

MARCIN LIS

WYDANIE III

C#



PRAKTYCZNY
KURS

Helion

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

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

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

Opieka redakcyjna: Ewelina Burska

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

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, zatrzymaj pod adres
http://helion.pl/user/opinie/cshpk3_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z ćwiczeniami i listingami wykorzystanymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/cshpk3.zip>

ISBN: 978-83-283-2893-8

Copyright © Helion 2016

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

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

Spis treści

Wstęp	9
Czym jest C#?	9
Dla kogo jest ta książka?	9
Standardy C#	10
Rozdział 1. Zanim zaczniesz programować	11
Lekcja 1. Podstawowe koncepcje C# i .NET	11
Jak to działa?	11
Narzędzia	12
Instalacja narzędzi	13
Lekcja 2. Pierwsza aplikacja, komilacja i uruchomienie programu	16
.NET Framework	16
Visual Studio	19
Mono	22
MonoDevelop (Xamarin Studio)	23
Struktura kodu	26
Lekcja 3. Komentarze	27
Komentarz blokowy	27
Komentarz liniowy	28
Komentarz XML	29
Ćwiczenia do samodzielnego wykonania	30
Rozdział 2. Elementy języka	31
Typy danych	31
Lekcja 4. Typy danych w C#	32
Typy danych w C#	32
Zapis wartości (literaly)	36
Zmienne	39
Lekcja 5. Deklaracje i przypisywanie	39
Proste deklaracje	39
Deklaracje wielu zmiennych	41
Nazwy zmiennych	42
Zmienne typów odnośnikowych	42
Ćwiczenia do samodzielnego wykonania	43

Lekcja 6. Wyprowadzanie danych na ekran	43
Wyświetlanie wartości zmiennych	43
Wyświetlanie znaków specjalnych	46
Instrukcja Console.WriteLine	48
Ćwiczenia do samodzielnego wykonania	49
Lekcja 7. Operacje na zmiennych	49
Operacje arytmetyczne	50
Operacje bitowe	57
Operacje logiczne	61
Operatory przypisania	63
Operatory porównywania (relacyjne)	64
Pozostałe operatory	65
Priorytety operatorów	65
Ćwiczenia do samodzielnego wykonania	66
Instrukcje sterujące	67
Lekcja 8. Instrukcja warunkowa if...else	67
Podstawowa postać instrukcji if...else	67
Zagnieżdżanie instrukcji if...else	69
Instrukcja if...else if	72
Ćwiczenia do samodzielnego wykonania	75
Lekcja 9. Instrukcja switch i operator warunkowy	76
Instrukcja switch	76
Przerwianie instrukcji switch	79
Operator warunkowy	81
Ćwiczenia do samodzielnego wykonania	82
Lekcja 10. Pętle	82
Pętla for	83
Pętla while	86
Pętla do...while	88
Pętla foreach	89
Ćwiczenia do samodzielnego wykonania	90
Lekcja 11. Instrukcje break i continue	91
Instrukcja break	91
Instrukcja continue	95
Ćwiczenia do samodzielnego wykonania	96
Tablice	97
Lekcja 12. Podstawowe operacje na tablicach	98
Tworzenie tablic	98
Inicjalizacja tablic	101
Właściwość Length	102
Ćwiczenia do samodzielnego wykonania	103
Lekcja 13. Tablice wielowymiarowe	104
Tablice dwuwymiarowe	104
Tablice tablic	107
Tablice dwuwymiarowe i właściwość Length	109
Tablice nieregularne	111
Ćwiczenia do samodzielnego wykonania	115
Rozdział 3. Programowanie obiektowe	117
Podstawy	117
Lekcja 14. Klasy i obiekty	118
Podstawy obiektowości	118
Pierwsza klasa	119
Jak użyć klasy?	121

Metody klas	122
Jednostki komplikacji, przestrzeń nazw i zestawy	126
Ćwiczenia do samodzielnego wykonania	130
Lekcja 15. Argumenty i przeciążanie metod	131
Argumenty metod	131
Obiekt jako argument	134
Przeciążanie metod	138
Argumenty metody Main	139
Sposoby przekazywania argumentów	140
Definicje metod za pomocą wyrażeń lambda	143
Ćwiczenia do samodzielnego wykonania	144
Lekcja 16. Konstruktory i destruktory	145
Czym jest konstruktor?	145
Argumenty konstruktorów	148
Przeciążanie konstruktorów	149
Słowo kluczowe this	151
Niszczanie obiektu	154
Ćwiczenia do samodzielnego wykonania	155
Dziedziczenie	156
Lekcja 17. Klasa potomne	156
Dziedziczenie	156
Konstruktory klasy bazowej i potomnej	160
Ćwiczenia do samodzielnego wykonania	164
Lekcja 18. Modyfikatory dostępu	164
Określanie reguł dostępu	165
Dlaczego ukrywamy wewnętrze klasy?	170
Jak zabronić dziedziczenia?	174
Tylko do odczytu	175
Ćwiczenia do samodzielnego wykonania	178
Lekcja 19. Przesłanianie metod i składowe statyczne	179
Przesłanianie metod	179
Przesłanianie pól	182
Składowe statyczne	183
Ćwiczenia do samodzielnego wykonania	186
Lekcja 20. Właściwości i struktury	186
Właściwości	187
Struktury	196
Ćwiczenia do samodzielnego wykonania	200
Rozdział 4. Wyjątki i obsługa błędów	203
Lekcja 21. Blok try...catch	203
Badanie poprawności danych	203
Wyjątki w C#	207
Ćwiczenia do samodzielnego wykonania	211
Lekcja 22. Wyjątki to obiekty	212
Dzielenie przez zero	212
Wyjątek jest obiektem	213
Hierarchia wyjątków	214
Przechwytywanie wielu wyjątków	215
Zagnieżdżanie bloków try...catch	218
Ćwiczenia do samodzielnego wykonania	220

Lekcja 23. Własne wyjątki	220
Zgłaszanie wyjątków	221
Ponowne zgłoszenie przechwyconego wyjątku	223
Tworzenie własnych wyjątków	225
Wyjątki warunkowe	226
Sekcja finally	228
Ćwiczenia do samodzielnego wykonania	231
Rozdział 5. System wejścia-wyjścia	233
Lekcja 24. Ciągi znaków	233
Znaki i łańcuchy znakowe	233
Znaki specjalne	237
Zamiana ciągów na wartości	238
Formatowanie danych	240
Przetwarzanie ciągów	242
Ćwiczenia do samodzielnego wykonania	247
Lekcja 25. Standardowe wejście i wyjście	247
Klasa Console i odczyt znaków	248
Wczytywanie tekstu z klawiatury	255
Wprowadzanie liczb	256
Ćwiczenia do samodzielnego wykonania	257
Lekcja 26. Operacje na systemie plików	258
Klasa FileInfo	258
Operacje na katalogach	259
Operacje na plikach	266
Ćwiczenia do samodzielnego wykonania	271
Lekcja 27. Zapis i odczyt plików	271
Klasa FileStream	272
Podstawowe operacje odczytu i zapisu	274
Operacje strumieniowe	278
Ćwiczenia do samodzielnego wykonania	287
Rozdział 6. Zaawansowane zagadnienia programowania obiektowego	289
Polimorfizm	289
Lekcja 28. Konwersje typów i rzutowanie obiektów	289
Konwersje typów prostych	290
Rzutowanie typów obiektowych	291
Rzutowanie na typ Object	295
Typy proste też są obiektowe!	297
Ćwiczenia do samodzielnego wykonania	299
Lekcja 29. Późne wiązanie i wywoływanie metod klas pochodnych	299
Rzeczywisty typ obiektu	300
Dziedziczenie a wywoływanie metod	302
Dziedziczenie a metody prywatne	307
Ćwiczenia do samodzielnego wykonania	308
Lekcja 30. Konstruktory oraz klasy abstrakcyjne	309
Klasy i metody abstrakcyjne	309
Wywołania konstruktorów	313
Wywoływanie metod w konstruktorach	316
Ćwiczenia do samodzielnego wykonania	318
Interfejsy	319
Lekcja 31. Tworzenie interfejsów	319
Czym są interfejsy?	319
Interfejsy a hierarchia klas	322

Interfejsy i właściwości	324
Ćwiczenia do samodzielnego wykonania	326
Lekcja 32. Implementacja kilku interfejsów	326
Implementowanie wielu interfejsów	327
Konflikty nazw	328
Dziedziczenie interfejsów	331
Ćwiczenia do samodzielnego wykonania	333
Klasy zagnieżdżone	334
Lekcja 33. Klasa wewnątrz klasy	334
Tworzenie klas zagnieżdżonych	334
Kilka klas zagnieżdżonych	336
Składowe klas zagnieżdżonych	338
Obiekty klas zagnieżdżonych	339
Rodzaje klas wewnętrznych	342
Dostęp do składowych klasy zewnętrznej	344
Ćwiczenia do samodzielnego wykonania	345
Typy uogólnione	346
Lekcja 34. Kontrola typów i typy uogólnione	346
Jak zbudować kontener?	346
Przechowywanie dowolnych danych	350
Problem kontroli typów	352
Korzystanie z typów uogólnionych	353
Ćwiczenia do samodzielnego wykonania	356
Rozdział 7. Aplikacje z interfejsem graficznym	359
Lekcja 35. Tworzenie okien	359
Pierwsze okno	359
Klasa Form	361
Tworzenie menu	366
Ćwiczenia do samodzielnego wykonania	370
Lekcja 36. Delegacje i zdarzenia	371
Koncepcja zdarzeń i delegacji	371
Tworzenie delegacji	371
Delegacja jako funkcja zwrotna	375
Delegacja powiązana z wieloma metodami	379
Zdarzenia	381
Ćwiczenia do samodzielnego wykonania	391
Lekcja 37. Komponenty graficzne	392
Wyświetlanie komunikatów	392
Obsługa zdarzeń	393
Menu	395
Etykiety	397
Przyciski	399
Pola tekstowe	401
Listy rozwijane	404
Ćwiczenia do samodzielnego wykonania	407
Zakończenie	409
Skorowidz	410



Wstęp

Czym jest C#?

Język C# (wym. *ce szarp* lub *si szarp*, ang. *c sharp*) został opracowany w firmie Microsoft. Jak już sama nazwa wskazuje, wywodzi się on z rodziny C i C++, choć zawiera również wiele elementów znanych programistom np. Javy, jak chociażby mechanizmy automatycznego odzyskiwania pamięci. Programiści korzystający na co dzień z wymienionych języków programowania będą się czuli doskonale w tym środowisku. Z kolei dla osób nieznających C# nie będzie on trudny do opanowania, a na pewno będzie dużo łatwiejszy niż tak popularny C++.

Głównym twórcą C# jest Anders Hejlsberg, czyli nie kto inny, jak projektant produkowanego niegdyś przez firmę Borland bardzo popularnego pakietu Delphi, a także Turbo Pascala! W Microsoftie Hejlsberg rozwijał m.in. środowisko Visual J++. To wszystko nie pozostało bez wpływu na C#, w którym można dojrzeć wyraźne związki zarówno z C i C++, jak i Javą i Delphi, czyli Object Pascalem.

C# jest językiem obiektowym (zorientowanym obiektowo, ang. *object oriented*), zawiera wspomniane już mechanizmy odzyskiwania pamięci i obsługę wyjątków. Jest też ściśle powiązany ze środowiskiem uruchomieniowym .NET, co oczywiście nie jest równoznaczne z tym, że nie powstają jego implementacje przeznaczone dla innych platform. Oznacza to jednak, że doskonale sprawdza się w najnowszym środowisku Windows oraz w sposób bezpośredni może korzystać z klas .NET, co pozwala na szybkie i efektywne pisanie aplikacji.

Dla kogo jest ta książka?

Książka przeznaczona jest dla osób, które chcieliby nauczyć się programować w C# — zarówno dla tych, które dotychczas nie programowały, jak i dla znających już jakiś inny język programowania, a pragnących nauczyć się nowego. Czytelnik nie musi

więc posiadać wiedzy o technikach programistycznych, powinien natomiast znać podstawy obsługi i administracji wykorzystywanego przez siebie systemu operacyjnego, takie jak instalacja oprogramowania, uruchamianie aplikacji czy praca w wierszu poleceń. Z pewnością nie są to zbyt wygórowane wymagania.

W książce przedstawiony został stosunkowo szeroki zakres zagadnień, począwszy od podstaw związanych z instalacją niezbędnych narzędzi, przez podstawowe konstrukcje języka, programowanie obiektowe, stosowanie wyjątków, obsługę systemu wejścia-wyjścia, aż po tworzenie aplikacji z interfejsem graficznym.

Materiał prezentowany jest od zagadnień najprostszych do coraz bardziej skomplikowanych — zarówno w obrębie całej książki, jak i poszczególnych lekcji — tak aby przyswojenie wiadomości nie sprawiało kłopotu Czytelnikom dopiero rozpoczynającym swoją przygodę z C#, a jednocześnie pozwalało na sprawne poruszanie się po treści książki osobom bardziej zaawansowanym. Prawie każda lekcja kończy się również zestawem ćwiczeń do samodzielnego wykonania, dzięki którym można w praktyce sprawdzić nabycie umiejętności. Przykładowe rozwiązania ćwiczeń zostały zamieszczone na serwerze FTP wydawnictwa Helion.

Standardy C#

Przedstawiona treść obejmuje najnowszy dostępny w trakcie powstawania książki standard języka C# — 6.0 (.NET 4.6, Visual Studio 2015), choć większość przykładów będzie poprawnie działać nawet w pierwotnych, powstały wiele lat temu, a obecnie rzadko spotykanych wersjach 1.0 i 1.2. Jest również w pełni zgodna z wersją 4.0 (.NET Framework 4, Visual Studio 2010) oraz 5.0 (.NET Framework 4.5, Visual Studio 2012 i 2013). Konstrukcje języka dostępne wyłącznie w wersji 6.0, która obecnie nie jest jeszcze tak mocno rozpowszechniona jak pozostałe, są wyraźnie zaznaczane w opisach.

Rozdział 1.

Zanim zaczniesz programować

Pierwszy rozdział zawiera wiadomości potrzebne do rozpoczęcia nauki programowania w C#. Znajdują się w nim informacje o tym, czym jest język programowania, co jest potrzebne do uruchamiania programów C# oraz jakie narzędzia programistyczne będą niezbędne w trakcie nauki. Zostanie pokazane, jak zainstalować platformę .NET Framework oraz jak używać popularnych pakietów, takich jak Visual Studio, Mono i MonoDevelop. Przedstawiona będzie też struktura prostych programów; nie zostanie także pominięty, zwykle bardzo niedoceniany przez początkujących, temat komentowania kodu źródłowego.

Lekcja 1. Podstawowe koncepcje C# i .NET

Jak to działa?

Program komputerowy to nic innego jak ciąg rozkazów dla komputera. Rozkazy te wyrażamy w języku programowania — w ten sposób powstaje tzw. kod źródłowy. Jednak komputer nie jest w stanie bezpośrednio go zrozumieć. Potrzebujemy więc aplikacji, która przetłumaczy kod zapisany w języku programowania na kod *zrozumiały* dla danego środowiska uruchomieniowego (sprzętu i systemu operacyjnego). Aplikacja taka nazywa się *kompilatorem*, natomiast proces tłumaczenia — *kompilacją*.

W przypadku klasycznych języków kompilowanych powstaje w tym procesie plik pośredni, który musi zostać połączony z dodatkowymi modułami umożliwiającymi współpracę z danym systemem operacyjnym, i dopiero po wykonaniu tej operacji powstaje plik wykonywalny, który można uruchamiać bez żadnych dodatkowych

zabiegów. Proces łączenia nazywamy inaczej *konsolidacją, łączeniem lub linkowaniem* (ang. *link* — łączyć), a program dokonujący tego zabiegu — linkerem. Współczesne narzędzia programistyczne najczęściej wykonują oba zadania (kompilację i łączenie) automatycznie.

C# współpracuje jednak z platformą .NET. Cóż to takiego? Otóż .NET to właśnie środowisko uruchomieniowe (tzw. CLR — ang. *Common Language Runtime*) wraz z zestawem bibliotek (tzw. FCL — ang. *Framework Class Library*) umożliwiających uruchamianie programów. Tak więc program pisany w technologii .NET — czy to w C#, Visual Basicu, czy innym języku — nie jest kompilowany do kodu natywnego danego procesora (czyli bezpośrednio zrozumiałego dla danego procesora), ale do kodu pośredniego (przypomina to w pewnym stopniu *byte-code* znany z Javy). Tenże *kod pośredni* (tak zwany CIL — ang. *Common Intermediate Language*) jest wspólny dla całej platformy. Innymi słowy, kod źródłowy napisany w dowolnym języku zgodnym z .NET jest tłumaczony na wspólny język *zrozumiały dla środowiska uruchomieniowego*. Pozwala to między innymi na bezpośrednią i bezproblemową współpracę modułów i komponentów pisanych w różnych językach¹.

Ponieważ kod pośredni nie jest zrozumiały dla procesora, w trakcie uruchamiania aplikacji środowisko uruchomieniowe dokonuje tłumaczenia z kodu pośredniego na kod natywny. Jest to nazywane komplikacją *just-in-time*, czyli *kompilacją w trakcie wykonania*. To dlatego, aby uruchomić program przeznaczony dla .NET, w systemie musi być zainstalowany pakiet .NET Framework (to właśnie implementacja środowiska uruchomieniowego) bądź inne środowisko tego typu, jak np. Mono.

Narzędzia

Najpopularniejszym środowiskiem programistycznym służącym do tworzenia aplikacji C# jest produkowany przez firmę Microsoft pakiet Visual C#, niegdyś dostępny jako osobny produkt, a obecnie jako część pakietu Visual Studio. Oczywiście wszystkie prezentowane w niniejszej książce przykłady mogą być tworzone przy użyciu tego właśnie produktu. Korzystać można z darmowej edycji Visual Studio Community (dawniej: Express) dostępnej pod adresem <http://www.visualstudio.com/>.

Oprócz tego istnieje również darmowy kompilator C# (*csc.exe*), będący częścią pakietu .NET Framework (pakietu tego należy szukać pod adresem <http://msdn.microsoft.com/netframework/> lub <http://www.microsoft.com/net/>). Jest to kompilator uruchamiany w wierszu poleceń, nie oferując więc dodatkowego wsparcia przy budowaniu aplikacji, tak jak Visual Studio, jednak do naszych celów jest całkowicie wystarczający. Z powodzeniem można zatem do wpisywania kodu przykładowych programów używać dowolnego edytora tekstowego, nawet tak prostego jak np. Notatnik (choć z pewnością nie byłby to najlepszy wybór), a do komplikacji — kompilatora *csc.exe*.

¹ W praktyce taka możliwość współpracy wymaga także stosowania wspólnego systemu typów (CTS — ang. *Common Type System*) i wspólnej specyfikacji języka (CLS — ang. *Common Language Specification*).

Na rynku dostępne są również inne narzędzia oferujące niezależne implementacje platformy .NET dla różnych systemów. Najpopularniejsze jest Mono, rozwijane jako produkt open source (<http://www.mono-project.com/>) wraz z narzędziem do budowania aplikacji *MonoDevelop* (<http://monodevelop.com/>). Te narzędzia również mogą być wykorzystywane do nauki C#.

W przypadku korzystania z czystego .NET Framework lub Mono do wpisywania tekstu programów konieczny będzie edytor tekstowy. Wspomniany wyżej Notatnik nie jest do tego celu najwygodniejszy, gdyż nie oferuje takich udogodnień, jak numerowanie wierszy czy kolorowanie składni. Dlatego lepiej użyć edytora przystosowanego do potrzeb programistów, jak np. Notepad++ (dostępny dla Windows, <http://notepad-plus-plus.org/>) czy jEdit (wieloplatformowy, dostępny dla Windows, Linuksa i Mac OS, <http://www.jedit.org/>).

Instalacja narzędzi

.NET Framework

Pakiet .NET Framework jest obecnie standardowym komponentem systemu Windows, więc wystarczy odnaleźć go na swoim komputerze. W tym celu należy sprawdzić, czy istnieje katalog:

```
\windows\Microsoft.NET\Framework\
```

(gdzie *windows* oznacza katalog systemowy), a w nim podkatalogi (podkatalog) o przykładowych nazwach:

```
v1.0.3705  
v1.1.4322  
v2.0.50727  
v3.0  
v3.5  
v4.0.30319
```

oznaczających kolejne wersje platformy .NET. Jeśli takie katalogi są obecne, oznacza to, że platforma .NET jest zainstalowana. Najlepiej gdyby była to wersja 3.5, 4.0.XXXXX lub wyższa. Jeśli jej nie ma, w każdej chwili da się ją doinstalować. Odpowiedni pakiet można znaleźć pod adresami wspomnianymi w poprzednim podrozdziale. Instalacja nikomu z pewnością nie przysporzy najmniejszego problemu. Po uruchomieniu pakietu i zaakceptowaniu umowy licencyjnej rozpoczęcie się proces instalacji (rysunek 1.1). Po jego zakończeniu będzie już można kompilować i uruchamiać programy w C#.

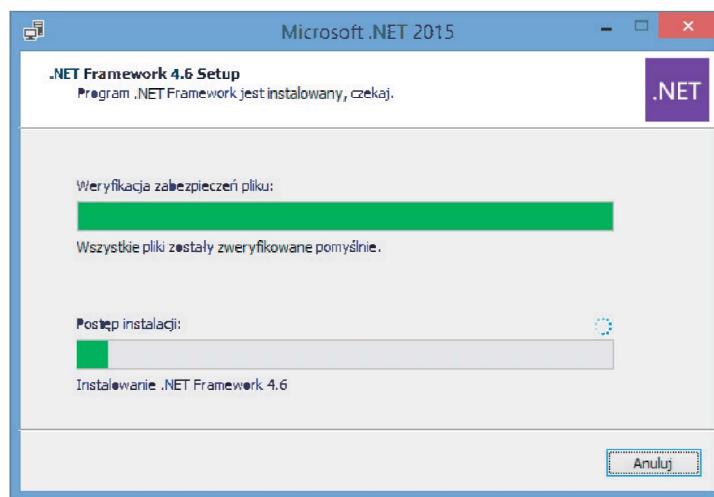
Dodatkowo można również zainstalować pakiet *.NET Framework SDK* (*SDK — Software Development Kit*), zawierający dodatkowe narzędzia i biblioteki. Do wykonywania zadań przedstawionych w książce nie jest to jednak konieczne.

Jeśli chcemy korzystać z kompilatora działającego w wierszu poleceń i konstrukcji języka charakterystycznych dla C# 6.0, konieczne będzie też zainstalowanie pakietu Microsoft Build Tools 2015 lub nowszego². Kompilator *csc.exe* dostępny wraz z platformą .NET nie uwzględnia bowiem tej wersji języka.

² W trakcie powstawania książki pakiet ten był dostępny pod adresem <http://www.microsoft.com/en-us/download/details.aspx?id=48159>.

Rysunek 1.1.

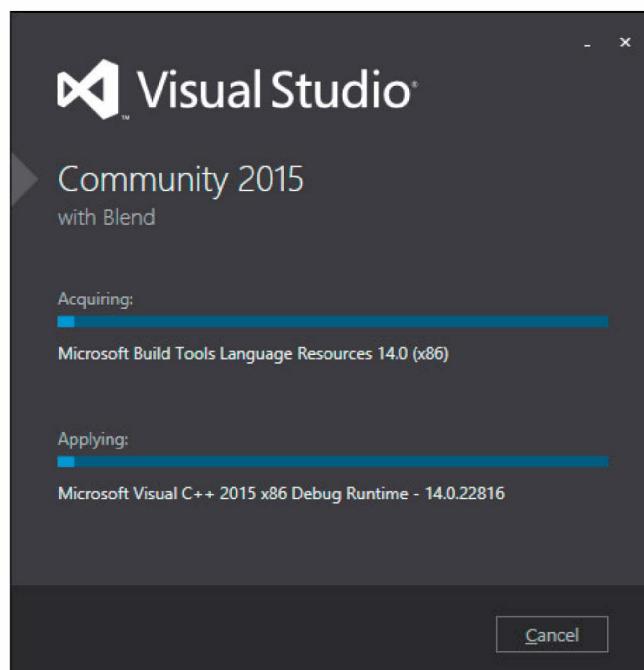
Postęp instalacji pakietu .NET Framework 4.6

**Visual Studio**

Instalacja bezpłatnego pakietu Visual Studio Community przebiega podobnie jak w przypadku każdej innej aplikacji dla systemu Windows. Kolejne okna umożliwiają wybór typowych opcji instalacyjnych, w tym ustalenie katalogu docelowego. W większości przypadków nie ma jednak potrzeby ich zmieniań (wystarczy kliknąć przycisk *Next*). Po kliknięciu przycisku *Install* rozpocznie się proces instalacji (rysunek 1.2).

Rysunek 1.2.

Postępy instalacji pakietu Visual Studio

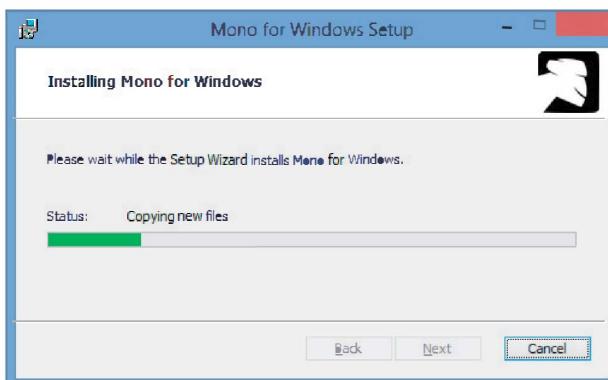


Mono

Mono to platforma rozwijana przez niezależnych programistów, pozwalająca uruchamiać aplikacje pisane dla .NET w wielu różnych systemach, takich jak Linux, Solaris, Mac OS X, Windows, Unix. Zawiera oczywiście również kompilator C#. Pakiet instalacyjny można pobrać pod adresem <http://www.mono-project.net>. Jego instalacja przebiega podobnie jak w przypadku innych aplikacji (rysunek 1.3).

Rysunek 1.3.

Instalacja platformy Mono

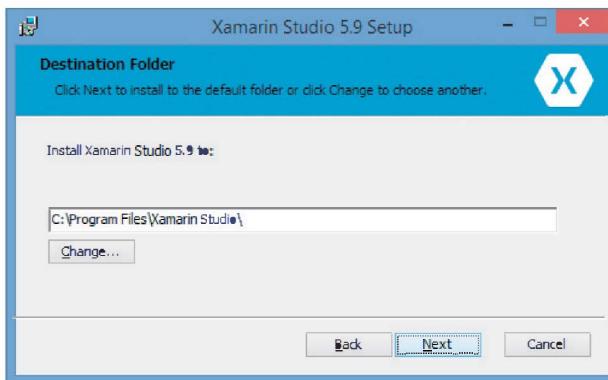


MonoDevelop (Xamarin Studio)

MonoDevelop (obecnie część pakietu Xamarin Studio) to zintegrowane środowisko programistyczne (IDE — ang. *Integrated Development Environment*) umożliwiające tworzenie aplikacji w C# (a także w innych językach dla platformy .NET). Jest dostępne pod adresem <http://monodevelop.com/>. Przed instalacją MonoDevelop należy zainstalować pakiet GTK# (o ile nie ma go w systemie), który również jest dostępny pod wymienionym adresem. Instalacja przebiega typowo. W jej trakcie można wybrać katalog, w którym zostaną umieszczone pliki pakietu (rysunek 1.4), jednak z reguły opcja domyślna jest wystarczająca, więc nie ma potrzeby jej zmieniać. Cała procedura instalacyjna sprawdza się zatem zwykle do klikania przycisków *Next*.

Rysunek 1.4.

Wybór katalogu docelowego



Lekcja 2. Pierwsza aplikacja, komplikacja i uruchomienie programu

Większość kursów programowania zaczyna się od napisania prostego programu, którego zadaniem jest jedynie wyświetlenie napisu na ekranie komputera. To bardzo dobry początek pozwalający oswoić się ze strukturą kodu, wyglądem instrukcji oraz procesem komplikacji. Zobaczmy zatem, jak wygląda taki program w C#. Jest on zaprezentowany na listingu 1.1.

Listing 1.1. Pierwsza aplikacja w C#

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Mój pierwszy program!");
    }
}
```

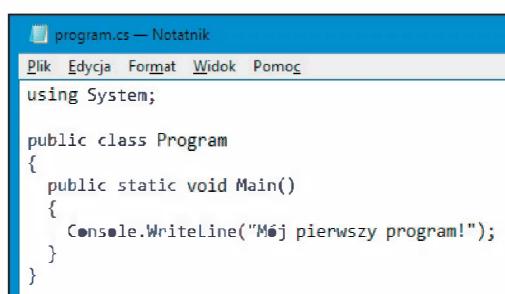
Dla osób początkujących wygląda to zapewne groźnie i zawile, w rzeczywistości nie ma tu jednak nic trudnego. Nie wnikając w poszczególne składowe tego programu, spróbujmy przetworzyć go na plik z kodem wykonywalnym, który da się uruchomić w systemie operacyjnym. Sposób wykonania tego zadania zależy od tego, którego z narzędzi programistycznych chcemy użyć. Zobaczmy więc, jak to będzie wyglądało w każdym z przypadków.

.NET Framework

Korzystając z systemowego Notatnika (rysunek 1.5) lub lepiej innego edytora tekstowego, zapisujemy program z listingu 1.1 w pliku o nazwie *program.cs* lub innej, bardziej nam odpowiadającej. Przyjmuje się jednak, że pliki zawierające kod źródłowy (tekst programu) w C# mają rozszerzenie *cs*. Jeżeli użyjemy programu takiego jak Notepad++ (rysunek 1.6), po zapisaniu pliku w edytorze elementy składowe języka otrzymają różne kolory oraz grubości czcionki, co zwiększy czytelność kodu.

Rysunek 1.5.

Tekst pierwszego programu w Notatniku



Rysunek 1.6.

Tekst pierwszego programu w edytorze Notepad++

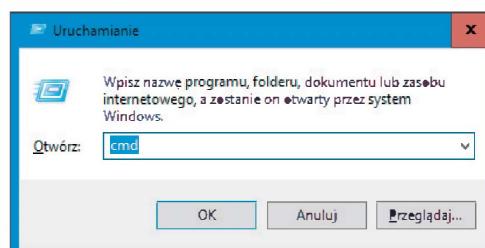
```
using System;
public class Program
{
    public static void Main()
    {
        Console.WriteLine("Mój pierwszy program!");
    }
}
```

length : 133 lines : Ln:1 Col:1 Sel:0|0 Dos\Windows ANSI INS

Uruchamiamy następnie wiersz poleceń (okno konsoli). W tym celu wykorzystujemy kombinację klawiszy *Windows+R*³, w polu *Uruchom* wpisujemy cmd lub cmd.exe⁴ i klikamy przycisk *OK* lub naciskamy klawisz *Enter* (rysunek 1.7). W systemach 2000 i XP (a także starszych) pole *Uruchom* jest dostępne bezpośrednio w menu *Start*⁵. Okno konsoli w systemie Windows 8 zostało przedstawione na rysunku 1.8 (w innych wersjach wygląda bardzo podobnie).

Rysunek 1.7.

Uruchamianie polecenia cmd w Windows 8



C:\Windows\system32\cmd.exe

Microsoft Windows [Version 10.0.10240]
(c) 2015 Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Users\main>

Rysunek 1.8. Okno konsoli (wiersz poleceń) w systemie Windows 8

Korzystamy z kompilatora uruchamianego w wierszu poleceń — csc.exe (dostępnego po zainstalowaniu w systemie pakietu .NET Framework, a także Visual Studio). Jako parametr należy podać nazwę pliku z kodem źródłowym; wywołanie będzie więc miało postać:

ścieżka dostępu do kompilatora\csc.exe program.cs

³ Klawisz funkcyjny *Windows* jest też opisywany jako *Start*.

⁴ W starszych systemach (Windows 98, Me) należy uruchomić aplikację *command.exe* (*Start/Uruchom/command.exe*).

⁵ W systemach Windows Vista i 7 pole *Uruchom* standardowo nie jest dostępne w menu startowym, ale można je do niego dodać, korzystając z opcji *Dostosuj*.

Na przykład:

```
c:\windows\Microsoft.NET\Framework\v4.0.30319\csc.exe program.cs
```

Nie można przy tym zapomnieć o podawaniu nazwy pliku zawsze z rozszerzeniem.

W celu ułatwienia sobie pracy warto dodać do zmiennej systemowej path ścieżkę do- stępu do pliku wykonywalnego kompilatora, np. wydając (w wierszu poleceń) polecenie:

```
path=%path%; "c:\windows\Microsoft.NET\Framework\v4.0.30319\"
```

Jeśli chcemy skorzystać z kompilatora dla C# 6.0 dostępnego po zainstalowaniu pa- kietu Microsoft Build Tools 2015, zamiast powyższej ścieżki należy użyć następującej (dostosowując odpowiednio literę dysku):

```
c:\Program Files\MSBuild\14.0\Bin
```

Po wykonaniu opisanych czynności komplikacja będzie mogła być wykonywana za pomocą polecenia:

```
csc.exe program.cs
```

Jeżeli plik *program.cs* nie znajduje się w bieżącym katalogu, konieczne będzie podanie pełnej ścieżki dostępu do pliku, np.:

```
csc.exe c:\cs\program.cs
```

W takiej sytuacji lepiej jednak zmienić katalog bieżący. W tym celu używa się pole- cenia cd, np.:

```
cd c:\cs\
```

Po komplikacji powstanie plik wynikowy *program.exe*, który można uruchomić w wier- szu poleceń (w konsoli systemowej), czyli tak jak każdą inną aplikację, o ile oczywiście został zainstalowany wcześniej pakiet .NET Framework. Efekt komplikacji i uru- chomienia jest widoczny na rysunku 1.9 (na rysunku pokazano również wspomniane wyżej komendy ustanawiające nową wartość zmiennej środowiskowej PATH oraz zmieniające katalog bieżący, które wystarczy wykonać raz dla jednej sesji konsoli; plik *program.cs* został umieszczony w katalogu *c:\cs*).

The screenshot shows a Windows Command Prompt window with the title 'C:\WINDOWS\system32\cmd.exe'. The window displays the following text:

```
Microsoft Windows [Version 10.0.10240]
(c) 2015 Microsoft Corporation. Wszelkie prawa zastrzeżone.

C:\Users\main>path=%path%;c:\Windows\Microsoft.NET\Framework\v4.0.30319\
C:\Users\main>cd c:\cs

C:\cs>csc program.cs
Microsoft (R) Visual C# Compiler version 4.6.0079.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

C:\cs>program.exe
Mój pierwszy program!

