ta platforma.

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

Spis treści

O Autorze ... 15

Przedmowa ... 17

Wstęp ... 21

Część I Struktura .NET i podstawy CLR-a ... 27

Rozdział 1. Wprowadzenie do zarządzanego środowiska ... 29

Krótka historia CLR ... 30
Omówienie zarządzanego środowiska wykonawczego ... 30
Bezpieczeństwo typów ... 32
Pełne bezpieczeństwo ... 34
Obsługa wielu języków ... 35
Wydajność i skalowalność ... 36
Wdrażanie ... 37
Metadane i samoopisujący kod ... 38
Oczyszczanie pamięci i zarządzanie zasobami ... 39
Ewolucja COM ... 39
Obsługa wątków ... 40
Obsługa sieci ... 40
Zdarzenia ... 40
Konsekwentna obsługa błędów ... 40
Uzyskiwanie informacji o typie w czasie pracy aplikacji ... 41
Globalizacja ... 41
Usuwanie błędów i profilowanie ... 41
Podsumowanie ... 41

Rozdział 2. Common Language Runtime — język i system typów ... 43

Common Type System (CTS) ... 43
Typy wartości ... 44
Typy odwołań ... 48
Cechy Common Language Specification ... 56
Nazewnictwo ... 56
Elementy składowe typu ... 61
Właściwości ... 64
Zdarzenia ... 67
Tablice ... 68
Wyliczenia ... 68
Wyjątki ... 69
Własne atrybuty ... 70
Podsumowanie ... 72

Rozdział 3. Common Language Runtime — omówienie środowiska wykonawczego 73
Wprowadzenie do środowiska wykonawczego 74
Uruchamianie metody 75
Typy obsługiwane przez IL 76
Wykonawczy wątek kontroli 79
Sterowanie działaniem metody 80
Podsumowanie 89

Część II Komponenty CLR 91

Rozdział 4. Podzespół 93
Przedstawienie podzespołu .NET 94
Niezależność od języka programowania 95
Metadane — podstawa dla równoległego wdrażania i obsługi wielu wersji 96
Metadane a dokładniejsze zabezpieczenia 96
Większa produktywność programisty 97
Bezproblemowa interakcja z niezarządzanymi interfejsami API 97
Praca zdalna 97
Tworzenie międzynarodowych wersji aplikacji 97
Koszt metadanych 98
Ogólna struktura podzespołu 98
Szczegółowe omówienie struktury podzespołu 103
Niezarządzany interfejs API i dostęp do metadanych podzespołu 111
Fizyczna struktura podzespołu 115
Tablica Module (0x00) 119
Tablica TypeRef (0x01) 121
Tablica TypeDef (0x02) 123
Tablica MethodDef (0x06) 125
Tablica MemberRef (0x0A) 130
Tablica CustomAttribute (0x0C) 130
Tablica Assembly (0x20) 131
Tablica AssemblyRef (0x23) 131
Podsumowanie 132

Rozdział 5. Podstawy języka Intermediate Language 133
Miejsce IL w procesie programowania 133
Podręcznik ILDASM 134
Podstawy języka IL 135
Najczęściej używane polecenia IL 139
Polecenia IL do ładowania wartości 139
Polecenia IL do składowania 142
Polecenia IL do sterowania działaniem kodu 143
Polecenia operacji IL 147
Polecenia modelu obiektów IL 148
Podsumowanie 149

Rozdział 6. Publikowanie aplikacji 151
Problemy z instalacją klienta Windows..151
Wdrażanie i publikowanie aplikacji .NET 153
Identyfikacja kodu poprzez silną nazwę..153
Wdrażanie prywatnego podzespołu..157
Instalacja współdzielonego kodu..159
Wyszukiwanie podzespołów 162
Badanie plików zasad..163
Sprawdzenie wcześniejszych odwołań do podzespołu 165
Sprawdzenie GAC..165
Wyszukiwanie podzespołu..165
Administracja zasadami..166
Domyślna zasada wiązania..166
Zmiana zasady wiązania poprzez plik zasad aplikacji 166
Zmiana zasady wiązania poprzez plik zasad wydawcy..168
Zmiana zasady wiązania poprzez plik zasad komputera..169
Podsumowanie..170

Część III Usługi wykonawcze zapewniane przez CLR 171

Rozdział 7. Wykorzystywanie istniejącego kodu poprzez P/Invoke..... 173
Omówienie P/Invoke 173
Dlaczego współpraca z klasycznym kodem? 175
Niektóre inne metody współpracy z klasycznym kodem 176
Użycie P/Invoke 177
Deklaracja funkcji — atrybut DLLImport 177
Szeregowanie 180
Użycie zarządzanych rozszerzeń VC++..190
Wpływ P/Invoke na wydajność aplikacji 194
Podsumowanie..194

Rozdział 8. Użycie COM i COM+ w zarządzanym kodzie..................... 195
Runtime Callable Wrapper 196
Przykładowa aplikacja demonstrująca użycie RCW..198
Programowe generowanie podzespołu współpracy..204
Późne wiązanie z komponentem COM 206
Współpraca z elementami sterującymi ActiveX 207
Współpraca z SOAP 211
Podsumowanie..212

Rozdział 9. Użycie zarządzanego kodu jako komponentu COM i COM+ 215
Dlaczego warto tworzyć komponent .NET zachowujący się jak składnik COM?.............................215
Podstawy współpracy niezarządzanego i zarządzanego kodu..216
Eksportowanie metadanych do biblioteki typów 217
Rejestracja informacji w bibliotece typów..230
Demonstracja współpracy COM z klasą Arithmetic..232
Demonstracja współpracy COM z Excelem 232
Zaawansowane funkcje współpracy COM..235
Podsumowanie..240

Rozdział 10. Zarządzanie pamięcią i zasobami 243

Omówienie sposobu zarządzania zasobami w strukturze .NET 243
- Porównanie alokacji w strukturze .NET i C-Runtime 247
- Optymalizacja oczyszczania pamięci poprzez generacje 250
- Finalizacja 250
- Zarządzanie zasobami poprzez obiekt Disposable 256

Duże obiekty 261

WeakReference lub wyścig z mechanizmem oczyszczania pamięci 262

Podsumowanie 264

Rozdział 11. Obsługa wątków 265

Prezentacja wątków 266
- Tworzenie i uruchamianie wątków 266
- Witaj świecie 268
- Wielozakresowy delegat wątku 268
- Przekazywanie informacji do wątku 270

Inne operacje wykonywane na wątkach 273

AppDomain 275

Synchronizacja 278
- Monitor i Lock 280
- Zsynchronizowane zbiory 282
- Klasy synchronizacji wątków 282
- Oczekiwanie poprzez WaitHandle 282
- Klasa Interlocked 283
- Słowo kluczowe volatile 283
- Metoda Join 283
- Invoke, BeginInvoke oraz EndInvoke 283
- Pierwszy przykład synchronizacji — obiad filozofów 284
- Drugi przykład synchronizacji — koszyk z kolorowymi piłkami 285

Pula wątków 286
- QueueUserWorkItem 286
- RegisterWaitForSingleObject 289

Podsumowanie 291

Rozdział 12. Obsługa sieci 293

Krótka historia rozproszonych aplikacji 293

Tradycyjne gniazda 294

WinSock 295

Klasy sieciowe .NET 295
- Produktywne tworzenie oprogramowania sieciowego 296
- Warstwowy stos sieci 296
- Scenariusze dla aplikacji serwerowych i klienckich 296
- Elastyczność 296
- Zgodność ze standardami 296

Gniazdo w .NET 297
- Gniazdo UDP 297
- Gniazdo TCP 306
- Opcje Socket 311
- Użycie IOControl 316
- Asynchroniczne operacje na gniazdach 317

Klasy transportowe .NET 323
Klasa UDP 324
Klasa TCP 326
Klasy protokołów .NET 333
Obsługa HTTP, HTTPS i FILE 334
Programowanie asynchroniczne 336
Proste metody pobierania i przekazywania danych 340
Aplikacje Windows 341
Zarządzanie połączeniem 344
ServicePoint i ServicePointManager 346
Zarządzanie połączeniem i wydajność 346
Bezpieczeństwo sieciowe 350
Uwierzytelnianie 351
Zabezpieczenia dostępu do kodu 352
Dostęp do HTTP 352
Dostęp do gniazda 352
Rozwiązywanie nazw DNS 353
Podsumowanie 353

