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

Tytuł oryginału: Programming C# 5.0
Tłumaczenie: Piotr Rajca
ISBN: 978-83-246-6987-5

Authorized Polish translation of the English edition Programming C# 5.0 ISBN 9781449320416

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/csh5pr_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/csh5pr.zip

Printed in Poland.

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

Spis treści

Wstęp

Od wprowadzenia języka C# minęło już dobrych kilkanaście lat. Rozwijał się on stopniowo, powiększając zarówno swe 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:

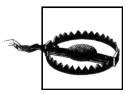
Kursywa
: Oznacza nowe pojęcia, adresy URL, adresy poczty elektronicznej, nazwy plików oraz rozszerzenia.

`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.

Przy użyciu tej ikony są oznaczane porady, sugestie lub ogólne uwagi.



Ta ikona symbolizuje 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*.

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 Kennediemu, 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 wspaniałe 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[1] 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.

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

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 skrótowo 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 przeszkód współpracować; a to z kolei oznacza, że biblioteki .NET zazwyczaj mogą być używane w kodzie pisanym 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.

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 równie 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 szczegóły 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, Javę 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 ściśle 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 „myśleniu" niż na robieniu czegoś, to F# może być odpowiednim rozwiązaniem.

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++ nić 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.

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 ściśle 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 kadrą 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[2], 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 wybierzemy, 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 ściśle 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 sprzed pojawienia 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

[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, określaną jako C++/CLI.

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 podzbiór 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 udostępniła więcej niż jedną implementację CLI. .NET Framework jest produktem o komercyjnej jakości i implementuje znacznie więcej niż jedynie możliwości CLI. Firma Microsoft udostępniła także bazę kodu, określaną 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*).

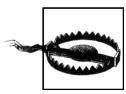
W przypadku kodu zarządzanego to środowisko uruchomieniowe, a nie kompilator generuje kod maszynowy wykonywany następnie przez procesor. Dzięki temu środowisko uruchomieniowe jest w stanie dostarczać usług, których udostępnianie w tradycyjnym modelu działania byłoby trudne lub nawet niemożliwe. Kompilator 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 kompilacją *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 kompilacji JIT. Takie działania mogą być zainicjowane przez narzędzia diagnostyczne, lecz również CLR może przeprowadzić ponowną kompilację, 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 kompilacja 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.



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 rozdziale 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 rozdziale 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 umieszczanie 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 jakiejkolwiek 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 asynchroniczne

Najważniejszą nowością wprowadzoną w C# 5.0 jest wsparcie dla programowania asynchronicznego. Platforma .NET Framework zawsze udostępniała asynchroniczne interfejsy programowania aplikacji (czyli takie, w których wywołanie metody może się zakończyć, nim operacja zostanie w całości wykonana). Asynchroniczność 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 asynchronicznymi API zawsze polegał na tym, że korzystanie z nich jest znacznie trudniejsze niż z ich synchronicznych odpowiedników. Jednak obecnie, jeśli tylko asynchroniczny 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 asynchronicznych 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, asynchronicznymi 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 asynchroniczne, 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 debuger, 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.

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[3]. 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.

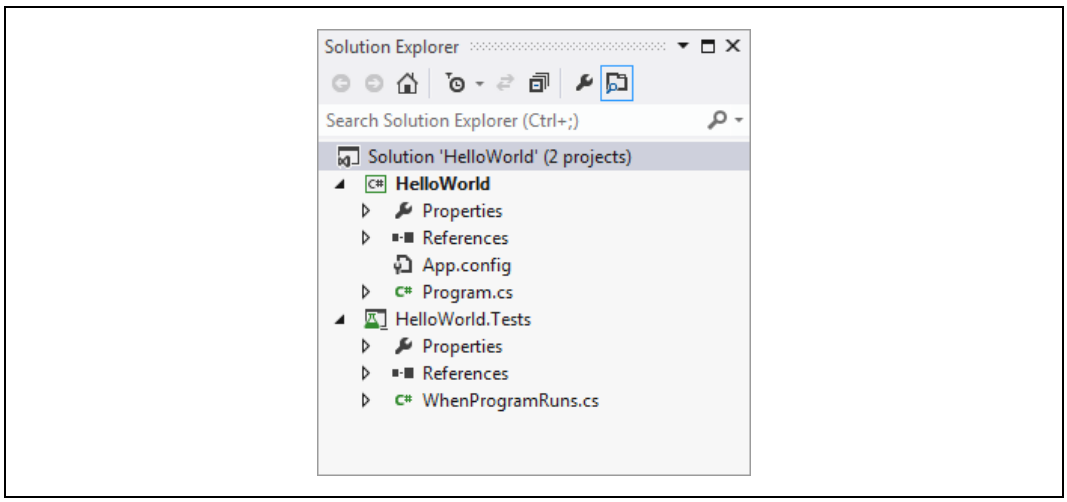
Pliki projektów zazwyczaj mają rozszerzenie kończące się na *proj*. Na przykład projekty C# mają rozszerzenie *.csproj*, a projekty C++ rozszerzenie *.vcxproj*. Jeśli przejrzymy te pliki przy użyciu edytora tekstów, przekonamy się, że zazwyczaj zawierają one kod XML. (Choć nie zawsze tak się dzieje.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

[3] W systemie Windows pliki wykonywalne zazwyczaj mają rozszerzenie *.exe*, natomiast biblioteki — rozszerzenie *.dll* (jest to skrót od angielskich słów *dynamic lik 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 rozdziale 12.