C:\cs>
```

Rysunek 1.9. Kompilacja i uruchomienie programu

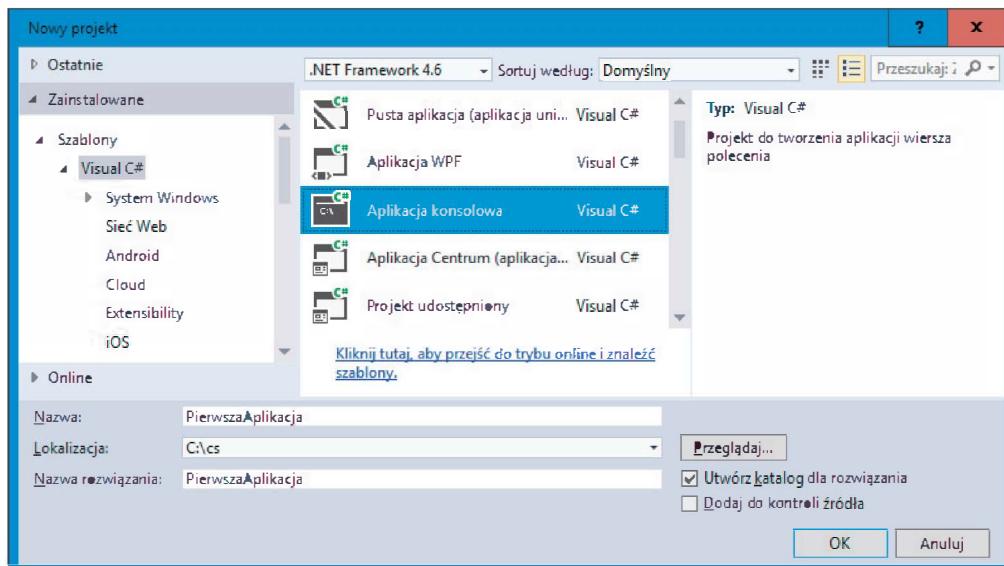
Kompilator *csc.exe* pozwala na stosowanie różnych opcji umożliwiających ingerencję w proces komplikacji; są one omówione w tabeli 1.1.

Tabela 1.1. Wybrane opcje kompilatora *csc*

Nazwa opcji	Forma skrócona	Parametr	Znaczenie
/out:	–	nazwa pliku	Nazwa pliku wynikowego — domyślnie jest to nazwa pliku z kodem źródłowym.
/target:	/t:	exe	Tworzy aplikację konsolową (opcja domyślna).
/target:	/t:	winexe	Tworzy aplikację okienkową.
/target:	/t:	library	Tworzy bibliotekę.
/platform:	/p:	x86, Itanium, x64, anycpu	Określa platformę sprzętowo-systemową, dla której ma być generowany kod. Domyślnie jest to każda platforma (anycpu).
/recurse:	–	maska	Kompiluje wszystkie pliki (z katalogu bieżącego oraz katalogów podrzędnych), których nazwa jest zgodna z maską.
/win32icon:	–	nazwa pliku	Dodała do pliku wynikowego podaną ikonę.
/debug	–	+ lub –	Włącza (+) oraz wyłącza (–) generowanie informacji dla debugera.
/optimize	/o	+ lub –	Włącza (+) oraz wyłącza (–) optymalizację kodu.
/warnaserror	–	–	Włącza tryb traktowania ostrzeżeń jako błędów.
/warn:	/w:	od 0 do 4	Ustawia poziom ostrzeżeń.
/nowarn:	–	lista ostrzeżeń	Wyłącza generowanie podanych ostrzeżeń.
/help	/?	–	Wyświetla listę opcji.
/nologo	–	–	Nie wyświetla noty copyright.

Visual Studio

Uruchamiamy Visual Studio, a następnie na stronie startowej klikamy ikonę *Nowy projekt* (*New Project*) bądź z menu *Plik (File)* wybieramy pozycję *Nowy projekt (New Project)* lub wykorzystujemy kombinację klawiszy *Ctrl+Shift+N*. Na ekranie pojawi się okno wyboru typu projektu (rysunek 1.10). Z zakładki (*Szablony/VisualC#/Windows*) wybieramy *Aplikacja konsolowa (Console Application)*, czyli aplikację konsolową, działającą w wierszu poleceń, a w polu tekstowym *Nazwa (Name)* wpisujemy nazwę projektu, np.: *PierwszaAplikacja* (lub pozostawiamy nazwę domyślną projektu zaproponowaną przez Visual Studio). W tym miejscu można też zmienić lokalizację projektu na dysku (pole *Lokalizacja (Location)*), np. na *c:\cs*. Domyślną lokalizacją projektu jest podkatalog *visual studio 2015\Projects* (dla wersji 2015) w katalogu dokumentów użytkownika (zwykle: *c:\users\nazwa_uzytkownika\documents*). Klikamy przycisk *OK*.



Rysunek 1.10. Okno wyboru typu projektu

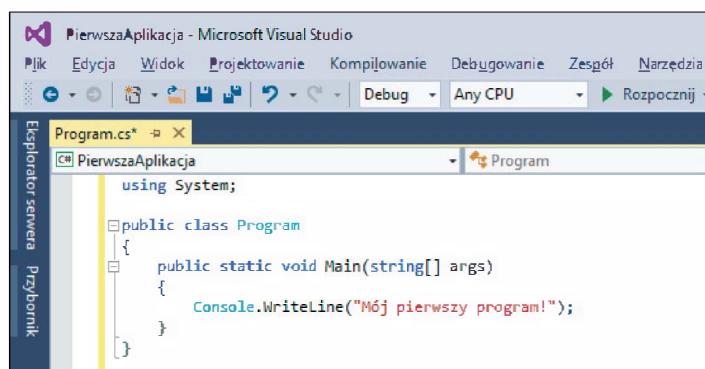
Wygenerowany zostanie wtedy szkielet aplikacji (rysunek 1.11). Nie będziemy jednak z niego korzystać. Usuwamy więc istniejący tekst i w to miejsce wpisujemy nasz własny kod z listingu 1.1 (rysunek 1.12).

Rysunek 1.11.
Szkielet aplikacji
wygenerowanej
przez Visual Studio

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

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

Rysunek 1.12.
Tekst programu z listingu 1.1 w środowisku Visual Studio



The screenshot shows the Microsoft Visual Studio interface with the title bar "PierwszaAplikacja - Microsoft Visual Studio". The menu bar includes "Plik", "Edycja", "Widok", "Projektowanie", "Kompilowanie", "Debugowanie", "Zeszyt", and "Narzędzia". The toolbar has icons for file operations like Open, Save, and Build. The status bar shows "Debug Any CPU Rozpocznij". The code editor displays the following C# code:

```
using System;
public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Mój pierwszy program!");
    }
}
```

Kompilacji dokonujemy przez wybranie z menu *Kompilowanie (Build)* pozycji *Kompiluj rozwiązanie (Build Solution)* lub naciśnięcie kombinacji klawiszy *Ctrl+Shift+B* (w starszych wersjach Visual C#: a. należy użyć klawisza *F6*; b. jeżeli menu *Build* nie jest dostępne, można je włączyć przez wybranie z menu *Tools* pozycji *Settings* i *Expert settings*). Plik wynikowy (*PierwszaAplikacja.exe*) znajdzie się w katalogu projektu (jeśli przyjmiemy dane z rysunku 1.10, będzie to katalog *C:\cs\PierwszaAplikacja*), w podkatalogu *PierwszaAplikacja\bin\Release* lub *PierwszaAplikacja\bin\Debug*, w zależności od tego, jaka została wybrana konfiguracja docelowa (pole wyboru z opcjami *Debug* i *Release* widoczne na pasku narzędziowym; opcja *Debug* oznacza program do dalszych testów programistycznych, opcja *Release* — wersję docelową, dystrybucyjną). Pełna ścieżka dostępu do pliku *exe* miałaby więc w tym przykładzie postać:

c:\cs\PierwszaAplikacja\PierwszaAplikacja\bin\Release\PierwszaAplikacja.exe

lub

c:\cs\PierwszaAplikacja\PierwszaAplikacja\bin\Debug\PierwszaAplikacja.exe

Aby ją uruchomić w wierszu poleceń, należy otworzyć konsolę i wpisać podaną ścieżkę dostępu (przyjmując konfigurację docelową w opcji *Release*):

c:\cs\PierwszaAplikacja\PierwszaAplikacja\bin\Release\PierwszaAplikacja.exe

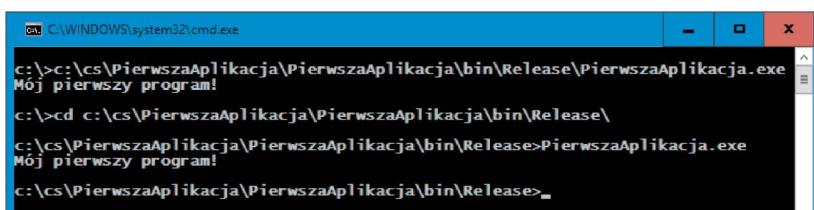
albo najpierw przejść do katalogu z plikiem wykonywalnym, wydając polecenie:

cd c:\cs\PierwszaAplikacja\PierwszaAplikacja\bin\Release\

a następnie uruchomić plik:

PierwszaAplikacja.exe

Efekt użycia obu sposobów jest widoczny na rysunku 1.13.



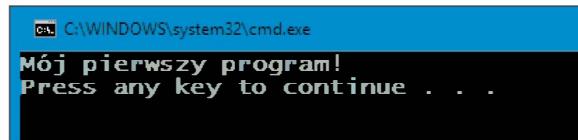
The screenshot shows a Windows Command Prompt window titled "cmd" with the path "C:\WINDOWS\system32\cmd.exe". The command entered is "c:\>c:\cs\PierwszaAplikacja\PierwszaAplikacja\bin\Release\PierwszaAplikacja.exe". The output shows the message "Mój pierwszy program!". The command then changes directory to "c:\>cd c:\cs\PierwszaAplikacja\PierwszaAplikacja\bin\Release\" and runs the same executable again, resulting in the same output "Mój pierwszy program!". Finally, the command ends with "c:\>c:\cs\PierwszaAplikacja\PierwszaAplikacja\bin\Release>".

Rysunek 1.13. Uruchomienie programu *PierwszaAplikacja* w wierszu poleceń

Aplikacja może zostać również uruchomiona bezpośrednio z poziomu Visual Studio. W tym celu należy z menu *Debug* wybrać pozycję *Start without debugging* lub zastosować kombinację klawiszy *Ctrl+F5*. W oknie konsoli oprócz wyników działania programu pojawi się wtedy dodatkowa informacja o potrzebie naciśnięcia dowolnego klawisza (rysunek 1.14).

Rysunek 1.14.

Uruchomienie aplikacji z poziomu Visual Studio



Gdyby taka informacja się nie pojawiła, a okno znikło od razu, uniemożliwiając obserwację wyniku, do kodu źródłowego można dodać na końcu instrukcję:

```
Console.ReadKey();
```

powodującą, że aplikacja będzie czekała zakończeniem działania, aż zostanie naciśnięty dowolny klawisz na klawiaturze. Cały program przyjabyłby wtedy postać widoczną na listingu 1.2.

Listing 1.2. Dodanie instrukcji oczekującej na naciśnięcie klawisza

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Mój pierwszy program!");
        Console.ReadKey();
    }
}
```

Mono

Postępujemy podobnie jak w przypadku korzystania z .NET Framework (omawiana jest wersja Mono dla Windows), to znaczy za pomocą edytora tekstopowego (Notepad++, Notatnik lub inny) zapisujemy program z listingu 1.1 w pliku o nazwie *program.cs*, a następnie w menu startowym odszukujemy grupę Mono (typowo: *Start/Programy (Wszystkie programy)/Mono X.Y.Z for Windows*) i wybieramy *Mono-X.Y.Z Command Prompt*⁶, gdzie *X.Y.Z* to numer wersji, np. *2.10.5*.

Przechodzimy do katalogu, w którym zapisaliśmy plik z kodem źródłowym, np. wydając polecenie:

```
cd c:\cs
```

⁶ Nazwy poszczególnych menu będą się różnić w zależności od wersji Mono oraz Windows.

Następnie korzystamy z kompilatora uruchamianego w wierszu poleceń. Obecnie (począwszy od Mono 2.11) jest to kompilator mcs. Dawniej wybór komendy zależał od tego, z której wersji języka chciało się korzystać:

- ♦ mcs — pierwotnie C# 1.1, następnie wersje 3.5 (mono 2.6) lub 4.0 (mono 2.8),
- ♦ gmcs — C# 2.0,
- ♦ smcs — C# 2.1 (aplikacje Moonlight — implementacja technologii Silverlight),
- ♦ dmcs — C# 4.0.

Wybierzmy zatem wersję współczesną, czyli *mcs.exe*. Jako parametr należy podać nazwę pliku z kodem źródłowym. Wywołanie będzie więc miało postać:

```
mcs.exe program.cs
```

Po komplikacji powstanie plik wynikowy *program.exe*, który można uruchomić w wierszu poleceń (w konsoli systemowej). Jeśli chcemy skorzystać ze środowiska uruchomieniowego .NET, aby uruchomić aplikację, piszemy po prostu:

```
program.exe
```

jeśli natomiast do uruchomienia chcemy wykorzystać środowisko uruchomieniowe Mono, piszemy:

```
mono program.exe
```

Efekt komplikacji i uruchomienia za pomocą obu sposobów został pokazany na rysunku 1.15.

Rysunek 1.15.

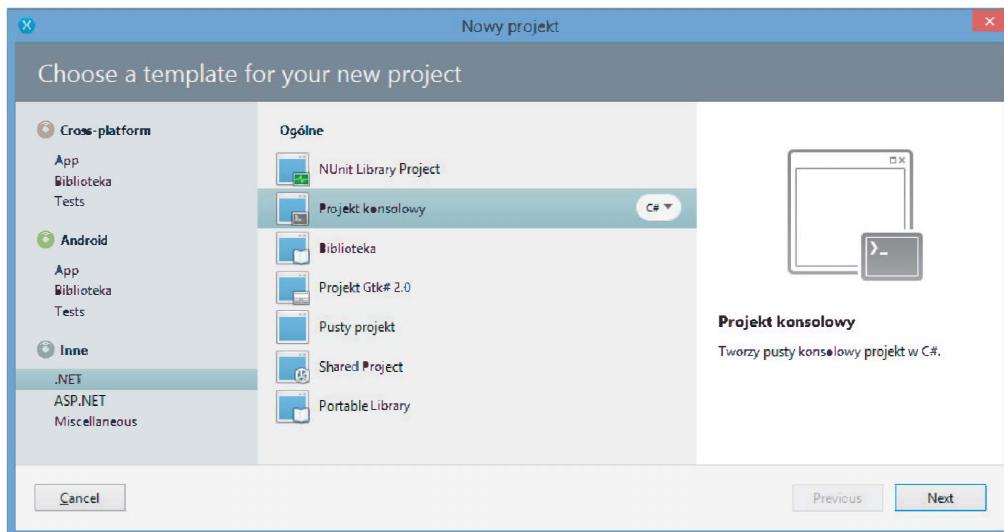
Kompilacja
i uruchamianie
w środowisku Mono

The screenshot shows a terminal window with the title "Open Mono Command Prompt". The command line output is as follows:

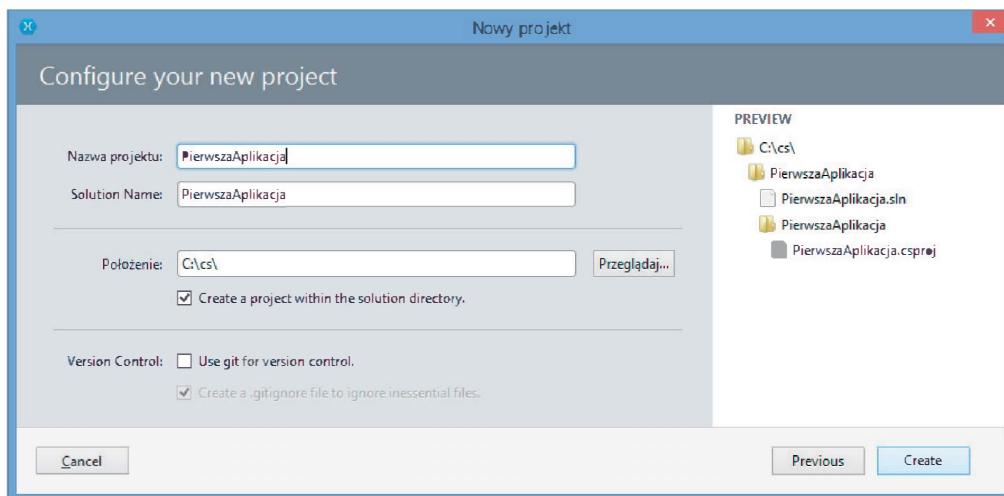
```
Mono version 4.0.1
Prepending 'C:\Program Files\Mono\bin\' to PATH
C:\Program Files\Mono>cd c:\cs\
c:\cs>mcs program.cs
c:\cs>program.exe
Mój pierwszy program!
c:\cs>mono program.exe
Mój pierwszy program!
c:\cs>_
```

MonoDevelop (Xamarin Studio)

Po uruchomieniu pakietu wybieramy z menu *Plik* pozycję *Nowy*, a następnie *Solution* (lub wykorzystujemy kombinację klawiszy *Ctrl+Shift+N*). W oknie, które się wtedy pojawi, wskazujemy .NET oraz *Projekt konsolowy*, upewniając się, że na liście z prawej strony opcji wybrana jest pozycja C# (rysunek 1.16), i klikamy przycisk *Next*. W kolejnym oknie, w polu *Nazwa projektu*, podajemy nazwę projektu, np. *PierwszaAplikacja*, a w polu *Położenie* wskazujemy lokalizację projektu na dysku (np. katalog *c:\cs*) i klikamy przycisk *Create* (rysunek 1.17).

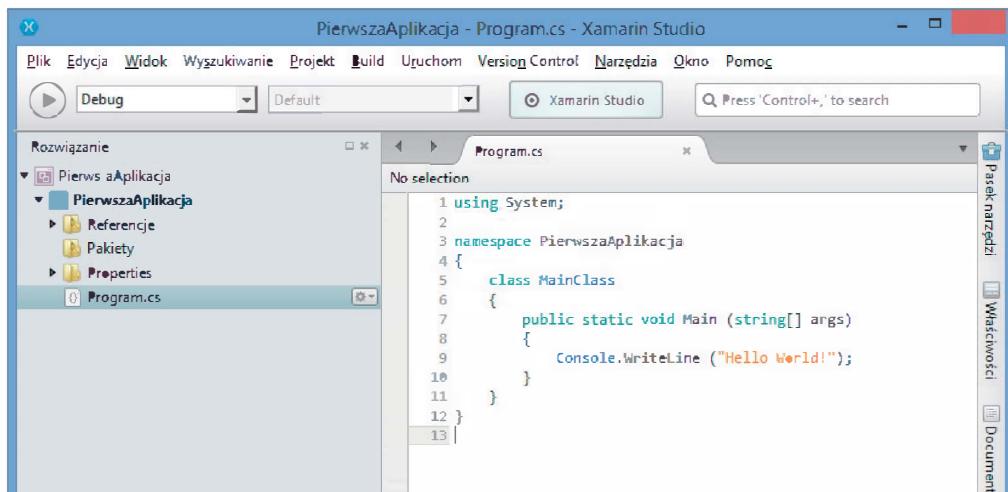


Rysunek 1.16. Wybór typu projektu w MonoDevelop (Xamarin Studio)

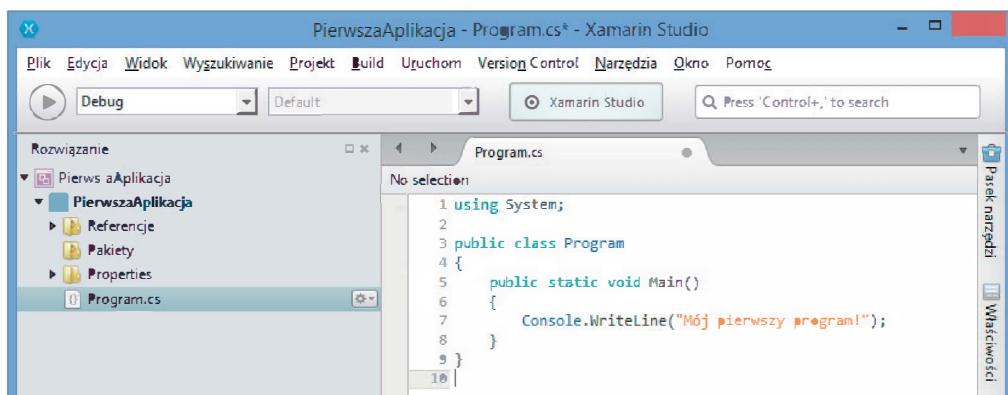


Rysunek 1.17. Określenie nazwy oraz położenia projektu

Zostanie wtedy wygenerowany szkielet kodu aplikacji (rysunek 1.18); podobnie jak miało to miejsce w przypadku Visual Studio, tu również nie będziemy z niego korzystać. Usuwamy więc istniejący tekst (warto zauważyc, że jest bardzo podobny do naszego listingu 1.1) i wpisujemy w to miejsce nasz własny kod z listingu 1.1 (rysunek 1.19).



Rysunek 1.18. Szkielet aplikacji wygenerowany przez Xamarin Studio



Rysunek 1.19. Kod z listingu 1.1 w oknie kodu pakietu Xamarin Studio

Zapisujemy całość na dysku, wybierając z menu *Plik* pozycję *Zapisz wszystko* bądź wciskając kombinację klawiszy *Ctrl+Shift+S*. Kompilacji dokonujemy przez wybranie z menu *Build* pozycji *Build PierwszaAplikacja* lub naciśnięcie klawisza *F7*. Plik wynikowy (*PierwszaAplikacja.exe*) znajdzie się w katalogu projektu (jeśli przyjmiemy dane z rysunku 1.17, będzie to katalog *C:\cs\PierwszaAplikacja\PierwszaAplikacja*), w podkatalogu *\Bin\Debug* (jeżeli bieżącą konfiguracją jest *Debug*; jest to opcja domyślna) lub *\Bin\Release* (jeżeli bieżącą konfiguracją jest *Release*)⁷. Pełna ścieżka dostępu do pliku *exe* miałaby więc w tym przykładzie postać:

`C:\cs\PierwszaAplikacja\PierwszaAplikacja\Bin\Debug\PierwszaAplikacja.exe`

⁷ Bieżącą konfigurację można zmienić za pomocą menu *Projekt/Aktywna konfiguracja* lub korzystając z listy rozwijanej widocznej w lewym górnym rogu okna.

Aby uruchomić aplikację w wierszu poleceń, należy otworzyć konsolę i wpisać podaną ścieżkę dostępu lub najpierw przejść do katalogu z plikiem wykonywalnym, wydając polecenie:

```
cd C:\cs\PierwszaAplikacja\PierwszaAplikacja\Bin\Debug\
```

a następnie uruchomić plik:

```
PierwszaAplikacja.exe
```

Efekt użycia obu sposobów będzie taki sam jak na rysunku 1.13, w podrozdziale „.NET Framework” (o ile użyte zostały te same ścieżki dostępu).

Aplikacja może zostać również uruchomiona bezpośrednio z poziomu środowiska MonoDevelop. Aby uruchomić program, należy z menu *Uruchom* wybrać pozycję *Start Without Debugging* (ewentualnie *Start Debugging*, co będzie się jednak wiązało z dodatkowym uruchomieniem procesu nadzorczeego — debugera) lub użyć kombinacji klawiszy *Ctrl+F5*. Efekt będzie taki sam jak w przypadku Visual Studio (rysunek 1.14). Gdyby się okazało, że okno konsoli zostanie od razu zamknięte, co uniemożliwi zaobserwowanie wyniku, w kodzie źródłowym należy zastosować dodatkową instrukcję:

```
Console.ReadKey();
```

(tak jak na listingu 1.2).

Struktura kodu

Struktura prostego programu w C# wygląda jak na listingu 1.3. Na jej dokładne wyjaśnienie przyjdzie czas w rozdziale 3., omawiającym podstawy technik obiektowych. Przyjmijmy więc, że właśnie tak ma to wyglądać, a w miejscu oznaczonym // należy wpisywać instrukcje do wykonania. Taka struktura będzie wykorzystywana w rozdziale 2., omawiającym podstawowe konstrukcje języka C#.

Listing 1.3. Struktura programu w C#

```
using System;

public class NazwaKlasy
{
    public static void Main()
    {
        // tutaj instrukcje do wykonania
    }
}
```

Ogólnie możemy tu wyróżnić dyrektywę `using`, określającą, z jakiej przestrzeni nazw chcemy korzystać, publiczną klasę `NazwaKlasy` (na listingach 1.1 i 1.2 miała ona nazwę `Program`) oraz funkcję `Main`, od której zaczyna się wykonywanie kodu programu.

Lekcja 3. Komentarze

W większości języków programowania znajdziemy konstrukcję komentarzy, dzięki którym można opisywać kod źródłowy w języku naturalnym. Innymi słowy, służą one do wyrażenia, co programista miał na myśli, stosując daną instrukcję programu. Choć początkowo komentowanie kodu źródłowego programu może wydawać się zupełnie niepotrzebne, okazuje się, że jest to bardzo pożyteczny nawyk. Nierzadko bowiem bywa, że po pewnym czasie sam programista ma problemy z analizą napisanego przez siebie programu, nie wspominając już o innych osobach, które miałyby wprowadzać poprawki czy modyfikacje.

W C# istnieją trzy rodzaje komentarzy. Są to:

- ♦ komentarz blokowy,
- ♦ komentarz liniowy (wierszowy),
- ♦ komentarz XML.

Komentarz blokowy

Komentarz blokowy rozpoczyna się od znaków `/*` i kończy znakami `*/`. Wszystko, co znajduje się pomiędzy nimi, jest traktowane przez kompilator jako komentarz i pomijane w procesie komilacji. Przykład takiego komentarza jest widoczny na listingu 1.4.

Listing 1.4. Użycie komentarza blokowego

```
using System;

public class Program
{
    public static void Main()
    {
        /*
         * To mój pierwszy program w C#.
         * Wyświetla on na ekranie napis.
         */
        Console.WriteLine("Mój pierwszy program!");
    }
}
```

Umiejscowienie komentarza blokowego jest praktycznie dowolne. Co ciekawe, może on znaleźć się nawet w środku instrukcji (pod warunkiem, że nie przedzielimy żadnego słowa). Przykład takiego zapisu znajduje się na listingu 1.5. Jest to możliwe dlatego, że zgodnie z tym, co zostało napisane wyżej, wszystko, co znajduje się między znakami `/*` i `*/` (oraz same te znaki), jest ignorowane przez kompilator. Należy jednak pamiętać, że to raczej ciekawostka — w praktyce zwykle nie ma potrzeby stosowania tego typu konstrukcji.

Listing 1.5. Komentarz blokowy wewnątrz instrukcji

```
using System;

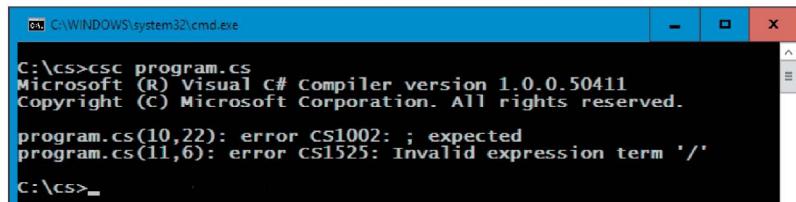
public class Program
{
    public /*komentarz*/ static void Main()
    {
        Console.WriteLine /*komentarz*/ ("To jest napis.");
    }
}
```

Komentarzy blokowych nie wolno zagnieździć, to znaczy jeden nie może znaleźć się w środku drugiego (w takiej sytuacji kompilator nie jest w stanie prawidłowo rozpoznać końców komentarzy). Jeśli zatem spróbujemy wykonać komplikację kodu przedstawionego na listingu 1.6, kompilator zgłosi błąd — zwykle w postaci serii komunikatów. Efekt próby komplikacji kodu z listingu 1.6 jest widoczny na rysunku 1.20.

Rysunek 1.20.

Próba

zagnieżdżenia
komentarza
blokowego
powoduje błąd
kompilacji

**Listing 1.6.** Zagnieżdżenie komentarzy blokowych

```
using System;

public class Program
{
    public static void Main()
    {
        /*
         * Komentarzy blokowych nie
         * w tym miejscu wystąpi błąd*
         * wolno zagnieździć.
        */
        Console.WriteLine("To jest napis.");
    }
}
```

Komentarz liniowy

Komentarz liniowy zaczyna się od znaków // i obowiązuje do końca danej linii programu. To znaczy wszystko, co występuje po tych dwóch znakach aż do końca bieżącej linii, jest ignorowane przez kompilator. Przykład wykorzystania takiego komentarza zilustrowano na listingu 1.7.

Listing 1.7. Użycie komentarza liniowego

```
using System;

public class Program
{
    public static void Main()
    {
        //Teraz wyświetlamy napis.
        Console.WriteLine("To jest napis.");
    }
}
```

Komentarza tego typu nie można oczywiście użyć w środku instrukcji, gdyż wtedy jej część stałaby się komentarzem i powstałby błąd komilacji. Można natomiast w środku komentarza liniowego wstawić komentarz blokowy, o ile zaczyna się i kończy w tej samej linii; konstrukcja taka wyglądałaby następująco:

```
// komentarz /* komentarz blokowy */ liniowy
```

Jest to dopuszczalne i zgodne z regułami składni języka, choć w praktyce niezbyt przydatne. Komentarz liniowy może też znaleźć się w środku komentarza blokowego:

```
/*
//Ta konstrukcja jest po prawna.
*/
```

Komentarz XML

Komentarz XML zaczyna się od znaków ///, po których powinien nastąpić znacznik XML wraz z jego treścią. Komentarze tego typu są przydatne, gdyż na ich podstawie da się wygenerować opisujący kod źródłowy dokument XML, który może być dalej automatycznie przetwarzany przez inne narzędzia. Rozpoznawane przez komplikator C# (zawarty w .NET Framework i Visual C#) znaczniki XML, które mogą być używane w komentarzach tego typu, zostały przedstawione w tabeli 1.2, natomiast przykład ich użycia jest widoczny na listingu 1.8. Aby wygenerować plik XML z dokumentacją, należy użyć opcji /doc kompilatora csc, np.:

```
csc program.cs /doc:dokumentacja.xml
```

Tabela 1.2. Znaczniki komentarza XML

Znacznik	Opis
<c>	Oznaczenie fragmentu komentarza jako kodu.
<code>	Oznaczenie wielowierszowego fragmentu komentarza jako kodu.
<example>	Oznaczenie przykładu użycia fragmentu kodu.
<exception>	Odniesienie do wyjątku.
<include>	Odniesienie do pliku zewnętrznego, który ma być dołączony do dokumentacji.
<list>	Oznaczenie wyliczenia.
<para>	Oznaczenie akapitu tekstu.

Tabela 1.2. Znaczniki komentarza XML — ciąg dalszy

Znacznik	Opis
<param>	Opis parametru metody.
<paramref>	Oznaczenie, że słowo w opisie odnosi się do parametru.
<permission>	Opis dostępu do składowej.
<remarks>	Opis składowej (np. metody).
<returns>	Opis wartości zwracanej.
<see>	Określenie odnośnika do danego miejsca w dokumentacji.
<seealso>	Określenie odnośnika do danego miejsca w dokumentacji.
<summary>	Opis typu bądź składowej.
<typeparam>	Opis typu uogólnionego (generycznego).
<typeparamref>	Dodatkowe informacje na temat typu uogólnionego.
<value>	Opis właściwości.

Listing 1.8. Użycie komentarza XML

```
using System;

/// <summary>Główna klasa aplikacji</summary>
public class Program
{
    /// <remarks>Metoda startowa aplikacji</remarks>
    public static void Main()
    {
        Console.WriteLine("To jest napis.");
    }
}
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 3.1

Na początku programu z listingu 1.1 wstaw komentarz blokowy opisujący działanie tego programu. Dokonaj komplikacji kodu.

Ćwiczenie 3.2

Wygeneruj dokumentację kodu z listingu 1.8, tak by została zapisana w pliku *program.xml*. Zapoznaj się ze strukturą tego pliku, wczytując go do przeglądarki lub edytora tekstowego.

Rozdział 2.

Elementy języka

C#, podobnie jak i inne języki programowania, zawiera szereg podstawowych instrukcji pozwalających programistom na wykonywanie najróżniejszych operacji programistycznych. Ten rozdział jest im w całości poświęcony. Pojawi się zatem pojęcie zmiennej; zostanie pokazane, w jaki sposób należy deklarować zmienne oraz jakie operacje można na nich wykonywać. Będzie też wyjaśnione, czym są i jak wykorzystywać typy danych. Przy przedstawianiu zmennych nie będą jednak dokładnie omawiane zmienne obiektowe, z którymi będzie można się zapoznać bliżej dopiero w kolejnym rozdziale.

Po omówieniu zmiennych zostaną przedstawione występujące w C# instrukcje sterujące wykonywaniem programu. Będą to instrukcje warunkowe pozwalające wykonywać różny kod w zależności od tego, czy zadany warunek jest prawdziwy, czy fałszywy, oraz pętle, umożliwiające łatwe wykonywanie powtarzających się instrukcji. Ostatnie dwie lekcje rozdziału 2. poświęcone są tablicom, i to zarówno jedno-, jak i wielowymiarowym. Osoby, które znają dobrze takie języki jak C, C++ czy Java, mogą jedynie przejrzeć ten rozdział, gdyż większość podstawowych instrukcji sterujących w C# jest bardzo podobna. Bliżej powinny zapoznać się jedynie z materiałem lekcji 6., omawiającej wyprowadzanie danych na ekran.

Typy danych

Typ danych to określenie rodzaju danych, czyli specyfikacja ich struktury i rodzaju wartości, które mogą przyjmować. W programowaniu typ może odnosić się do zmiennej, stałej, argumentu funkcji, metody, zwracanego wyniku itp. Przykładowo zmienna (tym pojęciem zajmiemy się już w kolejnej lekcji) to miejsce w programie, w którym można przechowywać jakieś dane, np. liczby czy ciągi znaków. Każda z nich ma swoją nazwę, która ją jednoznacznie identyfikuje, oraz typ określający, jakiego rodzaju dane może ona przechowywać. Na przykład zmienność typu `int` może przechowywać liczby całkowite, a zmienność typu `float` — liczby rzeczywiste.

Lekcja 4. Typy danych w C#

W tej lekcji zostanie wyjaśnione, jak można podzielić typy danych oraz jakie typy występują w C#. Będzie przedstawione pojęcie typu wartościowego oraz odnośnikowego, a także to, jakie zakresy wartości są przez te typy reprezentowane. Osoby poczatkujące mogą jedynie pokrótko przejrzeć tę lekcję i przejść do następnej, w której omawiane są pojęcie zmennych i typy wykorzystywane w praktyce, a następnie wracać tu w razie potrzeby. Czytelnicy, którzy znają już inny język programowania, powinni natomiast zwrócić uwagę na różnice występujące między tym językiem a C#.

Typy danych w C#

Typy danych można podzielić na **typy wartościowe** (ang. *value types*), do których zalicza się typy proste (inaczej podstawowe — ang. *primitive types*, *simple types*), wyliczeniowe (ang. *enum types*) i strukturalne¹ (ang. *struct types*) oraz **typy odnośnikowe** (referencyjne — ang. *reference types*), do których należą typy klasowe, interfejsowe, delegacyjne i tablicowe. Nie trzeba się jednak przerażać tą mnogością. Na początku nauki wystarczy zapoznać się z podstawowymi typami: prostymi arytmetycznymi, a także typami `boolean` oraz `string`.

Typy proste

Typy proste można podzielić na arytmetyczne całkowitoliczbowe, arytmetyczne zmienoprzecinkowe, typy `char` i `bool`. Przyjrzymy się im bliżej.

Typy arytmetyczne całkowitoliczbowe

Typy całkowitoliczbowe w C# to:

- ◆ `sbyte`,
- ◆ `byte`,
- ◆ `short`,
- ◆ `ushort`,
- ◆ `int`,
- ◆ `uint`,
- ◆ `long`,
- ◆ `ulong`.

¹ Z formalnego punktu widzenia typy wartościowe proste można uznać za typy strukturalne, tak więc typy strukturalne dzieliłyby się na typy proste wbudowane w język i struktury definiowane przez programistę. Są to jednak rozważania teoretyczne, którymi nie musimy się zajmować.

Zakresy możliwych do przedstawiania za ich pomocą wartości oraz liczby bitów, na których są one zapisywane, przedstawione są w tabeli 2.1. Określenie „ze znakiem” odnosi się do wartości, które mogą być dodatnie lub ujemne, natomiast „bez znaku” — do wartości nieujemnych.

Tabela 2.1. Typy całkowitoliczbowe w C#

Nazwa typu	Zakres reprezentowanych wartości	Znaczenie
sbyte	od -128 do 127	8-bitowa liczba ze znakiem
byte	od 0 do 255	8-bitowa liczba bez znaku
short	od -32 768 (-2^{15}) do 32 767 ($2^{15}-1$)	16-bitowa liczba ze znakiem
ushort	od 0 do 65 535 ($2^{16}-1$)	16-bitowa liczba bez znaku
int	od -2 147 483 648 (-2^{31}) do 2 147 483 647 ($2^{31}-1$)	32-bitowa liczba ze znakiem
uint	od 0 do 4 294 967 295 ($2^{32}-1$)	32-bitowa liczba bez znaku
long	od -9 223 372 036 854 775 808 (-2^{63}) do 9 223 372 036 854 775 807 ($2^{63}-1$)	64-bitowa liczba ze znakiem
ulong	od 0 do 18 446 744 073 709 551 615 ($2^{64}-1$)	64-bitowa liczba bez znaku

Typy arytmetyczne zmiennoprzecinkowe

Typy zmiennoprzecinkowe, czyli reprezentujące wartości rzeczywiste, z częścią ulamkową, występują tylko w trzech odmianach:

- ◆ float,
- ◆ double,
- ◆ decimal.

Zakres oraz precyzja liczb, jakie można za ich pomocą przedstawić, zaprezentowane są w tabeli 2.2. Typ decimal² służy do reprezentowania wartości, dla których ważniejsza jest precyzja niż maksymalny zakres reprezentowanych wartości (jest tak na przykład w przypadku danych finansowych).

Tabela 2.2. Typy zmiennoprzecinkowe w C#

Nazwa typu	Zakres reprezentowanych liczb	Precyzja
float	od $\pm 1,5 \times 10^{-45}$ do $\pm 3,4 \times 10^{38}$	7 miejsc po przecinku
double	od $\pm 5,0 \times 10^{-324}$ do $\pm 1,7 \times 10^{308}$	15 lub 16 cyfr
decimal	od $\pm 1,0 \times 10^{-28}$ do $\pm 7,9 \times 10^{28}$	28 lub 29 cyfr

² Ten typ bywa też uznawany za oddzielną kategorię typów arytmetycznych, odmienną od całkowitoliczbowych i zmiennoprzecinkowych.

Typ char

Typ **char** służy do reprezentacji znaków, przy czym w C# jest on 16-bitowy i zawiera znaki Unicode (Unicode to standard pozwalający na zapisanie znaków występujących w większości języków świata). Ponieważ kod Unicode to nic innego jak 16-bitowa liczba, czasami zalicza się go również do typów arytmetycznych całkowitoliczbowych. Aby umieścić znak w kodzie programu, należy go ująć w znaki apostrofu³, np.:

```
'a'
```

Pomiędzy tymi znakami można też użyć jednej z sekwencji specjalnych, przedstawionych w tabeli 2.3.

Typ bool

Ten typ określa tylko dwie wartości logiczne: **true** i **false** (prawda i falsz). Są one używane przy konstruowaniu wyrażeń logicznych, porównywaniu danych oraz wskazywaniu, czy dana operacja zakończyła się sukcesem. Uwaga dla osób znających C albo C++: wartości **true** i **false** nie mają przełożenia na wartości liczbowe jak w przypadku wymienionych języków. Oznacza to, że poniższy fragment kodu jest niepoprawny.