Rozdział 13. Tworzenie rozproszonych aplikacji przy użyciu .NET Remoting 355

Aplikacje rozproszone 356
Użycie .NET do rozpraszania aplikacji 356
Architektura Remoting 365
Zdalne obiekty 367
Prosty serwer czasu korzystający z .NET Remoting 369
Serwer czasu korzystający z hosta IIS 375
Tworzenie egzemplarza zdalnego obiektu przy użyciu Activator.GetObject 376
Tworzenie egzemplarza zdalnego obiektu przy użyciu RemotingServices.Connect 377
Tworzenie egzemplarza zdalnego obiektu przy użyciu RegisterWellKnownClientType 377
Użycie obiektu Client-Activated 377
Asynchroniczne wywołania do zdalnego obiektu 378
Generowanie i użycie opisu WSDL zdalnego obiektu 380
Użycie usługi WWW poprzez Remoting 383
Użycie usługi WWW poprzez interfejsy API Remoting 384
Użycie wywołań SOAP z poziomu COM 384
Konwersja komponentu COM+ do usługi WWW poprzez funkcje Remoting 387
Zaawansowane kwestie związane ze zdalną pracą 388
Implementacja własnego ujścia kanału do blokowania określonych adresów IP 388
Implementacja własnego ujścia kanału dziennika 393
Przedłużanie i kontrolowanie cyklu życia zdalnego obiektu 397
Wybranie najlepszej kombinacji kanału i formatera 398
Usuwanie błędów zdalnych aplikacji 401
Podsumowanie 402

Rozdział 14. Delegaty i zdarzenia 403

Dlaczego delegaty? 403
Podstawy używania delegatów 405
Porównywanie delegatów 409
Usuwanie delegatów 410

Klonowanie delegatów 412
Serializacja delegatów 413
Delegaty asynchroniczne 416
Rozwiązanie problemu obiadu filozofów przy wykorzystaniu delegatów 416
Zdarzenia i praca z delegatami 421
Podstawy użycia zdarzeń 421
Zdarzenia Microsoftu 424
Podsumowanie 427

Rozdział 15. Użycie zarządzanych wyjątków do skutecznej obsługi błędów 429

Obsługa błędów przy użyciu wyjątków 429
Trudności związane z unikaniem wyjątków 430
Właściwy sposób obsługi błędów 430
Omówienie wyjątków 430
Wyjątki C# 436
Wyjątki VC++ 443
Wyjątki VB 447
Podstawowe wyjątki VB 447
Zaawansowane wyjątki VB 449
Obsługa wyjątków w wielu językach 451
Wyjątki P/Invoke 453
Wyjątki COM 454
Zdalne wyjątki 455
Wyjątki wątku i asynchronicznego wywołania zwrotnego 457
Wyjątki asynchronicznego wywołania zwrotnego 458
Wyjątki wątków 459
Podsumowanie 460

Rozdział 16. Bezpieczeństwo .NET 465

Dwa różne, choć podobne, modele zabezpieczeń 465
Uprawnienia 466
Bezpieczeństwo typologiczne 468
Zasady zabezpieczeń 470
Główny obiekt 484
Uwierzytelnianie 484
Autoryzacja 484
Zabezpieczenia oparte na rolach 484
Zabezpieczenia dostępu kodu 491
Izolowane magazynowanie 499
Kryptografia w .NET 501
Podsumowanie 507

Rozdział 17. Refleksja 509

Użycie refleksji do uzyskania informacji o typie 510
Uzyskanie z podzespołu informacji o typie przy użyciu refleksji 510
Użycie klasy typu 518

Uzyskanie i wykorzystanie atrybutów przy użyciu refleksji 527
Dopasowanie metadanych przy użyciu własnych atrybutów 529
Użycie refleksji do serializacji typów 534
Serializacja używana do pracy zdalnej 535
Serializacja XML 540
Użycie refleksji do późnego wiązania obiektu 544
Dynamiczne generowanie kodu 545
Model Code Document Object (CodeDom) 545
Kompilacja kodu dla wielu języków 548
Mnożenie macierzy 550
Bezpośrednie generowanie kodu IL (Reflect.Emit) 558
Podsumowanie 559

Rozdział 18. Globalizacja i lokalizacja 561

Podstawy lokalizacji aplikacji 561
Droga do międzynarodowej aplikacji 564
Użycie klasy CultureInfo 566
Użycie klasy RegionInfo 572
Użycie zasobów w aplikacji 572
Dostęp do zasobów .NET 576
Wykorzystanie podzespołów satelitarnych 577
Przykład ręcznie tworzonej aplikacji 580
Przykład z użyciem Visual Studio .NET 582
Edytory IME 583
Podsumowanie 585

Rozdział 19. Usuwanie błędów aplikacji .NET 587

Dostarczenie informacji użytkownikowi 588
Polecenia Trace i Debug 589
Polecenie Assert 598
Dzienniki zdarzeń 598
Zapewnienie informacji poprzez stronę ASP.NET 602
Użycie klasy ErrorProvider 606
Weryfikacja wprowadzanych danych przy użyciu klasy ErrorProvider 606
Użycie ErrorProvider do weryfikacji danych wprowadzanych do bazy danych 607
Użycie debuggera 609
Ustawianie punktów kontrolnych 609
Krokowe uruchomienie kodu 610
Informacje o statusie 610
Przyłączanie do działającego procesu 612
Zdalne usuwanie błędów 614
Błędy ładowania podzespołów 616
Tworzenie własnego debuggera 617
Faza uruchomieniowa debuggera Cordbg 624
Podsumowanie 625

Rozdział 20. Profilowanie aplikacji .NET 627

Tradycyjne narzędzia do profilowania aplikacji .NET 627
Stworzenie obrazu użycia pamięci przy użyciu memsnap 628
Uzyskanie informacji o wykorzystaniu procesora przy użyciu pstat 629

Profilowanie wykorzystania pamięci przy użyciu vadump 630
Szczegółowy profil uruchomienia aplikacji .NET dostępny poprzez profile 633
Monitorowanie błędów stron przy użyciu pfmon 634
Monitorowanie synchronizacji procesów, pamięci i licznika wątków przy użyciu pview 635
Menedżer zadań 636
Monitorowanie ogólnosystemowych informacji przy użyciu perfmtr 637
Profilowanie przy użyciu exctrlst 638
Użycie Monitora wydajności i PerformanceCounters do profilowania aplikacji .NET 639
Programowy dostęp do liczników wydajności 652
Dodanie własnej kategorii liczników przy użyciu Server Explorera 657
Tworzenie usługi zapisującej we własnym liczniku PerformanceCounter 659
Monitorowanie własnego licznika wydajności 661
Dodanie licznika w sposób programowy 663
Użycie własnych interfejsów API do profilowania 663
General Code Profiler 664
Użycie Hot Spots Trackera do odnalezienia często wykorzystywanego kodu 671
Podsumowanie 676

Dodatki 677

Dodatek A Podstawy C# 679

Tworzenie programu w C# 679
Programowanie obiektowe w C# 682
Obiekty C# 683
Obiekty wartości 683
Obiekty typów odniesień 692
Typ wskaźnika 702
Podstawowe elementy programowania w C# 704
abstract 704
as 707
base 708
break 708
case 709
catch 709
const 710
continue 711
default 711
delegate 711
do 711
else 712
event 712
explicit 712
extern 713
finally 713
fixed 714
for 714
foreach 714
goto 717

if 717
implicit 717
in 717
interface 717
internal 719
is 720
lock 720
namespace 720
new 720
operator 720
out 721
override 721
params 721
private 722
protected 722
public 722
readonly 723
ref 723
return 724
sealed 724
stackalloc 724
static 725
switch 725
throw 726
try 726
typeof 726
unchecked 727
unsafe 727
using 727
virtual 728
volatile 729
while 729

Dodatek B Biblioteki klas struktury .NET 731

System.BitConverter 731
System.Buffer 732
System.Console 732
System.Convert 733
System.DateTime 734
System.Environment 735
System.Guid 737
System.IFormatProvider 737
System.Math 739
System.OperatingSystem 740
System.Random 740
System.TimeSpan 741
System.TimeZone 741
System.Version 742

System.Collections 742
System.Collections.ArrayList 742
System.Collections.BitArray..743
System.Collections.Hashtable..745
System.Collection.ICollection 745
System.Collections.IDictionary 746
System.Collections.Ienumerable..747
System.Collections.Ilist..747
System.Collections.Queue 748
System.Collections.ReadOnlyCollectionBase 749
System.Collections.SortedList 749
System.Collections.Stack...750
System.ComponentModel 755
System.ComponentModel.License...756
System.ComponentModel.TypeDescriptor...760
System.Configuration 761
System.Data..762
System.Diagnostics..762
System.Drawing 762
System.Drawing.Drawing2D 763
System.IO 765
System.Messaging..767
System.Text 769
System.Text.RegularExpression 770
System.Timers 771
System.Web..772
System.Windows.Forms...773
System.Xml 777
System.Xml.Xsl 779