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 przedstawia 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*. Co 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 zamykania 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 kompilacji 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.

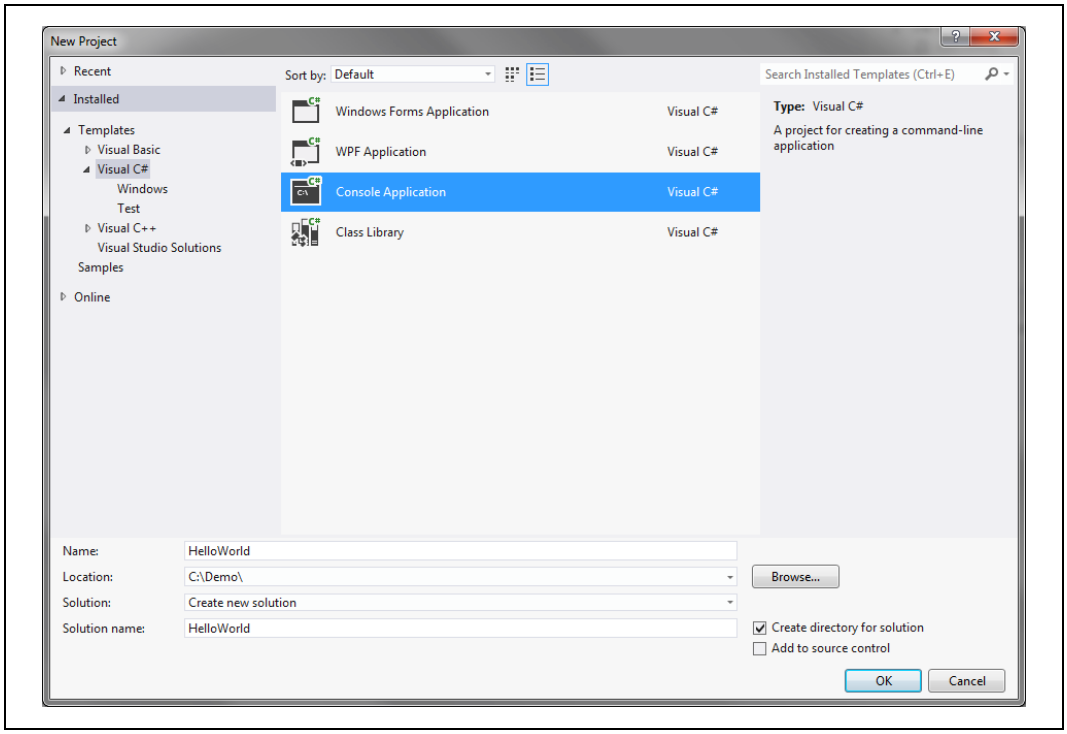
Mając to wszystko na uwadze, w dalszej części rozdziału pokażę, jak można utworzyć projekt wraz z solucją, a następnie w ramach wprowadzenia do języka C# pokażę różne elementy, które w Visual Studio można dodawać do nowego projektu. Pokażę także, w jaki sposób można dodawać do projektów testy jednostkowe.

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#. Znaczna 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 pobieżnie przejrzeć kolejny podrozdział, gdyż Visual Studio będzie Ci już znane.

Anatomia prostego programu

Aby stworzyć nowy projekt, należy skorzystać z opcji *FILE/New/Project*[4], a jeśli ktoś woli posługiwać się klawiaturą, to może użyć kombinacji klawiszy *Ctrl+Shift+N*. W efekcie na ekranie zostanie wyświetlone okno dialogowe *New Project*, przedstawione na rysunku 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 zaznaczymy 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.

[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 zapisywał nazwy opcji, jedynie zaczynając je wielką literą.



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 kompilacji 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 zaznaczymy 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 wyświetlonego 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 zaznaczymy 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 wyświetlone w panelu *Solution Explorer*, którego wygląd będzie teraz przypominał ten pokazany na rysunku 1.1.