```
int zmieniona = 0;
if(zmieniona){
    //instrukcje
}
```

W takim wypadku błąd zostanie zgłoszony już na etapie kompilacji, nie istnieje bowiem domyślna konwersja z typu **int** na typ **bool** wymagany przez instrukcję **if**.

Typy wyliczeniowe

Typ wyliczeniowy jest określany słowem **enum** i pozwala na tworzenie wyliczeń, czyli określonego zbioru wartości, które będą mogły być przyjmowane przez dane tego typu. W najprostszym wypadku schemat utworzenia wyliczenia wygląda następująco:

```
enum nazwa_wyliczenia {element1, element2, ... , elementN};
```

Na przykład:

```
enum Kolory {czervony, zielony, niebieski}
```

W rzeczywistości każde wyliczenie ma własny typ bazowy, a każdy element wyliczenia — wartość tego typu. Domyślnie typem bazowym jest **int**, a elementy są numerowane od 1. Zatem w powyższym przykładzie ciąg **czervony** otrzymał wartość 1, **zielony** — 2, a **niebieski** — 3. Istnieje jednak możliwość samodzielnego określenia zarówno typu bazowego, jak i wartości przypisanych poszczególnym elementom. W tym celu należy zastosować rozszerzoną definicję typu wyliczeniowego w postaci:

```
enum nazwa_typu:typ_bazowy {element1 = wartość1, element2 = wartość2, ...,
                             ↳elementN = wartośćN}
```

³ W rzeczywistości należałoby mówić o znaku zastępczym apostrofu lub pseudoapostrofie; spotykane jest też określenie „apostrof prosty”.

Na przykład:

```
enum Kolory:short {czarny = 10, zielony = 20, niebieski = 30}
```

Taką instrukcję w celu zwiększenia czytelności można też rozbić na kilka linii, np.:

```
enum Kolory:short
{
    czarny = 10,
    zielony = 20,
    niebieski = 30
}
```

Należy pamiętać, że typem bazowym może być taki, który reprezentuje wartości całkowite, czyli: byte, sbyte, short, ushort, int, uint, long i ulong.

Typy strukturalne

Struktury definiowane są za pomocą słowa struct. Przypominają one klasy, choć dotyczą ich pewne ograniczenia. Ten typ danych zostanie omówiony w rozdziale 3.

Ciągi znaków

Do reprezentacji **łańcuchów znakowych**, czyli napisów, służy typ string. Jest on zaliczany do typów prostych, w rzeczywistości jednak należałoby go traktować jako typ referencyjny. Nie wnikając jednak w dyskusje teoretyczne, na początku nauki C# trzeba jedynie wiedzieć, że jeśli chcemy umieścić w programie łańcuch znaków, napis, należy go ująć w cudzysłów, czyli na jego początku i końcu umieścić znaki cudzysłowi⁴, np.:

```
"To jest napis"
```

Warto zwrócić uwagę, że skorzystaliśmy już z takiego zapisu w naszym pierwszym programie (listing 1.1 w lekcji 2.).

W łańcuchach znakowych można stosować sekwencje znaków specjalnych. Zostały one przedstawione w tabeli 2.3.

Tabela 2.3. Sekwencje znaków specjalnych

Sekwencja	Znaczenie	Reprezentowany kod
\a	Sygnał dźwiękowy (ang. alert)	0x0007
\b	Cofnięcie o jeden znak (ang. backspace)	0x0008
\f	Nowa strona (ang. form feed)	0x000C
\n	Nowa linia (ang. new line)	0x000A
\r	Powrót karetki (przesunięcie na początek linii, ang. carriage return)	0x000D

⁴ Ścisłej rzeczą ujmując, należałoby tu mówić o znakach zastępczych cudzysłowu lub o znakach cudzysłowu prostego. W książce będzie jednak stosowane popularne określenie „cudzysłów”.

Tabela 2.3. Sekwencje znaków specjalnych

Sekwencja	Znaczenie	Reprezentowany kod
\t	Znak tabulacji poziomej (ang. <i>horizontal tab</i>)	0x0009
\v	Znak tabulacji pionowej (ang. <i>vertical tab</i>)	0x000B
\"	Znak cudzysłowu	0x0022
\'	Znak apostrofu	0x0027
\\"	Lewy ukośnik (ang. <i>backslash</i>)	0x005C
\xNNNN	Kod znaku w postaci szesnastkowej (heksadecymalnej)	0xNNNN
\uNNNN	Kod znaku w formacie Unicode	0xNNNN
\0	Znak pusty	0x0000

Typy referencyjne

Typy referencyjne (czy też obiektowe, klasowe) odnoszą się do programowania obiektowego⁵, dlatego też zostaną przedstawione dopiero w rozdziale 3.

Zapis wartości (literały)

Literały, czyli **stałe napisowe** (ang. *string constant, literal constant*), to ciągi znaków reprezentujące w kodzie źródłowym programu jawne wartości. Na przykład ciąg znaków 12 jest literałem interpretowanym przez kompilator jako wartość całkowita dodatnia równa 12, zapisana w systemie dziesiętnym. A zatem jeśli chce się umieścić w kodzie jakąś wartość jednego z typów prostych, trzeba wiedzieć, jakiego literała można użyć. Zwykle nie jest to skomplikowane; wiadomo, że gdy napiszemy 12, to chodzi nam o wartość 12, a jak 120 — o wartość 120. Możliwe są jednak różne sposoby zapisu w zależności od tego, jakiego typu danych chcemy użyć.

Literały całkowitoliczbowe

Literały całkowite reprezentują liczby całkowite. Są to zatem ciągi cyfr, które mogą być poprzedzone znakiem plus (+) lub minus (-). Jeżeli ciąg cyfr nie jest poprzedzony żadnym znakiem lub jest poprzedzony znakiem +, reprezentuje wartość dodatnią; jeżeli natomiast jest poprzedzony znakiem -, reprezentuje wartość ujemną. Jeżeli ciąg cyfr zostanie poprzedzony znakami 0x lub 0X, będzie traktowany jako wartość szesnastkowa (heksadecymalna). W zapisie wartości szesnastkowych mogą być wykorzystywane zarówno małe, jak i duże literały alfabetu od A do F. Poniżej przedstawione zostały przykładowe literały całkowite.

⁵ Purystycy językowi powiedzieliby „zorientowanego obiektowo”; uważam jednak, że potoczny i często stosowany termin „programowanie obiektowe” (choć teoretycznie oba te wyrażenia mają nieco inne znaczenia) jest w pełni adekwatny do tej techniki, i taki też będzie stosowany w tej książce.

123	dodatnia całkowita wartość dziesiętna 123
-123	ujemna całkowita wartość dziesiętna -123
0xFF	dodatnia całkowita wartość szesnastkowa równa 255 dziesiętnie
-0x0f	ujemna całkowita wartość szesnastkowa równa -15 dziesiętnie

Typ danych zostanie rozpoznany automatycznie na podstawie minimalnego dopuszczalnego zakresu, poczynając od `int`, a kończąc na `long`. Oznacza to, że wartość 123 będzie miała typ `int`, bo nie przekracza dopuszczalnego zakresu dla typu `int`, ale wartość 4 294 967 296 będzie miała typ `long`, ponieważ przekracza dopuszczalny zakres dla `uint`, ale nie przekracza zakresu dla `long` (por. tabela 2.1).

Jeżeli wartości ma być nadany konkretny typ, należy dodać do niej jeden z następujących przyrostków (sufiksów):

- ◆ `U` lub `u` — wartość będzie traktowana jako `unsigned int` (jeśli mieści się w zakresie tego typu) lub `long`,
- ◆ `L` lub `l` — wartość będzie traktowana jako `long long` (jeśli mieści się w zakresie tego typu) lub `ulong`⁶,
- ◆ `UL` lub `ul` — wartość będzie traktowana jako `ulong long`⁷.

Przykładem takiego zapisu będzie `123L` czy też `48ul`.

Literły zmiennoprzecinkowe

Literaly rzeczywiste reprezentują liczby rzeczywiste (**zmiennoprzecinkowe**, **zmiennopozycyjne**). Są to zatem ciągi cyfr zawierające separator dziesiętny (znak kropki) lub zapisane w notacji wykładniczej z literą E bądź e (jak w podanych niżej przykładach). Mogą być poprzedzone znakiem plus (+) lub minus (-). Jeżeli przed ciągiem cyfr nie występuje żaden dodatkowy znak lub też występuje znak +, literal reprezentuje wartość dodatnią, jeśli natomiast przed ciągiem cyfr występuje znak -, literal reprezentuje wartość ujemną.

Literaly rzeczywiste mogą być zapisywane w notacji wykładniczej, w postaci $X.YeZ$, gdzie X to część całkowita, Y część dziesiętna, natomiast Z to wykładnik potęgi liczby 10 (można używać zarówno malej, jak i wielkiej litery e). Zapis taki oznacza to samo co $X.Y \times 10^Z$. Oto przykłady literalów rzeczywistych:

1.1	dodatnia wartość rzeczywista 1,1
-1.1	ujemna wartość rzeczywista -1,1
0.2e100	dodatnia wartość rzeczywista 20

⁶ Ze względu na czytelność kodu zaleca się stosowanie wielkiej litery L.

⁷ Dopuszcza się stosowanie dowolnej kombinacji małych i wielkich liter `u` i `l` (`UL`, `U1`, `uL`, `LU`, `Lu`, `1U`, `1u`).

0.1E2	dodatnia wartość rzeczywista 10
2e-2	dodatnia wartość rzeczywista 0,02
-3.4E-1	ujemna wartość rzeczywista -0,34

Wartość opisana w powyższy sposób otrzymuje standardowo typ `double`. Możliwe jest określenie konkretnego typu za pomocą jednego z sufiksów:

- ◆ F lub f — wartość będzie traktowana jako `float`,
- ◆ D lub d — wartość będzie traktowana jako `double`,
- ◆ M lub m — wartość będzie traktowana jako `decimal`.

Przykładem takiego zapisu będzie `1.2d` czy też `22.54M`.

Literaty znakowe (dla typu `char`)

Literaty znakowe pozwalają na zapisywanie pojedynczych znaków. Znak, który ma się znaleźć w kodzie programu, należy objąć znakami apostrofu prostego, np.:

`'a'`

Dopuszczalne jest stosowanie sekwencji specjalnych, przedstawionych w tabeli 2.3 (mimo że sekwencja jest zapisywana za pomocą co najmniej dwóch znaków, jest traktowana jako jeden).

Literaty logiczne (dla typu `bool`)

Literaty logiczne występują jedynie w dwóch postaciach. Pierwsza to słowo `true`, oznaczające prawdę, a druga to słowo `false`, czyli fałsz. W obu przypadkach należy używać tylko małych liter.

Literaty łańcuchowe (dla typu `String`)

Literaty łańcuchowe umożliwiają umieszczanie w kodzie ciągów znaków (napisów). Ciąg należy ująć w znaki cudzysłowu prostego, tak jak zostało to omówione w poprzedniej części lekcji. Można stosować sekwencje specjalne z tabeli 2.3.

Literał null

Literał `null` jest stosowany jako określenie wartości typu specjalnego `null`. Zapisywany jest jako ciąg znaków `null`.

Zmienne

Zmienne można traktować jako konstrukcje programistyczne, które pozwalają na przechowywanie różnych danych niezbędnych w trakcie działania aplikacji. Każda zmienna ma swoją nazwę oraz typ. Nazwa to jednoznaczny identyfikator, dzięki któremu istnieje możliwość odwoływania się do zmiennej w kodzie programu, natomiast typ określa, jakiego rodzaju dane zmienna może przechowywać. Podstawowe typy danych zostały omówione w lekcji 5. Lekcja 6. jest poświęcona problemowi deklaracji i przypisywania wartości zmiennym, lekcja 7. — wyświetlanemu wartości zmiennych (ale także znaków specjalnych i napisów) na ekranie, natomiast w najdłuższej lekcji w tym podrozdziale, lekcji 8., zostanie pokazane, jakie operacje (arytmetyczne, logiczne, bitowe) można wykonywać na zmiennych.

Lekcja 5. Deklaracje i przypisania

Lekcja 5. jest poświęcona deklaracjom oraz przypisywaniu zmiennym wartości. Przedstawiono w niej, jak tworzy się zmienne i jaki ma to związek z wymienionymi już typami danych, jak zadeklarować wiele zmiennych w jednej instrukcji oraz jak spowodować, aby zmienne przechowywały dane, a także jakie obowiązują zasady związane z ich nazewnictwem. Na zakończenie będzie nieco o deklaracjach typów odnośnikowych, którymi bliżej zajmiemy się jednak dopiero w rozdziale 3.

Proste deklaracje

Każda zmienność przed wykorzystaniem w kodzie programu musi zostać zadeklarowana. **Deklaracja** (ang. *declaration*) polega na podaniu typu oraz nazwy zmiennej. Ogólnie taka konstrukcja wygląda następująco:

typ_zmiennej nazwa_zmiennej;

Należy zwrócić uwagę na średnik kończący deklarację. Jest on niezbędny, informuje bowiem kompilator o zakończeniu instrukcji programu (a deklaracja zmiennej jest instrukcją). Program, w którym umieszczono prostą deklarację zmiennej, został przedstawiony na listingu 2.1.

Listing 2.1. Deklaracja zmiennej

```
using System;

public class Program
{
    public static void Main()
    {
        int liczba;
    }
}
```

Po zapisaniu takiego programu w pliku *program.cs* i skompilowaniu w wierszu poleceń przez wydanie komendy

```
csc program.cs
```

kompilator wygeneruje ostrzeżenie widoczne na rysunku 2.1. Nie należy się nim jednak na razie przejmować. To tylko informacja, że zadeklarowaliśmy zmiennej, ale nie wykorzystaliśmy jej do niczego w programie.

Rysunek 2.1.

Kompilator informuje o niewykorzystanej zmiennej

```
C:\cs>csc program.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

program.cs(7,9): warning CS0168: The variable 'liczba' is declared but
never used

C:\cs>=
```

W kodzie z listingu 2.1 powstała zmienność o nazwie *liczba* i typie *int*. Jak już wspomniano, typ *int* pozwala na przechowywanie liczb całkowitych w zakresie od -2 147 483 648 do 2 147 483 647, można więc przypisać tej zmiennej dowolną liczbę mieszczącą się w tym przedziale. Przypisanie takie odbywa się za pomocą znaku (operatora) równości. Jeśli więc chcemy, aby zmienność *liczba* zawierała wartość 100, powinniśmy napisać:

```
liczba = 100;
```

Takie pierwsze przypisanie wartości zmiennej nazywamy jej **inicjacją** lub (częściej) **inicjalizacją**⁸ (listing 2.2). A ponieważ przypisanie wartości zmiennej jest instrukcją programu, nie możemy również zapomnieć o kończącym ją znaku średnika.

Listing 2.2. Inicjalizacja zmiennej

```
using System;

public class Program
{
    public static void Main()
    {
        int liczba;
        liczba = 100;
    }
}
```

Inicjalizacja zmiennej może odbywać się w dowolnym miejscu programu po deklaracji (jak na listingu 2.2), ale może być również równoczesna z deklaracją. W tym drugim przypadku jednocześnie deklarujemy i inicjujemy zmienność, co schematycznie wygląda następująco:

```
typ_zmiennej nazwa_zmiennej = wartość_zmiennej;
```

⁸ Z formalnego punktu widzenia prawidłowym terminem jest „inicjacja”. Wynika to jednak wyłącznie z tego, że termin ten został zapożyczony do języka polskiego znacznie wcześniej niż kojarzona chyba wyłącznie z informatyką „inicjalizacja” (kalka od ang. *initialization*). Co więcej, np. w języku angielskim funkcjonują oba terminy (*initiation* oraz *initialization*) i mają nieco inne znaczenia. Nie wnioskając jednak w niuanse językowe, można powiedzieć, że w przedstawionym znaczeniu oba te terminy mogą być (i są) używane zamiennie.

Zatem w konkretnym przypadku, kiedy mamy zmienną **liczba** typu **int** i chcemy jej przypisać wartość 100, konstrukcja taka miałaby postać:

```
int liczba = 100;
```

Deklaracje wielu zmiennych

Zmienne można deklarować w momencie, kiedy są nam one potrzebne w kodzie programu (inaczej jest np. w Pascalu, gdzie zmienne wykorzystywane w procedurze lub funkcji muszą być zadeklarowane na ich początku). W jednej linii możemy również zadeklarować kilka zmiennych, o ile tylko są one tego samego typu. Struktura takiej deklaracji wygląda wtedy następująco:

```
typ_zmiennej nazwa1, nazwa2, nazwa3;
```

W praktyce mogłoby to przedstawać się tak:

```
int pierwszaLiczba, drugaLiczba, trzeciaLiczba;
```

W przypadku takiej deklaracji można również część zmiennych od razu zainicjować:

```
int pierwszaLiczba = 100, drugaLiczba, trzeciaLiczba = 200;
```

W tym ostatnim przykładzie powstały trzy **zmiennne** o nazwach: **pierwszaLiczba**, **drugaLiczba** oraz **trzeciaLiczba**. Zmiennej **pierwszaLiczba** została przypisana wartość 100, zmiennej **trzeciaLiczba** wartość 200, natomiast zmienność **drugaLiczba** pozostała niezainicjowana. Na listingu 2.3 jest zaprezentowany kod wykorzystujący różne sposoby deklaracji i inicjalizacji **zmiennych**.

Listing 2.3. Różne sposoby deklaracji i inicjalizacji zmiennych

```
using System;

public class Program
{
    public static void Main()
    {
        int liczba1;
        byte liczba2, liczba3 = 100;
        liczba1 = 12842;
        liczba2 = 25;
    }
}
```

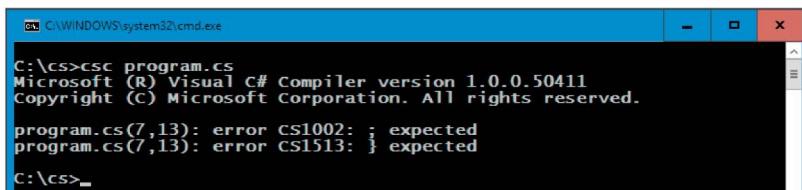
Powstały tutaj w sumie trzy zmienne: jedna typu **int** (**liczba1**) oraz dwie typu **byte** (**liczba2**, **liczba3**). Przypomnijmy, że typ **byte** pozwala na przechowywanie wartości całkowitych z zakresu od 0 do 255 (por. lekcja 4.). Jedynie zmienność **liczba3** została zainicjowana już w trakcie deklaracji i otrzymała wartość 100. Inicjacja zmiennych **liczba1** i **liczba2** odbyła się już po deklaracji i otrzymały one wartości odpowiednio: 12842 i 25.

Nazwy zmiennych

Przy nazywaniu zmiennych obowiązują pewne zasady, których należy przestrzegać. Nazwa taka może składać się z liter (zarówno małych, jak i dużych), cyfr oraz znaku podkreślenia, nie może jednak zaczynać się od cyfry. Tej zasady musimy bezwzględnie przestrzegać, gdyż umieszczenie innych znaków w nazwie (np. dwukropka, myślnika, wykryznika itp.) spowoduje natychmiast błąd komplikacji (rysunek 2.2).

Rysunek 2.2.

Reakcja kompilatora na nieprawidłową nazwę zmiennej



```
C:\cs>csc program.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

program.cs(7,13): error CS1002: ; expected
program.cs(7,13): error CS1513: } expected

C:\cs>
```

Można stosować dowolne znaki będące literami, również te spoza ścisłego alfabetu łacińskiego. Dopuszczalne jest zatem stosowanie wszelkich znaków narodowych (w tym oczywiście polskich). To, czy będą one stosowane, zależy wyłącznie od indywidualnych preferencji programisty i (lub) od specyfiki danego projektu programistycznego (często jednak polskie litery są pomijane, natomiast nazwy zmiennych wywodzą się z języka angielskiego).

Nazwy powinny również odzwierciedlać funkcje, które zmienna pełni w programie. Jeśli ma ona określać szerokość ekranu, nazwijmy ją po prostu szerokoscEkranu czy też screenWidth. To bardzo poprawia czytelność kodu oraz ułatwia jego późniejszą analizę. Przyjmuje się też, że nazwa zmiennej rozpoczyna się od małej litery, natomiast poszczególne słowa wchodzące w skład tej nazwy rozpoczynają się wielkimi literami.

Zmienne typów odnośnikowych

Zmienne typów odnośnikowych, inaczej obiektowych lub referencyjnych (ang. *reference types, object types*), deklaruje się w sposób bardzo podobny do zmiennych zaprezentowanych już typów (tak zwanych typów prostych). Występuje tu jednak bardzo ważna różnica. Otóż pisząc:

```
typ_zmiennej nazwa_zmiennej;
```

w przypadku typów prostych utworzyliśmy zmienną, z której od razu można korzystać. W przypadku typów referencyjnych zostanie w ten sposób zadeklarowane jedynie odniesienie, inaczej referencja (ang. *reference*), któremu domyślnie zostanie przypisana wartość pusta, nazywana null. Zmiennej referencyjnej po deklaracji należy przypisać odniesienie do utworzonego oddzielną instrukcją obiektu. Dopiero wtedy możemy zacząć z niej korzystać. Tym tematem zajmiemy się w rozdziale 3.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 5.1

Zadeklaruj i jednocześnie zainicjuj dwie zmienne typu short. Nazwy zmennych i przypisywane wartości możesz wybrać dowolnie. Pamiętaj o zasadach nazewnictwa zmennych oraz zakresie wartości, jakie mogą być reprezentowane przez typ short.

Ćwiczenie 5.2

Zadeklaruj zmenną typu byte. Przypisz jej wartość większą niż 255. Spróbuj wykonać komplikację i zaobserwuj działanie kompilatora.

Lekcja 6. Wyprowadzanie danych na ekran

Aby zobaczyć wyniki działania programu, można wyświetlić je na ekranie. W lekcji 6. zostanie pokazane, w jaki sposób wykonać to zadanie, jak wyświetlić zwykły napis oraz wartości wybranych zmiennych. Będzie również wyjaśnione, co to są znaki specjalne i co należy zrobić, aby one także mogły pojawić się na ekranie.

Wyświetlanie wartości zmiennych

W lekcji 5. przedstawiono sposoby deklarowania zmiennych oraz przypisywania im różnych wartości. W jaki sposób przekonać się jednak, że dane przypisanie faktycznie odniosło skutek; jak zobaczyć efekty działania programu? Najlepiej wyświetlić je na ekranie. Sposób wyświetlania określonego napisu opisano już w lekcji 1., była to instrukcja: `Console.WriteLine("Mój pierwszy program!");` (por. listing 1.1, rysunek 1.9). Instrukcję tę będziemy wykorzystywać również w dalszych przykładach.

Potrafimy wyświetlić ciąg znaków. Jak jednak wyświetlić wartość zmiennej? Jest to równie proste. Zamiast ujętego w znaki cudzysłowa napisu należy podać nazwę zmiennej, czyli schematycznie konstrukcja taka będzie wyglądała następująco:

```
Console.WriteLine(nazwa_zmiennej);
```

Zmodyfikujmy zatem program z listingu 2.2, tak aby deklaracja i inicjalizacja odbywały się w jednej linii, oraz wyświetlimy wartość zmiennej liczba na ekranie. Sposób realizacji tego zadania został przedstawiony na listingu 2.4 (wynik wykonania programu zaprezentowano na rysunku 2.3).

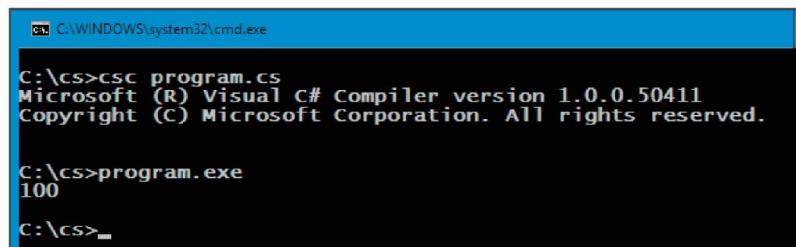
Listing 2.4. Wyświetlenie wartości zmiennej

```
using System;

public class Program
{
    public static void Main()
    {
        int liczba = 100;
        Console.WriteLine(liczba);
    }
}
```

Rysunek 2.3.

Efekt komplikacji i uruchomienia programu z listingu 2.4



W jaki sposób poradzić sobie jednak, kiedy chcemy jednocześnie mieć na ekranie zdefiniowany przez nas ciąg znaków oraz wartość danej zmiennej? Można dwukrotnie użyć instrukcji `Console.WriteLine`, jednak lepszym pomysłem jest zastosowanie operatora⁹ + (plus) w postaci:

```
Console.WriteLine("napis" + nazwa_zmiennej);
```

Konkretny przykład zastosowania takiej konstrukcji jest widoczny na listingu 2.5, a wynik jego działania — na rysunku 2.4.

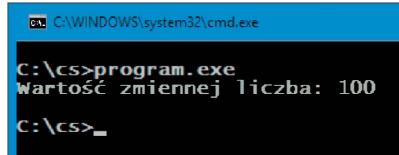
Listing 2.5. Wyświetlenie napisu i wartości zmiennej

```
using System;

public class Program
{
    public static void Main()
    {
        int liczba = 100;
        Console.WriteLine("Wartość zmiennej liczba: " + liczba);
    }
}
```

Rysunek 2.4.

Zastosowanie operatora + do jednoczesnego wyświetlenia napisu i wartości zmiennej



⁹ Operator to coś, co wykonuje jakąś operację. Operacjami na zmiennych i operatorami zajmiemy się w lekcji 7.

Jeśli chcemy, aby wartość zmiennej znalazła się w środku napisu albo żeby w jednej linii znalazły się wartości kilku zmiennych, musimy kilkakrotnie użyć operatora +, składając ciąg znaków, który ma się pojawić na ekranie, z mniejszych fragmentów. Jest zatem możliwa konstrukcja w postaci:

```
Console.WriteLine("napis1" + zmienna1 + "napis2" + zmienna2);
```

Załóżmy więc, że w programie zostaną zadeklarowane dwie zmienne typu byte o nazwach pierwszaLiczba oraz drugaLiczba, którym przypiszemy wartości początkowe równe 25 i 75. Naszym zadaniem będzie wyświetlenie napisu: Wartość zmiennej pierwszaLiczba to 25, a wartość zmiennej drugaLiczba to 75. Program ten jest przedstawiony na listingu 2.6.

Listing 2.6. Łączenie napisów

```
using System;

public class Program
{
    public static void Main()
    {
        byte pierwszaLiczba = 25;
        byte drugaLiczba = 75;
        Console.WriteLine(
            "Wartość zmiennej pierwszaLiczba to " +
            pierwszaLiczba +
            ", a wartość zmiennej drugaLiczba to " +
            drugaLiczba +
            ".");
    }
}
```

Pewnym zaskoczeniem może być rozbicie instrukcji wyświetlającej dane aż na siedem linii. Powód jest prosty: pełna linia ze względu na swoją długość nie zmieściłaby się na wydruku. Lepiej więc było samodzielnie podzielić ją na mniejsze części, aby poprawić czytelność listingu. Przy takim podziale kierujemy się zasadą, że nie wolno nam przedzielić łańcucha znaków ujętego w cudzysłów, natomiast w każdym innym miejscu, gdzie występuje spacja, możemy zamiast niej postawić znak końca linii (naciskając klawisz *Enter*).

Jeśli jednak zmiennych jest kilka i chcemy je wstawić w konkretne miejsca łańcucha znakowego (napisu), warto zastosować inny typ instrukcji, w której miejsce wstawienia zmiennej określa się za pomocą liczby umieszczonej w nawiasie klamrowym. Przykładowo można użyć instrukcji:

```
Console.WriteLine("zm1 = {0}, zm2 = {1}", zm1, zm2);
```

W takiej sytuacji w miejsce ciągu znaków {0} zostanie wstawiona wartość zmiennej zm1, natomiast w miejsce {1} — wartość zmiennej zm2. Przeróbmy zatem program z listingu 2.6 tak, aby użyć tego właśnie sposobu umieszczania wartości zmiennych w ciągu znaków. Odpowiedni kod został zaprezentowany na listingu 2.7.

Listing 2.7. Umieszczanie wartości zmiennych w łańcuchu znakowym

```
using System;

public class Program
{
    public static void Main()
    {
        byte pierwszaLiczba = 25;
        byte drugaLiczba = 75;
        Console.WriteLine(
            "Wartość zmiennej pierwszaLiczba to {0}, a wartość zmiennej drugaLiczba to {1}.",
            pierwszaLiczba, drugaLiczba);
    }
}
```

W C# 6.0 i wyższych można postąpić jeszcze inaczej. W łańcuchu znakowym mogą znaleźć się wyrażenia ujęte w znaki nawiasu klamrowego. Wartości tych wyrażeń zostaną wyliczone, a rezultat obliczeń będzie wstawiony do łańcucha (mówimy wtedy o **interpolicji łańcuchów znakowych**, ang. *string interpolation*). Przy tym, jeśli wyrażeniem jest po prostu nazwa zmiennej, do napisu zostanie wstawiona wartość tej zmiennej. Taki ciąg znaków musi być poprzedzony znakiem \$. Ogólnie:

\$"łańcuch {wyrażenie1}znakowy {wyrażenie2}łańcuch znakowy{wyrażenieN}"

np., jeżeli istnieją zmiennej liczba1, wartosc2 i napis3, można zastosować taką konstrukcję:

\$"łańcuch {liczba1}znakowy {wartosc2}łańcuch znakowy{napis3}"

Program z listingu 2.6 w C# 6.0 mógłby zatem wyglądać tak jak na listingu 2.8.

Listing 2.8. Przykład interpolacji łańcuchów znakowych

```
using System;

public class Program
{
    public static void Main()
    {
        byte pierwszaLiczba = 25;
        byte drugaLiczba = 75;
        Console.WriteLine(
            $"Wartość zmiennej pierwszaLiczba to {pierwszaLiczba}, a wartość zmiennej
            drugaLiczba to {drugaLiczba}.");
    }
}
```

Wyświetlanie znaków specjalnych

Wiemy, że aby wyświetlić na ekranie napis, musimy ująć go w znaki cudzysłowna oraz skorzystać z instrukcji `Console.WriteLine`, np. `Console.WriteLine("napis")`. Nasuwa się jednak pytanie, w jaki sposób wyprowadzić na ekran sam znak cudzysłowu,

skoro jest on częścią instrukcji. Odpowiedź jest prosta: należy go poprzedzić znakiem lewego ukośnika (ang. *backslash*)¹⁰, tak jak jest to zaprezentowane na listingu 2.9.

Listing 2.9. Wyświetlenie znaku cudzysłowu

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("To jest znak cudzysłowu: \" \"");
    }
}
```

Jest to tak zwana **sekwencja ucieczki** (ang. *escape sequence*). Zaczyna się ona od znaku \, po którym występuje określenie znaku specjalnego, jaki ma być wyświetlony na ekranie. Powstaje jednak kolejny problem: w jaki sposób wyświetlić teraz sam znak ukośnika \? Odpowiedź jest taka sama jak w poprzednim przypadku: należy poprzedzić go dodatkowym znakiem \. Konstrukcja taka wyglądać będzie zatem następująco:

```
Console.WriteLine("Oto lewy ukośnik: \\");
```

W ten sam sposób można wyprowadzić również inne znaki specjalne, takie jak znak nowej linii czy znak apostrofu, które zostały zaprezentowane w lekcji 4., w tabeli 2.3, przy omawianiu typu *string*. Można więc używać tabulatorów, znaków nowego wiersza itp. Jeśli użyjemy na przykład instrukcji:

```
Console.WriteLine("abc\t222\ndef\t444");
```

na ekranie pojawi się widok podobny do następującego:

abc	222
def	444

Między ciągami abc i 222 został wstawiony tabulator poziomy (\t), w związku z czym są one od siebie oddalone. Podobnie jest z ciągami def i 444. Z kolei między ciąg 222 a def został wstawiony znak nowego wiersza (\n), dzięki czemu nastąpiło przejście do nowej linii (na ekranie pojawią się dwa wiersze).

Choć używanie znaków specjalnych jest przydatne do formatowania tekstu, czasem jednak chcielibyśmy wyświetlić go w formie oryginalnej. Oczywiście można tekst przetworzyć w taki sposób, aby przed każdą sekwencją specjalną dodać znak \ (np. wszystkie sekwencje \n zamienić na \\n); to jednak wymagałoby dodatkowej pracy. Na szczęście istnieje prostsze rozwiązanie — wystarczy poprzedzić ciąg znakiem @, a sekwencje specjalne nie będą przetwarzane. Zostało to zilustrowane w przykładzie z listingu 2.10.

¹⁰ Spotyka się również określenie ukośnik wsteczny, w odróżnieniu od ukośnika zwykłego / (ang. *slash*).