Dodatek C Obsługa hostów Common Language Runtime 781

Dodanie własnego hosta do CLR 781

Dodatek D Porównanie Common Language Runtime i Java Virtual Machine.. 787

Historyczne tło wojny o zarządzanie kodem..787
Java kontra języki .NET 789
Java kontra C# 792
Typy zdefiniowane przez użytkownika...793
Wyjątki 797
Właściwości 799
Zdarzenia...801
Dostęp do bazy danych 803
Polimorfizm...806
Współpraca z klasycznym kodem 811
Atrybuty 816
Serializacja 817
Obsługa wersji...820
Uzyskanie informacji o typie w czasie pracy aplikacji 821
Analiza XML...823

Inne różnice między językami 824
Dostęp do Internetu 824
Graficzny interfejs użytkownika 825
Rozważania na temat firmy i pracowników 825

Dodatek E Bibliografia 827

Rozdział 2 827
Rozdział 3 827
Rozdział 4 828
Rozdział 5 828
Rozdział 6 829
Rozdział 7 829
Rozdział 8 829
Rozdział 9 830
Rozdział 10 830
Rozdział 11 831
Rozdział 12 832
Ogólne koncepcje sieci 832
Przykłady i porady dotyczące System.Net 832
Rozdział 13 833
Rozdział 14 834
Rozdział 16 835
Dodatek A 837
Dodatek B 837
Dodatek D 838

Skorowidz 841

Rozdział 3.
Common Language Runtime — omówienie środowiska wykonawczego

CLR jest mechanizmem, który pobiera polecenia IL, dokonuje ich przekształcenia do poleceń kodu maszynowego, a następnie wykonuje je. Nie oznacza to jednak, że CLR interpretuje polecenia. CLR po prostu tworzy środowisko, w którym możliwe jest wykonanie kodu IL. Aby wszystkie te operacje działały w sposób wydajny i przenośny, również mechanizm wykonawczy musi być wydajny i zapewniać przenośność. Wydajność to kluczowy czynnik; jeżeli kod nie działa wystarczająco szybko, to wszystkie inne funkcje tracą na znaczeniu.

Przenośność odgrywa ogromną rolę ze względu na ogromną liczbę urządzeń i procesorów, z którymi musi współpracować CLR. Przez długie lata Microsoft i Intel byli bliskimi partnerami. Microsoft wybrał linię procesorów Intela dla swojego oprogramowania, dzięki czemu możliwe było bezproblemowe tworzenie aplikacji związanych z obsługą różnych wersji architektur i poleceń procesora. Microsoft nie musiał tworzyć wersji dla procesora Motorola 68XXX, ponieważ nie był on obsługiwany. Ograniczenie zakresu obsługi procesorów stało się problemem, kiedy Win16 został zastąpiony przez Win32 (nie było interfejsów API Win16, ale użyłem tej nazwy dla API istniejących przed Win32). Programy wykorzystujące funkcje 32-bitowego procesora musiały zachować częściową zgodność ze starszymi API. Tworzenie takich aplikacji było poważnym przedsięwzięciem. Pojawienie się na horyzoncie Win64 oznacza, że Microsoft musi zaprzestać tworzenia wersji swojego oprogramowania do współpracy z każdym nowym procesorem, gdyż może zagrozić to problemami w funkcjonowaniu firmy. Microsoft próbuje penetrować rynek telefonów komórkowych, urządzeń przenośnych i tabletów. Wszystkie te urządzanie są sterowane przez ogromną liczbę procesorów o różnych architekturach. Oznacza to, iż nie jest możliwe już tworzenie oprogramowania powiązanego ściśle z procesorem.

Remedium na problemy z bazowym adresem i wielkością danych (Win32 kontra Win64) jest środowisko wykonawcze *Common Language Runtime*. CLR umożliwia także przenoszenie kodu na różne platformy. Poniższy rozdział omawia szczegółowo architekturę środowiska wykonawczego dla zarządzanych aplikacji. Informacje na temat konkretnych poleceń obsługiwanych przez CLR można odnaleźć w rozdziale 5., „Podstawy języka Intermediate Language”.

Wprowadzenie do środowiska wykonawczego

Przed pojawieniem się .NET plik wykonywalny (zwykle plik z rozszerzeniem *.exe*) był aplikacją. Innymi słowy, aplikacja znajdowała się w tym jednym pliku. Aby cały system działał bardziej efektywnie, aplikacja mogła wykorzystywać współdzielony kod znajdujący się najczęściej w pliku z rozszerzeniem *.dll*. Możliwe było użycie biblioteki importowej (plik zawierający odwołania funkcji do pliku DLL powiązanego z tą biblioteką) lub załadowanie biblioteki DLL w czasie pracy programu (przy użyciu `LoadLibrary`, `LoadLibraryEx` i `GetProcAddress`). W przypadku .NET jednostką wykonawczą i wdrożeniową jest podzespół. Uruchomienie aplikacji rozpoczyna się zwykle od podzespołu z rozszerzeniem *.exe*. Aplikacja może wykorzystywać współdzielony kod poprzez zaimportowanie podzespołu ze współdzielonym kodem. Odbywa się to przy użyciu określonego odwołania. Takie odwołanie można dodać w węźle *Add References* w Visual Studio .NET lub przy użyciu przełącznika wiersza poleceń `/r`. Ładowanie podzespołu przez aplikację odbywa się także poprzez metody `Assembly.Load` i `Assembly.LoadFrom`.

Przed przejściem do kolejnej części rozdziału należy zapoznać się z pewnymi definicjami obowiązującymi w .NET:

- Podzespół — *podzespół* jest główną jednostką wdrożeniową w strukturze .NET. W bibliotekach klas bazowych znajduje się klasa `Assembly`, która otacza fizyczny podzespół. Każde odniesienie w tej książce do tej klasy lub egzemplarza klasy będzie oznaczone właśnie nazwą `Assembly`. Klasa ta istnieje w przestrzeni nazw `System`. Podzespół może zawierać odwołania do innych podzespołów i modułów. Rozdział 4. zawiera bardziej szczegółowe informacje na temat podzespołów.
- Moduł — *moduł* to pojedynczy plik z wykonywalną zawartością. Podzespół może otaczać jeden lub więcej modułów; moduł nie może istnieć bez powiązanego z nim podzespołu. Podobnie jak w przypadku podzespołu, w bibliotece klas bazowych istnieje klasa `Module`, która zapewnia większość funkcji modułu. Każda wzmianka o `Module` odnosi się do tej klasy w bibliotece klas bazowych. Klasa `Module` istnieje w przestrzeni nazw `System`.
- *AppDomain* — domena aplikacji (może być nazwana lekką wersją procesu). Przed pojawieniem się .NET izolacja poszczególnych procesów odbywała się z wykorzystaniem systemu operacyjnego i komputera. Problemy z jednym procesem nie powodowały zawieszenia całego systemu. Dzięki ścisłej kontroli typów przez strukturę .NET dostępny jest mechanizm zapewniający ten sam poziom izolacji wewnątrz procesu. Ten mechanizm jest nazywany domeną aplikacji lub `AppDomain`. Podobnie jak w przypadku modułów i podzespołów, w bibliotece klas bazowych istnieje klasa `AppDomain`, która zapewnia większość funkcji domeny aplikacji. Ta klasa istnieje w przestrzeni nazw `System`. Każde odwołanie do tej klasy w tej książce będzie nazwane `AppDomain`.
- IL lub MSIL — IL to skrót oznaczający język *Intermediate Language*, a MSIL oznacza *Microsoft Intermediate Language*. Wszystkie podzespoły są napisane w języku IL; jest to zestaw poleceń reprezentujących kod aplikacji. IL jest nazwany językiem pośrednim, ponieważ polecenia nie są przekształcane do kodu macierzystego aż do momentu, w którym będzie to niezbędne. Kiedy konieczne jest wykonanie

kodu opisującego metodę, odbywa się jego kompilacja w locie do kodu macierzystego przy użyciu kompilatora JIT. Rozdział 5. zawiera informacje o poszczególnych poleceniach IL.

- JIT — skrót JIT (*Just-In-Time*) oznacza kompilację w locie. To określenie odnosi się do kompilatora wykonującego kod IL w razie potrzeby.