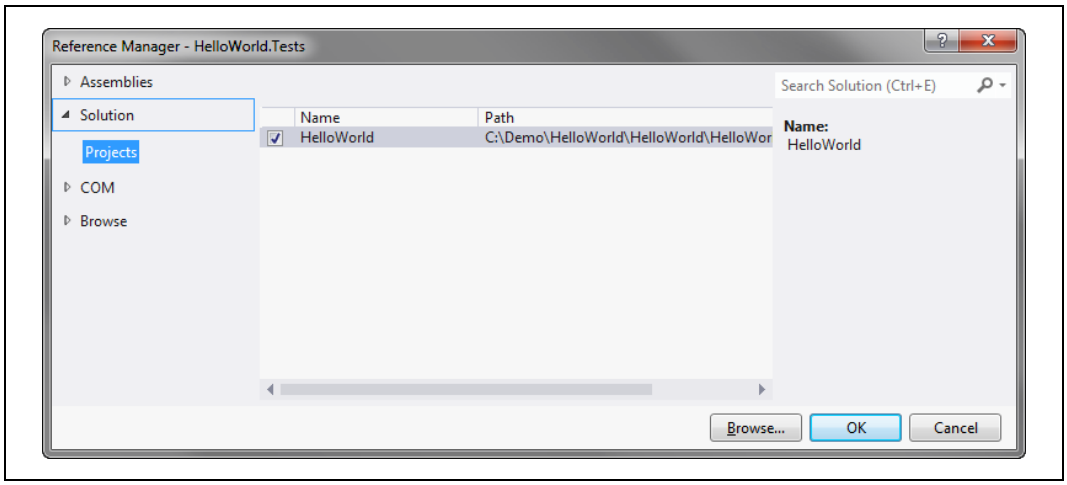
Ten dodatkowy projekt testowy będzie miał za zadanie zapewnić, że nasz główny projekt będzie działał zgodnie z naszymi oczekiwaniami. Osobiście preferuję styl programowania, w którym testy pisze się przed testowanym kodem, dlatego też zaczniemy właśnie od tego projektu. (Takie podejście określane jest 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

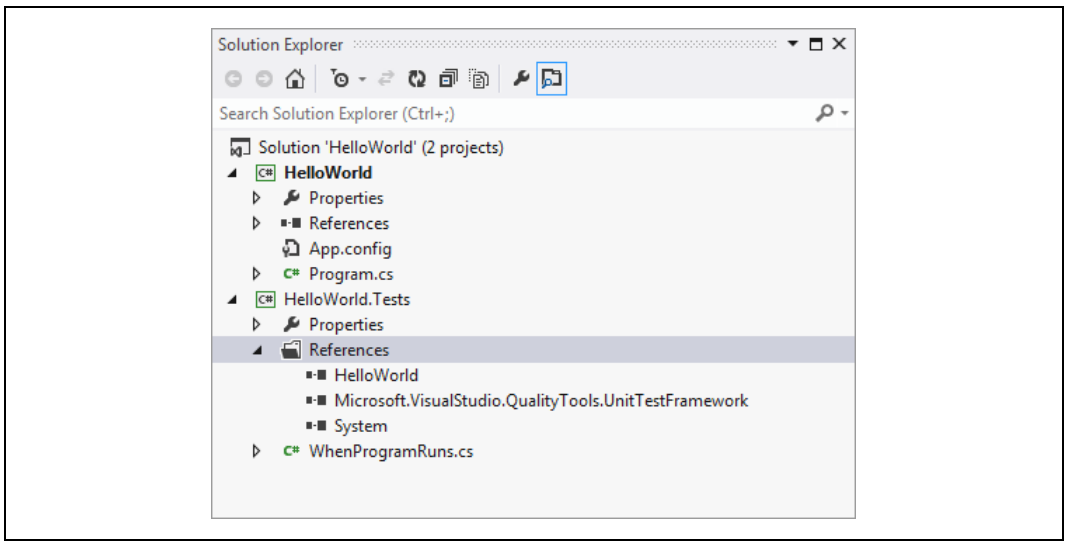
Aby przekazać Visual Studio informacje o zależnościach pomiędzy naszymi dwoma projektami, trzeba kliknąć prawym przyciskiem myszy węzeł *References* projektu *HelloWorld.Test* w panelu *Solution Explorer* i wybrać opcję *Add Reference*. Gdy to zrobimy, na ekranie zostanie wyświetlone okno dialogowe *Reference Manager* przedstawione na rysunku 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 rysunku 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 rysunku 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*[5]. 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 listingu 1.1; kod testu jest umieszczony na samym dole i został wyróżniony pogrubieniem.

Listing 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

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

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 udostępniłyby podrobioną wersję tej abstrakcji, służącą do celów testowych. My jednak chcielibyśmy, by nasza aplikacja (którą kod z listingu 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 rozdziale 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 listingu 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`. Listing 1.2 przedstawia odpowiedni wiersz kodu z listingu 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.

Listing 1.2. Wywoływanie metody

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

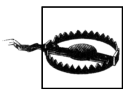
Niestety, takie rozwiązanie stwarza pewien problem. Punkt wejścia do programu jest zazwyczaj dostępny wyłącznie w trakcie jego działania — jest to szczegół implementacyjny programu i zazwyczaj nie ma żadnego powodu, by go udostępniać publicznie. Jednak w tym przykładzie zrobimy wyjątek, gdyż metoda `Main` 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 listingu 1.3. (Wszystko zostanie wyjaśnione już niebawem).