Listing 2.10. Wyłączanie przetwarzania sekwencji znaków specjalnych

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Z przetwarzaniem znaków specjalnych:");
        Console.WriteLine("abc\t222\ndef\t444\n");
        Console.WriteLine("Bez przetwarzania znaków specjalnych:");
        Console.WriteLine(@"abc\t222\ndef\t444\n");
    }
}
```

Na rysunku 2.5 został zaprezentowany wynik działania tego kodu. Jak widać w pierwszym przypadku, gdy został użyty standardowy ciąg, zarówno zawarte w nim tabulatory, jak i znak nowego wiersza zostały zinterpretowane i napis uzyskał zaplanowany układ. W przypadku drugim ciąg został potraktowany dokładnie tak, jak go zapisano, a znaki specjalne pojawiły się w postaci oryginalnych sekwencji.

Rysunek 2.5.

*Różnice
w przetwarzaniu
znaków specjalnych*

Przedstawiony sposób działa również w przypadku przypisywania ciągów znaków zmiennym. Można np. napisać:

```
String napis = @"abc\t222\ndef\t444\n";
Console.WriteLine(napis);
```

Instrukcja Console.WriteLine

Oprócz poznanej już dobrze instrukcji `Console.WriteLine` możemy wykorzystać do wyświetlania danych na ekranie również bardzo podobną instrukcję: `Console.Write`. Jej działanie jest analogiczne, z tą różnicą, że nie następuje przejście do nowego wiersza. Zatem wykonanie instrukcji w postaci:

```
Console.WriteLine("napis1");
Console.WriteLine("napis2");
```

spowoduje wyświetlenie dwóch wierszy tekstu, z których pierwszy będzie zawierał tekst *napis1*, a drugi *napis2*. Natomiast wykonanie instrukcji w postaci:

```
Console.Write("napis1");
Console.Write("napis2");
```

spowoduje wyświetlenie na ekranie tylko jednego wiersza, który będzie zawierał połączony tekst w postaci: *napis1napis2*.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 6.1

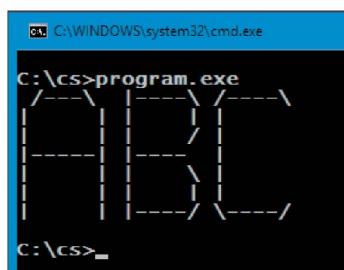
Zadeklaruj dwie zmienne typu `double`. Przypisz im dwie różne wartości zmiennoprzecinkowe, np. 14.5 i 24.45. Wyświetl wartości tych zmiennych na ekranie w dwóch wierszach. Nie korzystaj z instrukcji `Console.WriteLine`.

Ćwiczenie 6.2

Napisz program, który wyświetli na ekranie zbudowany ze znaków alfanumerycznych napis widoczny na rysunku 2.6. Pamiętaj o wykorzystaniu sekwencji znaków specjalnych.

Rysunek 2.6.

Efekt wykonania
ćwiczenia 6.2

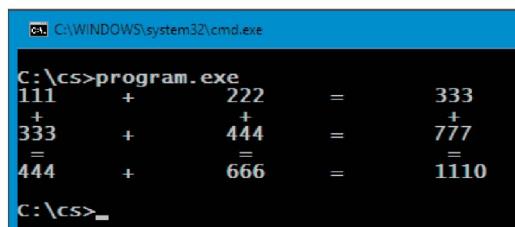


Ćwiczenie 6.3

Napisz program wyświetlający na ekranie znaki układające się w równania przedstawione na rysunku 2.7. Użyj tylko jednego ciągu znaków i jednej instrukcji `Console.WriteLine`.

Rysunek 2.7.

Ilustracja do
ćwiczenia 6.3



Lekcja 7. Operacje na zmiennych

Na zmiennych typów prostych (czyli tych przedstawionych w lekcji 3., z wyjątkiem typów obiektowych) można wykonywać różnorodne operacje, na przykład dodawanie, odejmowanie itp. Operacji tych dokonuje się za pomocą operatorów. Na przykład operacji dodawania dokonujemy za pomocą operatora plus zapiszanego jako `+`, a odejmowania za pomocą operatora minus zapiszanego jako `-`. W tej lekcji zostanie omówione, w jaki sposób są wykonywane operacje na zmiennych i jakie rządzą

nimi prawa. Będą w niej przedstawione operatory arytmetyczne, bitowe, logiczne, przypisania i porównywania, a także ich priorytety, czyli zostanie wyjaśnione, które z nich są silniejsze, a które słabsze.

Operacje arytmetyczne

Operatory arytmetyczne służą, jak nietrudno się domyślić, do wykonywania operacji arytmetycznych, czyli dobrze znanego wszystkim mnożenia, dodawania itp. Występują w tej grupie jednak również mniej znane operatory, takie jak operator inkrementacji i dekrementacji. Wszystkie one są zebrane w tabeli 2.4.

Tabela 2.4. Operatory arytmetyczne w C#

Operator	Wykonywane działanie
*	Mnożenie
/	Dzielenie
+	Dodawanie
-	Odejmowanie
%	Dzielenie modulo (reszta z dzielenia)
++	Inkrementacja (zwiększenie)
--	Dekrementacja (zmniejszanie)

Podstawowe działania

W praktyce korzystanie z większości z tych operatorów sprowadza się do wykonywania typowych działań znanych z lekcji matematyki. Jeśli zatem chcemy dodać do siebie dwie zmienne lub liczbę do zmiennej, wykorzystujemy operator +; gdy chcemy coś pomnożyć — operator * itp. Oczywiście operacje arytmetyczne wykonuje się na zmiennych typów arytmetycznych. Założmy zatem, że mamy trzy zmienne typu całkowitoliczbowego int i wykonajmy na nich kilka prostych operacji. Zobrazowano to na listingu 2.11; dla zwiększenia czytelności opisu poszczególne linie zostały ponumerowane.

Listing 2.11. Proste operacje arytmetyczne

```
using System;

public class Program
{
    public static void Main()
    {
        /*1*/ int a, b, c;
        /*2*/ a = 10;
        /*3*/ b = 20;
        /*4*/ Console.WriteLine("a = " + a + ", b = " + b);
        /*5*/ c = b - a;
        /*6*/ Console.WriteLine("b - a = " + c);
        /*7*/ c = a / 2;
```

```
/*8*/ Console.WriteLine("a / 2 = " + c);
/*9*/ c = a * b;
/*10*/ Console.WriteLine("a * b = " + c);
/*11*/ c = a + b;
/*12*/ Console.WriteLine("a + b = " + c);
}
```

Linie od 1. do 4. to deklaracje zmiennych a, b i c, przypisanie zmiennej a wartości 10, zmiennej b wartości 20 oraz wyświetlenie tych wartości na ekranie. W linii 5. przypisujemy zmiennej c wynik odejmowania b - a, czyli wartość 10 ($20 - 10 = 10$). W linii 7. przypisujemy zmiennej c wartość działania a / 2, czyli 5 ($10 / 2 = 5$). Podobnie postępujemy w linii 9. i 11., gdzie wykonujemy działanie mnożenia (c = a * b) oraz dodawania (c = a + b). W liniach 6., 8., 10. i 12. korzystamy z dobrze znanej nam instrukcji `Console.WriteLine` do wyświetlenia wyników poszczególnych działań. Efekt uruchomienia takiego programu jest widoczny na rysunku 2.8.

Rysunek 2.8.

Wynik prostych działań arytmetycznych na zmiennych

```
C:\>program.exe
a = 10, b = 20
b - a = 10
a / 2 = 5
a * b = 200
a + b = 30
C:\>
```

Do operatorów arytmetycznych należy również znak %, przy czym jak już wspomniano, nie oznacza on obliczania procentów, ale dzielenie modulo, czyli resztę z dzielenia. Przykładowo działanie $10 \% 3$ da w wyniku 1. Trójką zmieści się bowiem w dziesięciu trzy razy, pozostawiając resztę 1 ($3 \times 3 = 9$, $9 + 1 = 10$). Podobnie $21 \% 8$ to 5, gdyż $2 \times 8 = 16$, $16 + 5 = 21$.

Inkrementacja i dekrementacja

Operatory inkrementacji, czyli zwiększania (++), oraz dekrementacji, czyli zmniejszania (--), są z pewnością znane osobom знаjącym język takie jak C, C++ czy Java, nowością będą natomiast dla programujących w Pascalu. Operator ++ zwiększa po prostu wartość zmiennej o 1, a -- zmniejsza ją o 1. Mogą one występować w formie przedrostkowej lub przyrostkowej. Jeśli mamy na przykład zmienną o nazwie x, forma przedrostkowa (preinkrementacyjna) będzie miała postać `++x`, natomiast przyrostkowa (postinkrementacyjna) — `x++`.

Obie te postacie powodują zwiększenie wartości zapisanej w zmiennej x o 1, ale w przypadku formy przedrostkowej (`++x`) odbywa się to przed użyciem zmiennej, a w przypadku formy przyrostkowej (`x++`) dopiero po jej użyciu. Mimo iż osobom początkującym wydaje się to zapewne zupełnie niezrozumiałe, wszelkie wątpliwości rozwiąże praktyczny przykład. Spójrzmy na listing 2.12 i zastanówmy się, jakie będą wyniki działania takiego programu.

Listing 2.12. Użycie operatora inkrementacji

```

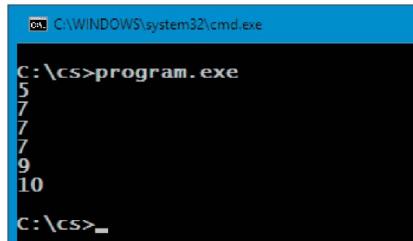
using System;

public class Program
{
    public static void Main()
    {
        /*1*/ int x = 5, y;
        /*2*/ Console.WriteLine(x++);
        /*3*/ Console.WriteLine(++x);
        /*4*/ Console.WriteLine(x);
        /*5*/ y = x++;
        /*6*/ Console.WriteLine(y);
        /*7*/ y = ++x;
        /*8*/ Console.WriteLine(y);
        /*9*/ Console.WriteLine(++y);
    }
}

```

Wynikiem jego działania będzie ciąg liczb 5, 7, 7, 7, 9, 10 (rysunek 2.9). Dlaczego? Otóż w linii 1. deklarujemy zmienne x i y oraz przypisujemy zmiennej x wartość 5. W linii 2. stosujemy formę przyrostkową operatora ++, zatem najpierw wyświetlamy wartość zmiennej x na ekranie ($x = 5$), a dopiero potem zwiększamy ją o 1 ($x = 6$). W linii 3. pośtepujemy dokładnie odwrotnie, to znaczy przez zastosowanie formy przedrostkowej najpierw zwiększamy wartość zmiennej x o 1 ($x = 7$), a dopiero potem wyświetlamy ją na ekranie. W linii 4. jedyną operacją jest ponowne wyświetlenie wartości x ($x = 7$).

Rysunek 2.9.
Wynik działania programu ilustrującego działanie operatora ++



W linii 5. najpierw przypisujemy aktualną wartość x ($x = 7$) zmiennej y ($y = 7$) i dopiero potem zwiększamy x o 1 ($x = 8$). W linii 6. wyświetlamy wartość y. W linii 7. najpierw zwiększamy x o 1 ($x = 9$), a następnie przypisujemy ją y ($y = 9$). W linii 8. wyświetlamy wartość y ($y = 9$). W ostatniej linii, 9., najpierw zwiększamy y o 1 ($y = 10$), a dopiero potem wyświetlamy tę wartość na ekranie. Wynik działania programu jest widoczny na rysunku 2.9.

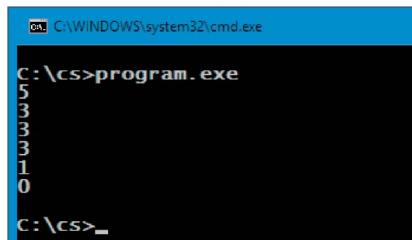
Operator dekrementacji (--) działa analogicznie do ++, z tą różnicą, że zmniejsza wartość zmiennej o 1. Zmodyfikujmy zatem kod z listingu 2.11 w taki sposób, że wszystkie wystąpienia ++ zamienimy na --. Otrzymamy wtedy program widoczny na listingu 2.13. Tym razem wynikiem będzie ciąg liczb: 5, 3, 3, 3, 1, 0 (rysunek 2.10). Prześledźmy jego działanie.

Listing 2.13. Użycie operatora dekrementacji

```
using System;

public class Program
{
    public static void Main()
    {
        /*1*/ int x = 5, y;
        /*2*/ Console.WriteLine(x--);
        /*3*/ Console.WriteLine(--x);
        /*4*/ Console.WriteLine(x);
        /*5*/ y = x--;
        /*6*/ Console.WriteLine(y);
        /*7*/ y = --x;
        /*8*/ Console.WriteLine(y);
        /*9*/ Console.WriteLine(--y);
    }
}
```

Rysunek 2.10.
Ilustracja działania
operatora
dekrementacji



W linii 1. deklarujemy zmienne x i y oraz przypisujemy zmiennej x wartość 5, dokładnie tak jak w programie z listingu 2.11. W linii 2. stosujemy formę przyrostkową operatora $--$, zatem najpierw wyświetlamy wartość zmiennej x ($x = 5$) na ekranie, a dopiero potem zmniejszamy ją o 1 ($x = 4$). W linii 3. postępujemy dokładnie odwrotnie, to znaczy przez zastosowanie formy przedrostkowej najpierw zmniejszamy wartość zmiennej x o 1 ($x = 3$), a dopiero potem wyświetlamy ją na ekranie. W linii 4. jedną operacją jest ponowne wyświetlenie wartości x ($x = 3$).

W linii 5. najpierw przypisujemy aktualną wartość x ($x = 3$) zmiennej y ($y = 3$) i dopiero potem zmniejszamy x o jeden ($x = 2$). W linii 6. wyświetlamy wartość y ($y = 3$). W 7. najpierw zmniejszamy x o 1 ($x = 1$), a następnie przypisujemy tę wartość y ($y = 1$). W linii 8. wyświetlamy wartość y ($y = 1$). W ostatniej linii, 9., najpierw zmniejszamy y o 1 ($y = 0$), a dopiero potem wyświetlamy tę wartość na ekranie.

Kiedy napotkamy problemy...

Zrozumienie sposobu działania operatorów arytmetycznych nikomu z pewnością nie przysporzyło żadnych większych problemów. Jednak i tutaj czyhają na nas pewne pułapki. Powróćmy na chwilę do listingu 2.11. Wykonywane było tam m.in. działanie $c = a / 2$. Zmienna a miała wartość 10, zmiennej c został zatem przypisany wynik działania $10 / 2$, czyli 5. To nie budzi żadnych wątpliwości.

Zwróćmy jednak uwagę, że zarówno a, jak i c były typu int, czyli mogły przechowywać jedynie liczby całkowite. Co się zatem stanie, jeśli wynikiem dzielenia a / 2 nie będzie liczba całkowita? Czy zobaczymy ostrzeżenie kompilatora lub czy program podczas działania niespodziewanie zasygnalizuje błąd? Możemy się o tym przekonać, komplikując i uruchamiając kod widoczny na listingu 2.14.

Listing 2.14. Automatyczne konwersje wartości

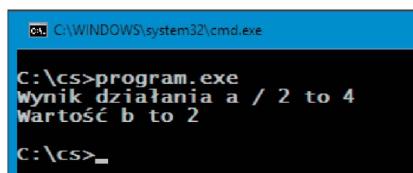
```
using System;

public class Program
{
    public static void Main()
    {
        int a, b;
        a = 9;
        b = a / 2;
        Console.WriteLine("Wynik działania a / 2 to " + b);
        b = 8 / 3;
        Console.WriteLine("Wartość b to " + b);
    }
}
```

Zmienne a i b są typu int i mogą przechowywać liczby całkowite. Zmiennej a przypisujemy wartość 9, zmiennej b wynik działania a / 2. Z matematyki wiemy, że wynikiem działania 9/2 jest 4,5. Nie jest to zatem liczba całkowita. Co się stanie w programie? Otóż wynik zostanie zaokrąglony w dół, do najbliższej liczby całkowitej. Zmienna b otrzyma zatem wartość 4. Jest to widoczne na rysunku 2.11.

Rysunek 2.11.

Wynik działania programu z listingu 2.14



Podobnie jeśli zmiennej b przypiszemy wynik bezpośredniego dzielenia liczb (linia b = 8 / 3;), prawdziwy wynik zostanie zaokrąglony w dół (odrzucona będzie część ułamkowa). Innymi słowy, C# sam dopasuje typy danych (mówiąc fachowo: dokona konwersji typu). Początkujący programiści powinni zwrócić na ten fakt szczególną uwagę, gdyż niekiedy prowadzi to do trudnych do wykrycia błędów w aplikacjach.

Dlaczego jednak kompilator nie ostrzega nas o tym, że taka konwersja zostanie dokonana? Przede wszystkim najczęściej wynik działań jest znany dopiero w trakcie działania programu, w większości przypadków nie ma więc możliwości sprawdzenia już na etapie komplikacji, czy wynik jest właściwego typu. Nie można też dopuścić do sytuacji, gdy program będzie zgłaszał błędy lub ostrzeżenia za każdym razem, kiedy wynik działania nie będzie dokładnie takiego typu jak zmienna, której jest przypisywany. Dlatego też istnieją zdefiniowane w języku programowania ogólne zasady konwersji typów, które są stosowane automatycznie (tzw. konwersje automatyczne),

kiedy tylko jest to możliwe. Jedna z takich reguł mówi właśnie, że jeśli wynikiem operacji jest wartość zmiennoprzecinkowa, a wynik ten ma być przypisany zmiennej całkowitoliczbowej, to część ułamkowa zostanie odrzucona.

Nie oznacza to jednak, że możemy bezpośrednio przypisać wartość zmiennoprzecinkową zmiennej typu całkowitoliczbowego. Kompilator nie dopuści do wykonania takiej operacji. Przekonajmy się o tym — spróbujmy dokonać komplikacji kodu z listingu 2.15.

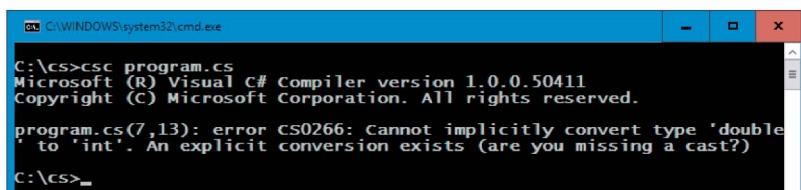
Listing 2.15. Próba przypisania wartości ułamkowej zmiennej całkowitej

```
using System;

public class Program
{
    public static void Main()
    {
        int a = 9.5;
        Console.WriteLine("Zmienna a ma wartość " + a);
    }
}
```

Jak widać na rysunku 2.12, komplikacja się nie udała, a komplikator zgłosił błąd. Tym razem bowiem próbujemy bezpośredniego przypisania liczby ułamkowej zmiennej typu int, która takich wartości przechowywać nie może.

Rysunek 2.12.
Próba przypisania
zmiennej int
wartości ułamkowej
kończy się błędem
komplikacji



Z podobną sytuacją będziemy mieć do czynienia, kiedy spróbujemy przekroczyć dopuszczalny zakres wartości, który może być reprezentowany przez określony typ danych. Sprawdźmy! Zmienna typu sbyte może przechowywać wartości z zakresu od -128 (to jest -2^7) do 127 (to jest $2^7 - 1$). Co się zatem stanie, jeśli zmiennej typu sbyte spróbujemy przypisać wartość przekraczającą 127 choćby o 1 ? Spodziewamy się, że komplikator zgłosi błąd. I tak jest w istocie. Na listingu 2.16 przedstawiono kod realizujący taką instrukcję. Próba komplikacji da efekt widoczny na rysunku 2.13. Błąd tego typu zostanie nam od razu zasygnalizowany, nie trzeba się więc takiej pomyłki obawiać. Wyeliminujemy ją praktycznie od ręki, tym bardziej że komplikator wskazuje konkretne miejsce jej wystąpienia.

Listing 2.16. Przypisanie zmiennej wartości przekraczającej dopuszczalny zakres

```
using System;

public class Program
{
    public static void Main()
    {
        sbyte liczba = 128;
```

```

        Console.WriteLine("Zmienna liczba ma wartość = " + liczba);
    }
}

```

Rysunek 2.13.

Próba przypisania zmiennej wartości przekraczającej dopuszczalny zakres

```
C:\>csc program.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

program.cs(7,20): error CS0031: Constant value '128' cannot be converted
to a 'sbyte'

C:\>_
```

Niestety, najczęściej przypisanie wartości przekraczającej zakres danego typu odbywa się już w trakcie działania programu. Jest to sytuacja podobna do przykładu z automatyczną konwersją liczby zmiennoprzecinkowej tak, aby mogła zostać przypisana zmiennej typu `int` (listing 2.14). Jeśli przypisanie wartości zmiennej jest wynikiem obliczeń wykonanych w trakcie działania programu, kompilator nie będzie w stanie ostrzec nas, że przekraczamy dopuszczalny zakres wartości. Taką sytuację zobrazowano na listingu 2.17.

Listing 2.17. *Przekroczenie dopuszczalnej wartości w trakcie działania aplikacji*

```

using System;

public class Program
{
    public static void Main()
    {
        sbyte liczba = 127;
        liczba++;
        Console.WriteLine("Zmienna liczba ma wartość " + liczba);
    }
}

```

Na początku deklarujemy tu zmienną `liczba` typu `sbyte` i przypisujemy jej maksymalną wartość, którą można za pomocą tego typu przedstawić (127), a następnie zwiększamy ją o 1 (`liczba++`). Tym samym maksymalna wartość została przekroczona. Dalej próbujemy wyświetlić ją na ekranie. Czy taki program da się skompilować? Okazuje się, że tak. Tym razem, odmiennie niż w poprzednim przykładzie, zakres zostaje przekroczony dopiero w trakcie działania aplikacji, w wyniku działań arytmetycznych. Co zatem zostanie wyświetcone na ekranie? Jest to widoczne na rysunku 2.14.

Rysunek 2.14.

Wynik działania programu z listingu 2.17

```
C:\>csc program.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

C:\>program.exe
Zmienna liczba ma wartość -128

C:\>_
```

Zauważmy, że wartość, która się pojawiła na ekranie, to dolny zakres dla typu sbyte (por. tabela 2.1), czyli minimalna wartość, jaką może przyjąć zmienna tego typu. Zatem przekroczenie dopuszczalnej wartości nie powoduje błędu, ale „zawinięcie” liczby. Zobrazowano to na rysunku 2.15. Arytmetyka wygląda zatem w tym przypadku następująco:

$$\begin{aligned} \text{SByte_MAX} + 1 &= \text{SByte_MIN} \\ \text{SByte_MAX} + 2 &= \text{SByte_MIN} + 1 \\ \text{SByte_MAX} + 3 &= \text{SByte_MIN} + 2, \end{aligned}$$

jak również:

$$\begin{aligned} \text{SByte_MIN} - 1 &= \text{SByte_MAX} \\ \text{SByte_MIN} - 2 &= \text{SByte_MAX} - 1 \\ \text{SByte_MIN} - 3 &= \text{SByte_MAX} - 2, \end{aligned}$$

gdzie SByte_MIN to minimalna wartość, jaką może przyjąć zmienna typu sbyte, czyli -128, a SBYTE_MAX to wartość maksymalna, czyli 127. Tak samo jest w przypadku pozostałych typów całkowitoliczbowych.

Rysunek 2.15.

Przekroczenie dopuszczalnego zakresu dla typu sbyte



Operacje bitowe

Operatory bitowe służą, jak sama nazwa wskazuje, do wykonywania operacji na bitach. Przypomnijmy więc przynajmniej podstawowe informacje na temat systemów liczbowych. W systemie dziesiętnym, z którego korzystamy na co dzień, wykorzystywanych jest dziesięć cyfr, od 0 do 9. W systemie dwójkowym będą zatem wykorzystywane jedynie dwie cyfry: 0 i 1, w systemie szesnastkowym cyfry od 0 do 9 i dodatkowo litery od A do F, a w systemie ósemkowym cyfry od 0 do 7. Kolejne liczby są budowane z tych cyfr i znaków dokładnie tak samo jak w systemie dziesiętnym (przedstawiono to w tabeli 2.5). Widac wyraźnie, że np. 4 dziesiętnie to 100 dwójkowo, a 10 dziesiętnie to 1010 dwójkowo.

Operatory bitowe pozwalają właśnie na wykonywanie operacji na poszczególnych bitach liczb. Są to z pewnością znane każdemu ze szkoły operacje: AND (iloczyn bitowy, koniunkcja bitowa), OR (suma bitowa, alternatywa bitowa) oraz XOR (bitowa alternatywa wykluczająca) i NOT (negacja bitowa, dopełnienie bitowe) oraz mniej znane operacje przesunięć bitów. Symbolem operatora AND jest znak amper-sand (&), operatora OR pionowa kreska (|), operatora XOR znak strzałki w góre, daszka (^), natomiast operatora NOT znak tyldy (~). Operatory te zostały zebrane w tabeli 2.6.

Tabela 2.5. Kolejne 15 liczb w systemie w różnych systemach liczbowych

system dwójkowy	system ósemkowy	system dziesiętny	system szesnastkowy
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F

Tabela 2.6. Operatory bitowe w C#

Operator	Symbol
Iloczyn bitowy AND	&
Suma bitowa OR	
Negacja bitowa NOT	~
Bitowa alternatywa wykluczająca XOR	^
Przesunięcie bitowe w prawo	>>
Przesunięcie bitowe w lewo	<<

Iloczyn bitowy

Iloczyn bitowy to operacja, w której wyniku włączone pozostają tylko bity włączone w obu argumentach. Wynik operacji AND na pojedynczych bitach zobrazowano w tabeli 2.7. Jeśli zatem wykonamy operację:

179 & 38;

to jej wynikiem będzie 34. Dlaczego właśnie tyle? Najłatwiej pokazać to, jeśli obie wartości (czyli 179 i 38) przedstawi się w postaci dwójkowej. 179 w postaci dwójkowej to 10110011, natomiast 38 to 00100110. Operacja AND będzie zatem miała postać:

10110011 (179)

00100110 (38)

00100010 (34)

Wynikiem jest więc 34.

Tabela 2.7. Wyniki operacji AND dla pojedynczych bitów

Argument 1	Argument 2	Wynik
1	1	1
1	0	0
0	1	0
0	0	0

Suma bitowa

Suma bitowa to operacja, w której pozostają włączone bity włączone w przynajmniej jednym z argumentów. Wynik operacji OR na pojedynczych bitach zobrazowano w tabeli 2.8. Jeśli zatem wykonamy operację:

34 | 65;

to jej wynikiem będzie 99. Jeżeli obie liczby rozpiszemy w postaci dwójkowej, otrzymamy: 00100010 (34) i 01000001 (65). Zatem całe działanie będzie miało postać:

00100010 (34)

01000001 (65)

01100011 (99)

Tabela 2.8. Wyniki operacji OR dla pojedynczych bitów

Argument 1	Argument 2	Wynik
1	1	1
1	0	1
0	1	1
0	0	0

Negacja bitowa

Negacja bitowa powoduje zmianę stanu bitów — czyli tam, gdzie dany bit miał wartość 0, będzie posiadał wartość 1, natomiast tam, gdzie dany bit miał wartość 1, będzie posiadał wartość 0. Działanie operacji NOT na pojedynczych bitach zobrazowano w tabeli 2.9.

Tabela 2.9. Wynik operacji NOT dla poszczególnych bitów

Argument	Wynik
1	0
0	1

Bitowa alternatywa wykluczająca

Bitowa alternatywa wykluczająca, czyli operacja XOR, powoduje, że włączone zostają te bity, które miały różne stany w obu argumentach. Pozostałe bity zostają wyłączone. Wynik operacji XOR na pojedynczych bitach zobrazowano w tabeli 2.10.

Tabela 2.10. Wyniki operacji XOR dla pojedynczych bitów

Argument 1	Argument 2	Wynik
1	1	0
1	0	1
0	1	1
0	0	0

Wykonanie przykładowej operacji:

`34 ^ 118;`

da w wyniku wartość 84. Jeśli bowiem zapiszemy obie wartości w postaci dwójkowej, to 34 przyjmie postać 00100010, natomiast 118 — postać 01110110. Operacja XOR będzie zatem wyglądała następująco:

$$\begin{array}{r}
 00100010 \text{ (34)} \\
 01110110 \text{ (118)} \\
 \hline
 01010100 \text{ (84)}
 \end{array}$$

Przesunięcie bitowe w lewo

Przesunięcie bitowe w lewo to operacja polegająca na przesunięciu wszystkich bitów argumentu znajdującego się z lewej strony operatora w lewo o liczbę miejsc wskazaną przez argument z prawej strony operatora. Tym samym wykonanie przykładowej operacji:

`84 << 1;`

da w wyniku wartość 168. Działanie `84 << 1` oznacza bowiem: „Przesuń wszystkie bity wartości 84 o jedno miejsce w lewo”. Skoro 84 w postaci dwójkowej ma postać 01010100, to po przesunięciu powstanie 10101000, czyli 168.

Warto zauważyć, że przesunięcie bitowe w lewo odpowiada mnożeniu wartości przez wielokrotność liczby 2. A zatem przesunięcie w lewo o jedno miejsce to pomnożenie przez 2, przesunięcie w lewo o dwa miejsca to pomnożenie przez 4, przesunięcie w lewo o trzy miejsca to pomnożenie przez 8 itd.

Przesunięcie bitowe w prawo

Analogicznie do przesunięcia w lewo **przesunięcie bitowe w prawo** polega na przesunięciu wszystkich bitów argumentu znajdującego się z lewej strony operatora w prawo o liczbę miejsc wskazaną przez argument z prawej strony operatora. A zatem wykonanie operacji:

```
84 >> 1;
```

da w wyniku wartość 42. Oznacza to bowiem przesunięcie wszystkich bitów wartości 01010100 o jedno miejsce w prawo, czyli powstanie wartości 00101010 dwójkowo (42 dziesiętnie).

Tu również należy zwrócić uwagę, że przesunięcie bitowe w prawo odpowiada podzieleniu wartości przez wielokrotność liczby 2. A zatem przesunięcie w prawo o jedno miejsce to podzielenie przez 2, przesunięcie w prawo o dwa miejsca to podzielenie przez 4, przesunięcie w prawo o trzy miejsca to podzielenie przez 8 itd. Należy jednak pamiętać, że jeżeli dzielona liczba będzie nieparzysta, to w wyniku takiego dzielenia zostanie utracona część ułamkowa.

Operacje logiczne

Operacje logiczne możemy wykonywać na argumentach, które mają wartość logiczną prawda lub fałsz. W językach programowania wartości te zwykle są oznaczane jako true i false. W przypadku C# oba te słowa muszą być zapisane dokładnie w taki sposób, to znaczy małymi literami, inaczej nie zostaną rozpoznane przez kompilator. Jeśli zatem mamy przykładowe wyrażenie `0 < 1`, to ma ono wartość logiczną true (prawda), jako że niewątpliwie zero jest mniejsze od jedności. Operacje logiczne to znane ze szkoły logicznej AND (iloczyn), OR (suma) oraz NOT (negacja). Przedstawiono to w tabeli 2.11.

Tabela 2.11. Operatory logiczne w C#

Operator	Symbol
Iloczyn logiczny AND	<code>&&</code>
Suma logiczna OR	<code> </code>
Negacja logiczna NOT	<code>!</code>
Iloczyn logiczny AND	<code>&</code>
Suma logiczna OR	<code> </code>

Występowanie w tabeli podwójnych operatorów sumy i iloczynu logicznego nie jest błędem. Różnią się one nieco sposobem działania.

Iloczyn logiczny (`&&`)

Wynikiem operacji AND (iloczyn logiczny) jest wartość true wtedy i tylko wtedy, kiedy oba argumenty mają wartość true. W każdym innym przypadku wynikiem jest false.

Przedstawiono to w tabeli 2.12. Jeśli w przypadku tego operatora wartością pierwszego argumentu jest `false`, to drugi nie jest obliczany (wynikiem całego wyrażenia, co wynika z tabeli 2.12, na pewno bowiem będzie `false`).

Tabela 2.12. Logiczny iloczyn

Argument 1	Argument 2	Wynik
<code>true</code>	<code>true</code>	<code>true</code>
<code>true</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>true</code>	<code>false</code>
<code>false</code>	<code>false</code>	<code>false</code>

Iloczyn logiczny (&)

Ten operator działa tak jak `&&`, z tą różnicą, że drugi argument jest obliczany zawsze, nawet w sytuacji, kiedy wartością pierwszego jest `false`.

Suma logiczna (||)

Wynikiem operacji OR (suma logiczna) jest wartość `false` wtedy i tylko wtedy, kiedy oba argumenty mają wartość `false`. W każdym innym przypadku wynikiem jest `true`. Przedstawiono to w tabeli 2.13. Jeśli w przypadku tego operatora wartością pierwszego argumentu jest `true`, to drugi nie jest obliczany (wynikiem całego wyrażenia na pewno bowiem będzie `true` — tabela 2.13).

Tabela 2.13. Logiczna suma

Argument 1	Argument 2	Wynik
<code>true</code>	<code>true</code>	<code>true</code>
<code>true</code>	<code>false</code>	<code>true</code>
<code>false</code>	<code>true</code>	<code>true</code>
<code>false</code>	<code>false</code>	<code>false</code>

Suma logiczna (|)

Ten operator działa tak jak `||`, z tą różnicą, że drugi argument jest obliczany zawsze, nawet w sytuacji, kiedy wartością pierwszego jest `true`.

Logiczna negacja

Operacja NOT (logiczna negacja) zamienia po prostu wartość argumentu na przeciwną. Jeśli więc argument miał wartość `true`, będzie miał wartość `false` i odwrotnie — jeśli miał wartość `false`, będzie miał wartość `true`. Zobrazowano to w tabeli 2.14.

Tabela 2.14. Logiczna negacja

Argument	Wynik
true	false
false	true

Operatory przypisania

Operacje przypisania są dwuargumentowe i powodują przypisanie argumentu prawostronnego argumentowi lewostronnemu. Taką najprostszą operację już przedstawiono; odbywa się ona przy wykorzystaniu operatora = (równa się). Jeśli napiszemy **liczba = 5**, będzie to oznaczać, że zmiennej **liczba** chcemy przypisać wartość 5.

Oprócz prostego operatora = w C# (tak jak i w wielu innych współczesnych, popularnych językach programowania) występuje szereg operatorów łączonych, tzn. takich, w których przypisaniu towarzyszy dodatkowa operacja arytmetyczna lub bitowa. Istnieje na przykład operator +=, który oznacza: „Przypisz argumentowi z lewej strony wartość wynikającą z dodawania argumentu znajdującego się z lewej strony i argumentu znajdującego się z prawej strony operatora”.

Choć brzmi to z początku nieco zawile, w rzeczywistości jest bardzo proste i znacznie upraszcza niektóre konstrukcje programistyczne. Po prostu przykładowy zapis:

a += b

tłumaczymy jako:

a = a + b

Zatem wykonanie kodu widocznego na listingu 2.18 spowoduje przypisanie zmiennej **liczba** wartości 15, następnie dodanie do niej wartości 10 oraz wyświetlenie ostatecznego stanu zmiennej (**liczba = 25**) na ekranie.

Listing 2.18. Użycie operatora +=

```
using System;

public class Program
{
    public static void Main()
    {
        int liczba = 15;
        liczba += 10;
        Console.WriteLine("Zmienna liczba ma wartość = " + liczba);
    }
}
```

W C# występuje cała grupa tych operatorów (są one zebrane w tabeli 2.15). Schematycznie można przedstawić ich znaczenie następująco:

arg1 op= arg2

oznacza działanie:

`arg1 = arg1 op arg2`

czyli `a += b` oznacza `a = a + b`, `a *= b` oznacza `a = a * b`, `a %= b` oznacza `a = a % b` itd.

Tabela 2.15. Operatory przypisania i ich znaczenie w C#

Argument 1	Operator	Argument 2	Znaczenie
x	=	y	<code>x = y</code>
x	+=	y	<code>x = x + y</code>
x	-=	y	<code>x = x - y</code>
x	*=	y	<code>x = x * y</code>
x	/=	y	<code>x = x / y</code>
x	%=	y	<code>x = x % y</code>
x	&=	y	<code>x = x & y</code>
x	=	y	<code>x = x y</code>
x	^=	y	<code>x = x ^ y</code>
x	<<=	y	<code>x = x << y</code>
x	>>=	y	<code>x = x >> y</code>

Operatory porównywania (relacyjne)

Operatory porównywania służą oczywiście do porównywania argumentów. Wynikiem ich działania jest wartość logiczna `true` lub `false`, czyli prawda lub fałsz. Operatory te są zebrane w tabeli 2.16. Przykładowo wynikiem operacji `argument1 == argument2` będzie `true`, jeżeli argumenty są sobie równe, oraz `false`, jeżeli są różne. A zatem `4 == 5` ma wartość `false`, a `2 == 2` ma wartość `true`. Podobnie `2 < 3` ma wartość `true` (2 jest bowiem mniejsze od 3), ale `4 < 1` ma wartość `false` (gdyż 4 jest większe, a nie mniejsze od 1). Jak je wykorzystywać w praktyce, będzie wyjaśnione w omówieniu instrukcji warunkowych w lekcji 8.

Tabela 2.16. Operatory porównywania w C#

Operator	Opis
<code>==</code>	Wynikiem jest <code>true</code> , jeśli argumenty są sobie równe.
<code>!=</code>	Wynikiem jest <code>true</code> , jeśli argumenty są różne.
<code>></code>	Wynikiem jest <code>true</code> , jeśli argument prawostronny jest mniejszy od lewostronnego.
<code><</code>	Wynikiem jest <code>true</code> , jeśli argument prawostronny jest większy od lewostronnego.
<code>>=</code>	Wynikiem jest <code>true</code> , jeśli argument prawostronny jest mniejszy od lewostronnego lub jest mu równy.
<code><=</code>	Wynikiem jest <code>true</code> , jeśli argument prawostronny jest większy od lewostronnego lub jest mu równy.

Pozostałe operatory

Oprócz operatorów wymienionych na poprzednich stronach w języku C# występuje jeszcze kilkanaście innych, niektóre z nich zostaną omówione w dalszej części książki, m.in. operator warunkowy, wyboru składowej, indeksowania tablic itp. Wszystkie występujące w języku operatory zostały za to uwzględnione w tabeli 2.17, w podrozdziale „Priorytety operatorów”.

Priorytety operatorów

Sama znajomość operatorów to jednak nie wszystko. Trzeba jeszcze wiedzieć, jaki mają one **priorytet**, czyli jaka jest kolejność ich wykonywania. Wiemy np. z matematyki, że mnożenie jest silniejsze od dodawania, zatem najpierw mnożymy, potem dodajemy. W C# jest podobnie — siła każdego operatora jest ścisłe określona. Przedstawiono to w tabeli 2.17 (uwzględnione zostały również operatory nieomawiane w książce — nie należy więc przejmować się ich dużą liczbą). Im wyższa pozycja w tabeli, tym wyższy priorytet operatora. Operatory znajdujące się na jednym poziomie (w jednym wierszu) mają ten sam priorytet. Należy przy tym zwrócić uwagę, że priorytet niektórych operatorów zmieniał się wraz z wersjami języka C# (jest tak m.in. z sizeof, null-coalescing, lambda). W tabeli uwzględniono priorytety aktualne dla wersji 6.0.

Tabela 2.17. Priorytety operatorów w C#

Operatory	Symbol
wybór składowej, wywołanie funkcji, indeksowanie tablicy, inkrementacja i dekrementacja przyrostkowa, tworzenie obiektów, kontrola typów, delegacje, ustalanie rozmiaru, dereferencja z dostępem do składowej	., (), [], ++, --, new, typeof, default, checked, unchecked, delegate, sizeof ¹¹ , ->
ustalenie znaku wartości, negacje, inkrementacja i dekrementacja przedrostkowa, rzutowanie typów, ustalanie rozmiaru, pobranie adresu, dereferencja	+, -, !, ~, ++, --, (), sizeof ¹² , &, * ¹³
mnożenie, dzielenie, dzielenie modulo	*, /, %
dodawanie, odejmowanie	+, -
przesunięcia bitowe	<<, >>
relacyjne, testowanie typów	<, >, <=, >=, is, as
porównania	==, !=
bitowe AND	&
bitowe XOR	^

¹¹ Dotyczy C# 5.0 i wyższych.

¹² Dotyczy C# 4.0 i niższych.

¹³ Dostępny od C# 5.0.

Tabela 2.17. Priorytety operatorów w C# — ciąg dalszy

Operatory	Symbol
bitowe OR	
logiczne AND	&&
logiczne OR	
obsługa przypisania null (<i>null-coalescing</i>)	??
warunkowy	?:
przypisania, lambda	=, +=, -=, *=, /=, %=%, >>=, <<=, &=, ^=, =, =>

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 7.1

Zadeklaruj trzy zmienne typu int: a, b i c. Przypisz im dowolne wartości całkowite. Wykonaj działanie a % b % c. Wynik tego działania przypisz czwartej zmiennej o nazwie wynik, jej wartość wyświetl na ekranie. Spróbuj dobrze wartości zmennych tak, aby wynikiem nie było 0.

Ćwiczenie 7.2

Zadeklaruj zmienną typu int o dowolnej nazwie. Przypisz jej wartość 256. Wykonaj na niej działania: przesunięcie bitowe w prawo o dwa miejsca i przesunięcie bitowe w lewo o dwa miejsca. Wyniki działań wyświetl na ekranie.

Ćwiczenie 7.3

Zadeklaruj dwie zmienne typu int. Przypisz im dowolne wartości i wykonaj na nich przykładowe działania sumy bitowej oraz iloczynu bitowego. Wyświetl wyniki na ekranie.

Ćwiczenie 7.4

Zadeklaruj zmienną typu int o dowolnej nazwie i przypisz jej dowolną wartość całkowitą. Dwukrotnie wykonaj na niej operację XOR, wykorzystując jako drugi argument również dowolną liczbę całkowitą. Wykorzystaj operator przypisania ^=. Zaobserwuj otrzymany wynik i zastanów się, dlaczego ma on taką właśnie postać.

Ćwiczenie 7.5

Zadeklaruj zmienną typu dowolnego całkowitoliczbowego i przypisz jej początkową wartość 1. Wykonaj operacje powodujące zwiększenie wartości zmiennej o 3 bez używania operatorów =, +, +=. Wykonaj zadanie co najmniej trzema różnymi sposobami.

Ćwiczenie 7.6

Umieść w programie dwie zmienne typu całkowitoliczbowego i przypisz im dowolne wartości. Napisz kod zamieniający wartości zmiennych, tzn. w pierwszej zmiennej ma się znaleźć wartość zapisana w drugiej, a w drugiej — wartość zapisana w pierwszej. Przy realizacji zadania nie używaj dodatkowych zmiennych.

Instrukcje sterujące

Lekcja 8. Instrukcja warunkowa if...else

Lekcja 8. w całości poświęcona jest instrukcjom warunkowym, a dokładniej — różnym postaciom instrukcji warunkowej `if...else`. Może ona bowiem występować zarówno w formie prostej służącej do zbadania jednego tylko warunku, jak i w formach złożonych składających się z wielu członów. Instrukcje warunkowe mogą też być swobodnie zagnieżdżane, co pozwala na badanie różnych, zależnych od siebie warunków. Przedstawione zostaną również formy zapisu pozwalające na utworzenie mniej rozwlekłego kodu.

Podstawowa postać instrukcji if...else

Instrukcje warunkowe służą, jak sama nazwa wskazuje, do sprawdzania warunków. Dzięki temu w zależności od tego, czy dany warunek jest prawdziwy, czy nie, można wykonać różne bloki instrukcji. W C# instrukcja warunkowa ma ogólną postać:

```
if (warunek)
{
    instrukcje do wykonania, kiedy warunek jest prawdziwy
}
else
{
    instrukcje do wykonania, kiedy warunek jest fałszywy
}
```

Zatem po słowie kluczowym `if` w nawiasie okrągłym umieszcza się warunek do sprawdzenia, a za nim w nawiasie klamrowym blok instrukcji do wykonania, gdy warunek jest prawdziwy. Dalej następuje słowo kluczowe `else`, a za nim, również w nawiasie klamrowym, blok instrukcji, które zostaną wykonane, gdy warunek będzie fałszywy.

Spotyka się też nieco inny sposób zapisu (charakterystyczny dla języków takich jak C, C++, Java), w którym pierwsze znaki nawiasu klamrowego znajdują się w tych samych wierszach co słowa `if` i `else`, schematycznie:

```
if (warunek) {  
    instrukcje do wykonania, kiedy warunek jest prawdziwy  
}  
else {  
    instrukcje do wykonania, kiedy warunek jest fałszywy  
}
```

Nie ma to znaczenia formalnego, a jedynie estetyczne i można stosować tę formę, która jest dla nas czytelniejsza (lub która jest wymagana w danym projekcie programistycznym). Referencyjnie w C# stosowana jest pierwsza z zaprezentowanych form.

Nawiasy klamrowe można pominąć, jeśli w bloku ma być wykonana tylko jedna instrukcja. Wówczas instrukcja przyjmie skondensowaną postać:

```
if(warunek)  
    instrukcja1;  
else  
    instrukcja2;
```

Nie należy wtedy zapomnieć o średnikach kończących instrukcje.

Blok `else` jest jednak opcjonalny, zatem prawidłowa jest również konstrukcja:

```
if (warunek)  
{  
    instrukcje do wykonania, kiedy warunek jest prawdziwy  
}
```

lub:

```
if(warunek)  
    instrukcja;
```

Zobaczmy, jak to wygląda w praktyce. Sprawdzamy, czy zmienna typu `int` jest większa od 0, i wyświetlimy odpowiedni komunikat na ekranie. Kod realizujący takie zadanie jest widoczny na listingu 2.19. Na początku deklarujemy zmienną `liczba` typu `int` i przypisujemy jej wartość 15. Następnie za pomocą instrukcji `if` sprawdzamy, czy jest ona większa od 0. Wykorzystujemy w tym celu operator porównywania `>` (Lekcja 7.). Ponieważ zmiennej `liczba` przypisaliśmy stałą wartość równą 15, która na pewno jest większa od 0, zostanie oczywiście wyświetlony napis: `Zmienna liczba jest większa od zera.`. Jeśli przypiszemy jej wartość ujemną lub równą 0, to zostanie wykonany blok `else`.

Listing 2.19. Użycie instrukcji `if`

```
using System;  
  
public class Program  
{  
    public static void Main()  
    {  
        int liczba = 15;  
        if (liczba > 0)  
        {  
            Console.WriteLine("Zmienna liczba jest większa od zera.");  
        }  
    }  
}
```

```
        else
    {
        Console.WriteLine("Zmienna liczba nie jest większa od zera.");
    }
}
```

Zgodnie z tym, co zostało napisane wcześniej — jeśli w blokach po `if` lub `else` znajduje się tylko jedna instrukcja, to można pominąć nawiasy klamrowe. A zatem program mógłby również mieć postać przedstawioną na listingu 2.20. To, która z form zostanie wykorzystana, zależy od indywidualnych preferencji programisty. W trakcie dalszej nauki będzie jednak stosowana głównie postać z listingu 2.19 (jako forma zalecana ze względu na większą czytelność i łatwość dalszej rozbudowy kodu).

Listing 2.20. Pominiecie nawiasów klamrowych w instrukcji `if...else`

```
using System;

public class Program
{
    public static void Main()
    {
        int liczba = 15;
        if (liczba > 0)
            Console.WriteLine("Zmienna liczba jest większa od zera.");
        else
            Console.WriteLine("Zmienna liczba nie jest większa od zera.");
    }
}
```

Zagnieżdżanie instrukcji `if...else`

Ponieważ w nawiasach klamrowych występujących po `if` i po `else` mogą znaleźć się dowolne instrukcje, można tam również umieścić kolejne instrukcje `if...else`. Innymi słowy, instrukcje te można zagnieżdżać. Schematycznie wygląda to następująco:

```
if (warunek1)
{
    if(warunek2)
    {
        instrukcje1
    }
    else
    {
        instrukcje2
    }
}
else
{
    if (warunek3)
    {
        instrukcje3
    }
}
```

```

    else
    {
        instrukcje4
    }
}

```

Taka struktura ma następujące znaczenie: *instrukcje1* zostaną wykonane, kiedy prawdziwe będą warunki *warunek1* i *warunek2*; *instrukcje2* — kiedy prawdziwy będzie warunek *warunek1*, a fałszywy — *warunek2*; *instrukcje3* — kiedy fałszywy będzie warunek *warunek1* i prawdziwy będzie *warunek3*; instrukcje *instrukcje4*, kiedy będą fałszywe warunki *warunek1* i *warunek3*. Oczywiście nie trzeba się ograniczać do przedstawionych tu dwóch poziomów zagnieżdżenia — może ich być dużo więcej — należy jednak zwrócić uwagę, że każdy kolejny poziom zagnieżdżenia zmniejsza czytelność kodu.

Spróbujmy wykorzystać taką konstrukcję do wykonania bardziej skomplikowanego przykładu. Napiszemy program rozwiązujący klasyczne równanie kwadratowe. Jak wiadomo ze szkoły, równanie takie ma postać: $A \times x^2 + B \times x + C = 0$, gdzie *A*, *B* i *C* to parametry równania. Równanie ma rozwiązanie w zbiorze liczb rzeczywistych, jeśli parametr Δ (delta) równy $B^2 - 4 \times A \times C$ jest większy od 0 lub równy 0. Jeśli Δ równa jest 0, mamy jedno rozwiązanie równe $\frac{-B}{2 \times A}$; jeśli Δ jest większa od 0, mamy dwa

rozwiązań: $x_1 = \frac{-B + \sqrt{\Delta}}{2 \times A}$ i $x_2 = \frac{-B - \sqrt{\Delta}}{2 \times A}$. Taka liczba warunków doskonale predysponuje to zadanie do przećwiczenia działania instrukcji *if..else*. Jedyną niedogodnością programu będzie to, że parametry *A*, *B* i *C* będą musiały być wprowadzone bezpośrednio w kodzie programu, nie przedstawiono bowiem jeszcze sposobu na wczytywanie danych z klawiatury (zostanie to omówione dopiero w rozdziale 5.). Cały program jest pokazany na listingu 2.21.

Listing 2.21. Program rozwiązujący równania kwadratowe

```

using System;

public class Program
{
    public static void Main()
    {
        //deklaracja zmiennych
        int A = 2, B = 3, C = -2;

        //wyświetlenie parametrów równania
        Console.WriteLine("Parametry równania:\n");
        Console.WriteLine("A = " + A + ", B = " + B + ", C = " + C + "\n");

        //sprawdzenie, czy jest to równanie kwadratowe
        //a jest równe zero, równanie nie jest kwadratowe
        if (A == 0)
        {
            Console.WriteLine("To nie jest równanie kwadratowe: A = 0!");
        }
    }
}

```

```
        }
        //A jest różne od zera, równanie jest kwadratowe
        else
        {
            //obliczenie delty
            double delta = B * B - 4 * A * C;

            //jeśli delta mniejsza od zera
            if (delta < 0)
            {
                Console.WriteLine("Delta < 0.");
                Console.WriteLine("To równanie nie ma rozwiązań w zbiorze liczb rzeczywistych");
            }
            //jeśli delta większa lub równa zero
            else
            {
                //deklaracja zmiennej pomocniczej
                double wynik;
                //jeśli delta równa zero
                if (delta == 0)
                {
                    //obliczenie wyniku
                    wynik = -B / (2 * A);
                    Console.WriteLine("Rozwiązanie: x = " + wynik);
                }
                //jeśli delta większa od zera
                else
                {
                    //obliczenie wyników
                    wynik = (-B + Math.Sqrt(delta)) / (2 * A);
                    Console.WriteLine("Rozwiązanie: x1 = " + wynik);
                    wynik = (-B - Math.Sqrt(delta)) / (2 * A);
                    Console.WriteLine(", x2 = " + wynik);
                }
            }
        }
    }
}
```

Zaczynamy od zadeklarowania i zainicjowania trzech **zmiennych**, A, B i C, odzwierciedlających parametry równania. Następnie wyświetlamy je na ekranie. Za pomocą instrukcji **if** sprawdzamy, czy zmienna A jest równa 0. Jeśli tak, oznacza to, że równanie nie jest kwadratowe — na ekranie pojawia się wtedy odpowiedni komunikat i program kończy działanie. Jeśli jednak A jest różne od 0, można przystąpić do obliczenia delty. Wynik obliczeń przypisujemy zmiennej o nazwie **delta**. Zmienna ta jest typu **double**, to znaczy może przechowywać liczby **zmiennoprzecinkowe**. Jest to konieczne, jako że **delta** nie musi być liczbą całkowitą.

Kolejny krok to sprawdzenie, czy **delta** nie jest mniejsza od 0. Jeśli jest, oznacza to, że równanie nie ma rozwiązań w zbiorze liczb rzeczywistych, wyświetlamy więc stosowny komunikat na ekranie. Jeśli jednak **delta** nie jest mniejsza od 0, przystępujemy do sprawdzenia kolejnych warunków. Przede wszystkim badamy, czy **delta** jest równa 0

— w takiej sytuacji można od razu obliczyć rozwiązanie równania ze wzoru $-B / (2 * A)$. Wynik tych obliczeń przypisujemy zmiennej pomocniczej o nazwie wynik i wyświetlamy komunikat z rozwiązaniem na ekranie.

W przypadku gdy delta jest większa od 0, mamy dwa pierwiastki (rozwiązań) równania. Obliczamy je w liniach:

```
wynik = (-B + Math.Sqrt(delta)) / (2 * A);
```

oraz:

```
wynik = (-B - Math.Sqrt(delta)) / (2 * A);
```

Rezultat obliczeń wyświetlamy oczywiście na ekranie, tak jak jest to widoczne na rysunku 2.16. Nieomawiana do tej pory instrukcja `Math.sqrt(delta)` powoduje obliczenie pierwiastka kwadratowego (drugiego stopnia) z wartością zawartej w zmiennej `delta`.

Rysunek 2.16.

Przykładowy wynik działania programu rozwiązującego równania kwadratowe

```
C:\WINDOWS\system32\cmd.exe
C:\cs>program.exe
Parametry równania:
A = 2, B = 3, C = -2
Rozwiązanie: x1 = 0,5, x2 = -2
C:\cs>-
```

Instrukcja if...else if

Zagnieżdżanie instrukcji `if` sprawdza się dobrze w tak prostym przykładzie jak omówiony wyżej, jednak z każdym kolejnym poziomem staje się coraz bardziej nieczytelne. Nadmiernemu zagnieżdżaniu można zapobiec przez zastosowanie nieco zmodyfikowanej instrukcji w postaci `if...else if`. Założmy, że mamy znaną nam już konstrukcję instrukcji `if` w postaci:

```
if(warunek1)
{
    instrukcje1
}
else
{
    if(warunek2)
    {
        instrukcje2
    }
    else
    {
        if(warunek3)
        {
            instrukcje3
        }
        else
        {
            instrukcje4
        }
    }
}
```

```
    }
}
}
```

Innymi słowy, mamy sprawdzić po kolej warunki *warunek1*, *warunek2* i *warunek3* i w zależności od tego, które są prawdziwe, wykonać instrukcje *instrukcje1*, *instrukcje2*, *instrukcje3* lub *instrukcje4*. Zatem *instrukcje1* są wykonywane, kiedy *warunek1* jest prawdziwy; *instrukcje2*, kiedy *warunek1* jest fałszywy, a *warunek2* prawdziwy; *instrukcje3* — kiedy prawdziwy jest *warunek3*, natomiast fałszywe są *warunek1* i *warunek2*; *instrukcje4* są natomiast wykonywane, kiedy wszystkie warunki są fałszywe. Jest to zobrazowane w tabeli 2.18.

Tabela 2.18. Wykonanie instrukcji w zależności od stanu warunków

Wykonaj instrukcję	<i>warunek1</i>	<i>warunek2</i>	<i>warunek3</i>
<i>instrukcje1</i>	Prawdziwy	Bez znaczenia	Bez znaczenia
<i>instrukcje2</i>	Fałszywy	Prawdziwy	Bez znaczenia
<i>instrukcje3</i>	Fałszywy	Fałszywy	Prawdziwy
<i>instrukcje4</i>	Fałszywy	Fałszywy	Fałszywy

Konstrukcję taką możemy zamienić na identyczną znaczeniowo (semantycznie), ale prostszą w zapisie instrukcję *if...else if* w postaci:

```
if(warunek1)
{
    instrukcje1
}
else if (warunek2)
{
    instrukcje2
}
else if(warunek3)
{
    instrukcje3
}
else
{
    instrukcje4
}
```

Zapis taki tłumaczymy następująco: „Jeśli prawdziwy jest *warunek1*, wykonaj *instrukcje1*; w przeciwnym wypadku, jeżeli prawdziwy jest *warunek2*, wykonaj *instrukcje2*; w przeciwnym wypadku, jeśli prawdziwy jest *warunek3*, wykonaj *instrukcje3*; w przeciwnym wypadku wykonaj *instrukcje4*”.

Zauważmy, że konstrukcja ta pozwoli nam uprościć nieco kod przykładowu z listingu 2.21, obliczający pierwiastki równania kwadratowego. Zamiast sprawdzać, czy delta jest mniejsza, większa, czy równa 0, za pomocą zagnieżdzonej instrukcji *if*, łatwiej będzie skorzystać z instrukcji *if...else if*. Zobrazowano to na listingu 2.22.

Listing 2.22. Użycie instrukcji if...else if do rozwiązywania równania kwadratowego

```
using System;

public class Program
{
    public static void Main()
    {
        //deklaracja zmiennych
        int A = 2, B = 3, C = -2;

        //wyświetlenie parametrów równania
        Console.WriteLine("Parametry równania:\n");
        Console.WriteLine("A = " + A + ", B = " + B + ", C = " + C + "\n");

        //sprawdzenie, czy jest to równanie kwadratowe

        //A jest równe zero, równanie nie jest kwadratowe
        if (A == 0)
        {
            Console.WriteLine("To nie jest równanie kwadratowe: A = 0!");
        }
        //A jest różne od zera, równanie jest kwadratowe
        else
        {
            //obliczenie deltę
            double delta = B * B - 4 * A * C;

            //jeśli delta mniejsza od zera
            if (delta < 0)
            {
                Console.WriteLine("Delta < 0.");
                Console.WriteLine("To równanie nie ma rozwiązań w zbiorze liczb rzeczywistych.");
            }
            //jeśli delta równa zero
            else if(delta == 0)
            {
                //obliczenie wyniku
                double wynik = -B / (2 * A);
                Console.WriteLine("Rozwiązanie: x = " + wynik);

            }
            //jeśli delta większa od zera
            else
            {
                double wynik;
                //obliczenie wyników
                wynik = (-B + Math.Sqrt(delta)) / (2 * A);
                Console.WriteLine("Rozwiązanie: x1 = " + wynik);
                wynik = (-B - Math.Sqrt(delta)) / (2 * A);
                Console.WriteLine(", x2 = " + wynik);
            }
        }
    }
}
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 8.1

Zadeklaruj dwie zmienne typu int: a i b. Przypisz im dowolne wartości całkowite. Użyj instrukcji if do sprawdzenia, czy dzielenie modulo (reszta z dzielenia) a przez b daje w wyniku 0. Wyświetl stosowny komunikat na ekranie.

Ćwiczenie 8.2

Napisz program, którego zadaniem będzie ustalenie, czy równanie kwadratowe ma rozwiązanie w zbiorze liczb rzeczywistych.

Ćwiczenie 8.3

Napisz program, w którego kodzie znajdzie się zmienna typu int (lub innego typu liczbowego). Wyświetl wartość bezwzględną tej zmiennej. Użyj instrukcji warunkowej if.

Ćwiczenie 8.4

Zawrzyj w kodzie programu zmienne określające współrzędne dwóch punktów na płaszczyźnie (mogą być typu int). Wyświetl na ekranie informację, czy prosta przechodząca przez te punkty będzie równoległa do osi OX (czyli będzie pozioma) lub OY (czyli będzie pionowa), a jeśli tak, to do której.

Ćwiczenie 8.5

Napisz program zawierający dane prostokąta (współrzędna lewego górnego rogu oraz szerokość i wysokość) oraz punktu. Wszystkie dane mogą być w postaci wartości całkowitych typu int (lub podobnego). Wyświetl informację, czy punkt zawiera się w prostokącie.

Ćwiczenie 8.6

Napisz program, który będzie ustalać, czy kwadrat o zadanej długości boku $dł$ zmieści się wewnątrz okręgu o zadanym promieniu r (dane $dł$ i r umieść w postaci zmiennych o zadanach wartościach).

Lekcja 9. Instrukcja switch i operator warunkowy

W lekcji 8. omówiono instrukcję warunkową `if` w kilku różnych postaciach. W praktyce wystarczyłaby ona do obsługi wszelkich zadań programistycznych związanych ze sprawdzaniem warunków. Okazuje się jednak, że czasami wygodniejsze są inne konstrukcje warunkowe, i właśnie im jest poświęcona lekcja 9. Zostaną dokładnie opisane: instrukcja `switch`, nazywana instrukcją wyboru, oraz tak zwany operator warunkowy. Osoby znające języki takie jak C, C++ i Java powinny zwrócić uwagę na różnice związane z przekazywaniem sterowania w instrukcji wyboru. Z kolei osoby dopiero rozpoczynające naukę czy też znające np. Pascala powinny dokładniej zapoznać się z całym przedstawionym materiałem.

Instrukcja switch

Instrukcja `switch` pozwala w wygodny i przejrzysty sposób sprawdzić ciąg warunków i wykonywać różny kod w zależności od tego, czy są one prawdziwe, czy fałszywe. W najprostszej postaci może być ona odpowiednikiem ciągu `if...else if`, w którym jako warunek jest wykorzystywane porównywanie zmiennej do wybranej liczby. Daje ona jednak programistę dodatkowe możliwości, jak choćby wykonanie tego samego kodu dla kilku warunków. Jeśli mamy przykładowy zapis:

```
if(liczba == 1)
{
    instrukcje1
}
else if (liczba == 2)
{
    instrukcje2
}
else if(liczba == 3)
{
    instrukcje3
}
else
{
    instrukcje4
}
```

można go przedstawić za pomocą instrukcji `switch`, która będzie wyglądać następująco:

```
switch(liczba)
{
    case 1 :
        instrukcje1;
        break;
    case 2 :
        instrukcje2;
        break;
    case 3 :
        instrukcje3;
```

```
        break;
    default :
        instrukcje4;
        break;
}
```

W rzeczywistości w nawiasie okrągły występującym po switch nie musi występować nazwa zmiennej, ale może się tam znaleźć dowolne wyrażenie, którego wynikiem będzie wartość arytmetyczna całkowitoliczbową, typu char lub string¹⁴. W postaci ogólnej cała konstrukcja wygląda zatem następująco:

```
switch(wyrażenie)
{
    case wartość1 :
        instrukcje1;
        break;
    case wartość2 :
        instrukcje2;
        break;
    case wartość3 :
        instrukcje3;
        break;
    default :
        instrukcje4;
        break;
}
```

Należy ją rozumieć następująco: „Sprawdź wartość wyrażenia *wyrażenie*; jeśli jest to *wartość1*, wykonaj *instrukcję1* i przerwij wykonywanie bloku switch (instrukcja break); jeżeli jest to *wartość2*, wykonaj *instrukcję2* i przerwij wykonywanie bloku switch; jeśli jest to *wartość3*, wykonaj *instrukcję3* i przerwij wykonywanie bloku switch; jeżeli nie zachodzi żaden z wymienionych przypadków, wykonaj *instrukcję4* i zakończ blok switch”.

Zobaczmy, jak to działa na konkretnym przykładzie. Został on zaprezentowany na listingu 2.23.

Listing 2.23. Proste użycie instrukcji wyboru switch

```
using System;

public class Program
{
    public static void Main()
    {
        int liczba = 25;
        switch(liczba)
        {
            case 25 :
                Console.WriteLine("liczba = 25");
                break;
            case 15 :
                Console.WriteLine("liczba = 15");
                break;
        }
    }
}
```

¹⁴ Lub też istnieje niejawnia konwersja do jednego z tych typów.

```
        break;
    default :
        Console.WriteLine("Zmienna liczba nie jest równa ani 15, ani 25.");
        break;
    }
}
```

Na początku deklarujemy zmienną o nazwie `liczba` i typie `int`, czyli taką, która może przechowywać liczby całkowite, i przypisujemy jej wartość 25. Następnie wykorzystujemy instrukcję `switch` do sprawdzenia stanu zmiennej i wyświetlenia odpowiedniego napisu. W tym wypadku wartością wyrażenia będącego parametrem instrukcji `switch` jest oczywiście wartość zapisana w zmiennej `liczba`. Nic nie stoi jednak na przeszkodzie, aby parametr ten był wyliczany dynamicznie w samej instrukcji. Jest to widoczne w przykładzie na listingu 2.24.

Listing 2.24. Obliczanie wartości wyrażenia bezpośrednio w instrukcji `switch`

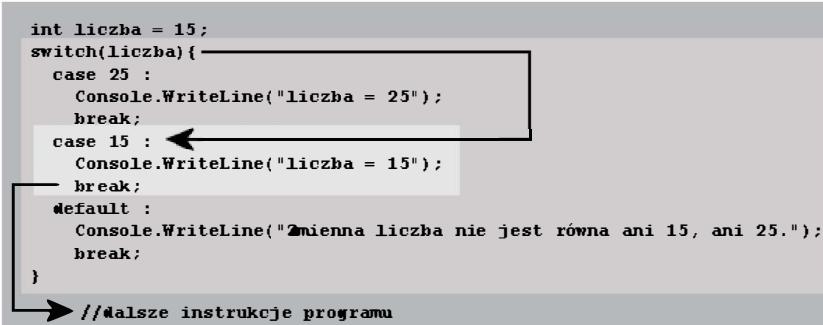
```
using System;

public class Program
{
    public static void Main()
    {
        int liczba1 = 2;
        int liczba2 = 1;
        switch(liczba1 * 5 / (liczba2 + 1))
        {
            case 5 :
                Console.WriteLine("liczba = 5");
                break;
            case 15 :
                Console.WriteLine("liczba = 15");
                break;
            default :
                Console.WriteLine("Zmienna liczba nie jest równa ani 5, ani 15.");
                break;
        }
    }
}
```

Zatem instrukcja `switch` najpierw oblicza wartość wyrażenia występującego w nawiasie okrągłym (jeśli jest to zmienna, wtedy w to miejsce jest podstawiana jej wartość), a następnie próbuje ją dopasować do jednej z wartości występujących po słowach `case`. Jeśli zgodność zostanie stwierdzona, będą wykonane instrukcje występujące w danym bloku `case`. Jeżeli nie uda się dopasować wartości wyrażenia do żadnej z wartości występujących po słowach `case`, będzie wykonywany blok `default`. Blok `default` nie jest jednak obligatoryjny i jeśli nie jest nam w programie potrzebny, można go pominąć.

Przerywanie instrukcji switch

W przypadku instrukcji switch w każdym bloku case musi wystąpić instrukcja przekazująca sterowanie w inne miejsce programu. W przykładach z listingów 2.23 i 2.24 była to instrukcja break, powodująca przerwanie działania całej instrukcji switch. Tak więc w tych przypadkach break powodowało przekazanie sterowania za instrukcję switch, tak jak zostało to schematycznie pokazane na rysunku 2.17.



Rysunek 2.17. Schemat przerwania instrukcji switch przez break

Istnieją jednak inne możliwości przerwania danego bloku case. Można też użyć instrukcji return (jednak zostanie ona omówiona dopiero w dalszej części książki) lub goto. Instrukcja goto (ang. go to — „idź do”) powoduje przejście do wyznaczonego miejsca w programie. Może wystąpić w trzech wersjach:

- ◆ goto etykiet; — powoduje przejście do etykiety o nazwie *etykiet*;
- ◆ goto case przypadek_case; — powoduje przejście do wybranego bloku case;
- ◆ goto default; — powoduje przejście do bloku default.

Każdy z tych przypadków został użyty na listingu 2.25. Etykieta powstaje poprzez umieszczenie przed jakąś instrukcją nazwy etykiety zakończonej znakiem dwukropka, schematycznie:

nazwa_etykiety:
instrukcja;



Kod z listingu 2.24 to jedynie przykład działania różnych wersji instrukcji goto w jednym bloku switch oraz tego, jak NIE należy pisać programów. Używanie tego typu konstrukcji prowadzi jedynie do zaciemniania kodu i nie powinno się ich stosować w praktyce.

Listing 2.25. Ilustracja użycia instrukcji goto

```

using System;

public class Program
{
    public static void Main()

```

```
{  
    int liczba = 1;  
    etykieta:  
    switch(liczba)  
    {  
        case 1 :  
            Console.WriteLine("liczba = 1");  
            goto default;  
        case 2 :  
            Console.WriteLine("liczba = 2");  
            goto default;  
        case 3 :  
            liczba--;  
            goto case 4;  
        case 4 :  
            goto etykieta2;  
        default :  
            liczba++;  
            goto etykieta;  
    }  
    etykieta2:  
    Console.Write("Blok switch został zakończony: ");  
    Console.WriteLine("liczba = {0}", liczba);  
}  
}
```

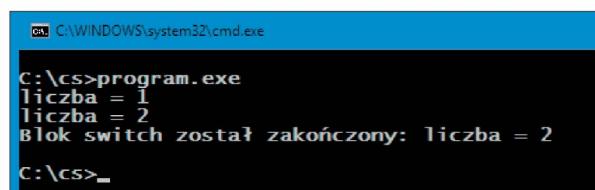
Jak więc działa ten program? Na początku została zadeklarowana zmienna `liczba` i została jej przypisana wartość 1, natomiast tuż przed instrukcją `switch` została umieszczona etykieta o nazwie `etykieta`. Ponieważ `liczba` ma wartość 1, w instrukcji `switch` zostanie wykonany blok `case 1`. W tym bloku na ekranie jest wyświetlany napis określający stan zmiennej oraz wykonywana jest instrukcja `goto default`. Tym samym sterowanie zostanie przekazane do bloku `default`.

W bloku `default` zmienna `liczba` jest zwiększana o 1 (`liczba++`), a następnie jest wykonywana instrukcja `goto etykieta`. Oznacza to przejście do miejsca w programie oznaczonego etykietą o nazwie `etykieta` (a dokładniej — do instrukcji oznaczonej etykietą, czyli pierwszej instrukcji za etykietą). Tym miejscem jest oczywiście początek instrukcji `switch`, zostanie więc ona ponownie wykonana.

Ponieważ jednak tym razem `liczba` jest już równa 2 (jak pamiętamy, zwiększyliśmy jej początkową wartość o 1), zostanie wykonany blok `case 2`. W tym bloku, podobnie jak w przypadku `case 1`, na ekranie jest wyświetlany napis określający stan zmiennej i wykonywana jest instrukcja `goto default`. Zmienna `liczba` w bloku `default` ponownie zostanie zwiększona o jeden (będzie więc równa już 3), a sterowanie będzie przekazane na początek instrukcji `switch` (`goto etykieta;`).

Trzecie wykonanie instrukcji `switch` spowoduje wejście do bloku `case 3` (ponieważ zmienna `liczba` będzie wtedy równa 3). W tym bloku wartość `liczba` jest zmniejszana o 1 (`liczba--`), a następnie sterowanie jest przekazywane do bloku `case 4`. W bloku `case 4` znajduje się instrukcja `goto etykieta2`, powodująca opuszczenie instrukcji `switch` i udanie się do miejsca oznaczonego jako `etykieta2`. Ostatecznie na ekranie zobaczymy widok przedstawiony na rysunku 2.18.

Rysunek 2.18.
Wynik działania
programu
z listingu 2.25



```
C:\> C:\WINDOWS\system32\cmd.exe
C:\>program.exe
liczba = 1
liczba = 2
Blok switch został zakończony: liczba = 2
C:\>
```

Jeśli ktoś sądzi, że przedstawiony kod jest niepotrzebnie zagmatwany, ma całkowitą rację. To tylko ilustracja możliwości przekazywania sterowania programu w instrukcji switch i działania różnych wersji goto. Stosowanie takich konstrukcji w praktyce nie prowadzi do niczego dobrego. Potraktujmy to więc jako przykład tego, jak nie należy programować.

Operator warunkowy

Operator warunkowy ma postać:

```
warunek ? wartość1 : wartość2
```

Oznacza ona: „Jeżeli *warunek* jest prawdziwy, podstaw za wartość wyrażenia *wartość1*; w przeciwnym wypadku podstaw *wartość2*”. Zobaczmy w praktyce, jak może wyglądać jego wykorzystanie. Zobrazowano to w kodzie widocznym na listingu 2.26.

Listing 2.26. Użycie operatora warunkowego

```
using System;

public class Program
{
    public static void Main()
    {
        int liczba = 10;
        int liczba2;
        liczba2 = liczba < 0 ? -1 : 1;
        Console.WriteLine(liczba2);
    }
}
```

Najważniejsza jest tu oczywiście linia `liczba2 = liczba < 0 ? -1 : 1;`. Po lewej stronie operatora przypisania = znajduje się zmienna (`liczba2`), natomiast po stronie prawej wyrażenie warunkowe, czyli linia ta oznacza: „Przypisz zmiennej `liczba2` wartość wyrażenia warunkowego”. Jaka jest ta wartość? Trzeba przeanalizować samo wyrażenie: `liczba < 0 ? -1 : 1`. Oznacza ono zgodnie z tym, co zostało napisane w poprzednim akapicie: „Jeżeli wartość zmiennej `liczba` jest mniejsza od 0, przypisz wyrażeniu wartość `-1`, w przeciwnym wypadku (zmienna `liczba` większa lub równa 0) przypisz wartość `1`”. Ponieważ zmiennej `liczba` przepisaliśmy wcześniej 10, wartością całego wyrażenia będzie 1 i ta właśnie wartość zostanie przypisana zmiennej `liczba2`. Dla zwiększenia czytelności do wyrażenia można by też dodać nawias okrągły:

```
liczba2 = (liczba < 0) ? -1 : 1;
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 9.1

Napisz instrukcję switch zawierającą 10 bloków case sprawdzających kolejne wartości całkowite od 0 do 9. Pamiętaj o instrukcjach break.

Ćwiczenie 9.2

Zadeklaruj zmienną typu bool. Wykorzystaj wyrażenie warunkowe do sprawdzenia, czy wynikiem dowolnego dzielenia modulo jest wartość 0. Jeśli tak, przypisz zmiennej typu bool wartość true, w przeciwnym razie — wartość false.

Ćwiczenie 9.3

Napisz program działający podobnie do kodu z ćwiczenia 8.3. Zamiast instrukcji if użyj jednak operatora warunkowego.

Ćwiczenie 9.4

Zmodyfikuj program z listingu 2.26 w taki sposób, aby badanie stanu zmiennej liczba1 było wykonywane za pomocą instrukcji warunkowej if...else.

Ćwiczenie 9.5

Napisz instrukcję switch badającą wartość pewnej zmiennej typu całkowitoliczbowego i wyświetlającą komunikaty w następujących sytuacjach: a) zmienna ma wartość 1, 4, 8; b) zmienna ma wartość 2, 3, 7; c) wszystkie inne przypadki. W każdym wymienionym przypadku na ekranie ma się pojawić dokładnie jeden komunikat o dowolnej treści. Postaraj się całość zapisać możliwie krótko.

Lekcja 10. Pętle

Pętle są konstrukcjami programistycznymi, które pozwalają na wykonywanie powtarzających się czynności. Jeśli na przykład chcemy wyświetlić na ekranie dziesięć razy dowolny napis, najłatwiej będzie skorzystać właśnie z odpowiedniej pętli. Oczywiście można też dziesięć razy napisać w kodzie programu `Console.WriteLine("napis")`, jednak będzie to z pewnością niezbyt wygodne. W tej lekcji będą omówione wszystkie występujące w C# rodzaje pętli, czyli pętle for, while oraz do...while i foreach. Przedstawione zostaną także występujące między nimi różnice oraz przykłady wykorzystania.

Pętla for

Pętla for ma ogólną postać:

```
for (wyrażenie początkowe; wyrażenie warunkowe; wyrażenie modyfikujące)
{
    instrukcje do wykonania
}
```

wyrażenie początkowe jest stosowane do zainicjalizowania zmiennej używanej jako licznik liczby wykonania pętli. *wyrażenie warunkowe* określa warunek, jaki musi być spełniony, aby dokonać kolejnego przejścia w pętli, natomiast *wyrażenie modyfikujące* jest zwykle używane do modyfikacji zmiennej będącej licznikiem. Najłatwiej wyjaśnić to wszystko na praktycznym przykładzie. Spójrzmy na listing 2.27.

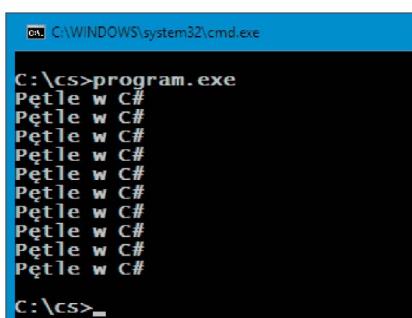
Listing 2.27. Podstawowa pętla for

```
using System;

public class Program
{
    public static void Main()
    {
        for(int i = 0; i < 10; i++)
        {
            Console.WriteLine("Pętla w C#");
        }
    }
}
```

Taką konstrukcję należy rozumieć następująco: „Zadeklaruj zmenną i i przypisz jej wartość 0 (int i = 0), następnie tak długo, jak długo wartość i będzie mniejsza od 10, wykonuj instrukcję Console.WriteLine(“Pętle w C#”) oraz zwiększaj i o 1 (i++”). Tym samym na ekranie pojawi się dziesięć razy napis Pętle w C# (rysunek 2.19). Zmienna i jest nazywana **zmenną iteracyjną**, czyli kontrolującą kolejne przebiegi (iteracje) pętli. Zwyczajowo nazwy zmiennych iteracyjnych zaczynają się od litery *i*, po czym w razie potrzeby używa się kolejnych liter alfabetu (*j*, *k*, *l*). Zobaczmy to już na kolejnych stronach przy omawianiu pętli zagnieżdżonych.

Rysunek 2.19.
*Wynik działania
prostej pętli typu for*



Jeśli chcemy zobaczyć, jak zmienia się stan zmiennej i w trakcie kolejnych przebiegów, możemy modyfikować instrukcję wyświetlającą napis, tak aby pojawiała się również i ta informacja. Wystarczy drobna modyfikacja w postaci:

```
for(int i = 0; i < 10; i++)
{
    Console.WriteLine("[i = " + i + "] Pętle w C#");
}
```

lub też, co wydaje się czytelniejsze:

```
for(int i = 0; i < 10; i++)
{
    Console.WriteLine("[i = {0}] Pętle w C#", i);
}
```

Po jej wprowadzeniu w każdej linii będzie wyświetlana również wartość i, tak jak zostało to przedstawione na rysunku 2.20.

Rysunek 2.20.

Pętla for wyświetlająca stan zmiennej iteracyjnej

```
C:\>program.exe
[i = 0] Pętle w C#
[i = 1] Pętle w C#
[i = 2] Pętle w C#
[i = 3] Pętle w C#
[i = 4] Pętle w C#
[i = 5] Pętle w C#
[i = 6] Pętle w C#
[i = 7] Pętle w C#
[i = 8] Pętle w C#
[i = 9] Pętle w C#
C:\>-
```

Wyrażenie początkowe to w powyższym przykładzie `int i = 0`, wyrażenie warunkowe `i < 10`, a wyrażenie modyfikujące — `i++`. Okazuje się, że mamy dużą dowolność umiejscawiania tych wyrażeń. Na przykład wyrażenie modyfikujące, które najczęściej jest wykorzystywane do modyfikacji zmiennej iteracyjnej, można umieścić wewnątrz samej pętli, to znaczy zastosować konstrukcję o następującej schematycznej postaci:

```
for (wyrażenie początkowe; wyrażenie warunkowe;
{
    instrukcje do wykonania
    wyrażenie modyfikujące
})
```

Zmieńmy zatem program z listingu 2.27, tak aby wyrażenie modyfikujące znalazło się wewnątrz pętli. Zostało to zobrazowane na listingu 2.28.

Listing 2.28. Wyrażenie modyfikujące wewnątrz pętli for