Po załadowaniu kodu rozpoczyna się jego wykonywanie. Od tego momentu występują znaczące różnice pomiędzy starymi i nowymi aplikacjami .NET. W przypadku nie zarządzanego kodu kompilator i konsolidator już zdążyły przekształcić źródło do kodu macierzystego, dzięki czemu wykonywanie poleceń może rozpocząć się natychmiast. Oczywiście oznacza to kompilowanie oddzielnych wersji kodu dla każdego macierzystego środowiska. W niektórych przypadkach konieczność dostarczenia i obsługi oddzielnej wersji aplikacji dla każdego możliwego środowiska byłaby niepożądana, dlatego klient otrzymuje jedynie kompatybilną wersję. Prowadzi to do zastosowania najniższego wspólnego mianownika, gdyż producenci chcą sprzedawać oprogramowanie działające w wielu różnych środowiskach. W chwili obecnej niewiele firm oferuje aplikacje obsługujące systemy z akceleratorami grafiki. Dzieje się tak, gdyż producenci musieliby nie tylko dostarczać inną wersję programu dla każdej karty akceleratora, ale także umożliwić pracę aplikacji w środowisku bez takiej karty. Sytuacja wygląda bardzo podobnie w przypadku obsługi innych urządzeń sprzętowych, takich jak pamięć podręczna dysku twardego, pamięć podręczna procesora, szybkie urządzenia sieciowe, systemy wieloprocesorowe, wyspecjalizowane systemy do przetwarzania obrazu itp. W wielu innych przypadkach kompilacja kodu powoduje uzyskanie zoptymalizowanego i wydajnego programu dla konkretnego systemu, lub nie zoptymalizowanego programu dla większej liczby systemów.

Jednym z pierwszych kroków wykonywanych przez CLR w celu uruchomienia programu jest sprawdzenie, czy dana metoda została przekształcona do postaci macierzystego kodu. Jeżeli tak się nie stało, wykonywana jest kompilacja w locie. Opóźnienie kompilacji metod zapewnia dwie ogromne korzyści. Po pierwsze, producent może opublikować jedną wersję oprogramowania, a CLR na konkretnym komputerze będzie odpowiedzialny za optymalizację kodu dla określonego środowiska sprzętowego. Po drugie, kompilator JIT może wykorzystać dostępne funkcje optymalizacji, dzięki czemu program będzie działał szybciej, niż analogiczny kod niezarządzany. Systemy zbudowane w oparciu o 64-bitowy procesor posiadają tryb kompatybilności, który umożliwia uruchamianie 32-bitowych programów bez konieczności modyfikacji. Ten tryb nie zapewnia jednak maksymalnej wydajności kodu. Jeżeli aplikacja została skompilowana do kodu IL, to może w pełni wykorzystać moc 64-bitowego procesora pod warunkiem, że mechanizm JIT może obsłużyć taki procesor.

Proces ładowania i kompilacji metody jest powtarzany aż do momentu skompilowania wszystkich metod lub zakończenia pracy aplikacji. Pozostała część rozdziału omawia środowisko, w którym CLR wykonuje metody klas.

Uruchamianie metody

CLR wymaga przedstawionych poniżej informacji o każdej metodzie. Wszystkie te dane są udostępniane CLR poprzez metadane podzespołu.

- Polecenia — CLR wymaga listy poleceń MSIL. Jak zostanie to pokazane w następnym rozdziale, każda metoda zawiera wskaźnik do zestawu poleceń. Ten wskaźnik stanowi część metadanych powiązanych z metodą.
- Podpis — każda metoda ma swoją sygnaturę, a CLR wymaga obecności takiego podpisu dla każdej metody. Podpis określa sposób wywoływania, typ zwrotny oraz liczbę i typy parametrów.
- Tablica obsługi wyjątków — nie istnieją konkretne polecenia IL do obsługi wyjątków, a jedynie dyrektywy. Zamiast takich poleceń podzespół dostarcza listę wyjątków, która zawiera typ wyjątku, adres przesunięcia do pierwszego polecenia po bloku `try` wyjątku oraz długość bloku `try`. W tabeli znajduje się także informacja o przesunięciu (offset) do kodu procedury obsługi, długości tego kodu oraz o żetonie opisującym klasę używaną do kapsułkowania wyjątku.
- Wielkość stosu oceny — te dane są dostępne poprzez metadane podzespołu; po uruchomieniu narzędzia *ILDasm* zwykle można zauważyć zapis `.maxstack x`, gdzie `x` jest wielkością tego stosu. Logiczna wielkość stosu reprezentuje maksymalną liczbę elementów, jakie należy umieścić na stosie. Fizyczna wielkość elementów i stosu jest ustalana przez CLR w czasie kompilacji metody w locie.
- Opis lokalnych tablic — każda metoda musi zadeklarować z góry liczbę wymaganych elementów do lokalnego składowania. Podobnie jak stos oceny, jest to logiczna tabela elementów, choć w tym przypadku deklarowany jest także typ każdego elementu. Oprócz tego w metadanych znajduje się znacznik wskazujący, czy na początku każdego wywołania metody lokalne zmienne powinny być zainicjalizowane do wartości zerowej.

Dzięki tym informacjom CLR może stworzyć abstrakcję, która normalnie byłaby ramką stosu macierzystego. Zwykle każdy procesor lub komputer tworzy ramkę stosu zawierającą argumenty (parametry) lub odwołania do argumentów metody. W ramce stosu umieszczane są także zwrotne zmienne w oparciu o konwencje wywoływania używane przez dany komputer lub procesor. Kolejność parametrów wejściowych i wyjściowych, a także sposób podania liczby parametrów, jest zależny od konkretnego komputera. Ponieważ wszystkie wymagane informacje są dostępne dla każdej metody, to CLR może ustalać rodzaj ramki stosu w czasie pracy aplikacji.

Wywołanie metody odbywa się w taki sposób, aby zapewnić CLR minimalną kontrolę nad sposobem wykonania metody i jej stanem. Kiedy CLR wywołuje lub uruchamia metodę, dostaje się ona pod kontrolę CLR przy użyciu tzw. wątku kontroli (*Thread of Control*).

Typy obsługiwane przez IL

Na poziomie IL obsługiwany jest prosty zestaw typów. Możliwa jest manipulacja tymi typami przy użyciu poleceń IL:

- `int8` — 8-bitowa wartość ze znakiem i uzupełnieniem do 2;
- `unsigned int8` (`byte`) — 8-bitowa wartość binarna bez znaku;

- `int16` (`short`) — 16-bitowa wartość ze znakiem i uzupełnieniem do 2;
- `unsigned int16` (`ushort`) — 16-bitowa wartość binarna bez znaku;
- `int32` (`int`) — 32-bitowa wartość ze znakiem i uzupełnieniem do 2;
- `unsigned int32` (`uint`) — 32-bitowa wartość binarna bez znaku;
- `int64` (`long`) — 64-bitowa wartość ze znakiem i uzupełnieniem do 2;
- `unsigned` (`ulong`) — 64-bitowa wartość binarna bez znaku;
- `float32` (`float`) — 32-bitowa wartość zmiennoprzecinkowa IEC 60559:1989;
- `float64` (`double`) — 64-bitowa wartość zmiennoprzecinkowa IEC 60559:1989;
- `native int` — wartość macierzystej wielkości ze znakiem i uzupełnieniem do 2;
- `native unsigned int` — wartość macierzystej wielkości bez znaku i z uzupełnieniem do 2;
- `F` — wartość zmiennoprzecinkowa macierzystej wielkości; ta zmienna jest wykorzystywana wewnętrznie przez CLR i nie jest widoczna dla użytkownika;
- `0` — odwołanie obiektu macierzystej wielkości do zarządzanej pamięci;
- `&` — zarządzany wskaźnik macierzystej wielkości.

Są to typy, które mogą być reprezentowane w pamięci, ale występują pewne ograniczenia przetwarzania tych elementów danych. Jak zostanie to omówione w kolejnej części rozdziału, CLR przetwarza te elementy na stosie oceny, który stanowi część danych stanu dla każdej metody. Stos stanu może przedstawiać element dowolnej wielkości, ale jedyne operacje, jakie mogą być wykonane na zdefiniowanych przez użytkownika typach wartości, to kopiowanie z i do pamięci oraz obliczanie adresów tych typów wartości. Wszystkie operacje związane z wartościami zmiennoprzecinkowymi wykorzystują wewnętrzne przedstawienie wartości zmiennoprzecinkowej, które jest zależne od sposobu implementacji (wartość `F`).