Listing 1.3. Udostępnienie punktu wejścia do programu

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

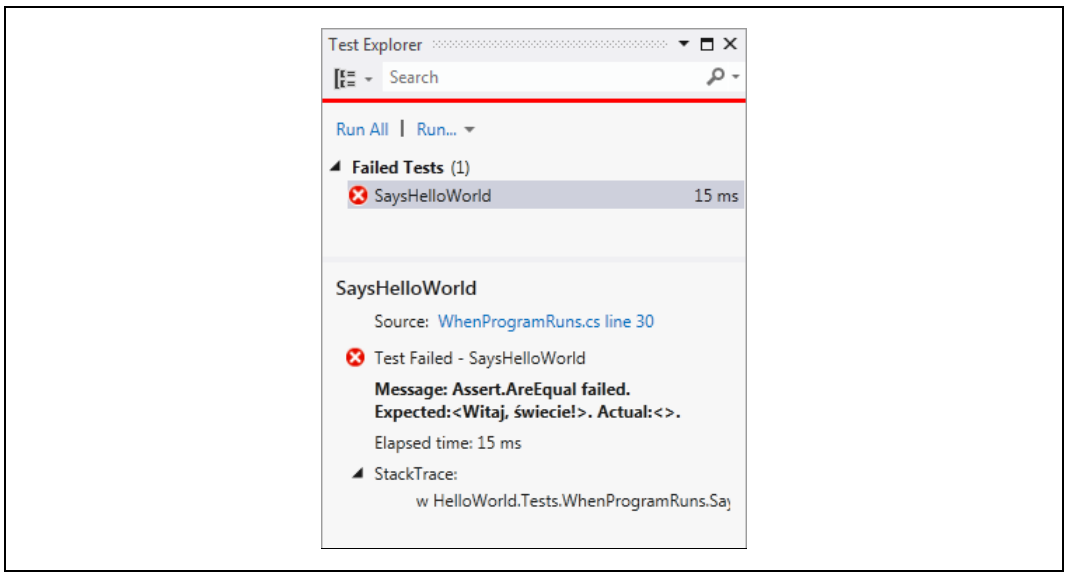
Na początku obu wierszy kodu dodaliśmy słowo kluczowe `public`, dzięki czemu kod programu stanie się dostępny dla testów i w końcu będzie można skompilować kod z listingu 1.1. Istnieją także inne sposoby pozwalające uzyskać taki sam efekt. Można pozostawić klasę w niezmienionej postaci, lecz oznaczyć 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 rozdziałach 3. i 15.), dlatego też chciałem, by ten pierwszy przykład był możliwie jak najprostszy. Alternatywne rozwiązanie przedstawiłem w rozdziale 15.



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 rysunku 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 rysunku 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 listingu 1.4, przy czym zawiera on już modyfikacje przedstawione na listingu 1.3. W ramach użytecznej prezentacji najważniejszych elementów składni oraz struktury C# w kolejnych punktach rozdziału opisałem poszczególne elementy przedstawionego kodu.

Listing 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.

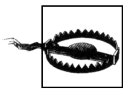
Przestrzenie nazw zazwyczaj stanowią także pewną podpowiedź dotyczącą przeznaczenia typu. Na przykład wszystkie typy związane z obsługą plików należą do przestrzeni nazw `System.IO`, natomiast te związane z komunikacją sieciową — do przestrzeni `System.Net`. Przestrzenie nazw mogą także tworzyć hierarchie. 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 listingu 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.



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 rozdziale 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 listingu 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 listing 1.5, dla każdej z tych klas można zdefiniować unikatową nazwę zastępczą.

Listing 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 listingu 1.4 został przedstawiony na listingu 1.6. Bezpośrednio za deklaracją przestrzeni nazw jest umieszczany otwierający nawias klamrowy ({). Cały kod znajdujący się pomiędzy tym nawiasem oraz odpowiadającym mu zamykającym nawiasem klamrowym umieszczonym 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`.

Listing 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 listingu 1.7.

Listing 1.7. Zagnieżdżanie deklaracji przestrzeni nazw

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

Alternatywnym rozwiązaniem jest podanie pełnej nazwy przestrzeni w jednej deklaracji, jak pokazałem na listingu 1.8. To rozwiązanie jest stosowane znacznie częściej.

Listing 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 listingów 1.7 i 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 umieszczane 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 listing 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.

Listing 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 rozdziale 3.

Punkt wejścia do programu

Domyślnie kompilator C# poszukuje metody o nazwie `Main` i jeśli uda mu się ją znaleźć, to automatycznie użyje 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 listingu 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 listingu 1.9).