```
using System;

public class Program
{
    public static void Main()
    {
```

```
for(int i = 0; i < 10;)
{
    Console.WriteLine("Pętle w C#");
    i++;
}
}
```

Ten program jest funkcjonalnym odpowiednikiem poprzedniego przykładu. Szczególną uwagę należy zwrócić na znak średnika występujący po wyrażeniu warunkowym. Mimo iż wyrażenie modyfikujące znalazło się teraz wewnątrz bloku pętli, ten średnik wciąż jest niezbędny. Jeśli go zabraknie, komplikacja z pewnością się nie powiedzie.

Skoro udało się nam przenieść wyrażenie modyfikujące do wnętrza pętli, spróbujmy dokonać podobnego zabiegu również z wyrażeniem początkowym. Jest to prosty zbieg techniczny. Schematycznie taka konstrukcja wygląda następująco:

```
wyrażenie początkowe;
for (; wyrażenie warunkowe;) {
    instrukcje do wykonania
    wyrażenie modyfikujące;
}
```

Spójrzmy na listing 2.29. Całe wyrażenie początkowe przeniesliśmy po prostu przed pętlę. To jest nadal w pełni funkcjonalny odpowiednik programu z listingu 2.27. Ponownie należy zwrócić uwagę na umiejscowienie średników pętli `for`. Oba są niezbędne do prawidłowego działania kodu.

Listing 2.29. Wyrażenie początkowe przed pętlą `for`

```
using System;

public class Program
{
    public static void Main()
    {
        int i = 0;
        for(; i < 10;)
        {
            Console.WriteLine("Pętle w C#");
            i++;
        }
    }
}
```

Kolejną ciekawą możliwością jest połączenie wyrażenia warunkowego i modyfikującego. Pozostawimy wyrażenie początkowe przed pętlą, natomiast wyrażenie modyfikujące ponownie wprowadzimy do konstrukcji pętli, łącząc je jednak z wyrażeniem warunkowym. Taka konstrukcja jest przedstawiona na listingu 2.30.

Listing 2.30. Połączenie wyrażenia modyfikującego z warunkowym

```
using System;

public class Program
{
    public static void Main()
    {
        int i = 0;
        for(; i++ < 10;)
        {
            Console.WriteLine("Pętle w C#");
        }
    }
}
```

Istnieje również możliwość przeniesienia wyrażenia warunkowego do wnętrza pętli, jednak wymaga to zastosowania instrukcji `break`, która będzie omówiona w lekcji 11. Zwróćmy też uwagę, że przedstawiony wyżej kod nie jest w pełni funkcjonalnym odpowiednikiem pętli z listingów 2.27 – 2.29, choć w pierwszej chwili wyniki działania wydają się identyczne. Warto samodzielnie zastanowić się, dlaczego tak jest, oraz w ramach ćwiczeń wprowadzić odpowiednie modyfikacje (podobna kwestia została także omówiona w przykładach dotyczących pętli `while`).

Pętla while

Pętla typu `while` służy, podobnie jak `for`, do wykonywania powtarzających się czynności. Pętlę `for` najczęściej wykorzystuje się, kiedy liczba powtarzanych operacji jest znana (na przykład zapisana w jakiejś zmiennej), natomiast `while`, kiedy nie jest z góry znana (na przykład wynika z działania jakiejś funkcji). Jest to jednak podział zupełnie umowny: oba typy pętli można zapisać w taki sposób, aby były swoimi funkcjonalnymi odpowiednikami. Ogólna postać pętli `while` wygląda następująco:

```
while (wyrażenie warunkowe)
{
    instrukcje;
}
```

Instrukcje są wykonywane dopóty, dopóki wyrażenie warunkowe jest prawdziwe. Zobaczmy zatem, jak za pomocą pętli `while` wyświetlić na ekranie dziesięć razy dowolny napis. Zobrazowano to w kodzie widocznym na listingu 2.31. Pętlę taką rozumiemy następująco: „Dopóki `i` jest mniejsze od 10, wyświetlaj napis na ekranie, za każdym razem zwiększając `i o 1` (`i++`)”.

Listing 2.31. Prosta pętla `while`

```
using System;

public class Program
{
    public static void Main()
    {
```

```
int i = 0;
while(i < 10)
{
    Console.WriteLine("Pętle w C#");
    i++;
}
}
```

Nic nie stoi na przeszkodzie, aby tak jak w przypadku pętli `for` wyrażenie warunkowe było jednocześnie wyrażeniem modyfikującym. Taka pętla została przedstawiona na listingu 2.32. Ponieważ w wyrażeniu został użyty operator `++`, najpierw `i` jest porównywane z 10, a dopiero potem zwiększone o 1.

Listing 2.32. Połączenie wyrażenia warunkowego z modyfikującym

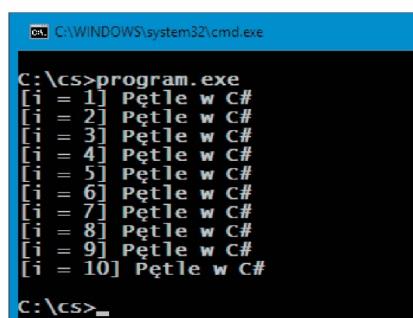
```
using System;

public class Program
{
    public static void Main()
    {
        int i = 0;
        while(i++ < 10)
        {
            Console.WriteLine("Pętle w C#");
        }
    }
}
```

Należy zwrócić uwagę, że mimo iż programy z listingów 2.31 i 2.32 wykonują to samo zadanie, nie są to w pełni funkcjonalne odpowiedniki. Można to zauważyć, dodając instrukcję wyświetlającą stan zmiennej `i` w obu wersjach kodu. Wystarczy zmodyfikować instrukcję `Console.WriteLine("Pętle w C#")` identycznie jak w przypadku pętli `for`: `Console.WriteLine("[i = {0}] Pętle w C#", i)`. Wynik działania kodu, kiedy zmienność `i` jest modyfikowana wewnętrz pętli (tak jak na listingu 2.31), będzie taki sam jak na rysunku 2.20, natomiast wynik jego działania, kiedy zmienność ta jest modyfikowana w wyrażeniu warunkowym (tak jak na listingu 2.32), jest przedstawiony na rysunku 2.21.

Rysunek 2.21.

Stan zmiennej `i`,
kiedy jest
modyfikowana
w wyrażeniu
warunkowym



Widać wyraźnie, że w pierwszym przypadku wartości zmiennej zmieniają się od 0 do 9, natomiast w przypadku drugim od 1 do 10. Nie ma to znaczenia, kiedy jedynie wyświetlamy serię napisów, jednak już w przypadku, gdybyśmy wykorzystywali zmienną i do jakichś celów, np. dostępu do komórek tablicy (tak jak w podrozdziale „Tablice”), ta drobna z pozoru różnica spowodowałaby poważne konsekwencje w działaniu programu. Dobrym ćwiczeniem do samodzielnego wykonania będzie poprawienie programu z listingu 2.32 tak, aby działał dokładnie tak samo jak ten z listingu 2.31 (sekcja „Ćwiczenia do samodzielnego wykonania”).

Pętla do...while

Odmiana pętli `while` jest pętlą `do...while`, której schematyczna postać wygląda następująco:

```
do
{
    instrukcje;
}
while(warunek);
```

Konstrukcję tę należy rozumieć następująco: „Wykonuj *instrukcje*, dopóki *warunek* jest prawdziwy”. Zobaczmy zatem, jak wygląda znane nam zadanie wyświetlenia dziesięciu napisów, jeśli do jego realizacji wykorzystamy pętlę `do...while`. Zobrazowano to w kodzie znajdującym się na listingu 2.33.

Listing 2.33. Użycie pętli `do...while`

```
using System;

public class Program
{
    public static void Main()
    {
        int i = 0;
        do
        {
            Console.WriteLine("[i = {0}] Pętle w C#", i);
        }
        while(i++ < 9);
    }
}
```

Zwróćmy uwagę, jak w tej chwili wygląda warunek. Od razu daje się zauważyc podstawową różnicę w stosunku do pętli `while`. Otóż w pętli `while` najpierw jest sprawdzany warunek, a dopiero potem wykonywane są instrukcje. W przypadku pętli `do...while` jest dokładnie odwrotnie — najpierw są wykonywane instrukcje, a dopiero potem sprawdzany jest warunek. Dlatego też tym razem sprawdzamy, czy *i* jest mniejsze od 9. Gdybyśmy pozostawili starą postać wyrażenia warunkowego, tj. *i++ < 10*, napis zostałby wyświetlony jedenaście razy.

Takiemu zachowaniu można zapobiec, wprowadzając wyrażenie modyfikujące zmienną *i* do wnętrza pętli, czyli sama pętla miałaby wtedy postać:

```
int i = 0;
do
{
    Console.WriteLine("[i = {0}] Pętla w C#", i);
    i++;
}
while(i < 10);
```

Teraz warunek pozostaje w starej postaci i otrzymujemy odpowiednik pętli `while`.

Ta cecha (czyli wykonywanie instrukcji przed sprawdzeniem warunku) pętli `do...while` jest bardzo ważna, oznacza bowiem, że pętla tego typu jest wykonywana zawsze co najmniej raz, nawet jeśli warunek jest fałszywy. Można się o tym przekonać w bardzo prosty sposób — wprowadzając fałszywy warunek i obserwując zachowanie programu, np.:

```
int i = 0;
do
{
    Console.WriteLine("[i = {0}] Pętla w C#", i);
    i++;
}
while(i < 0);
```

lub wręcz:

```
int i = 0;
do
{
    Console.WriteLine("[i = {0}] Pętla w C#", i);
    i++;
}
while(false);
```

Warunki w tych postaciach są ewidentnie fałszywe. W pierwszym przypadku zmienią *i* już w trakcie inicjacji jest równa 0 (nie może być więc jednocześnie mniejsza od 0!), natomiast w drugim warunkiem kontynuacji pętli jest `false`, a więc na pewno nie może być ona kontynuowana. Mimo to po wykonaniu powyższego kodu na ekranie pojawi się jeden napis `[i = 0] Pętla w C#.` Jest to najlepszy dowód na to, że warunek jest sprawdzany nie przed każdym przebiegiem pętli, ale po nim.

Pętla `foreach`

Pętla typu `foreach` pozwala na automatyczną iterację po tablicy lub też po kolekcji¹⁵. Jej działanie zostanie pokazane w tym pierwszym przypadku (Czytelnicy, którzy nie znają pojęcia tablic, powinni najpierw zapoznać się z materiałem zawartym w lekcji 12., kolekcje nie będą natomiast omawiane w tej publikacji). Jeśli bowiem mamy tablicę

¹⁵ Dokładniej rzecz ujmując, „po obiekcie implementującym odpowiedni interfejs pozwalający na iterację po jego elementach” lub prościej „po obiekcie zawierającym tzw. iterator”.

tab zawierającą wartości pewnego typu, to do przejrzenia wszystkich jej elementów możemy użyć konstrukcji o postaci:

```
foreach(typ identyfikator in tablica)
{
    instrukcje
}
```

W takim wypadku w kolejnych przebiegach pętli pod *identyfikator* będzie podstawa- wiana wartość kolejnej komórki. Pętla będzie działała tak długo, aż zostaną przejrzane wszystkie elementy tablicy (lub kolekcji) typu *typ* bądź też zostanie przerwana za pomocą jednej z instrukcji pozwalających na taką operację (lekcia 11.). Przykład użycia pętli typu *foreach* został przedstawiony na listingu 2.34.

Listing 2.34. Użycie pętli *foreach* do odczytania zawartości tablicy

```
using System;

public class Program
{
    public static void Main()
    {
        int[] tab = new int[10] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        foreach(int wartosc in tab)
        {
            Console.WriteLine(wartosc);
        }
    }
}
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 10.1

Wykorzystując pętlę *for*, napisz program, który wyświetli liczby całkowite od 1 do 10 podzielne przez 2.

Ćwiczenie 10.2

Nie zmieniając żadnej instrukcji wewnętrz pętli, zmodyfikuj kod z listingu 2.32 w taki sposób, aby był funkcjonalnym odpowiednikiem kodu z listingu 2.31.

Ćwiczenie 10.3

Wykorzystując pętlę *while*, napisz program, który wyświetli liczby całkowite od 1 do 20 podzielne przez 3.

Ćwiczenie 10.4

Zmodyfikuj kod z listingu 2.33 w taki sposób, aby w wyrażeniu warunkowym pętli *do...while* zamiast operatora < wykorzystać operator <=.

Ćwiczenie 10.5

Napisz program, który wyświetli na ekranie liczby od 1 do 20 i zaznaczy przy każdej z nich, czy jest to liczba parzysta, czy nieparzysta. Zrób to:

- a) wykorzystując pętlę `for`,
- b) wykorzystując pętlę `while`,
- c) wykorzystując pętlę `do...while`.

Ćwiczenie 10.6

Napisz program wyświetlający w porządku malejącym liczby od 100 do 1 podzielne przez trzy, ale niepodzielne przez 2. Zadanie wykonaj za pomocą trzech rodzajów pętli (tak jak w ćwiczeniu 10.5).

Lekcja 11. Instrukcje `break` i `continue`

W lekcji 10. omówiono cztery rodzaje pętli, czyli konstrukcji programistycznych pozwalających na łatwe wykonywanie powtarzających się czynności. Były to pętle `for`, `while`, `do...while` i `foreach`. W lekcji bieżącej zostaną przedstawione dwie dodatkowe, współpracujące z pętlami instrukcje: `break` i `continue`. Pierwsza powoduje przerwanie wykonywania pętli i opuszczenie jej bloku, natomiast druga — przerwanie bieżącej iteracji i przejście do kolejnej. Sprawdzimy zatem, jak je stosować w przypadku prostych pętli pojedynczych, a także pętli dwu- lub wielokrotnie zagnieżdżonych. Omówiony będzie również temat etykiet, które dodatkowo zmieniają zachowanie `break` i `continue`, a tym samym umożliwiają tworzenie bardziej zaawansowanych konstrukcji pętli.

Instrukcja `break`

Instrukcję `break` przedstawiono już w lekcji 9., przy omawianiu instrukcji `switch`. Znaczenie `break` w języku programowania jest zgodne z nazwą, w języku angielskim `break` znaczy „przerywać”. Dokładnie tak zachowywała się ta konstrukcja w przypadku instrukcji `switch`, tak też zachowuje się w przypadku pętli — po prostu przerwa ich wykonanie. Dzięki temu możemy np. zmodyfikować pętlę `for` tak, aby wyrażenie warunkowe znalazło się wewnątrz niej. Kod realizujący takie zadanie jest przedstawiony na listingu 2.35.

Listing 2.35. Użycie `break` wewnątrz pętli `for`

```
using System;  
  
public class Program  
{  
    public static void Main()  
}
```

```
{  
    for(int i = 0; ; i++)  
    {  
        Console.WriteLine("[i = {0}] Pętle w C#", i);  
        if(i == 9)  
        {  
            break;  
        }  
    }  
}
```

Ponownie szczególną uwagę należy zwrócić na wyrażenia znajdujące się w nawiasie okrągłym pętli. Mimo iż usuneliśmy wyrażenie warunkowe, znajdujący się po nim średnik musi pozostać na swoim miejscu; inaczej nie uda nam się komplikacja programu.

Sama pętla działa w taki sposób, że w każdym przebiegu wykonywana jest instrukcja warunkowa `if`, sprawdzająca, czy zmienna `i` osiągnęła wartość 9, czyli badająca warunek `i == 9`. Jeśli warunek ten będzie prawdziwy, będzie to oznaczało, że na ekranie zostało wyświetlonych 10 napisów, zostanie więc wykonana instrukcja `break`, która przerwie działanie pętli.

Instrukcja `break` pozwala również na pozbycie się wszystkich wyrażeń z nawiasu okrągłego pętli! Jak to zrobić? Otóż wystarczy wyrażenie początkowe przenieść przed pętlę, a wyrażenia modyfikujące i warunkowe do jej wnętrza. Sposób ten jest przedstawiony na listingu 2.36. Jest to raczej ciekawostka, pokazująca jak elastyczny jest język C# (a także inne języki oparte na podobnej składni), choć obecnie tego typu konstrukcje coraz częściej spotykane są też w praktyce. Warto bowiem zauważyć, że:

```
for(; ;)  
{  
    //instrukcje  
}
```

oznacza po prostu nieskończoną pętlę `for` (taka pętla nie kończy się samodzielnie — w jej wnętrzu musi wystąpić instrukcja przekazująca sterowanie poza pętlę, np. instrukcja `break`) i jest odpowiednikiem nieskończonej pętli `while` w postaci:

```
while(true)  
{  
    //instrukcje  
}
```

Listing 2.36. Usumięcie wyrażeń sterujących z pętli `for`

```
using System;  
  
public class Program  
{  
    public static void Main()  
    {  
        int i = 0;  
        for(; ;)
```

```
{  
    Console.WriteLine("[i = {0}] Pętle w C# ", i);  
    if(i++ >= 9)  
    {  
        break;  
    }  
}  
}  
}
```

W przedstawionym kodzie na wstępie zmiennej *i* przypisywana jest wartość początkowa 0, a następnie rozpoczyna się pętla *for*. W pętli najpierw wyświetlany jest napis zawierający stan zmiennej *i*, a później sprawdzany jest warunek *i++ >= 9*. Oznacza to, że najpierw bada się, czy *i* jest większe od 9 lub równe 9 (*i >= 9*), a następnie *i* jest zwiększane o 1 (*i++*). Jeżeli warunek jest prawdziwy (*i* osiągnęło wartość 9), wykonywana jest instrukcja *break* przerywająca pętlę. W przeciwnym razie (*i* mniejsze od 9) pętla jest kontynuowana.

Należy pamiętać, że instrukcja *break* przerywa działanie pętli, w której się znajduje. Jeśli zatem mamy zagnieżdżone pętle *for*, a instrukcja *break* występuje w pętli wewnętrznej, zostanie przerwana jedynie pętla wewnętrzna. Pętla zewnętrzna nadal będzie działać. Spójrzmy na kod znajdujący się na listingu 2.37. To właśnie dwie zagnieżdżone pętle *for*.

Listing 2.37. Zagnieżdżone pętle *for*

```
using System;  
  
public class Program  
{  
    public static void Main()  
    {  
        for(int i = 0; i < 3; i++)  
        {  
            for(int j = 0; j < 3; j++)  
            {  
                Console.WriteLine("{0} {1} ", i, j);  
            }  
        }  
    }  
}
```

Wynikiem działania takiego programu będzie ciąg liczb widoczny na rysunku 2.22. Pierwszy pionowy ciąg liczb określa stan zmiennej *i*, natomiast drugi — stan zmiennej *j*. Ta konstrukcja działa w ten sposób, że w każdym przebiegu pętli zewnętrznej (w której zmienną iteracyjną jest *i*) są wykonywane trzy przebiegi pętli wewnętrznej (w której zmienną iteracyjną jest *j*). Stąd też biorą się ciągi liczb pojawiające się na ekranie.

Rysunek 2.22.

*Wynik działania
dwóch zagnieżdżonych
pętli typu for*

```
C:\>program.exe
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
C:\>
```

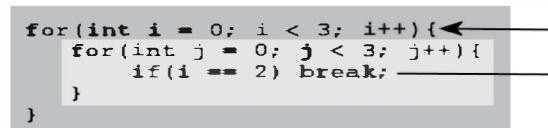
Jeśli teraz w pętli wewnętrznej umieścimy instrukcję warunkową `if(i == 2) break;`, tak aby cała konstrukcja wyglądała następująco:

```
for(int i = 0; i < 3; i++)
{
    for(int j = 0; j < 3; j++)
    {
        if(i == 2) break;
        Console.WriteLine("{0} {1} ", i, j);
    }
}
```

zgodnie z tym, co zostało napisane wyżej, za każdym razem, kiedy `i` osiągnie wartość 2, przerywana będzie pętla wewnętrzna, a sterowanie będzie przekazywane do pętli zewnętrznej. Zobrazowano to na rysunku 2.23. Tym samym po uruchomieniu programu znikną ciągi liczb wyświetlane, kiedy `i` było równe 2 (rysunek 2.24). Instrukcja `break` powoduje bowiem przejście do kolejnej iteracji zewnętrznej pętli.

Rysunek 2.23.

*Instrukcja break
powoduje
przerwanie
wykonywania
pętli wewnętrznej*

**Rysunek 2.24.**

*Zastosowanie
instrukcji
warunkowej
w połączeniu
z break*

```
C:\>program.exe
0 0
0 1
0 2
1 0
1 1
1 2
C:\>
```

Zastosowanie instrukcji `break` nie ogranicza się oczywiście jedynie do pętli typu `for`. Może być ona również stosowana w połączeniu z pozostałymi rodzajami pętli, czyli `while`, `do...while` i `foreach`. Jej działanie w każdym z tych przypadków będzie takie samo jak w przypadku pętli `for`.

Instrukcja continue

O ile instrukcja `break` powodowała przerwanie wykonywania pętli oraz jej opuszczenie, o tyle instrukcja `continue` powoduje przejście do kolejnej iteracji. Jeśli zatem wewnątrz pętli znajdzie się instrukcja `continue`, bieżąca iteracja (przebieg) zostanie przerwana oraz rozpoczęte się kolejna (chyba że bieżąca iteracja była ostatnią). Zobaczmy jednak, jak to działa, na konkretnym przykładzie. Na listingu 2.38 jest widoczna pętla `for`, która wyświetla liczby całkowite z zakresu 1 – 20 podzielne przez 2 (por. Ćwiczenia do samodzielnego wykonania z lekcji 10.).

Listing 2.38. Użycie instrukcji `continue`

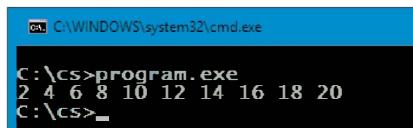
```
using System;

public class Program
{
    public static void Main()
    {
        for(int i = 1; i <= 20; i++)
        {
            if(i % 2 != 0) continue;
            Console.WriteLine("{0} ", i);
        }
    }
}
```

Wynik działania tego programu jest widoczny na rysunku 2.25. Sama pętla jest skonstruowana w dobrze nam już znany sposób. W środku znajduje się instrukcja warunkowa sprawdzająca warunek `i % 2 != 0`, czyli badająca, czy reszta z dzielenia `i` przez 2 jest różna od 0. Jeśli warunek ten jest prawdziwy (reszta jest różna od 0), oznacza to, że wartość zawarta w `i` nie jest podzielna przez 2, wykonywana jest zatem instrukcja `continue`. Jak już wiemy, powoduje ona rozpoczęcie kolejnej iteracji pętli, czyli zwiększenie wartości zmiennej `i` o 1 i przejście na początek pętli (do pierwszej instrukcji). Tym samym jeśli wartość `i` jest niepodzielna przez 2, nie zostanie wykonana znajdująca się za warunkiem instrukcja `Console.WriteLine("{0} ", i);`, więc dana wartość nie pojawi się na ekranie, a to właśnie było naszym celem.

Rysunek 2.25.

Wynik działania programu
wyświetlającego liczby
z zakresu 1 – 20 podzielne
przez 2



Rzecz jasna, zadanie to można wykonać bez użycia instrukcji `continue` (sekcja „Ćwiczenia do samodzielnego wykonania”), ale bardzo dobrze ilustruje ono istotę jej działania. Schematyczne działanie `continue` dla pętli `for` przedstawiono na rysunku 2.26.

Rysunek 2.26.

Działanie instrukcji
continue
w przypadku
pętli for

```
for (wyrażenie1; wyrażenie2; wyrażenie3) { ←  
    instrukcja1;  
    instrukcja2;  
    ...  
    instrukcjaN;  
    continue; ←  
    instrukcjaM;  
}
```

Instrukcja `continue` w przypadku pętli zagnieźdzonych działa w sposób znany z opisu instrukcji `break`, to znaczy jej działanie dotyczy tylko pętli, w której się znajduje. Jeśli zatem znajduje się w pętli wewnętrznej, powoduje przejście do kolejnej iteracji pętli wewnętrznej, a jeśli znajduje się w pętli zewnętrznej — do kolejnej iteracji pętli zewnętrznej. Zobrazowano to schematycznie na rysunku 2.27. Instrukcja `continue`, podobnie jak `break`, może być również stosowana w przypadku pozostałych typów pętli.

Rysunek 2.27.

Sposób działania
instrukcji `continue`
w przypadku
zagnieźdzonych
pętli for

```
for (wyrażenie1; wyrażenie2; wyrażenie3) { ←  
    instrukcja1;  
    for (wyrażenie1; wyrażenie2; wyrażenie3) { ←  
        instrukcja1; ←  
        ...  
        instrukcjaN;  
        continue; ←  
        instrukcjaM;  
    }  
    ...  
    instrukcjaN;  
    continue; ←  
    instrukcjaM;  
}
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 11.1

Napisz program, który wyświetli na ekranie nieparzyste liczby z zakresu 1 – 20. Wykorzystaj pętlę `for` i instrukcję `continue`.

Ćwiczenie 11.2

Napisz program, który wyświetli na ekranie nieparzyste liczby z zakresu 1 – 20. Wykorzystaj pętlę `while` i instrukcję `continue`.

Ćwiczenie 11.3

Napisz program, który wyświetli na ekranie liczby z zakresu 100 – 1 (w porządku malejącym) niepodzielne przez 3 i 4. Wykonaj dwa warianty ćwiczenia: z użyciem instrukcji `continue` i bez jej użycia.

Ćwiczenie 11.4

Napisz program, który wyświetli na ekranie liczby z zakresu –100 – 100 podzielne przez 4, ale niepodzielne przez 8 i przez 10. Wykorzystaj instrukcję `continue`.

Ćwiczenie 11.5

Zmodyfikuj program znajdujący się na listingu 2.38 tak, aby wynik jego działania pozostał bez zmian, ale nie było potrzeby użycia instrukcji `continue`.

Ćwiczenie 11.6

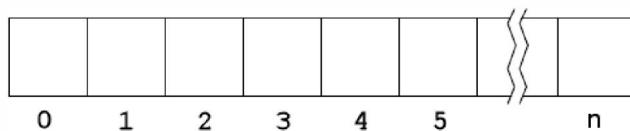
Napisz program wyświetlający 20 wierszy z liczbami. W pierwszym mają się znajdować liczby od 20 do 1, w drugim — od 19 do 1, w trzecim — od 18 do 1 itd. aż do ostatniego wiersza w pojedynczą liczbą 20. Wykonaj dwa warianty ćwiczenia: a) z użyciem pętli `for`, b) z użyciem pętli `while`.

Tablice

Tablica to stosunkowo prosta struktura danych, pozwalająca na przechowanie uporządkowanego zbioru elementów danego typu. Obrazowo przedstawiono to na rysunku 2.28. Jak widać, struktura ta składa się z ponumerowanych kolejno komórek (numeracja zaczyna się od 0). Każda taka komórka może przechowywać pewną porcję danych. Jego rodzaju będą to dane, określa typ tablicy. Jeśli zatem zadeklarujemy tablicę typu całkowitoliczbowego (np. `int`), będzie ona mogła zawierać liczby całkowite. Jeżeli będzie to natomiast typ znakowy (`char`), poszczególne komórki będą mogły zawierać różne znaki.

Rysunek 2.28.

Schematyczna struktura tablicy



Lekcja 12. Podstawowe operacje na tablicach

Tablice to struktury danych występujące w większości popularnych języków programowania. Nie mogło ich zatem zabraknąć również w C#. W tej lekcji zostaną przedstawione podstawowe typy tablic jednowymiarowych, będzie wyjaśnione, jak należy je deklarować, oraz będą zaprezentowane sposoby ich wykorzystywania. Zostanie omówiona również bardzo ważna dla tej struktury właściwość `Length`. Ponadto zwrócimy uwagę na sposób numerowania komórek każdej tablicy, który zawsze zaczyna się od 0, jak pokazano na rysunku 2.28.

Tworzenie tablic

Tablice w C# są obiektami (więcej o obiektach w rozdziale 3.). Aby móc skorzystać z tablicy, musimy najpierw zadeklarować zmienną tablicową, a następnie utworzyć samą tablicę (obiekt tablicy). Schematycznie sama deklaracja wygląda następująco:

```
typ_tablicy[] nazwa_tablicy;
```

Jest ona zatem bardzo podobna do deklaracji zwykłej zmiennej typu prostego (takiego jak `int`, `char`, `short` itp.), wyróżnikiem są natomiast znaki nawiasu kwadratowego. Taka deklaracja to jednak nie wszystko; powstała dopiero zmienna o nazwie `nazwa_tablicy`, dzięki której będziemy mogli odwoływać się do tablicy, ale samej tablicy jeszcze wcale nie ma! Musimy ją dopiero utworzyć, korzystając z operatora `new` w postaci:

```
new typ_tablicy[liczba_elementów];
```

Możemy jednocześnie zadeklarować i utworzyć tablicę, korzystając z konstrukcji:

```
typ_tablicy[] nazwa_tablicy = new typ_tablicy[liczba_elementów];
```

bądź też rozbić te czynności na dwie instrukcje. Schemat postępowania wygląda wtedy następująco:

```
typ_tablicy[] nazwa_tablicy;  
/* tutaj mogą znaleźć się inne instrukcje */  
nazwa_tablicy = new typ_tablicy[liczba_elementów];
```

Jak widać, pomiędzy deklaracją a utworzeniem tablicy można umieścić również inne instrukcje. Najczęściej wykorzystuje się jednak sposób pierwszy, to znaczy jednoczesną deklarację zmiennej tablicowej i samo utworzenie tablicy.

Zobaczmy zatem, jak to wygląda w praktyce. Zadeklarujemy tablicę liczb całkowitych (typu `int`) o nazwie `tab` i wielkości jednego elementu. Elementowi temu przypiszemy dowolną wartość, a następnie wyświetlimy ją na ekranie. Kod realizujący to zadanie jest widoczny na listingu 2.39.

Listing 2.39. Utworzenie tablicy w C#

```
using System;

public class Program
{
    public static void Main()
    {
        int[] tab = new int[1];
        tab[0] = 10;
        Console.WriteLine("Pierwszy element tablicy ma wartość: " + tab[0]);
    }
}
```

W pierwszym kroku zadeklarowaliśmy zmienną tablicową `tab` i przypisaliśmy jej nowo utworzoną tablicę typu `int` o rozmiarze 1 (`int[] tab = new int[1]`). Oznacza to, że tablica ta ma tylko jedną komórkę i może przechowywać naraz tylko jedną liczbę całkowitą. W kroku drugim jedynemu elementowi tej tablicy przypisaliśmy wartość 10. Zwrócić uwagę na sposób odwołania się do tego elementu: `tab[0] = 10`. Ponieważ w C# (podobnie jak w C, C++ czy Java) elementy tablic są numerowane od 0 (rysunek 2.28), pierwszy z nich ma indeks 0! To bardzo ważne: pierwszy element to indeks 0, drugi to indeks 1, trzeci to indeks 2 itd. Jeśli zatem chcemy odwołać się do pierwszego elementu tablicy `tab`, piszemy `tab[0]` (indeks żądanego elementu umieszczamy w nawiasie kwadratowym za nazwą tablicy). W kroku trzecim po prostu wyświetlimy zawartość wskazanego elementu na ekranie przy użyciu znanej nam już dobrze instrukcji `Console.WriteLine`.

Sprawdźmy teraz, co się stanie, jeśli się pomylimy i spróbujemy się odwołać do nieistniejącego elementu tablicy — na przykład zapomniemy, że tablice są indeksowane od 0, zadeklarujemy tablicę 10-elementową i spróbujemy odwołać się do elementu o indeksie 10 (element o takim indeksie oczywiście nie istnieje, ostatni element ma indeks 9). Taki scenariusz realizujemy za pomocą kodu przedstawionego na listingu 2.40.

Listing 2.40. Odwołanie do nieistniejącego elementu tablicy

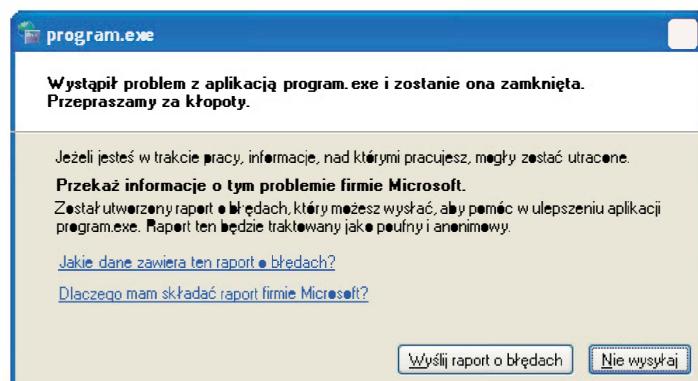
```
using System;

public class Program
{
    public static void Main()
    {
        int[] tab = new int[10];
        tab[10] = 1;
        Console.WriteLine("Element o indeksie 10 ma wartość: " + tab[10]);
    }
}
```

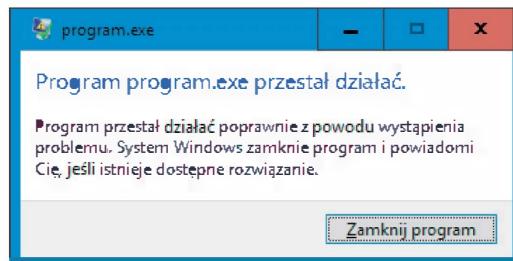
Program taki da się bez problemu skompilować, jednak próba jego uruchomienia spowoduje pojawienie się na ekranie okna z informacją o błędzie. Może ono mieć różną postać w zależności od tego, w jakiej wersji systemu została uruchomiona aplikacja. Przykładowo dla Windows XP będzie to okno przedstawione na rysunku 2.29, a w przypadku Windows 10 — okno z rysunku 2.30.

Rysunek 2.29.

Próba odwołania się do nieistniejącego elementu tablicy powoduje błąd aplikacji (Windows XP)

**Rysunek 2.30.**

Informacja o błędzie w systemie Windows 10



Jednocześnie na konsoli (w Windows XP dopiero po zamknięciu okna dialogowego) zobaczymy komunikat podający konkretne informacje o typie błędu oraz miejscu programu, w którym on wystąpił (rysunek 2.31). W przypadku uruchomienia programu z wykorzystaniem platformy Mono pojawi się natomiast jedynie informacja widoczna na rysunku 2.32.

Rysunek 2.31.

Systemowa informacja o błędzie

```
C:\>program.exe
Wystąpił nieobsłużony: System.IndexOutOfRangeException: Indeks wykracza
    poza granice tablicy.
        w Program.Main()
C:\>
```

Rysunek 2.32.

Informacja o błędzie na platformie Mono

```
Open Mono Command Prompt
C:\>mono program.exe
Unhandled Exception:
System.IndexOutOfRangeException: Array index is out of range.
    at Program.Main () [0x00000] in <filename unknown>:0
[ERROR] FATAL UNHANDLED EXCEPTION: System.IndexOutOfRangeException: Ar
ray index is out of range.
    at Program.Main () [0x00000] in <filename unknown>:0
C:\>
```

Wygląda to dosyć groźnie, jednak tak naprawdę nie stało się nic strasznego. Wykonanie programu, rzecz jasna, zostało przerwane, jednak najważniejsze jest to, że próba nieprawidłowego odwołania do tablicy została wykryta przez środowisko uruchomieniowe i samo odwołanie, które mogłoby naruszyć stabilność systemu, nie na-

stąpiło. Zamiast tego został wygenerowany tak zwany wyjątek (ang. *exception*, wyjątkami zajmiemy się w rozdziale 4.) o nazwie `IndexOutOfRangeException` (indeks poza zakresem) i program zakończył działanie. To bardzo ważna cecha nowoczesnych języków programowania.

Inicjalizacja tablic

Ważną sprawą jest inicjalizacja tablicy, czyli przypisanie jej komórkom wartości początkowych. W przypadku niewielkich tablic takiego przypisania można dokonać, ujmując żądane wartości w nawiasie klamrowym. Nie trzeba wtedy, choć można, korzystać z operatora `new`. System utworzy tablicę za nas i zapisze w jej kolejnych komórkach podane przez nas wartości. Schematycznie deklaracja taka wygląda następująco:

```
typ_tablicy[] nazwa_tablicy = {wartość1, wartość2, ..., wartośćN}
```

lub:

```
typ_tablicy[] nazwa_tablicy = new typ_tablicy[liczba_elementów]{wartość1,  
    ↴wartość2, ..., wartośćN}
```

Jeśli na przykład chcemy zadeklarować 6-elementową tablicę liczb całkowitych typu `int` i przypisać jej kolejnym komórkom wartości od 1 do 6, powinniśmy zastosować konstrukcję:

```
int[] tablica = {1, 2, 3, 4, 5, 6};
```

lub:

```
int[] tablica = new int[6] {1, 2, 3, 4, 5, 6};
```

O tym, że tego typu konstrukcja jest prawidłowa, przekonamy się, uruchamiając kod widoczny na listingu 2.41, gdzie taka tablica została zadeklarowana. Do wyświetlenia jej zawartości na ekranie zostały natomiast wykorzystane pętla typu `for` (lekcja 10.) i instrukcja `Console.WriteLine`. Wynik działania tego programu jest widoczny na rysunku 2.33.

Listing 2.41. Inicjalizacja tablicy

```
using System;

public class Program
{
    public static void Main()
    {
        int[] tablica = {1, 2, 3, 4, 5, 6};
        //lub
        //int[] tablica = new int[6]{1, 2, 3, 4, 5, 6};
        for(int i = 0; i < 6; i++)
        {
            Console.WriteLine("tab[{0}] = {1}", i, tablica[i]);
        }
    }
}
```

Rysunek 2.33.

Wynik działania pętli for użytej do wyświetlenia zawartości tablicy