Pozostałe typy danych (poza omówioną wartością zmiennoprzecinkową) z macierzystą wielkością to `native int`, `native unsigned int`, odwołanie obiektu macierzystej wielkości (`0`) i zarządzany wskaźnik macierzystej wielkości (`&`). Te typy danych tworzą mechanizm pozwalający CLR na opóźnienie wyboru wielkości wartości. Dzięki temu mechanizmowi `native int` może mieć 64 bity na procesorze IA64 lub 32 bity na procesorze Pentium.

Dwa typy danych mogą wydawać się znajome; są to typy `0` i `&`. Zmienna `0` wskazuje zarządzany obiekt, ale jej użycie jest ograniczone do poleceń jednoznacznie wskazujących operacje na zarządzanym typie lub poleceń, których metadane wskazują możliwość użycia odwołań do zarządzanego obiektu. Typ `0` wskazuje „poza” obiekt lub obiekt jako całość. Typ `&` również jest odwołaniem do zarządzanego obiektu, ale może być użyty w celu wskazania pola obiektu lub elementu tablicy. Typy `0` i `&` są śledzone przez CLR i mogą się zmieniać w zależności od rezultatów oczyszczania pamięci.

Typ macierzystej wielkości może być użyty także dla niezarządzanych wskaźników. Choć takie wskaźniki mają zdefiniowany typ w metadanych, to w kodzie IL są reprezentowane jako `native unsigned int`. Daje to CLR możliwość przydzielania niezarzą-

dzanego wskaźnika do większej przestrzeni adresowej na procesorach ją obsługujących bez niepotrzebnego blokowania pamięci w przypadku procesorów nieobsługujących tak dużej przestrzeni adresowej.

Niektóre polecenia IL wymagają obecności adresu na stosie, na przykład:

- `calli` — …, `argument1, argument2, …, argumentn, funkcja` → … `wartość_zwrotna`
- `cpblk` — …, `adres_docelowy, adres_źródłowy, wielkość` →…
- `initblk` — …, `adres, wartość, wielkość` → …
- `ldind.*` — …, `adres`→ …, `wartość`
- `stind.*` — …, `adres, wartość`→ …

Użycie macierzystego typu gwarantuje przenośność operacji związanych z tym typem. Jeżeli adres został podany w formie 64-bitowej liczby całkowitej, to może on być przenośny, W tym celu należy jednak upewnić się, czy wartość została prawidłowo przekonwertowana do postaci adresu. Jeżeli adres jest podany jako wartość 32-bitowa lub mniejszy, to nigdy nie będzie on przenośny, nawet jeżeli będzie działał poprawnie na większości komputerów 32-bitowych. W wielu przypadkach jest to kwestia związana z generatorem IL lub kompilatorem i nie należy się tym martwić. Należy jednak pamiętać, iż można uniemożliwić przenośność kodu poprzez niewłaściwe zastosowanie tych poleceń.

Krótkie wartości numeryczne (których wartość jest mniejsza niż 4 bajty) są rozszerzane do 4 bajtów po załadowaniu (a konkretnie po skopiowaniu z pamięci na stos) oraz zawężane w czasie składowania (czyli przy kopiowaniu ze stosu do pamięci). Wszystkie operacje związane z krótkimi wartościami numerycznymi tak naprawdę są wykonywane na 4-bajtowych wartościach. Do obsługi krótkich wartości służą specjalne polecenia IL:

- ładowanie i składowanie z i do pamięci — `ldelem`, `ldind`, `stind` i `stelem`;
- konwersja danych — `conv` i `conv.ofv`;
- tworzenie tablicy — `newarr`.

IL obsługuje jedynie operacje ze znakiem. Operacje ze znakiem i bez znaku różnią się sposobem interpretacji wartości. W przypadku gdy sposób interpretacji nie ma znaczenia, dana operacja występuje w wersji ze znakiem i bez znaku. Dla przykładu, polecenia `cgt` i `cgt.un` porównują dwie wartości w celu wybrania większej z nich.

Macierzyste położenie wartości

CLR wprowadza koncepcję macierzystego położenia obiektu, co pozwala na śledzenie obiektów. *Macierzyste położenie obiektu* jest miejscem przechowywania wartości obiektu i musi zapewniać mechanizm ustalania typu obiektu dla kompilatora JIT. Obiekt przekazywany jako odwołanie musi posiadać swoje macierzyste położenie, ponieważ przekazywany jest również jego adres. Dwa typy danych nie mają takiego położenia i nie mogą być przekazane jako odwołanie; są to stałe oraz wartości pośrednie na stosie oceny dla poleceń IL lub wartości zwrotnych metod. CLR obsługuje następujące położenia obiektów:

- Przychodzący argument — polecenia `ldarg` i `ldarga` sprawdzają adres położenia argumentu. Podpis metody decyduje o jej typie.
- Lokalna zmienna — polecenia `idloca` i `ldloc` ustalają adres lokalnej zmiennej. Lokalny stos oceny ustala typ takiej zmiennej jako część metadanych.
- Pole (egzemplarza lub statyczne) — dla pola egzemplarza należy użyć polecenia `ldflda`, natomiast dla statycznego pola `ldsflda`. Oba polecenia umożliwiają ustalenie adresu pola. Metadane powiązane z interfejsem klasy lub modułem decydują o typie pola.
- Element tablicy — polecenie `ldelema` pozwala na uzyskanie adresu elementu tablicy. Typ tablicy decyduje o typie elementu.

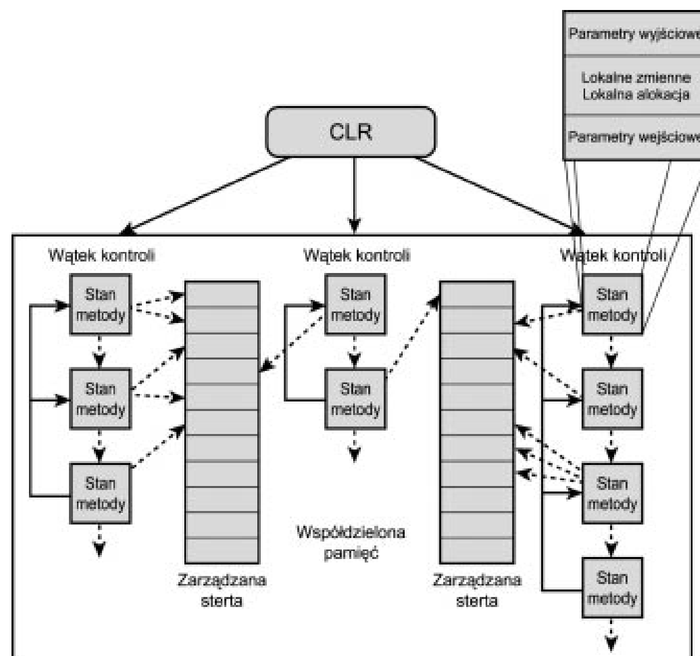
Wykonawczy wątek kontroli

Wątek kontroli w CLR (*Thread of Control*) niekoniecznie musi odpowiadać macierzystym wątkom systemu operacyjnego. Klasa `System.Threading.Thread` z biblioteki klas bazowych zapewnia logiczne kapsułkowanie dla wątku kontroli.

Więcej informacji na temat wątków można znaleźć w rozdziale 11., „Obsługa wątków".

Przy każdym wywołaniu metody konieczne jest wykonanie procedury sprawdzającej, czy dana metoda została skompilowana w locie. Rysunek 3.1 przedstawia schemat stanu CLR. Nie jest to dokładny schemat, ponieważ pokazane jest proste powiązanie między dwoma metodami. Rysunek byłby znacznie bardziej skomplikowany w przypadku użycia instrukcji skoku lub obsługi wyjątków.