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 powiódł się, ponieważ program nie spełnił zakładanego warunku: nie wyświetlił w oknie konsoli odpowiedniego komunikatu. Spełnienie tego warunku wymaga dopisania jednego wiersza kodu (przedstawionego na listingu 1.10) i umieszczenia go wewnątrz metody.

Listing 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 debugerze, 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 wybranie 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 listing 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 listing 1.11 przedstawia interesujący nas aktualnie fragment kodu z listingu 1.1.

Listing 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 listingu 1.1 zostały przedstawione na listingu 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 tekstów. Jeśli natomiast chodzi o atrybut `[TestMethod]`, to jak się zapewne domyślasz, informuje on, które metody reprezentują testy.

Listing 1.12. Metody z atrybutami

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

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

Warto zwrócić uwagę na jeszcze jeden fragment kodu z listingu 1.1: zawartość klasy rozpoczyna się od pola, które przedstawiłem ponownie na listingu 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.

Listing 1.13. Pole

```
private string _consoleOutput;
```

W ten sposób poznałeś 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 nim solucję 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 przestrzenia 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 rozdziale 3., jednak zanim do niego dotrzemy, w rozdziale 2. zajmiemy się kodem umieszczanym wewnątrz metod, który pozwala nam wyrażać to, co program ma robić.

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 rozdziale 1. W tym rozdziale przyjrzymy się dokładniej kodowi umieszczanemu w metodach. C# wymaga stosowania określonej struktury: kod składa się z instrukcji, które są umieszczane wewnątrz 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 listingu 2.1), a nie całe programy.

Listing 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 listingu 2.1 odpowiada programowi z listingu 2.2.

Listing 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.

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 kompilacji. Różni się to znacząco od sposobu działania języków o **typowaniu dynamicznym**, takich jak JavaScript, w których to typy danych są określane podczas wykonywania programu[1].

Najprostszym sposobem zaobserwowania statycznego typowania w działaniu jest posłużenie się bardzo prostymi deklaracjami zmiennych, takimi jak te z listingu 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`.

Listing 2.3. Deklaracje zmiennych

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

[1] 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 rozdziale 14.

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 listingu 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. Listing 2.4 stanowi kontynuację poprzedniego i pokazuje, że nowe wartości można przypisywać zmiennym niezależnie od tego, czy zostały one wcześniej zainicjowane, czy nie.

Listing 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 listingu 2.3, oddając do niego instrukcję przedstawioną na listingu 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.

Listing 2.5. Błąd: dane niewłaściwego typu

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

Języki korzystające z typowania dynamicznego, takie jak JavaScript, pozwoliłyby 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 rozdziale 14.). Jednak najczęściej stosowaną w C# praktyką jest korzystanie ze zmiennych, których typy są określane bardziej precyzyjnie.

Ze względu na dziedziczenie ten statyczny typ zmiennej nie zawsze zapewnia pełny obraz sytuacji. Tymi zagadnieniami zajmiemy się w rozdziale 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 rozdziale 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` umieszczanego zamiast nazwy typu, możemy zażądać, by kompilator zrobił to za nas. Listing 2.6 przedstawia deklaracje pierwszych trzech zmiennych z listingu 2.3, w których zamiast jawnie podanego typu danych zostało użyte słowo kluczowe `var`.

Listing 2.6. Niejawne 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 listingu 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 określane typy danych; wystarczy spróbować przypisać takiej zmiennej wartość innego typu. Możemy zatem przeprowadzić ten sam eksperyment co w kodzie z listingu 2.5, lecz tym razem używając zmiennej zadeklarowanej z użyciem słowa kluczowego `var`. Właśnie to robi fragment kodu przedstawiony na listingu 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.

Listing 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. „`var`to czy nie `var`to?”, 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. Listing 2.8 pokazuje, w jaki sposób utworzyć trzy zmienne tego samego typu, używając przy tym jednej deklaracji.

Listing 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. Listing 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 jawne 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", którą 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 jawne 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 jawne 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).

Listing 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 listingu 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;
```

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 listingu 2.10, to kompilator C# zgłosi błąd.

Listing 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[2]. 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. Rozwiązaniem tego problemu jest użycie inicjalizatora, dzięki któremu zmienna zawsze będzie miała jakąś wartość. Dla typów liczbowych

[2] 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).

taką nieużywaną wartością początkową może być `0`, natomiast dla zmiennych logicznych — wartość `false`. W rozdziale 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 jakiejkolwiek 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 listingu 2.11, pojawiłby się komunikat o błędzie o następującej treści: `The name 'thisWillNotWork' does not exist in the current context`[3].

Listing 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 listingu 2.12

[3] Nazwa „thisWillNotWork" nie istnieje w bieżącym kontekście — *przyp. tłum.*

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.

Listing 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 zamienną zadeklarujemy w bloku zagnieżdżonym, to jej zakres będzie obejmował wyłączenie ten blok. A zatem nie uda się skompilować kodu z listingu 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.

Listing 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 listingu 2.14. Deklaruje on zmienną o nazwie `anotherValue`, umieszczoną wewnątrz zagnieżdżonego bloku kodu. Jak wiemy, zmienna znajduje się w zakresie wyłącznie do końca zagnieżdżonego bloku kodu. Po zakończeniu bloku ponownie deklarujemy inną zmienną o tej samej nazwie.

Listing 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[4]
```