```
C:\>program.exe
tab[0] = 1
tab[1] = 2
tab[2] = 3
tab[3] = 4
tab[4] = 5
tab[5] = 6
C:\>
```

Właściwość Length

W przypadku gdy rozmiar tablicy jest większy niż kilka komórek, zamiast stosować przedstawione w poprzedniej sekcji przypisanie w nawiasie klamrowym, lepszym rozwiązaniem jest wykorzystanie do wypełnienia jej danymi zwyczajnej pętli. Jeśli zatem mamy np. 20-elementową tablicę liczb typu `int` i chcemy zapisać w każdej z jej komórek liczbę 10, najlepiej wykorzystać w tym celu następującą konstrukcję:

```
for(int i = 0; i < 20, i++)
{
    tab[i] = 10;
}
```

Gdy piszemy w ten sposób, łatwo jednak o pomyłkę. Może się np. zdarzyć, że zmienimy rozmiar tablicy, a zapomnimy zmodyfikować pętlę. Nierazko spotkamy się też z sytuacją, kiedy rozmiar nie jest z góry znany i zostaje ustalony dopiero w trakcie działania programu. Na szczęście ten problem został rozwiązany w bardzo prosty sposób. Dzięki temu, że tablice w C# są obiektami, każda z nich ma przechowującą jej rozmiar właściwość `Length`, która może być tylko odczytywana. Jeśli zatem zastosujemy konstrukcję w postaci:

`nazwa_tablicy.Length`

otrzymamy rozmiar dowolnej tablicy. Należy oczywiście pamiętać o numerowaniu poszczególnych komórek tablicy od 0. Jeśli zatem chcemy odwołać się do ostatniego elementu tablicy, należy odwołać się do indeksu o wartości `Length - 1`. W praktyce więc program wypełniający prostą tablicę liczb typu `int` kolejnymi wartościami całkowitymi oraz wyświetlający jej zawartość na ekranie będzie wyglądał jak na listingu 2.42.

Listing 2.42. Użycie pętli for do wypełniania tablicy danymi

```
using System;

public class Program
{
    public static void Main()
    {
        int[] tablica = new int[10];
        for(int i = 0; i < tablica.Length; i++)
        {
            tablica[i] = i;
        }
        for(int i = 0; i < tablica.Length; i++)
```

```
{  
    Console.WriteLine("tablica[{0}] = {1}", i, tablica[i]);  
}  
}
```

Została tu utworzona tablica 10-elementowa typu int. W pierwszej pętli for wypełniamy ją danymi w taki sposób, że w komórkach wskazywanych przez zmienną iteracyjną i zapisywana jest wartość tej zmiennej. A więc w komórce o indeksie 0 będzie umieszczona wartość 0, w komórce o indeksie 1 — wartość 1 itd. Pętla działa, dopóki i jest mniejsze od wartości wskazywanej przez tablica.Length. Dzięki temu zostaną wypełnione danymi wszystkie komórki.

Zadaniem drugiej pętli jest odczytanie wszystkich danych z tablicy i wyświetlenie ich na ekranie. Do wyświetlania używana jest typowa instrukcja Console.WriteLine, natomiast konstrukcja pętli jest taka sama jak w pierwszym przypadku. Zmienna i przyjmuje wartości od 0 do tablica.Length – 1.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 12.1

Napisz program, w którym zostanie utworzona 10-elementowa tablica liczb typu int. Za pomocą pętli for zapisz w kolejnych komórkach liczby od 101 do 110. Zawartość tablicy wyświetl na ekranie.

Ćwiczenie 12.2

Napisz program, w którym zostanie utworzona 10-elementowa tablica liczb typu int. Użyj pętli for do wypełnienia jej danymi w taki sposób, aby w kolejnych komórkach znalazły się liczby od 10 do 100 (czyli 10, 20, 30 itd.). Zawartość tablicy wyświetli na ekranie.

Ćwiczenie 12.3

Napisz program, w którym zostanie utworzona 20-elementowa tablica typu bool. Komórkom o indeksie parzystym przypisz wartość true, a o indeksie nieparzystym — false. Zawartość tablicy wyświetli na ekranie.

Ćwiczenie 12.4

Napisz program, w którym zostanie utworzona 100-elementowa tablica liczb typu int. Komórkom o indeksach 0, 10, 20, ..., 90 przypisz wartość 0, komórkom 1, 11, 21, ..., 91 wartość 1, komórkom 2, 12, 22, ..., 92 wartość 2 itd.

Ćwiczenie 12.5

Utwórz 26-elementową tablicę typu char. Zapisz w kolejnych komórkach małe litery alfabetu od a do z.

Ćwiczenie 12.6

Zmodyfikuj program z listingu 2.42 w taki sposób, aby do wypełnienia tablicy danymi została wykorzystana pętla typu while.

Lekcja 13. Tablice wielowymiarowe

Tablice jednowymiarowe przedstawione w lekcji 12. to nie jedyny rodzaj tablic, jakie można tworzyć w C#. Istnieją bowiem również tablice wielowymiarowe, które zostaną omówione właśnie w tej lekcji. Pokazane zostanie, w jaki sposób je deklarować i tworzyć oraz jak odwoływać się do poszczególnych komórek. Opisane będą przy tym zarówno tablice o regularnym, jak i nieregularnym układzie komórek. Okaże się też, jak ważna w przypadku tego rodzaju struktur jest znana nam już właściwość Length.

Tablice dwuwymiarowe

W lekcji 12. omówiono podstawowe tablice jednowymiarowe. Jednowymiarowe to znaczy takie, które są wektorami elementów, czyli strukturami, schematycznie zaprezentowanymi na rysunku 2.28. Tablice nie muszą być jednak jednowymiarowe, wymiarów może być więcej, np. dwa. Wtedy struktura taka może wyglądać tak jak na rysunku 2.34. Widać wyraźnie, że do wyznaczenia konkretnej komórki trzeba podać tym razem dwie liczby określające rząd oraz kolumnę.

Rysunek 2.34.

Struktura tablicy dwuwymiarowej

	0	1	2	3	4
0					
1					

Tablicę taką trzeba zadeklarować oraz utworzyć. Odbywa się to w sposób podobny jak w przypadku tablic jednowymiarowych. Jeśli schematyczna deklaracja tablicy jednowymiarowej wyglądała tak:

```
typ[] nazwa_tablicy;
```

to w przypadku tablicy dwuwymiarowej będzie ona następująca:

```
typ[ , ] nazwa_tablicy;
```

Jak widać, dodajemy po prostu przecinek.

Kiedy mamy już zmienną tablicową, możemy utworzyć i jednocześnie zainicjować samą tablicę. Tu również konstrukcja jest analogiczna do tablic jednowymiarowych:

```
typ_tablicy[ , ] nazwa_tablicy = {{wartość1, wartość2, ... , wartośćn}, {wartość1,
→wartość2, ... , wartośćN}}
```

Zapis ten można rozbić na kilka linii w celu zwiększenia jego czytelności, np.:

```
typ_tablicy[ , ] nazwa_tablicy =
{
    {wartość1, wartość2, ... , wartośćn},
    {wartość1, wartość2, ... , wartośćN}
}
```

Jeśli zatem chcemy utworzyć tablicę o strukturze z rysunku 2.34, wypełnić ją kolejnymi liczbami całkowitymi (zaczynając od 0) i wyświetlić jej zawartość na ekranie, możemy zastosować program widoczny na listingu 2.43. Tablica ta będzie wtedy miała postać widoczną na rysunku 2.35.

Listing 2.43. Utworzenie, inicjalizacja i wyświetlenie zawartości tablicy dwuwymiarowej

```
using System;

public class Program
{
    public static void Main()
    {
        int[,] tab =
        {
            {0, 1, 2, 3, 4},
            {5, 6, 7, 8, 9}
        };
        for(int i = 0; i < 2; i++)
        {
            for(int j = 0; j < 5; j++)
            {
                Console.WriteLine("tab[{0},{1}] = {2}", i, j, tab[i, j]);
            }
        }
    }
}
```

Rysunek 2.35.

Tablica
dwuwymiarowa po
wypełnieniu danymi

	0	1	2	3	4
0	0	1	2	3	4
1	5	6	7	8	9

Do wyświetlenia zawartości tablicy na ekranie zostały użyte dwie pętle typu `for`. Pętla zewnętrzna ze zmienną iteracyjną `i` przebiega kolejne wiersze tablicy, a wewnętrzna, ze zmienną iteracyjną `j`, kolejne komórki w danym wierszu. Aby odczytać zawartość konkretnej komórki, stosujemy odwołanie:

```
tab[i, j]
```

Wynik działania programu z listingu 2.43 jest widoczny na rysunku 2.36.

Rysunek 2.36.

Wynik działania programu wypełniającego tablicę dwuwymiarową

```
C:\cs>program.exe
tab[0,0] = 0
tab[0,1] = 1
tab[0,2] = 2
tab[0,3] = 3
tab[0,4] = 4
tab[1,0] = 5
tab[1,1] = 6
tab[1,2] = 7
tab[1,3] = 8
tab[1,4] = 9
C:\cs>-
```

Drugi sposób utworzenia tablicy dwuwymiarowej to wykorzystanie omówionego już wcześniej operatora `new`. Tym razem będzie on miał następującą postać:

```
new typ_tablicy[liczba_wierszy, liczba_kolumn];
```

Można jednocześnie zadeklarować i utworzyć tablicę, korzystając z konstrukcji:

```
typ_tablicy[ , ] nazwa_tablicy = new typ_tablicy[liczba_wierszy, liczba_kolumn];
```

lub też rozbić te czynności na dwie instrukcje. Schemat postępowania wygląda wtedy następująco:

```
typ_tablicy[ , ] nazwa_tablicy;
/* tutaj mogą się znaleźć inne instrukcje */
nazwa_tablicy = new typ_tablicy[liczba_wierszy, liczba_kolumn];
```

Jak widać, oba sposoby są analogiczne do przypadku tablic jednowymiarowych.

Jeśli zatem ma powstać tablica liczb typu `int`, o dwóch wierszach i pięciu komórkach w każdym z nich (czyli dokładnie taka jak w poprzednich przykładach), należy zastosować instrukcję:

```
int[ , ] tab = new int[2, 5];
```

Tablicę taką można wypełnić danymi przy użyciu zagnieżdżonych pętli `for`, tak jak jest to zaprezentowane na listingu 2.44.

Listing 2.44. Utworzenie tablicy dwuwymiarowej z użyciem operatora `new`

```
using System;

public class Program
{
    public static void Main()
```

```
{  
    int[ , ] tab = new int[2, 5];  
    int licznik = 0;  
  
    for(int i = 0; i < 2; i++)  
    {  
        for(int j = 0; j < 5; j++)  
        {  
            tab[i, j] = licznik++;  
        }  
    }  
  
    for(int i = 0; i < 2; i++)  
    {  
        for(int j = 0; j < 5; j++)  
        {  
            Console.WriteLine("tab[{0},{1}] = {2}", i, j, tab[i, j]);  
        }  
    }  
}
```

Za wypełnienie tablicy danymi odpowiadają dwie pierwsze zagnieżdżone pętle `for`. Kolejnym komórkom jest przypisywany stan zmiennej `licznik`. Zmienna ta ma wartość początkową równą 0 i w każdej iteracji jest zwiększana o 1. Stosujemy w tym celu operator `++` (lekция 7.). Po wypełnieniu danymi zawartość tablicy jest wyświetla na ekranie za pomocą kolejnych dwóch zagnieżdżonych pętli `for`, dokładnie tak jak w przypadku programu z listingu 2.43.

Tablice tablic

Oprócz już przedstawionego istnieje jeszcze jeden sposób tworzenia tablic wielowymiarowych. Zauważmy bowiem, że np. dwuwymiarową tablicę liczb typu `int` moglibyśmy potraktować jako tablicę tablic liczb typu `int`. Jaki zatem typ powinna przyjąć taka struktura? Przeanalizujmy jeszcze raz deklaracje. Otóż jeśli zapis:

```
int[]
```

oznacza tablicę liczb typu `int`, to w takim razie:

```
int[] []
```

będzie oznaczał właśnie tablicę tablic typu `int`, innymi słowy: tablicę składającą się z innych tablic typu `int`. Zatem pełna deklaracja:

```
int[] [] tablica = new int[n] []
```

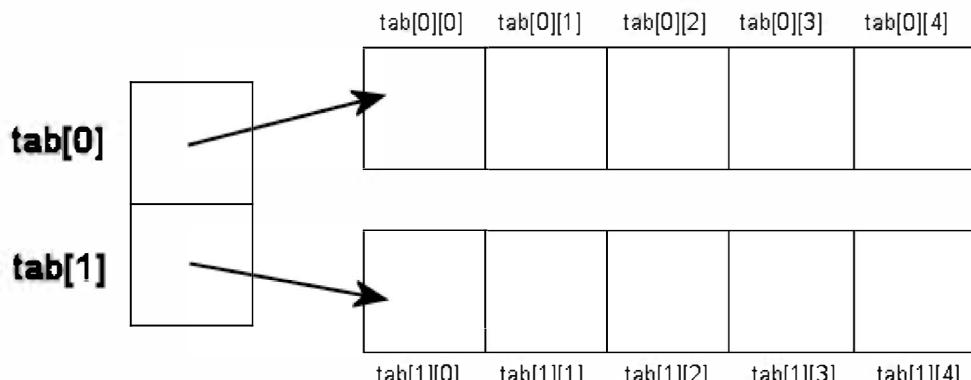
oznacza zadeklarowanie i utworzenie n -elementowej tablicy, której elementami będą mogły być inne tablice liczb typu `int`. Te tablice trzeba utworzyć dodatkowo, korzystając z operatora `new`. Można to zrobić oddzielnymi instrukcjami:

```
tablica[0] = new int[m];  
tablica[1] = new int[m];
```

itd. lub też używając pętli:

```
for(int i = 0; i < n; i++)
{
    tab[i] = new int[m];
}
```

W ten sposób powstanie n -elementowa tablica zawierająca m -elementowe tablice liczb typu int. Przykładową tablicę tego typu (gdzie $n = 2$, a $m = 5$) zilustrowano na rysunku 2.37, a sposób jej zadeklarowania, utworzenia i wypełnienia danymi został przedstawiony na listingu 2.45.



Rysunek 2.37. Budowa dwuwymiarowej tablicy o postaci int[2][5]

Listing 2.45. Utworzenie tablicy składającej się z innych tablic

```
using System;

public class Program
{
    public static void Main()
    {
        int[] [] tab = new int[2][];

        for(int i = 0; i < 2; i++)
        {
            tab[i] = new int[5];

            int licznik = 0;

            for(int j = 0; j < 5; j++)
            {
                tab[i][j] = licznik++;
            }
        }

        for(int i = 0; i < 2; i++)
        {
            for(int j = 0; j < 5; j++)
            {
                tab[i][j] = licznik++;
            }
        }
    }
}
```

```
{  
    Console.WriteLine("tab[{0}][{1}] = {2}", i, j, tab[i][j]);  
}  
}  
}  
}
```

Najpierw została tu zadeklarowana i utworzona tablica główna o odpowiednim typie (`int[][] tab = new int[2][];`). Następnie w pętli `for` poszczególnym jej komórkom zostały przypisane utworzone za pomocą operatora `new` nowe tablice liczb typu `int` (`tab[i] = new int[5];`). Zamiast pętli można by też użyć dwóch instrukcji:

```
tab[0] = new int[5];  
tab[1] = new int[5];
```

Skutek byłby taki sam. Tak więc po wykonaniu wszystkich opisanych instrukcji powstała tablica tablic o strukturze takiej jak na rysunku 2.37. Musi ona zostać wypełniona danymi. W tym celu zostały zastosowane dwie zagnieżdżone pętle `for`. W kolejnych komórkach zapisywany jest stan zmiennej `licznik`, która po każdym przypisaniu jest zwiększana o 1. Należy zwrócić uwagę na sposób odwoływanego się do poszczególnych komórek, nie może on być bowiem taki jak w przykładach z poprzedniej części lekcji. Tym razem mamy do czynienia z tablicą tablic, a zatem odwołanie składa się tak naprawdę z dwóch kroków. Najpierw wybieramy tablicę liczb typu `int` spośród tych zapisanych w komórkach tablicy `tab` (`tab[i]`), a następnie konkretną komórkę tej tablicy (`tab[i][j]`). Formalnie instrukcję przypisania można by więc zapisać też jako:

```
(tab[i])[j] = licznik++;
```

Na końcu programu znajdują się dwie kolejne zagnieżdżone pętle `for`, które zajmują się wyświetleniem zawartości wszystkich komórek. Efekt działania będzie taki sam jak w przypadku poprzednich przykładów (rysunek 2.36).

Tablice dwuwymiarowe i właściwość Length

W przykładzie z listingu 2.45 zarówno do wypełniania danymi, jak i wyświetlania zawartości komórek tablicy dwuwymiarowej zbudowanej jako tablica tablic były używane zagnieżdżone pętle `for`. Założeniem było jednak, że rozmiary są znane, a więc warunki zakończenia tych pętli miały postać typu `i < 2, j < 5`. Co jednak zrobić w sytuacji, kiedy rozmiar tablicy nie będzie z góry znany (np. zostanie wyliczony w trakcie działania programu) lub też chcielibyśmy stworzyć bardziej uniwersalny kod, pasujący do tablic o dowolnych rozmiarach? Intuicja podpowiada skorzystanie z właściwości `Length`, i w istocie to najlepszy sposób. Co ona jednak oznacza w przypadku tablicy dwuwymiarowej? Skoro, jak już wiemy, deklaracja w postaci:

```
int[][] tablica = new int[n] []
```

oznacza zadeklarowanie i utworzenie n -elementowej tablicy, to jej właściwość `Length` będzie wskazywała właśnie liczbę jej elementów, czyli liczbę tablic jednowymiarowych. Każda z tablic składowych oczywiście też będzie miała swoją właściwość `Length`. Tak więc po wykonaniu instrukcji:

```
int[][] tab = new int[2][];
tab[0] = new int[5];
tab[1] = new int[5];
```

będą zachodziły następujące zależności:

- ◆ `tab.Length = 2;`
- ◆ `tab[0].Length = 5;`
- ◆ `tab[1].Length = 5.`

Te wiadomości wystarczają do wprowadzenia takich modyfikacji kodu z [listingu 2.45](#), aby występujące w nim pętle `for` były uniwersalne i obsługiwały tablice dwuwymiarowe niezależnie od ich rozmiarów. Odpowiedni kod został zaprezentowany na [listingu 2.46](#).

Listing 2.46. Obsługa tablic dwuwymiarowych z użyciem właściwości `Length`

```
using System;

public class Program
{
    public static void Main()
    {
        int[][] tab = new int[2][];

        for(int i = 0; i < tab.Length; i++)
        {
            tab[i] = new int[5];
        }

        int licznik = 0;

        for(int i = 0; i < tab.Length; i++)
        {
            for(int j = 0; j < tab[i].Length; j++)
            {
                tab[i][j] = licznik++;
            }
        }

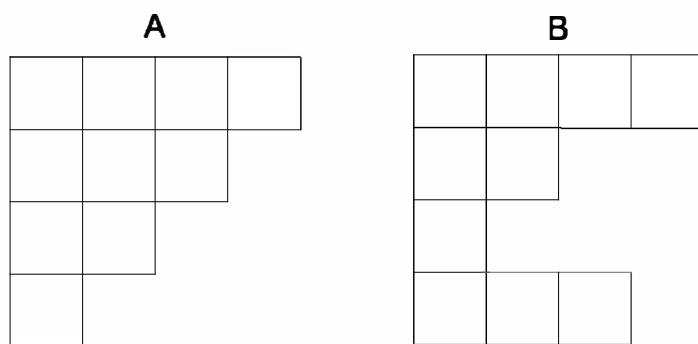
        for(int i = 0; i < tab.Length; i++)
        {
            for(int j = 0; j < tab[i].Length; j++)
            {
                Console.WriteLine("tab[{0}][{1}] = {2}", i, j, tab[i][j]);
            }
        }
    }
}
```

Tablice nieregularne

Tablice wielowymiarowe wcale nie muszą mieć regularnie prostokątnych kształtów, tak jak dotychczas prezentowane. Prostokątnych to znaczy takich, w których w każdym wierszu znajduje się taka sama liczba komórek (tak jak pokazano to na rysunkach 2.34 i 2.35). Nic nie stoi na przeszkodzie, aby utworzyć strukturę trójkątną (rysunek 2.38 A) lub też całkiem nieregularną (rysunek 2.38 B). Przy tworzeniu takich tablic czeka nas jednak więcej pracy niż w przypadku tablic regularnych, gdyż często każdy wiersz trzeba będzie tworzyć oddzielnie.

Rysunek 2.38.

Przykłady nieregularnych tablic dwuwymiarowych



Jak tworzyć tego typu struktury? Wiemy już, że tablice wielowymiarowe to tak naprawdę tablice tablic jednowymiarowych. To znaczy, że tablica dwuwymiarowa to tablica jednowymiarowa zawierająca szereg tablic jednowymiarowych, tablica trójwymiarowa to tablica jednowymiarowa zawierająca w sobie tablice dwuwymiarowe itd. Spróbujmy zatem stworzyć strukturę widoczną na rysunku 2.38 B. Zaczniemy od samej deklaracji — nie przysporzy nam ona z pewnością żadnego kłopotu, wykorzystaliśmy ją już kilkakrotnie:

```
int[][] tab
```

Ta deklaracja tworzy zmienną tablicową o nazwie `tab`, której można przypisywać tablice dwuwymiarowe przechowujące liczby typu `int`. Skoro struktura ma wyglądać jak na rysunku 2.38 B, trzeba teraz utworzyć 4-elementową tablicę, która będzie mogła przechowywać tablice jednowymiarowe liczb typu `int`; wykorzystać należy więc operator `new` w postaci:

```
new int[4][]
```

Zatem deklaracja i jednoczesna inicjalizacja zmiennej `tab` wyglądać będzie następująco:

```
int[][] tab = new int[4][];
```

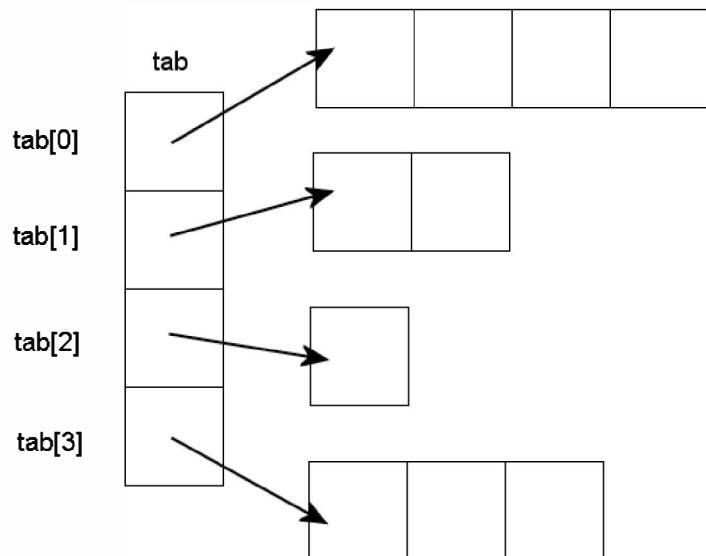
Teraz kolejnym elementom: `tab[0]`, `tab[1]`, `tab[2]` i `tab[3]` trzeba przypisać nowo utworzone tablice jednowymiarowe liczb typu `int`, tak aby w komórce `tab[0]` znalazła się tablica 4-elementowa, w `tab[1]` — 2-elementowa, w `tab[2]` — 1-elementowa, w `tab[3]` — 3-elementowa (rysunek 2.39). Trzeba zatem wykonać ciąg instrukcji:

```
tab[0] = new int[4];
tab[1] = new int[2];
tab[2] = new int[1];
tab[3] = new int[3];
```

To wszystko — cała tablica jest gotowa. Można ją już wypełnić danymi.

Rysunek 2.39.

*Struktura
nieregularnej tablicy
dwuwymiarowej*



Załóżmy, że chcemy, aby kolejne komórki zawierały liczby od 1 do 10, to znaczy w pierwszym wierszu tablicy znajdują się liczby 1, 2, 3, 4, w drugim — 5, 6, w trzecim — 7, a w czwartym — 8, 9, 10, tak jak zobrazowano to na rysunku 2.40. Do wypełnienia tablicy można użyć zagnieżdzonej pętli `for`, analogicznie do przypadku tablicy regularnej w przykładach z listingów 2.45 lub 2.46. Podobnie zagnieżdzonych pętli `for` użyjemy do wyświetlenia zawartości tablicy na ekranie. Pełny kod programu realizującego postawione zadania jest widoczny na listingu 2.47.

Rysunek 2.40.

*Tablica
nieregularna
wypełniona danymi*

1	2	3	4
5	6		
7			
8	9	10	

Listing 2.47. Tworzenie i wypełnianie danymi tablicy nieregularnej

```
using System;

public class Program
{
    public static void Main()
    {
        int[][] tab = new int[4][];
        tab[0] = new int[4];
        tab[1] = new int[2];
        tab[2] = new int[1];
        tab[3] = new int[3];

        int licznik = 1;
        for(int i = 0; i < tab.Length; i++)
        {
            for(int j = 0; j < tab[i].Length; j++)
            {
                tab[i][j] = licznik++;
            }
        }

        for(int i = 0; i < tab.Length; i++)
        {
            Console.Write("tab[{0}] = ", i);
            for(int j = 0; j < tab[i].Length; j++)
            {
                Console.Write("{0} ", tab[i][j]);
            }
            Console.WriteLine("");
        }
    }
}
```

W pierwszej części kodu tworzymy dwuwymiarową tablicę o strukturze takiej jak na rysunkach 2.38 B i 2.39 (wszystkie wykorzystane konstrukcje zostały wyjaśnione w poprzednich akapitach). Następnie wypełniamy otrzymaną tablicę danymi, tak aby uzyskać w poszczególnych komórkach wartości widoczne na rysunku 2.39. W tym celu używamy zagnieżdżonych pętli `for` i zmiennej `licznik`. Sposób ten był już wykorzystywany w programie z listingu 2.46. Pętla zewnętrzna, ze zmienną iteracyjną `i`, odpowiada za przebieg po kolejnych wierszach tablicy, a wewnętrzna, ze zmieniąną iteracyjną `j`, za przebieg po kolejnych komórkach w każdym wierszu.

Do wyświetlenia danych również zostały użyte dwie zagnieżdżone pętle `for`, odmienność jednak niż w przykładzie z listingu 2.46 dane dotyczące jednego wiersza tablicy są wyświetlane w jednej linii ekranu, dzięki czemu otrzymujemy obraz widoczny na rysunku 2.41. Osiągamy to dzięki wykorzystaniu instrukcji `Console.Write` zamiast dotychczasowego `Console.WriteLine`. W pętli zewnętrznej jest umieszczona instrukcja `Console.Write("tab[{0}] = ", i);`, wyświetlająca numer aktualnie przetwarzanego wiersza tablicy, natomiast w pętli wewnętrznej znajduje się instrukcja `Console.Write("{0} ", tab[i][j]);`, wyświetlająca zawartość komórek w danym wierszu.

Rysunek 2.41.
Wyświetlenie danych z tablicy nieregularnej

```
C:\>program.exe
tab[0] = 1 2 3 4
tab[1] = 5 6
tab[2] = 7
tab[3] = 8 9 10
C:\>
```

Spróbujmy teraz utworzyć tablicę, której struktura została przedstawiona na rysunku 2.38 A. Po wyjaśnieniach z ostatniego przykładu nikomu nie powinno przysporzyć to najmniejszych problemów. Wypełnimy ją danymi w sposób analogiczny do poprzedniego przypadku, czyli umieszczając w komórkach kolejne wartości od 1 do 10. Deklaracja i inicjalizacja wyglądać będzie następująco:

```
int[][] tab = new int[4][];
```

Kolejne wiersze tablicy utworzymy za pomocą serii instrukcji:

```
tab[0] = new int[4];
tab[1] = new int[3];
tab[2] = new int[2];
tab[3] = new int[1];
```

Wypełnienie takiej tablicy danymi (zgodnie z podanymi zasadami) oraz wyświetlenie tych danych na ekranie odbywać się będzie identycznie jak w przypadku kodu z listingu 2.47. Pełny program będzie zatem wyglądał tak, jak jest to przedstawione na listingu 2.48.

Listing 2.48. Tworzenie tablicy o trójkątnym kształcie

```
using System;

public class Program
{
    public static void Main()
    {
        int[][] tab = new int[4][];
        tab[0] = new int[4];
        tab[1] = new int[3];
        tab[2] = new int[2];
        tab[3] = new int[1];

        int licznik = 1;
        for(int i = 0; i < tab.Length; i++)
        {
            for(int j = 0; j < tab[i].Length; j++)
            {
                tab[i][j] = licznik++;
            }
        }

        for(int i = 0; i < tab.Length; i++)
        {
            Console.Write("tab[{0}] = ", i);
            for(int j = 0; j < tab[i].Length; j++)
            {
```

```
{  
    Console.Write("{0} ", tab[i][j]);  
}  
Console.WriteLine("");  
}  
}
```

Zauważmy jednak jedną rzecz. Tablica taka jak na rysunku 2.38 A, mimo że jest nazwana nieregularną, powinna raczej być nazwana nie prostokątną, gdyż w rzeczywistości jej trójkątny kształt można traktować jako regularny. Skoro tak, nie trzeba jej tworzyć ręcznie za pomocą serii (w prezentowanym wypadku czterech) instrukcji. Można również wykorzystać odpowiednio skonstruowaną pętlę typu `for`. Skoro każdy kolejny wiersz ma o jedną komórkę mniej, powinna to być pętla zliczająca od 4 do 1, czyli wyglądająca np. w następujący sposób:

```
for(int i = 0; i < 4; i++)  
{  
    tab[i] = new int[4 - i];  
}
```

Zmienna `i` zmienia się tu w zakresie 0 – 3, zatem `tab[i]` przyjmuje kolejne wartości `tab[0]`, `tab[1]`, `tab[2]`, `tab[3]`, natomiast wyrażenie `new int[4 - i]` wartości `new int[4]`, `new int[3]`, `new int[2]`, `new int[1]`. Tym samym otrzymamy dokładny odpowiednik czterech ręcznie napisanych instrukcji.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 13.1

Zmodyfikuj kod z listingu 2.47 tak, aby w kolejnych komórkach tablicy znalazły się liczby malejące od 10 do 1.

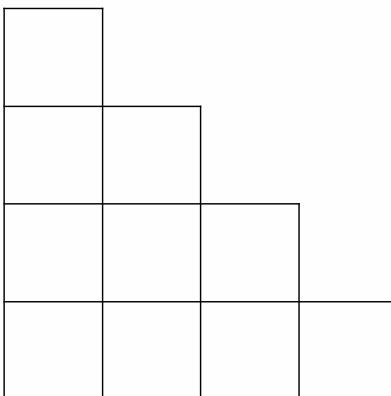
Ćwiczenie 13.2

Zmodyfikuj program z listingu 2.48 tak, aby do wypełnienia tablicy danymi były wykorzystywane pętle typu `while`.

Ćwiczenie 13.3

Utwórz tablicę liczb typu `int` zaprezentowaną na rysunku 2.42. Wypełnij kolejne komórki wartościami malejącymi od 10 do 1. Do utworzenia tablicy i wypełnienia jej danymi wykorzystaj pętlę typu `for`.

Rysunek 2.42.
*Odwrócona tablica
trójkątna do
ćwiczenia 13.3*



Ćwiczenie 13.4

Utwórz tablicę dwuwymiarową typu `bool` o rozmiarze 5×8 komórek. Wypełnij ją danymi w taki sposób, aby komórki o parzystych indeksach wiersza i kolumny zawierały wartość `true`, a pozostałe — wartość `false` (przyjmij, że 0 jest wartością parzystą). Zawartość tablicy wyświetl na ekranie w taki sposób, aby komórki o wartości `true` były reprezentowane przez wartość 1, a komórki o indeksie `false` — przez wartość 0.

Ćwiczenie 13.5.

Utwórz przykładową tablicę trójwymiarową i wypełnij ją przykładowymi danymi (będzie to struktura, którą można sobie wyobrazić jako prostopadłościian składający się z sześciianów; każdy sześciian będzie pojedynczą komórką). Zawartość tablicy wyświetl na ekranie.

Ćwiczenie 13.6.

Utwórz tablicę dwuwymiarową, w której liczba komórek w kolejnych rzędach będzie równa dziesięciu kolejnym wartościami ciągu Fibonacciego, poczynając od elementu o wartości 1 (1, 1, 2, 3, 5 itd.). Wartość każdej komórki powinna być jej numerem w danym wierszu w kolejności malejącej (czyli dla wiersza o długości pięciu komórek kolejne wartości to 5, 4, 3, 2, 1). Zawartość tablicy wyświetl na ekranie.

Rozdział 3.

Programowanie obiektowe

Każdy program w C# składa się z jednej lub wielu klas. W dotychczas prezentowanych przykładach była to tylko jednak klasa o nazwie `Program`. Przypomnijmy sobie naszą pierwszą aplikację, wyświetlającą na ekranie napis. Jej kod wyglądał następująco:

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Mój pierwszy program!");
    }
}
```

Założyliśmy wtedy, że szkielet kolejnych programów, na których demonstrowano struktury języka programowania, ma właśnie tak wyglądać. Teraz nadszedł czas, aby dowiedzieć się, dlaczego właśnie tak. Wszystko zostanie wyjaśnione w niniejszym rozdziale.

Podstawy

Pierwsza część rozdziału 3. składa się z trzech lekcji, w których podjęto tematykę podstaw programowania obiektowego w C#. W lekcji 14. jest omawiana budowa klas oraz tworzenie obiektów. Zostały w niej przedstawione pola i metody, sposoby ich deklaracji oraz wywoływanie. Lekcja 15. jest poświęcona argumentom metod oraz technice przeciążania metod, została w niej również przybliżona wykorzystywana już wcześniej metoda `Main`. W ostatniej, 16. lekcji, zaprezentowano temat konstruktorów, czyli specjalnych metod wywoływanych podczas tworzenia obiektów.

Lekcja 14. Klasy i obiekty

Lekcja 14. rozpoczyna rozdział przedstawiający podstawy programowania obiektowego w C#. Najważniejsze pojęcia zostaną tu wyjaśnione na praktycznych przykładach. Zajmiemy się tworzeniem klas, ich strukturą i deklaracjami, przeanalizujemy związek między klasą i obiektem. Zostaną przedstawione składowe klasy, czyli pola i metody, będzie też wyjaśnione, czym są wartości domyślne pól. Opisane zostaną również relacje między zadeklarowaną na stosie zmiennej obiektową (inaczej referencyjną, odnośnikową) a utworzonym na stercie obiektem.

Podstawy obiektowości

Program w C# składa się z klas, które są z kolei opisami obiektów. To podstawowe pojęcia związane z programowaniem obiektowym. Osoby, które nie zetknęły się dotychczas z programowaniem obiektowym, mogą potraktować **obiekt** (ang. *object*) jako pewien typ programistyczny, który może przechowywać dane i wykonywać operacje, czyli różne zadania. **Klasa** (ang. *class*) to z kolei definicja, opis takiego obiektu.

Skoro klasa definiuje obiekt, jest zatem również jego typem. Czym jest **typ obiektu**? Przytoczymy jedną z definicji: „Typ jest przypisany zmiennej, wyrażeniu lub innemu bytowi programistycznemu (danej, obiektowi, funkcji, procedurze, operacji, metodzie, parametrowi, modułowi, wyjątkowi, zdarzeniu). Specyfikuje on rodzaj wartości, które może przybierać ten byt. (...) Jest to również ograniczenie kontekstu, w którym odwołanie do tego bytu może być użyte w programie”¹. Innymi słowy, typ obiektu określa po prostu, czym jest dany obiekt. Tak samo jak miało to miejsce w przypadku zmiennych typów prostych. Jeśli mieliśmy zmenną typu int, to mogła ona przechowywać wartości całkowite. Z obiektami jest podobnie. Zmenna obiektowa hipotetycznej klasy Punkt może przechowywać obiekty klasy (typu) Punkt². Klasa to zatem nic innego jak definicja nowego typu danych.

Co może być obiektem? Tak naprawdę — wszystko. W życiu codziennym mianem tym określić możemy stół, krzesło, komputer, dom, samochód, radio... Każdy z obiektów ma pewne cechy, właściwości, które go opisują: wielkość, kolor, powierzchnię, wysokość. Co więcej, każdy obiekt może składać się z innych obiektów (rysunek 3.1). Na przykład mieszkanie składa się z poszczególnych pomieszczeń, z których każde może być obiektem; w każdym pomieszczeniu mamy z kolei inne obiekty: sprzęt domowe, meble itd.

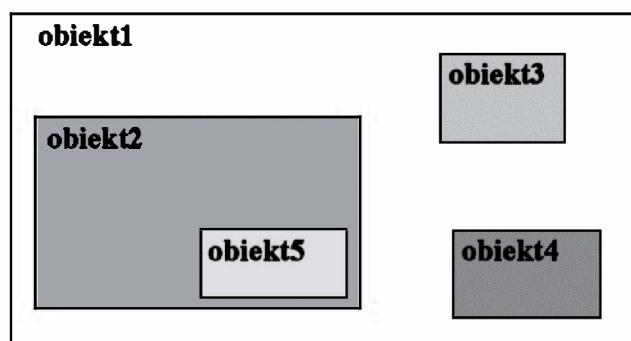
Obiekty oprócz tego, że mają właściwości, mogą wykonywać różne funkcje, zadania. Innymi słowy, każdy obiekt ma przypisany pewien zestaw poleceń, które potrafi wykonywać. Na przykład samochód „rozumie” polecenia „uruchom silnik”, „wyłącz silnik”, „skręć w prawo”, „przyspiesz” itp. Funkcje te składają się na pewien interfejs udostępniany nam przez tenże samochód. Dzięki interfejsowi możemy wpływać na zachowanie samochodu i wydawać mu polecenia.