Rysunek 3.1.
Stan komputera z perspektywy CLR



Zarządzana sterta na rysunku 3.1 odnosi się do pamięci zarządzanej przez CLR. Szczegółowe informacje o zarządzanych stosach i funkcjach oczyszczania pamięci zostaną przedstawione w rozdziale 10., „Zarządzanie pamięcią i zasobami". Przy każdym wywołaniu pamięci tworzony jest stan metody, który obejmuje następujące elementy:

- Wskaźnik polecenia — wskazuje kolejne polecenie IL, które aktualna metoda ma wykonać.
- Stos oceny — jest to stos podany przez dyrektywę `.maxstack`. Kompilator ustala w czasie kompilacji liczbę gniazd wymaganych na stosie.
- Tablica lokalnych zmiennych — jest to ta sama tablica, która została zadeklarowana i prawdopodobnie zainicjalizowana w metadanych. Dostęp do tych zmiennych z poziomu IL odbywa się poprzez indeks tej tablicy. Odwołania do tej tablicy w kodzie IL można odnaleźć, na przykład, w poleceniu `ldloc.0` (załadowanie na stos zmiennej `0` z tablicy lokalnych zmiennych) lub `stloc.1` (umieszczenie wartości na stosie w lokalnej zmiennej `1`).
- Tablica argumentów — jest to tablica argumentów przekazywanych do metody. Manipulacja tymi argumentami odbywa się przy użyciu poleceń IL, na przykład `ldarg.0` (załadowanie argumentu `0` na stos) lub `starg.1` (zapisanie wartości do argumentu `1` na stosie).
- Uchwyt `MethodInfo` — ten zbiór informacji jest dostępny w metadanych podzespołu. Uchwyt wskazuje informacje o podpisie metody (liczba i typ argumentów oraz typy zwrotne), typach lokalnych zmiennych i o danych wyjątków.
- Pula lokalnej pamięci — IL posiada instrukcje umożliwiające alokację pamięci adresowalnej przez aktualną metodę (`localloc`). Po powrocie metody pamięć jest odzyskiwana.
- Uchwyt zwrotnego stanu — uchwyt ten jest używany do zwrócenia stanu po zakończeniu metody.
- Deskryptor zabezpieczeń — CLR wykorzystuje ten deskryptor do zapisu ustawień zabezpieczeń zmienianych w sposób programowy lub poprzez własne atrybuty.

Stos oceny nie musi równać się fizycznej reprezentacji. Za obsługę fizycznej reprezentacji stosu oceny odpowiedzialny jest CLR oraz docelowy procesor. Z logicznego punktu widzenia stos oceny jest zbudowany z gniazd mogących przechowywać dowolny typ danych. Wielkość takiego stosu nie może być nieustalona. Nie jest możliwe, na przykład, użycie kodu, który powoduje umieszczenie na stosie zmiennej nieskończoną lub nieustaloną liczbę razy.

Polecenia związane z operacjami na stosie oceny nie mają typu; na przykład polecenie `add` dodaje dwie liczby, a `mul` powoduje ich pomnożenie. CLR śledzi typy danych i używa je w czasie kompilacji metody w locie.

Sterowanie działaniem metody

CLR zapewnia obsługę bogatego zestawu poleceń do kontroli pracy metody:

- Warunkowe lub bezwarunkowe rozgałęzienie — sterowanie może być przekazywane w dowolne miejsce metody pod warunkiem, że nie przekracza granicy chronionego regionu. *Chroniony region* jest definiowany w metadanych jako region powiązany z procedurą obsługi zdarzenia. W języku C# region ten jest znany jako blok `try`, a powiązany blok `catch` jest *procedurą obsługi*. CLR pozwala na obsługę wielu różnych procedur obsługi zdarzeń, które zostaną opisane w dalszej części książki. Warto pamiętać, że warunkowe lub bezwarunkowe rozgałęzienie nie może kierować do miejsca docelowego poza granicami wyjątku. W przypadku C# nie jest możliwe opuszczenie bloku `try` lub `catch`. Nie jest to ograniczenie języka C#, a raczej kodu IL, dla którego C# służy jako generator kodu.
- Wywołanie metody — wiele poleceń umożliwia metodom wywołanie innych metod, co powoduje utworzenie nowych stanów w sposób opisany wcześniej.
- Przyrostek wywołania — jest to specjalny przedrostek umieszczony bezpośrednio przed wywołaniem metody; nakazuje wywołującej metodzie odrzucenie swojej ramki stosu przed wywołaniem metody. Powoduje to powrót wywoływanej metody do miejsca, do którego powróciłaby metoda wywołująca.
- Powrót — jest to po prostu powrót z metody.
- Skok metody — jest to optymalizacja wywołania z przyrostkiem, które przekazuje argumenty i sterowanie metodą do innej metody o tym samym podpisie, co powoduje usunięcie aktualnej metody. Poniższy fragment kodu przedstawia prosty skok metody:

```
// Funkcja A
.method static public void A()
{
// Rezultat A
  ret
}
// Funkcja B
.method static public void B()
{
  jmp void A()
// Rezultat B
  ret
}
```

  Polecenia rozpoczynające się od komentarza `// Rezultat B` nie zostaną nigdy wykonane, ponieważ powrót z `B` jest zastąpiony przez powrót z `A`.
- Wyjątek — obejmuje zestaw poleceń generujących wyjątek i przekazujących sterowanie poza chroniony region.

CLR wymusza zastosowanie wielu reguł w czasie przekazywania sterowania wewnątrz metody. Po pierwsze, nie jest możliwe przekazanie sterowania do procedury obsługi wyjątku (blok `catch`, `finally` itd.), chyba że jest to wynik wyjątku. To ograniczenie jest związane z regułą mówiącą, że miejsce docelowe rozgałęzienia nie może przekraczać chronionego regionu. Po drugie, jeżeli wykonywany jest kod procedury obsługi dla chronionego regionu, to nie jest możliwe przekazanie sterowania poza tę procedurę, używając innych poleceń niż ograniczony zestaw poleceń obsługi wyjątków (`leave`, `end.finally`, `end.filter` i `end.catch`). W przypadku próby powrotu z metody z wew-

nątrz bloku `finally` w C# kompilator wygeneruje błąd. Nie jest to ograniczenie języka C#, ale samego kodu IL. Po trzecie, każde gniazdo w stosie oceny musi zachować swój typ przez cały cykl życia stosu (czyli cykl życia metody). Innymi słowy, nie jest możliwa zmiana typu gniazda (zmiennej) na stosie oceny. Zwykle nie stanowi to problemu, ponieważ taki stos nie jest w żaden sposób dostępny dla użytkownika. Po czwarte, wszystkie ścieżki programu muszą zakończyć się poleceniem powrotu (`ret`), skokiem metody (`jmp`), wywołaniem z przyrostkiem (`tail.*`) lub zgłoszeniem wyjątku.

Wywołanie metody

CLR może wywoływać metody na trzy różne sposoby. Poszczególne wywołania różnią się jedynie sposobem podania deskryptora miejsca wywołania. Taki deskryptor dostarcza CLR i mechanizmowi JIT informacji niezbędnych do wygenerowania macierzystego wywołania metody, udostępnia metodzie właściwe argumenty, a także umożliwia powrót metody.

Polecenie `calli` jest najprostszym wywołaniem metody i jest używane w przypadku, gdy docelowy adres jest obliczany w czasie pracy aplikacji. To polecenia pobiera dodatkowy argument w postaci wskaźnika funkcji, który istnieje w miejscu wywołania jako argument. Wskaźnik ten jest obliczany przy użyciu poleceń `ldftn` lub `ldvirftn`. Miejsce wywołania jest podane w metadanych w tablicy `StandAloneSig` (por. rozdział 4.).

Polecenie `call` jest używane, jeżeli adres funkcji jest znany w czasie kompilacji, na przykład w przypadku statycznej metody. Deskryptor miejsca wywołania wywodzi się z żetonu `MethodDef` lub `MethodRef`, który stanowi część polecenia. Opis tych dwóch tabel znajduje się w rozdziale 4.

Polecenie `callvirt` wywołuje metodę konkretnego egzemplarza obiektu. Również to polecenie obejmuje żeton `MethodDef` lub `MethodRef`, ale wymagany jest dodatkowy argument, który odnosi się do konkretnego egzemplarza, dla którego wywoływana jest metoda.

Konwencja wywoływania metod

CLR wykorzystuje prostą, jednakową konwencję wywoływania metod w kodzie IL. Jeżeli wywoływana metoda jest metodą egzemplarza, to na stosie najpierw umieszczane jest odwołanie do egzemplarza obiektu, a następnie poszczególne argumenty metody w kolejności od lewej do prawej. W wyniku tej operacji ze stosu pobierany jest najpierw wskaźnik `this`, po którym następuje wywoływana metoda i poszczególne argumenty w kolejności od numeru 0 do n. Jeżeli wywoływana jest statyczna metoda, to nie istnieje powiązany wskaźnik egzemplarza, a stos zawiera tylko jeden argument. W przypadku polecenia `calli` na stos trafiają najpierw argumenty w kolejności od lewej do prawej, a następnie wskaźnik funkcji. CLR i JIT muszą dokonać translacji tych danych do najwydajniejszej macierzystej konwencji wywołania.