[4] 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 „podrzędnym" do oznaczenia czegoś innego — *przyp. tłum.*

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 zagnieżdżonym blokiem, wewnątrz którego została ona zadeklarowana. Co więcej, druga deklaracja znajduje się poza zakresem zagnieżdżonego 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. Listing 2.15 przedstawia kod ilustrujący problem, którego staramy się uniknąć. Zadeklarowaliśmy w nim zmienną o nazwie `errorCount`, a podczas wykonywania programu zaczynamy modyfikować jej wartość. Jednak w pewnym momencie w zagnieżdżonym 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.

Listing 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 zagnieżdżonym 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 listingu 2.14? Zakresy obu zmiennych nie pokrywały się ze sobą. Cóż, okazuje się, że reguła zilustrowana na listingu 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 zagnieżdżonej przestrzeni deklaracji nie można deklarować nazw występujących

już w przestrzeni nadrzędnej. I to właśnie ta reguła została tu naruszona — najbardziej zewnętrzna przestrzeń deklaracji istniejąca w kodzie z listingu 2.15 zawiera już zmienną o nazwie `errorCount`, a przestrzeń zagnieżdżonego 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 listingu 2.15 było dopuszczalne, to założenie to nie byłoby spełnione. I to wyjaśnia, dlaczego kod z listingu 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 ściśle określonym miejscu kodu i także jej zakres kończy się w ściśle 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 zagnieżdżonych, które w przyszłości 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 listingu 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 listingu 2.16 oba są prawdziwe.

Listing 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 listingu 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 listingach 2.16 oraz 2.17. Oba robią dokładnie to samo, gdyż poszczególne czynności, które chcemy wykonać, są dokładnie takie same. Jednak kod z listingu 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 listingu 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.

Listing 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? Zacznijmy zatem od odpowiedzi na pytanie, a dopiero później wrócimy 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 ściśle określoną 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.

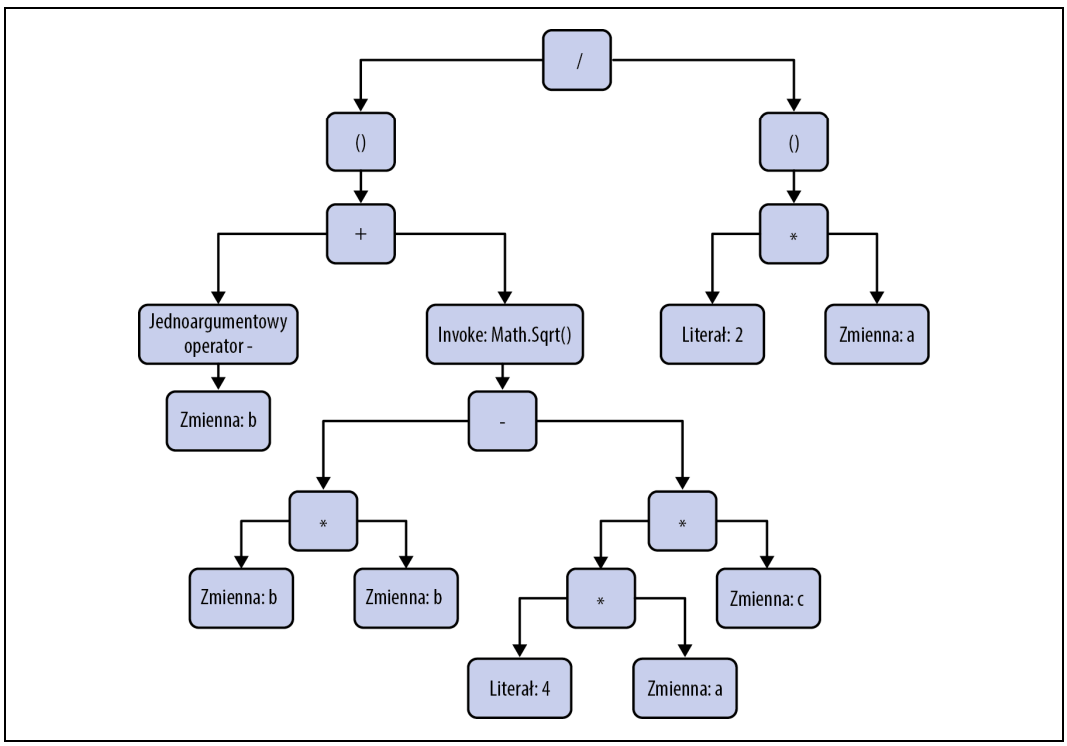
Niektóre symbole mają różne znaczenie zależnie od kontekstu. Znak minusa nie jest używany wyłącznie do oznaczania liczb ujemnych. Jeśli zostanie umieszczony pomiędzy dwoma wyrażeniami, to pełni także rolę operatora odejmowania, wymagającego określenia dwóch operandów.