Przekazywanie parametrów metody

CLR obsługuje trzy mechanizmy przekazywania parametrów:

- Według wartości — wartość obiektu jest umieszczana na stosie. W przypadku wbudowanych typów, takich jak wartości całkowite i zmiennoprzecinkowe, oznacza to po prostu ich położenie na stosie. Jeżeli jest to obiekt, to na stos trafia odwołanie typu `O`. W przypadku zarządzanych i niezarządzanych wskaźników na stosie umieszczany jest adres. Trochę inaczej wygląda obsługa własnych typów wartości, gdyż wartość poprzedzająca wywołanie metody jest umieszczana na stosie oceny na dwa sposoby. Po pierwsze, wartość może być bezpośrednio położona na stosie przy użyciu polecenia `ldarg`, `ldloc`, `ldfld` lub `ldsfld`. Po drugie, możliwe jest obliczenie wartości, która trafi na stos z wykorzystaniem polecenia `ldobj`.
- Według odwołania — w przypadku tej konwencji do metody przekazywany jest adres parametru, a nie jego wartość. Pozwala to na modyfikację parametru przez metodę. Tylko wartości posiadające własne domy mogą być przekazywane według odwołania, ponieważ przekazywany jest adres ich domu. W przypadku kodu, którego bezpieczeństwo typologiczne może być zweryfikowane, parametry powinny być przekazywane i wykorzystywane tylko przy użyciu poleceń `ldind.*` i `stind.*`.
- Odwołanie typu — przypomina normalne odwołanie, ale oprócz odwołania danych przesyłany jest również statyczny typ danych. Pozwala to IL na obsługę takich języków jak VB, które mogą posiadać metody nie będące statycznie ograniczonymi do akceptowanych typów danych, ale wymagają przekazania nie spakowanej wartości według odwołania. Aby wywołać taką metodę, należy skopiować istniejące odwołanie typu lub użyć polecenia `mkrefany` w celu utworzenia typu odwołania danych. W przypadku obecności takiego typu adres jest obliczany przez polecenie `refanyval`. Parametr odwołania typu musi odnosić się do danych posiadających dom.

Obsługa wyjątków

CLR umożliwia obsługę wyjątkowych warunków lub błędów dzięki obiektom wyjątków i chronionym blokom kodu. Blok `try` jest przykładem chronionego bloku w C#. CLR obsługuje cztery różne rodzaje procedur obsługi wyjątków:

- `Finally` — blok ten będzie wykonany po zakończeniu metody niezależnie od sposobu wyjścia — może to być normalna ścieżka wykonania (bezpośrednio lub przy użyciu `ret`) lub nieobsługiwany wyjątek.
- `Fault` — ten blok będzie wykonany w przypadku wystąpienia wyjątku, ale już nie, kiedy metoda zakończy pracę w normalny sposób.
- Filtrowanie według typu — ten blok kodu zostanie wykonany, jeżeli wykryto zgodność typu wyjątku dla tego kodu oraz zgłoszonego wyjątku. Odpowiada to blokowi `catch` w C#.
- Filtrowanie przez użytkownika — obsługa wyjątku przez blok kodu jest zależna od zestawu poleceń IL, które mogą nakazać zignorowanie wyjątku albo obsługę wyjątku przez aktualną lub następną procedurę obsługi. Przypomina to procedurę obsługi `_except` w *Structured Exception Handling* (SEH).

Nie każdy język generujący kod IL wykorzystuje wszystkie sposoby obsługi wyjątków, na przykład C# nie obsługuje procedur obsługi wyjątków filtrowanych przez użytkownika, natomiast VB pozwala na to.

Kiedy wystąpi wyjątek, CLR przeszukuje tablicę obsługi wyjątków stanowiącą część metadanych każdej metody. Tablica ta definiuje chroniony region oraz blok kodu obsługujący konkretny wyjątek. W tablicy znajduje się także informacja o przesunięciu od początku metody oraz o wielkości bloku kodu. Każdy wiersz w tablicy obsługi wyjątków podaje chroniony region (przesunięcie i wielkość), typ procedury obsługi (jeden z czterech wcześniej omówionych typów) oraz blok procedury obsługi (przesunięcie i wielkość). Oprócz tego wiersz procedury filtrowania według typu zawiera informacje o docelowym typie wyjątku. Procedura obsługi z filtrowaniem przez użytkownika zawiera etykietę rozpoczynającą blok kodu, który zostanie wykonany w celu ustalenia, czy blok procedury ma być wykonany wraz ze specyfikacją regionu procedury. Wydruk 3.1 przedstawia fragment pseudokodu C# do obsługi wyjątków.

Wydruk 3.1. *Pseudokod obsługi wyjątków w C#*

```
try
{
  // Chroniony blok
  . . .
}
catch(ExceptionOne e)
{
  // Procedura obsługi z filtrowaniem według typu
  . . .
}
finally
{
  // Procedura obsługi finally
  . . .
}
```

Dla kodu z wydruku 3.1 istnieją dwa wiersze w tablicy obsługi wyjątków; jeden z nich jest wymagany przez procedurę obsługi z filtrowaniem według typu, a drugi dla bloku `finally`. Oba wiersze odnoszą się do tego samego chronionego bloku kodu, a mianowicie do bloku `try`.

Wydruk 3.2 zawiera jeszcze jeden przykład schematu obsługi wyjątków, ale tym razem w Visual Basicu.

Wydruk 3.2. *Pseudokod obsługi wyjątków w VB*

```
Try
  'Chroniony region kodu
  . . .
Catch e As ExceptionOne When i = 0
  'Procedura obsługi wyjątku z filtrowaniem przez użytkownika
  . . .
Catch e As ExceptionTwo
Finally
  'Procedura obsługi finally
```

```
. . .
End Try
```

Pseudokod z wydruku 3.2 powoduje utworzenie trzech wierszy w tablicy obsługi wyjątków. Pierwszy blok `Catch` jest procedurą obsługi z filtrowaniem przez użytkownika — jest ona przekształcona do pierwszego wiersza tabeli. Drugi blok `Catch` to procedura obsługi z filtrowaniem typu; jest ona identyczna jak analogiczna procedura w języku C#. Trzeci i ostatni wiersz w tabeli to procedura obsługi `Finally`.

Po wygenerowaniu wyjątku CLR wyszukuje pierwszą zgodną procedurę w tablicy obsługi wyjątków. Zgodność oznacza, że wyjątek został zgłoszony w momencie, gdy zarządzany kod znajdował się w chronionym bloku podanym przez konkretny wiersz. Oprócz tego zgodna procedura obsługi musi „chcieć" obsłużyć wyjątek (filtr użytkownika ma wartość `true`, typ jest zgodny ze zgłoszonym wyjątkiem, kod opuszcza metodę, tak jak w `finally` itd.). Pierwszy wiersz w tablicy obsługi wyjątków, jaki zostanie dopasowany przez CLR, staje się wybraną procedurą obsługi. Jeżeli takiej procedury dla danej metody nie znaleziono, to następuje badanie metody wywołującej aktualną metodę. Takie wyszukiwanie jest kontynuowane aż do momentu odnalezienia właściwej procedury lub osiągnięcia szczytu stosu; w takim przypadku wyjątek jest deklarowany jako nieobsługiwany.

Sterowanie przepływem wyjątków

Przepływem wyjątków poprzez chronione regiony i powiązane procedury obsługi zarządza wiele reguł. Zastosowanie tych reguł jest wymuszane przez kompilator (generator kodu IL) lub CLR, ponieważ metoda jest kompilowana w locie. Należy pamiętać, że chroniony region i powiązana procedura obsługi są umieszczane na szczycie istniejącego bloku kodu IL. Ustalenie znajdującej się w metadanych struktury obsługi wyjątków nie jest możliwe na poziomie kodu IL. Poniżej przedstawiono zestaw reguł używanych przez CLR w czasie przekazywania sterowania z i do bloków obsługi wyjątków:

- Sterowanie może być przekazane do bloku obsługi wyjątków tylko poprzez mechanizm wyjątków.
- Istnieją dwie metody przekazania sterowania do chronionego regionu (blok `try`). Po pierwsze, proces wykonawczy może przejść do pierwszego polecenia chronionego regionu. Po drugie, polecenie `leave` w procedurze z filtrowaniem typu może zdefiniować przesunięcie (offset) do dowolnego polecenia w chronionym bloku, przy czym nie musi to być pierwsze polecenie.
- W momencie przejścia do chronionego regionu stos oceny musi być pusty. Oznacza to, że żaden element nie może umieścić wartości na stosie przed wejściem do chronionego regionu.
- Wyjście z chronionego regionu poprzez powiązane procedury obsługi jest rygorystycznie kontrolowane.