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 listingu 2.18 wykorzystują tę możliwość, by wyliczyć jeden z pierwiastków równania kwadratowego (jest to standardowy sposób rozwiązywania równań tego typu).

Listing 2.18. Wyrażenia wewnątrz 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 rysunku 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. Listing 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 wejściowy. 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.

Listing 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 próbują wykonać pewne obliczenia, a następnie ignorują 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 listingu 2.19 nie mogłoby być instrukcją, co wydaje się być dobrym pomysłem, gdyż ten kod nie robi niczego użytecznego. W trzecim wierszu kodu z listingu 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. Listing 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.

Listing 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 listingu 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`.

Listing 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 struktura. 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[5].

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 listing 2.22.

Listing 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 listingu 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 —

[5] 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`.

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 `acdebf5`. 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 zagnieżdżenia. Rozpatrując osobno każde z wyrażeń, widzimy, że jego operandy były przetwarzane od lewej do prawej, a to, że wyniki oddają kolejność zagnieżdżenia, 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 listingu 2.23. Są one tworzone przez zapisanie sekwencji dwóch znaków ukośnika (`/`); cała dalsza zawartość wiersza będzie ignorowana przez kompilator.

Listing 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 listingu 2.24). Ten przykład pokazuje także, że komentarze oddzielone mogą zajmować kilka wierszy kodu.

Listing 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 rozpoczynania i zakańczania 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. Listing 2.25 pokazuje problem, którego powodem jest komentarz rozpoczynający się w połowie pierwszego wiersza kodu i kończący w czwartym wierszu.

Listing 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 zagnieżdżać. 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 zagnieżdżanie komentarzy, zatem za sekwencją zamykającą 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.

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 listingu 2.26 są prawidłowe i mają takie samo znaczenie.

Listing 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 wewnątrz literałów łańcuchowych, gdyż wszystkie znaki umieszczone wewnątrz takiego literału 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 umieszczane w osobnych wierszach.

Dyrektywy 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.

Symbole kompilacji

C# udostępnia dyrektywę `#define`, pozwalającą definiować **symbole kompilacji**. Są one powszechnie używane do kompilowania 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 kompilacji definiuje się przy użyciu ustawień budowania kompilatora. Visual Studio pozwala określać 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 ustawiać przy użyciu odpowiednich przełączników.

W nowych projektach Visual Studio domyślnie definiuje pewne symbole kompilacji. 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 kompilacji są zazwyczaj używane wraz z dyrektywami `#if`, `#else`, `#elif` oraz `#endif`. Kod przedstawiony na listingu 2.27 używa niektórych spośród tych dyrektyw, by zapewnić, że pewne wiersze kodu zostaną skompilowane wyłącznie w przypadku użycia konfiguracji *Debug*.

Listing 2.27. Kompilacja warunkowa

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

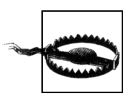
C# udostępnia nieco bardziej subtelny mechanizm przeznaczony do stosowania w takich sytuacjach, są to tak zwane **metody warunkowe** (ang. *conditional methods*). Kompilator rozpoznaje atrybut `ConditionalAttribute` definiowany przez .NET Framework i nadaje mu specjalne możliwości. Atrybutem tym można oznaczyć dowolną metodę. W kodzie z listingu 2.28 atrybut ten został użyty, by zaznaczyć, że metoda ma być skompilowana wyłącznie w przypadku, gdy został zdefiniowany symbol `DEBUG`.

Listing 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 listingu 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`.



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 początkujący 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*.

Dyrektywy #error oraz #warning

C# udostępnia dyrektywy `#error` oraz `#warning`, które pozwalają generować błędy i ostrzeżenia kompilatora. Są one zazwyczaj używane wewnątrz fragmentów kodu kompilowanych warunkowo, takich jak te przedstawione na listingu 2.29; choć bezwarunkowa dyrektywa `#warning` doskonale nadaje się także do przypominania o tym, że jeszcze nie napisaliśmy jakiegoś niezwykle ważnego fragmentu kodu.

Listing 2.29. Generowanie błędu kompilatora

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

Dyrektywa #line

Dyrektywa `#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. Listing 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*.

Listing 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`.

Dyrektywa `#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`.

Dyrektywa #pragma

Dyrektywa `#pragma` pozwala na wyłączanie 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).

Listing 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.

Listing 2.31. Wyłączanie ostrzeżeń kompilatora

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

Ogólnie rzecz biorąc, należy unikać wyłączania ostrzeżeń. Dyrektywa ta jest głównie wykorzystywana w rozwiązaniach związanych z generowaniem kodu, w których może ona okazać się jedynym sposobem zapewniania, że podczas kompilacji 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 jedyną 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 kompilacji. Jednak prawidłowe pary tych dyrektyw są przez kompilator ignorowane. Regiony tworzone przy ich użyciu można zagnieżdżać.

Dyrektywy te zostały wprowadzone wyłącznie z myślą o edytorach 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 zwinięciu 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 listingu 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 dysponujemy łańcuchem 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 oba łańcuchy.

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

Typy liczbowe

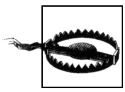
C# obsługuje działania arytmetyczne na liczbach całkowitych oraz zmiennoprzecinkowych. Dostępne są zarówno typy danych reprezentujące liczby ze znakiem, jak i wyłącznie liczby dodatnie, jak również typy o różnych zakresach wartości (przedstawiłem je w tabeli 2.1). Najczęściej używanym typem liczb całkowitych 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 jakich 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[6], 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 tabeli 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 tabeli 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.

[6] 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 rozdziale 21.



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[7]; a zgodnie z tym, co sugerują ich nazwy CLR podane w tabeli 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
`float`	`System.Single`	32	23 bity (~7 cyfr po przecinku)	1.5×10^{-45} do 3.4×10^{38}
`double`	`System.Double`	64	52 bity (~15 cyfr po przecinku)	5.0×10^{-324} do 1.7×10^{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

[7] *pl.wikipedia.org/wiki/IEEE_754*

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 pieniędzmi 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.

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[8]. 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 określać 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` oznacza się odpowiednio przy użyciu następujących przyrostków: `D`, `F` oraz `M`.

[8] Oznacza to, że typ `decimal` nie wykorzystuje wszystkich 128 bitów. Zastosowanie mniejszej liczby bitów sprawiłoby 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.

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 literał 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 literałów 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 rozdziale 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. Listing 2.32 pokazuje niektóre przypadki, w których zostanie wykonana taka automatyczna konwersja.

Listing 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 uzmysłowić 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 wyrazu[9].

C# wykonuje konwersje wartości liczbowych w sposób niejawny, 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 listingu 2.33 nie uda się skompilować, gdyż próbują 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.

Listing 2.33. Błąd: niedostępna konwersja niejawna

```
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. Listing 2.34 stanowi zmodyfikowaną wersję listingu 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.

Listing 2.34. Jawna konwersja przy użyciu rzutowania

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

[9] 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.

A zatem konwersje polegające na zawężaniu wymagają użycia jawnego rzutowania, natomiast 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.

Można się zastanawiać, co się stanie, gdy w sytuacji takiej jak ta z listingu 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 precyzyjnie 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 taktowane 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`[10]. (Rozwiązanie 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). Listing 2.35 pokazuje jeden ze sposobów, jak można to zrobić.

Listing 2.35. Wykorzystanie niesprawdzanego przepełnienia danych typu całkowitego

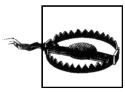
```
int start = Environment.TickCount;
DoSomeWork();
int end = Environment.TickCount;

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

Podstępną cechą właściwości `Environment.TickCount` 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 oczekiwaniami 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).

[10] *Właściwość* jest składową typu reprezentującą wartość, którą można odczytać lub zmodyfikować; można też wykonywać obie te czynności; właściwości zostały opisane w rozdziale 3.



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 listingu 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 odejmiemy 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 oczekiwać uzyskania wartości 4 294 967 185. Jednak ze względu na przepełnienie, 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 listingu 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.

Listing 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 listingu 2.37). Instrukcje sprawdzane zawsze odnoszą się do bloku kodu — słowa kluczowego `chcecked` nie można umieścić przed `int` na listingu 2.36, by zmienić przypisanie w instrukcję sprawdzaną. Dodatkowo konieczne byłoby umieszczenie jej w nawiasach klamrowych.

Listing 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 jawne 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 znacząco 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 listingu 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`.

Listing 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 właśnie 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[11] 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 rozdziale 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.

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

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 tabeli 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 tabeli 2.4. Operatory te nie działają na liczbach zmiennoprzecinkowych.

Tabela 2.4. Operatory bitowe działające na liczbach całkowitych

Nazwa	Przykład
Negacja bitowa	`~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 przeciwne — 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 tabeli 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)	`!x`
Koniunkcja warunkowa (AND)	`x && y`
Alternatywa warunkowa (OR)	`x || y`

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 listingu 2.39.

Listing 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 listingu 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 tabeli 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	`x < y`
Większy	`x > y`
Mniejszy lub równy	`x <= y`
Większy lub równy	`x >= y`
Równy	`x == y`
Różny	`x != y`

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 umieszczony 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`[12].

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 listingu 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).

Listing 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)`,

[12] 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 rozdziale 3.