  Opuszczenie dowolnego bloku obsługi wyjątków jest możliwe poprzez zgłoszenie kolejnego wyjątku.

  Kiedy sterowanie znajduje się w chronionym regionie lub bloku procedury obsługi (a nie w `finally` lub `fault`), możliwe jest użycie polecenia `leave`, co przypomina

bezwarunkowe rozgałęzienie. Skutkiem ubocznym jest jednak wyczyszczenie stosu oceny. Miejscem docelowym polecenia `leave` może być dowolne polecenie w chronionym regionie.

Blok obsługi z filtrowaniem przez użytkownika musi kończyć się poleceniem `endfilter`. To polecenie pobiera pojedynczy argument ze stosu oceny w celu ustalenia sposobu kontynuacji procedury obsługi wyjątku.

Blok `finally` lub `fault` kończy się poleceniem `endfinally`. To polecenie powoduje wyczyszczenie stosu oceny i powrót do otaczającej metody.

Sterowanie może być przekazane poza blok procedury z filtrowaniem typu poprzez ponowne zgłoszenie wyjątku. Jest to specjalny przypadek, w którym aktualnie obsługiwany wyjątek jest zgłaszany po raz wtóry.

- Żaden blok procedury obsługi ani chroniony region nie może wykonać polecenia `ret` w celu powrotu do otaczającej metody.
- Bloki procedury obsługi wyjątków nie mogą wykonywać lokalnej alokacji. Polecenie `localloc` nie jest dostępne w żadnej procedurze obsługi.

Typy wyjątków

W dokumentacji można odnaleźć informacje mówiące o wyjątkach, jakie mogą być wygenerowane przez pojedyncze polecenie. CLR potrafi wygenerować następujące wyjątki jako efekt wykonania konkretnych poleceń IL:

- `ArithmeticException`;
- `DivideByZeroException`;
- `ExcecutionEngineException`;
- `InvalidAddressException`;
- `OverflowException`;
- `SecurityException`;
- `StackOverflowException`.

Następujące wyjątki są generowane w wyniku błędów i niezgodności modelu obiektów:

- `TypeLoadException`;
- `IndexOutOfRangeException`;
- `InvalidAddressException`;
- `InvalidCastException`;
- `MissingFieldException`;
- `MissingMethodException`;
- `NullReferenceException`;
- `OutOfMemoryException`;

- `SecurityException`;
- `StackOverflowException`.

Wyjątek `ExecutionEngineException` może być zgłoszony przez dowolne polecenie i informuje o napotkaniu przez CLR nieoczekiwanej niezgodności. Ten wyjątek nie zostanie nigdy zgłoszony, jeżeli zweryfikowano kod.

Wiele wyjątków jest zgłaszanych w efekcie nieudanego odwzorowania, na przykład, jeżeli nie odnaleziono metody lub metoda ma niewłaściwy podpis. Poniżej przedstawiono listę takich wyjątków:

- `BadImageFormatException`;
- `EntryPointNotFoundException`;
- `MissingFieldException`;
- `MissingMemberException`;
- `MissingMethodException`;
- `NotSupportedException`;
- `TypeLoadException`;
- `TypeUnloadedException`.

Wiele wyjątków może być zgłoszonych wcześnie, ponieważ kod je wywołujący jest aktualnie uruchomiony. Zwykle jest to spowodowane wykryciem błędu w czasie konwersji kodu IL do postaci macierzystego kodu (błąd kompilacji w locie). Poniżej przedstawiono listę takich wyjątków:

- `MissingFieldException`;
- `MissingMethodException`;
- `SecurityException`;
- `TypeLoadException`.

Wyjątki zostaną omówione bardziej szczegółowo w rozdziale 15., „Użycie zarządzanych wyjątków do efektywnej obsługi błędów”.

Zdalne uruchomienie

Jeżeli ustalono, że tożsamość obiektu nie może być współdzielona, to CLR ustanawia granicę pracy zdalnej przy użyciu proxy. Proxy reprezentuje obiekt po jednej stronie granicy, natomiast wszystkie pola egzemplarza i odwołania metod są przekazywane na drugą stronę. Proxy jest automatycznie tworzone dla obiektów wywodzących się z `System.MarshalByRefObject`.

Funkcje pracy zdalnej zostaną omówione bardziej szczegółowo w rozdziale 13., „Tworzenie rozproszonych aplikacji przy użyciu .NET Remoting”.

CLR posiada mechanizm umożliwiający izolację aplikacji działających w tym samym procesie systemu operacyjnego. Ten mechanizm jest nazywany *domeną aplikacji*. Klasa w bibliotece klas bazowych udostępnia funkcje tej *AppDomain*. W celu zapewnienia efektywnej komunikacji między dwoma izolowanymi obiektami niezbędne jest utworzenie granicy pomiędzy domenami aplikacji.

Dostęp do pamięci

Dostęp do pamięci w środowisku wykonawczym musi być prawidłowo zorganizowany. Oznacza to, że adres dostępu do wartości `int16` lub `unsigned int16` (`short` lub `ushort`, dwubajtowe wartości) musi być parzysty. Dostęp do wartości `int32`, `unsigned int32` i `float32` (`int`, `uint` i `float`, 4-bajtowe wartości) musi być określony adresami podzielnymi przez 4. Dostęp do wartości `int64`, `unsigned int64` i `float64` (`long`, `ulong` i `double`, 8-bitowe wartości) musi cechować się adresami podzielnymi przez 4 lub 8 (w zależności od architektury). Z kolei adres używany do dostępu do macierzystych typów (`native int`, `native unsigned int` oraz `&`) musi być podzielny przez 4 lub 8 (w zależności od macierzystego środowiska).

Prawidłowa organizacja danych gwarantuje niepodzielność zapisywanych i odczytywanych danych o wielkości nie przekraczającej wielkości `native int`.

Dostęp do pamięci ulotnej

Niektóre polecenia dostępu do pamięci w IL umożliwiają użycia przedrostka `volatile`. Oznaczenie dostępu do pamięci w ten sposób nie gwarantuje niepodzielności danych, ale zapewnia odczyt zmiennej z pamięci przed uzyskaniem dostępu do pamięci. Innymi słowy, słowo kluczowe gwarantuje zapis do pamięci przed przyznaniem dostępu do zmiennej w pamięci.

Zadaniem przedrostka `volatile` jest symulacja sprzętowego rejestru procesora.

Wątki i blokady w CLR

CLR obsługuje wiele różnych mechanizmów gwarantujących synchronizację dostępu do danych. Synchronizacja wątków zostanie omówiona szczegółowo w rozdziale 11. Poniżej przedstawiono niektóre blokady stanowiące część modelu wykonawczego CLR:

- Blokady zsynchronizowanych metod — CLR umożliwia blokadę zarówno konkretnego egzemplarza (blokada wskaźnika `this`), jak i blokadę typu, dla którego zdefiniowano metodę (statyczna blokada). Blokada umożliwia wielokrotny dostęp do metody z tego samego wątku (rekurencja lub inne metody wywołania); dostęp do blokady z innego wątku będzie niemożliwy aż do momentu zwolnienia blokady.
- Jawne blokady — ten typ blokad jest zapewniany przez bibliotekę klas bazowych.
- Ulotny zapis i odczyt — jak już wspomniano wcześniej, oznaczenie metody jako `volatile` nie gwarantuje podzielności, chyba że wielkość wartości jest mniejsza lub równa `native int`, oraz jest prawidłowo zorganizowana.

- Operacje podzielne — biblioteka klas bazowych umożliwia wiele takich operacji dzięki klasie `System.Threading.Interlocked`.

Podsumowanie

W tym rozdziale omówiono krótko strukturę, w której wykonywany jest kod. Należy pamiętać, że na najniższym poziomie CLR jest mechanizmem umożliwiającym wykonanie poleceń IL. Pozwoli to lepiej zrozumieć sposób wykonywania IL i kodu aplikacji w CLR.

Kolejną kwestią są reguły ładowania podzespołu i wykonywania metody. W rozdziale można odnaleźć szczegółowe informacje o sterowaniu działaniem metody, a także o wbudowanych mechanizmach obsługi błędów i wyjątków w tym środowisku wykonawczym. Przedstawiłem także wbudowane w CLR funkcje pracy zdalnej. Ostatnim tematem rozdziału było omówienie sposobów dostępu kodu do pamięci oraz synchronizacji dostępu do metod w sytuacji, gdy wiele wątków może uzyskać dostęp do magazynu metod.