¹ K. Subieta, *Wytwarzanie, integracja i testowanie systemów informatycznych*, PJWSTK, Warszawa 1997.

² W dalszej części książki zostanie pokazane, że takiej zmiennej można również przypisać obiekty klas potomnych lub nadzędnych w stosunku do klasy Punkt.

Rysunek 3.1.

Obiekt może zawierać inne obiekty



W programowaniu jest bardzo podobnie. Za pomocą klas staramy się opisać obiekty, ich właściwości, zbudować konstrukcje, interfejs, dzięki któremu będziemy mogli wydawać polecenia realizowane potem przez obiekty. Obiekt powstaje jednak dopiero w trakcie działania programu jako instancja (wystąpienie, egzemplarz) danej klasy. Obiektów danej klasy może być bardzo dużo. Jeśli na przykład klasą będzie *Samochód*, to instancją tej klasy będzie konkretny egzemplarz o danym numerze seryjnym.

Ponieważ dla osób nieobeznanych z programowaniem obiektowym może to wszystko brzmieć nieco zawile, od razu zobaczymy, jak to będzie wyglądało w praktyce.

Pierwsza klasa

Załóżmy, że pisany przez nas program wymaga przechowywania danych odnoszących się do punktów na płaszczyźnie, ekranie. Każdy taki punkt jest charakteryzowany przez dwie wartości: współrzędną x oraz współrzędną y. Utwórzmy więc klasę opisującą obiekty tego typu. Schematyczny szkielet klasy wygląda następująco:

```
class nazwa_klasy
{
    //treść klasy
}
```

W treści klasy definiujemy pola i metody. Pola służą do przechowywania danych, metody do wykonywania różnych operacji. W przypadku klasy, która ma przechowywać dane dotyczące współrzędnych x i y, wystarczą dwa pola typu int (przy założeniu, że wystarczające będzie przechowywanie wyłącznie współrzędnych całkowitych). Pozostaje jeszcze wybór nazwy dla takiej klasy. Występują tu takie same ograniczenia jak w przypadku nazewnictwa zmiennych (por. lekcja 5.), czyli nazwa klasy może składać się jedynie z liter (zarówno małych, jak i dużych), cyfr oraz znaku podkreślenia, ale nie może zaczynać się od cyfry. Można stosować znaki polskie znaki (choć wielu programistów używa wyłącznie alfabetu łacińskiego, nawet jeśli nazwy pochodzą z języka polskiego). Przyjęte jest również, że w nazwach nie używa się znaku podkreślenia.

Naszą klasę nazwiemy zatem, jakżeby inaczej, *Punkt* i będzie ona miała postać widoczną na listingu 3.1. Kod ten zapiszemy w pliku o nazwie *Punkt.cs*.

Listing 3.1. Klasa przechowująca współrzędne punktów

```
class Punkt
{
    int x;
    int y;
}
```

Ta klasa zawiera dwa pola o nazwach `x` i `y`, które opisują współrzędne położenia punktu. Pola definiujemy w taki sam sposób jak zmienne.

Kiedy mamy zdefiniowaną klasę `Punkt`, możemy zadeklarować zmienną typu `Punkt`. Robimy to podobnie jak wtedy, gdy deklarowaliśmy zmienne typów prostych (np. `short`, `int`, `char`), czyli pisząc:

`typ_zmiennej nazwa_zmiennej;`

Ponieważ typem zmiennej jest nazwa klasy (klasa to definicja typu danych), to jeśli nazwa zmiennej ma być `przykładowyPunkt`, deklaracja przyjmie postać:

`Punkt przykładowyPunkt;`

W ten sposób powstała zmienna odnośnikowa (referencyjna, obiektowa), która domyślnie jest pusta, tzn. nie zawiera żadnych danych. Dokładniej rzecz ujmując, po deklaracji zmienna taka zawiera wartość specjalną `null`, która określa, że nie ma ona odniesienia do żadnego obiektu. Musimy więc sami utworzyć obiekt klasy `Punkt` i przypisać go tej zmiennej³. Obiekty tworzy się za pomocą operatora `new` w postaci:

`new nazwa_klasy();`

zatem cała konstrukcja schematycznie wyglądać będzie następująco:

`nazwa_klasy nazwa_zmiennej = new nazwa_klasy();`

a w przypadku naszej klasy `Punkt`:

`Punkt przykładowyPunkt = new Punkt();`

Oczywiście, podobnie jak w przypadku zmiennych typów prostych (por. lekcja 5.), również i tutaj można oddzielić deklarację zmiennej od jej inicjalizacji, zatem również poprawna jest konstrukcja w postaci:

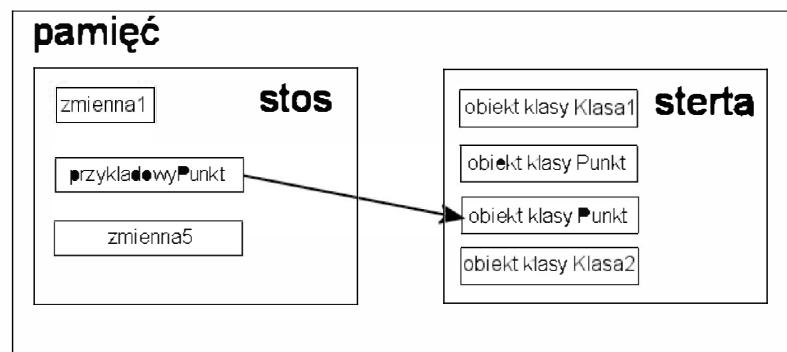
`Punkt przykładowyPunkt;
przykładowyPunkt = new Punkt();`

Koniecznie trzeba sobie dobrze uzmysłowić, że po wykonaniu tych instrukcji w pamięci powstają dwie różne struktury. Pierwszą z nich jest powstała na tak zwanym *stosie* (ang. *stack*) zmienna referencyjna `przykładowyPunkt`, drugą jest powstały na tak zwanej *stercie* (ang. *heap*) obiekt klasy (typu) `Punkt`. Zmienna `przykładowyPunkt` zawiera odniesienie do przypisanego jej obiektu klasy `Punkt` i tylko poprzez nią możemy się do tego obiektu odwoływać. Schematycznie zobrazowano to na rysunku 3.2.

³ Osoby programujące w C++ powinny zwrócić na to uwagę, gdyż w tym języku już sama deklaracja zmiennej typu klasowego powoduje wywołanie domyślnego konstruktora i utworzenie obiektu.

Rysunek 3.2.

Zależność
między zmienną
odnośnikową
a wskazywanym
przez nią obiektem



Jeśli chcemy odwołać się do danego pola klasy, korzystamy z operatora . (kropka), czyli używamy konstrukcji:

nazwa_zmiennej_obejektowej.nazwa_pola_obejktu

Przykładowo przypisanie wartości 100 polu x obiektu klasy Punkt reprezentowanego przez zmienną *przykładowyPunkt* będzie wyglądało następująco:

przykładowyPunkt.x = 100;

Jak użyć klasy?

Spróbujmy teraz przekonać się, że obiekt klasy Punkt faktycznie jest w stanie przechowywać dane. Jak wiadomo z poprzednich rozdziałów, aby program mógł zostać uruchomiony, musi zawierać metodę Main (więcej o metodach już w kolejnym podpunkcie, a o metodzie Main w jednej z kolejnych lekcji). Dopiszmy więc do klasy Punkt taką metodę, która utworzy obiekt, przypisze jego polom pewne wartości oraz wyświetli je na ekranie. Kod programu realizującego takie zadanie jest widoczny na listingu 3.2.

Listing 3.2. Użycie klasy Punkt

```
using System;

class Punkt
{
    int x;
    int y;

    public static void Main()
    {
        Punkt punkt1 = new Punkt();
        punkt1.x = 100;
        punkt1.y = 200;
        Console.WriteLine("punkt.x = " + punkt1.x);
        Console.WriteLine("punkt.y = " + punkt1.y);
    }
}
```

Struktura klasy Punkt jest taka sama jak w przypadku listingu 3.1, z tą różnicą, że do jej treści została dodana metoda Main. W tej metodzie deklarujemy zmienną klasy Punkt o nazwie punkt1 i przypisujemy jej nowo utworzony obiekt tej klasy. Dokonujemy zatem jednoczesnej deklaracji i inicjalizacji. Od tej chwili zmienna punkt1 wskazuje na obiekt klasy Punkt, możemy się zatem posługiwać nią tak, jakbyśmy posługiwali się samym obiektem. Pisząc zatem punkt1.x = 100, przypisujemy wartość 100 polu x, a pisząc punkt.y = 200, przypisujemy wartość 200 polu y. W ostatnich dwóch liniach korzystamy z instrukcji Console.WriteLine, aby wyświetlić wartość obu pól na ekranie. Efekt jest widoczny na rysunku 3.3.

Rysunek 3.3.

Wynik działania
klasy Punkt
z listingu 3.2

```
C:\>csc Punkt.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

C:\>Punkt.exe
punkt.x = 100
punkt.y = 200

C:\>_
```

Metody klas

Klasy oprócz pól przechowujących dane zawierają także metody, które wykonują zapisane przez programistę operacje. Definiujemy je w ciele (czyli wewnętrz) klasy pomiędzy znakami nawiasu klamrowego. Każda metoda może przyjmować argumenty oraz zwracać wynik. Schematyczna deklaracja metody wygląda następująco:

```
typ_wyniku nazwa_metody(argumenty_metody)
{
    instrukcje metody
}
```

Po umieszczeniu w ciele klasy deklaracja taka będzie natomiast wyglądała tak:

```
class nazwa_klasy
{
    typ_wyniku nazwa_metody(argumenty_metody)
    {
        instrukcje metody
    }
}
```

Jeśli metoda nie zwraca żadnego wyniku, jako typ wyniku należy zastosować słowo void; jeśli natomiast nie przyjmuje żadnych parametrów, pomiędzy znakami nawiasu okrągłego nie należy nic wpisywać. Aby zobaczyć, jak to wygląda w praktyce, do klasy Punkt dodamy prostą metodę, której zadaniem będzie wyświetlenie wartości współrzędnych x i y na ekranie. Nadamy jej nazwę WyswietlWspolrzedne, zatem jej wygląd będzie następujący:

```
void WyswietlWspolrzedne()
{
    Console.WriteLine("współrzędna x = " + x);
    Console.WriteLine("współrzędna y = " + y);
}
```

Słowo `void` oznacza, że metoda nie zwraca żadnego wyniku, a brak argumentów pomiędzy znakami nawiasu okrągłego wskazuje, że metoda ta żadnych argumentów nie przyjmuje. We wnętrzu metody znajdują się dwie dobrze nam znane instrukcje, które wyświetlały na ekranie współrzędne punktu. Po umieszczeniu powyższego kodu w klasie `Punkt` przyjmie ona postać widoczną na listingu 3.3.

Listing 3.3. Dodanie metody do klasy `Punkt`

```
using System;

class Punkt
{
    int x;
    int y;

    void WyswietlWspolrzedne()
    {
        Console.WriteLine("współrzędna x = " + x);
        Console.WriteLine("współrzędna y = " + y);
    }
}
```

Po utworzeniu obiektu danej klasy możemy **wywołać** (uruchomić) metodę w taki sam sposób, w jaki odwołujemy się do pól klasy, tzn. korzystając z operatora `.` (kropka). Jeśli zatem przykładowa zmieniona `punk1` zawiera referencję do obiektu klasy `Punkt`, prawidłowym wywołaniem metody `WyswietlWspolrzedne` będzie:

```
punk1.WyswietlWspolrzedne();
```

Ogólnie wywołanie metody wygląda następująco:

```
nazwa_zmiennej.nazwa_metody(argumenty_metody);
```

Oczywiście, jeśli dana metoda nie ma argumentów, po prostu je pomijamy. Przy czym termin *wywołanie* oznacza po prostu wykonanie kodu (instrukcji) zawartego w metodzie.

Użyjmy zatem metody `Main` do przetestowania nowej konstrukcji. W tym celu zmodyfikujemy program z listingu 3.2 tak, aby wykorzystywał metodę `WyswietlWspolrzedne`. Odpowiedni kod jest zaprezentowany na listingu 3.4. Wynik jego działania jest łatwy do przewidzenia (rysunek 3.4).

Listing 3.4. Wywołanie metody `WyswietlWspolrzedne`

```
using System;

class Punkt
{
    int x;
    int y;

    void WyswietlWspolrzedne()
    {
        Console.WriteLine("współrzędna x = " + x);
        Console.WriteLine("współrzędna y = " + y);
    }
}
```

```

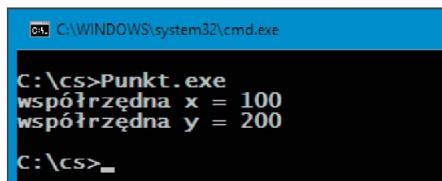
    }

    public static void Main()
    {
        Punkt punkt1 = new Punkt();
        punkt1.x = 100;
        punkt1.y = 200;
        punkt1.WyswietlWspolrzedne();
    }
}

```

Rysunek 3.4.

*Wynik działania metody
WyswietlWspolrzedne
z klasy Punkt*



Przedstawiony kod jest w istocie złożeniem przykładów z listingów 3.2 i 3.3. Klasa Punkt z listingu 3.3 została uzupełniona o nieco zmodyfikowany kod metody Main, poowany z listingu 3.2. W metodzie tej jest więc tworzony nowy obiekt typu Punkt i ustalane są wartości jego pól x i y. Do wyświetlenia wartości zapisanych w x i y jest natomiast używana metoda WyswietlWspolrzedne.

Zobaczmy teraz, w jaki sposób napisać metody, które będą mogły zwracać wyniki. Typ wyniku należy podać przed nazwą metody, zatem jeśli ma ona zwracać wartość typu int, deklaracja powinna wyglądać następująco:

```

int nazwa_metody()
{
    //instrukcje metody
}

```

Sam wynik zwracamy natomiast przez zastosowanie instrukcji return. Najlepiej zobaczyć to na praktycznym przykładzie. Do klasy Punkt dodamy zatem dwie metody — jedna będzie podawała wartość współrzędnej x, druga y. Nazwiemy je odpowiednio PobierzX i PobierzY. Wygląd metody PobierzX będzie następujący:

```

int PobierzX()
{
    return x;
}

```

Przed nazwą metody znajduje się określenie typu zwracanego przez nią wyniku — skoro jest to int, oznacza to, że metoda ta musi zwrócić jako wynik liczbę całkowitą z przedziału określonego przez typ int (tabela 2.1). Wynik jest zwracany dzięki instrukcji return. Zapis return x oznacza zwrócenie przez metodę wartości zapisanej w polu x.

Jak łatwo się domyślić, metoda PobierzY będzie wyglądała analogicznie, z tym że będzie w niej zwracana wartość zapisana w polu y. Pełny kod klasy Punkt po dodaniu tych dwóch metod będzie wyglądał tak, jak przedstawiono na listingu 3.5.

Listing 3.5. Metody zwracające wyniki

```
using System;

class Punkt
{
    int x;
    int y;

    int PobierzX()
    {
        return x;
    }

    int PobierzY()
    {
        return y;
    }

    void WyswietlWspolrzedne()
    {
        Console.WriteLine("współrzędna x = " + x);
        Console.WriteLine("współrzędna y = " + y);
    }
}
```

Jeśli teraz zechcemy przekonać się, jak działają nowe metody, możemy wyposażyć klasę Punkt w metodę Main testującą ich działanie. Mogłaby ona mieć postać widoczną na listingu 3.6.

Listing 3.6. Metoda Main testująca działanie klasy Punkt

```
public static void Main()
{
    Punkt punkt1 = new Punkt();
    punkt1.x = 100;
    punkt1.y = 200;
    int wspX = punkt1.PobierzX();
    int wspY = punkt1.PobierzY();
    Console.WriteLine("współrzędna x = " + wspX);
    Console.WriteLine("współrzędna y = " + wspY);
}
```

Początek kodu jest tu taki sam jak we wcześniej prezentowanych przykładach — powstaje obiekt typu Punkt i są w nim zapisywane przykładowe współrzędne. Następnie tworzone są dwie zmienne typu int: wspX i wspY. Pierwszej przypisywana jest wartość zwrócona przez metodę PobierzX, a drugiej — wartość zwrócona przez metodę PobierzY. Wartości zapisane w zmiennych są następnie wyświetlane w standardowy sposób na ekranie.

Warto tu zauważyc, że zmienne wspX i wspY pełnią funkcję pomocniczą — dzięki nim kod jest czytelniejszy. Nic jednak nie stoi na przeszkodzie, aby wartości zwrócone przez metody były używane bezpośrednio w instrukcjach Console.WriteLine⁴. Metoda

⁴ Po wyjaśnieniach przedstawionych w lekcji 3. można się domyślić, że to, co do tej pory było nazywane instrukcją WriteLine, jest w rzeczywistości wywołaniem metody o nazwie WriteLine.

Main mogłaby więc mieć również postać przedstawioną na listingu 3.7. Efekt działania byłby taki sam.

Listing 3.7. Alternatywna wersja metody Main

```
public static void Main()
{
    Punkt punkt1 = new Punkt();
    punkt1.x = 100;
    punkt1.y = 200;
    Console.WriteLine("współrzędna x = " + punkt1.PobierzX());
    Console.WriteLine("współrzędna y = " + punkt1.PobierzY());
}
```

Jednostki komplikacji, przestrzenie nazw i zestawy

Każdą klasę można zapisać w pliku o dowolnej nazwie. Często przyjmuje się jednak, że nazwa pliku powinna być zgodna z nazwą klasy. Jeśli zatem istnieje klasa Punkt, to jej kod powinien znaleźć się w pliku *Punkt.cs*. W jednym pliku może się też znaleźć kilka klas. Wówczas jednak zazwyczaj są to tylko jedna klasa główna oraz dodatkowe klasy pomocnicze. W przypadku prostych aplikacji tych zasad nie trzeba przestrzegać, ale w przypadku większych programów umieszczenie całej struktury kodu w jednym pliku spowodowałoby duże trudności w zarządzaniu nim. Pojedynczy plik można nazywać jednostką komplikacji lub modelem.

Wszystkie dotychczasowe przykłady składały się zawsze z jednej klasy zapisywanej w jednym pliku. Zobaczmy więc, jak mogą współpracować ze sobą dwie klasy. Na listingu 3.8 znajduje się nieco zmodyfikowana treść klasy Punkt z listingu 3.1. Przed składowymi zostały dodane słowa public, dzięki którym będzie istniała możliwość odwoływania się do nich z innych klas. Ta kwestia zostanie wyjaśniona dokładniej w jednej z kolejnych lekcji. Na listingu 3.9 jest natomiast widoczny kod klasy Program, która korzysta z klasy Punkt. Tak więc treść z listingu 3.8 zapiszemy w pliku o nazwie *Punkt.cs*, a kod z listingu 3.9 w pliku *Program.cs*.

Listing 3.8. Prosta klasa Punkt

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

Listing 3.9. Klasa Program korzystająca z obiektu klasy Punkt

```
using System;

public class Program
{
    public static void Main()
    {
        Punkt punkt1 = new Punkt();
```

```
punkt1.x = 100;
punkt1.y = 200;
Console.WriteLine("punkt1.x = " + punkt1.x);
Console.WriteLine("punkt1.y = " + punkt1.y);
}
}
```

W klasie `Program` znajduje się metoda `Main`, od której rozpoczyna się wykonywanie kodu aplikacji. W tej metodzie tworzony jest obiekt `punkt1` klasy `Punkt`, jego składowym przypisywane są wartości 100 i 200, a następnie są one wyświetlane na ekranie. Tego typu konstrukcje były wykorzystywane już kilkakrotnie we wcześniejszych przykładach.

Jak teraz przetworzyć oba kody na plik wykonywalny? Nie jest to skomplikowane, po prostu nazwy obu plików (`Program.cs` i `Punkt.cs`) należy zastosować jako argumenty wywołania kompilatora, czyli w wierszu poleceń wydać komendę:

```
csc Program.cs Punkt.cs
```

Trzeba też wiedzieć, że plik wykonywalny powstały po komplikacji nie zawiera tylko kodu wykonywalnego. W rzeczywistości kod wykonywany na platformie .NET składa się z tak zwanych zestawów (ang. *assembly*). Pojedynczy **zestaw** składa się z manifestu, metadanych oraz kodu języka pośredniego IL. **Manifest** to wszelkie informacje o zestawie, takie jak nazwy plików składowych, odwołania do innych zestawów, numer wersji itp. **Metadane** natomiast to opis danych i kodu języka pośredniego w danym zestawie, zawierający m.in. definicje zastosowanych typów danych.

Wszystko to może być umieszczone w jednym lub też kilku plikach (*exe*, *dll*). We wszystkich przykładach w tej książce będziemy mieli do czynienia tylko z zestawami jednoplikowymi i będą to pliki wykonywalne typu *exe*, generowane automatycznie przez kompilator, tak że nie będziemy musieli zagłębiać się w te kwestie. Nie można jednak pominąć zagadnienia przestrzeni nazw.

Przestrzeń nazw to ograniczenie widoczności danej nazwy, ograniczenie kontekstu, w którym jest ona rozpoznawana. Czemu to służy? Otóż pojedyncza aplikacja może się składać z bardzo dużej liczby klas, a jeszcze więcej klas znajduje się w bibliotekach udostępnianych przez .NET. Co więcej, nad jednym projektem zwykle pracują zespoły programistów. W takiej sytuacji nietrudno o pojawianie się konfliktów nazw, czyli powstawanie klas o takich samych nazwach. Tymczasem nazwa każdej klasy musi być unikatowa. Ten problem rozwiązują właśnie przestrzenie nazw. Jeśli bowiem klasa zostanie umieszczona w danej przestrzeni, to będzie widoczna tylko w niej. Będą więc mogły istnieć klasa o takiej samej nazwie, o ile tylko zostaną umieszczone w różnych przestrzeniach nazw. Taką przestrzeń definiuje się za pomocą słowa `namespace`, a jej składowe należy umieścić w występującym dalej nawiasie klamrowym. Schematycznie wygląda to tak:

```
namespace nazwa_przestrzeni
{
    elementy przestrzeni nazw
}
```

Przykładowo jeden programista może pracować nad biblioteką klas dotyczących grafiki trójwymiarowej, a drugi nad biblioteką klas wspomagających tworzenie grafiki na płaszczyźnie. Można zatem przygotować dwie osobne przestrzenie nazw, np. o nazwach *Grafika2D* i *Grafika3D*. W takiej sytuacji każdy programista będzie mógł utworzyć własną klasę o nazwie *Punkt* i obie te klasy będzie można jednocześnie wykorzystać w jednej aplikacji. Klasy te mogłyby mieć definicje takie jak na listingach 3.10 i 3.11.

Listing 3.10. Klasa *Punkt* w przestrzeni nazw *Grafika2D*

```
namespace Grafika2D
{
    class Punkt
    {
        public int x;
        public int y;
    }
}
```

Listing 3.11. Klasa *Punkt* w przestrzeni nazw *Grafika3D*

```
namespace Grafika3D
{
    class Punkt
    {
        public double x;
        public double y;
    }
}
```

Jak skorzystać z jednej z tych klas w jakimś programie? Istnieją dwie możliwości. Pierwsza z nich to podanie pełnej nazwy klasy wraz z nazwą przestrzeni nazw. Po między nazwą klasy a nazwą przestrzeni należy umieścić znak kropki. Na przykład odwołanie do klasy *Punkt* z przestrzeni *Grafika2D* miałoby postać:

Grafika2D.Punkt

Sposób ten został przedstawiony na listingu 3.12.

Listing 3.12. Użycie klasy *Punkt* przestrzeni nazw *Grafika2D*

```
using System;

public class Program
{
    public static void Main()
    {
        Grafika2D.Punkt punkt1 = new Grafika2D.Punkt();
        punkt1.x = 100;
        punkt1.y = 200;
        Console.WriteLine("punkt1.x = " + punkt1.x);
        Console.WriteLine("punkt1.y = " + punkt1.y);
    }
}
```

Drugi sposób to użycie dyrektywy `using` w postaci:

```
using nazwa_przestrzeni;
```

Należy ją umieścić na samym początku pliku. Nie jest ona niczym innym jak informacją dla kompilatora, że chcemy korzystać z klas zdefiniowanych w przestrzeni o nazwie `nazwa_przestrzeni`. Liczba umieszczonych na początku pliku instrukcji `using` nie jest ograniczona. Jeśli chcemy skorzystać z kilku przestrzeni nazw, używamy kilku dyrektyw `using`. Jasne jest więc już, co oznacza fragment:

```
using System;
```

wykorzystywany w praktycznie wszystkich dotychczasowych przykładach. To deklaracja, że chcemy korzystać z przestrzeni nazw o nazwie `System`. Była ona niezbędna, gdyż w tej właśnie przestrzeni jest umieszczona klasa `Console` zawierająca metody `Write` i `WriteLine`. Łatwo się domyślić, że moglibyśmy pominąć dyrektywę `using System`, ale wtedy instrukcja wyświetlająca wiersz tekstu na ekranie musiałaby przyjmować postać:

```
System.Console.WriteLine("tekst");
```

Tak więc nasz pierwszy program z listingu 1.1 (w rozdziale 1.) równie dobrze mógłby mieć postać widoczną na listingu 3.13.

Listing 3.13. Pominiecie dyrektywy `using System`

```
public class Program
{
    public static void Main()
    {
        System.Console.WriteLine("Mój pierwszy program!");
    }
}
```

W C# od wersji 6.0 w dyrektywie `using` można także użyć **klasy statycznej**, tak aby nie było konieczne powtarzanie nazwy tej klasy przy korzystaniu z metod. Dodatkowo należy wtedy użyć słowa `static`. W związku z tym program z listingu 1.1 w C# 6.0 i wyższych mógłby również wyglądać tak jak na listingu 3.14. Po użyciu zapisu `using static Console` w treści metody `Main` zamiast pisać `Console.WriteLine` można użyć samego zapisu `WriteLine`. Sposób działania kodu się nie zmieni.

Listing 3.14. Klasa statyczna w dyrektywie `using`

```
using static System.Console;

public class Program
{
    public static void Main()
    {
        WriteLine("Mój pierwszy program!");
    }
}
```

Nie będzie zaskoczeniem, że gdybyśmy chcieli, aby w programie z listingu 3.12 nie trzeba było odwoływać się do przestrzeni nazw `Grafika2D` przy każdym wystąpieniu klasy `Punkt`, należałoby użyć instrukcji `using Grafika2D`, tak jak zostało to zaprezentowane na listingu 3.15.

Listing 3.15. Użycie instrukcji `using Grafika2D`

```
using System;
using Grafika2D;

public class Program
{
    public static void Main()
    {
        Punkt punkt1 = new Punkt();
        punkt1.x = 100;
        punkt1.y = 200;
        Console.WriteLine("punkt1.x = " + punkt1.x);
        Console.WriteLine("punkt1.y = " + punkt1.y);
    }
}
```

Pozostaje jeszcze kwestia jednoczesnego użycia klas `Punkt` z przestrzeni `Grafika2D` i `Grafika3D`. Można oczywiście użyć dwóch następujących po sobie instrukcji `using`:

```
using Grafika2D;
using Grafika3D;
```

W żaden sposób nie rozwiąże to jednak problemu. Jak bowiem kompilator (ale także i programista) miałby wtedy ustalić, o którą z klas chodzi, kiedy nazywają się one tak samo? Dlatego też w takim wypadku za każdym razem trzeba w odwołaniu podawać, o którą przestrzeń nazw chodzi. Jeśli więc chcemy zdefiniować dwa obiekty: `punkt1` klasy `Punkt` z przestrzeni `Grafika2D` i `punkt2` klasy `Punkt` z przestrzeni `Grafika3D`, należy użyć instrukcji:

```
Grafika2D.Punkt punkt1 = new Grafika2D.Punkt();
Grafika3D.Punkt punkt2 = new Grafika3D.Punkt();
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 14.1

Napisz przykładową klasę `LiczbaCalkowita`, która będzie przechowywała wartość całkowitą. Klasa ta powinna zawierać metodę `WyswietlLiczbe`, która będzie wyświetlała na ekranie przechowywaną wartość, oraz metodę `PobierzLiczbe` zwracającą przechowywaną wartość.

Ćwiczenie 14.2

Napisz kod przykładowej klasy `Prostokat` zawierającej cztery pola przechowujące współrzędne czterech rogów prostokąta.

Ćwiczenie 14.3

Do utworzonej w ćwiczeniu 14.2 klasy `Prostokat` dopisz metody zwracające współrzędne wszystkich czterech rogów oraz metodę wyświetlającą wartości współrzędnych.

Ćwiczenie 14.4

Do klas `LiczbaCalkowita` (ćwiczenie 4.1) i `Prostokat` (ćwiczenie 4.3) dopisz przykładową metodę `Main` testującą ich zachowanie.

Ćwiczenie 14.5

Napisz klasę `Protokat` przechowującą jedynie współrzędne lewego górnego i prawego dolnego rogu (wyystarczą one do jednoznacznego wyznaczenia prostokąta na płaszczyźnie). Dodaj metody podające współrzędne każdego rogu.

Ćwiczenie 14.6

Do klasy `Prostokat` z ćwiczeń 14.2 i 14.3 dopisz metodę sprawdzającą, czy wprowadzone współrzędne faktycznie definiują prostokąt (cztery punkty na płaszczyźnie dają dowolny czworokąt, który nie musi mieć kształtów prostokąta).

Lekcja 15. Argumenty i przeciążanie metod

O tym, że metody mogą mieć argumenty, wiadomo z lekcji 14. Czas dowiedzieć się, jak się nimi posługiwać. Właśnie temu tematowi została poświęcona cała lekcja 15. Będzie w niej wyjaśnione, w jaki sposób przekazuje się metodom argumenty typów prostych oraz referencyjnych, będzie też omówione przeciążanie metod, czyli technika umożliwiająca umieszczenie w jednej klasie kilku metod o tej samej nazwie. Nieco miejsca zostanie także poświęcone metodzie `Main`, od której zaczyna się wykonywanie aplikacji. Zobaczmy również, w jaki sposób przekazać aplikacji parametry z wiersza poleceń.

Argumenty metod

W lekcji 14. powiedziano, że każda metoda może mieć argumenty. **Argumenty metody** (stosowane jest także określenie **parametry**) to inaczej dane, które można jej przekazać. Metoda może mieć dowolną liczbę argumentów umieszczonych w nawiasie okrągłym za jej nazwą. Poszczególne argumenty oddzielamy od siebie znakiem przecinka. Schematycznie wygląda to następująco:

```
typ_wyniku nazwa_metody(typ_argumentu_1 nazwa_argumentu_1, typ_argumentu_2
nazwa_argumentu_2, ... , typ_argumentu_N nazwa_argumentu_N)
{
    /* treść metody */
}
```

Przykładowo w klasie `Punkt` przydałyby się metody umożliwiające ustawianie współrzędnych. Jest tu możliwych kilka wariantów — zacznijmy od najprostszych: napiszemy dwie metody, `UstawX` i `UstawY`. Pierwsza będzie odpowiedzialna za przypisanie przekazanej jej wartości polu `x`, a druga — polu `y`. Zgodnie z podanym powyżej schematem pierwsza z nich powinna wyglądać następująco:

```
void UstawX(int wspX)
{
    x = wspX;
}
```

natomiast druga:

```
void UstawY(int wspY)
{
    y = wspY;
}
```

Metody te nie zwracają żadnych wyników, co sygnalizuje słowo `void`, przyjmują natomiast jeden parametr typu `int`. W ciele każdej z metod następuje z kolei przypisanie wartości przekazanej w parametrze odpowiedniemu polu: `x` w przypadku metody `UstawX` oraz `y` w przypadku metody `UstawY`.

W podobny sposób można napisać metodę, która będzie jednocześnie ustawiała pola `x` i `y` klasy `Punkt`. Oczywiście będzie ona przyjmowała dwa argumenty, które w deklaracji należy oddzielić przecinkiem. Zatem cała konstrukcja będzie wyglądała następująco:

```
void UstawXY(int wspX, int wspY)
{
    x = wspX;
    y = wspY;
}
```

Metoda `UstawXY` nie zwraca żadnego wyniku, ale przyjmuje dwa argumenty: `wspX`, `wspY`, oba typu `int`. W ciele tej metody argument `wspX` (dokładniej — jego wartość) zostaje przypisany polu `x`, a `wspY` — polu `y`. Jeśli teraz dodamy do klasy `Punkt` wszystkie trzy powstałe wyżej metody, otrzymamy kod widoczny na listingu 3.16.

Listing 3.16. Metody ustawiające pola klasy `Punkt`

```
using System;

class Punkt
{
    int x;
    int y;

    int PobierzX()
    {
        return x;
```

```
        }
        int PobierzY()
        {
            return y;
        }
        void UstawX(int wspX)
        {
            x = wspX;
        }
        void UstawY(int wspY)
        {
            y = wspY;
        }
        void UstawXY(int wspX, int wspY)
        {
            x = wspX;
            y = wspY;
        }
        void WyswietlWspolrzedne()
        {
            Console.WriteLine("współrzędna x = " + x);
            Console.WriteLine("współrzędna y = " + y);
        }
    }
```

Warto teraz napisać dodatkową metodę `Main`, która przetestuje nowe metody klasy `Punkt`. Dzięki temu będziemy mogli sprawdzić, czy wszystkie trzy działają zgodnie z naszymi założeniami. Taka przykładowa metoda jest widoczna na listingu 3.17⁵.

Listing 3.17. Metoda Main testująca metody ustawiające współrzędne

```
public static void Main()
{
    Punkt pierwszyPunkt = new Punkt();
    Punkt drugiPunkt = new Punkt();

    pierwszyPunkt.UstawX(100);
    pierwszyPunkt.UstawY(100);

    Console.WriteLine("pierwszyPunkt:");
    pierwszyPunkt.WyswietlWspolrzedne();

    drugiPunkt.UstawXY(200, 200);

    Console.WriteLine("\ndrugiPunkt:");
    drugiPunkt.WyswietlWspolrzedne();
}
```

Na początku tworzymy dwa obiekty typu (klasy) `Punkt`, jeden z nich przypisujemy zmiennej o nazwie `pierwszyPunkt`, drugi zmiennej o nazwie `drugiPunkt`⁶. Następnie wy-

⁵ W listingach dostępnych na FTP znajduje się pełny kod klasy `Punkt`, zawierający widoczną metodę `Main`.

⁶ Jak wynika z wcześniej przedstawionych informacji, w rzeczywistości zmiennym zostały przypisane referencje (odniesienia) do utworzonych na stercie obiektów. Często spotka się jednak przedstawioną tu uproszczoną terminologię, w której referencję utożsamia się z obiektem.

korzystujemy metody `UstawX` i `UstawY` do przypisania polom obiektu `pierwszyPunkt` wartości 100. W kolejnym kroku za pomocą metody `WyswietlWspolrzedne` wyświetlamy te wartości na ekranie. Dalej wykorzystujemy metodę `UstawXY`, aby przypisać polom obiektu `drugiPunkt` wartości 200, oraz wyświetlamy je na ekranie, również za pomocą metody `WyswietlWspolrzedne`. Po skompilowaniu i uruchomieniu tego programu otrzymamy widok jak na rysunku 3.5. Do uzyskania dodatkowego odstępu między wyświetlonymi informacjami została użyta sekwencja specjalna `\n`.

Rysunek 3.5.
Efekt wykonania
programu
z listingu 3.17

```
C:\>Punkt.exe
pierwszyPunkt:
współrzędna x = 100
współrzędna y = 100

drugiPunkt:
współrzędna x = 200
współrzędna y = 200

C:\>
```

Obiekt jako argument

Argumentem przekazanym metodzie może być również obiekt (ściślej: referencja do obiektu; w książce będzie jednak stosowana również i taka uproszczona terminologia łatwiejsza do przyswojenia dla początkujących), nie musimy ograniczać się jedynie do typów prostych. Podobnie metoda może zwracać obiekt w wyniku swojego działania. W obu wymienionych sytuacjach postępowanie jest takie same jak w przypadku typów prostych. Przykładowo metoda `UstawXY` w klasie `Punkt` mógłaby przyjmować jako argument obiekt tej klasy, a nie dwie liczby typu `int`, tak jak zostało to zaprogramowane we wcześniejszych przykładach (listing 3.15). Metoda taka wyglądałaby następująco:

```
void UstawXY(Punkt punkt)
{
    x = punkt.x;
    y = punkt.y;
}
```

Argumentem jest w tej chwili obiekt `punkt` klasy `Punkt`. W ciele metody następuje skopiowanie wartości pól z obiektu przekazanego jako argument do obiektu bieżącego, czyli przypisanie polu `x` wartości zapisanej w `punkt.x`, a polu `y` wartości zapisanej w `punkt.y`.

Podobnie możemy umieścić w klasie `Punkt` metodę o nazwie `PobierzXY`, która zwróci w wyniku nowy obiekt klasy `Punkt` o współrzędnych takich, jakie zostały zapisane w polach obiektu bieżącego. Metoda taka będzie miała postać:

```
Punkt PobierzXY()
{
    Punkt punkt = new Punkt();
    punkt.x = x;
    punkt.y = y;
    return punkt;
}
```

Jak widać, nie przyjmuje ona żadnych argumentów, nie ma przecież takiej potrzeby; z deklaracji wynika jednak, że zwraca obiekt klasy Punkt. W ciele metody najpierw tworzymy nowy obiekt klasy Punkt, przypisując go zmiennej referencyjnej o nazwie punkt, a następnie przypisujemy jego polom wartości pól x i y z obiektu bieżącego. Ostatecznie za pomocą instrukcji return powodujemy, że obiekt punkt staje się wartością zwracaną przez metodę. Klasa Punkt po wprowadzeniu takich modyfikacji będzie miała postać widoczną na listingu 3.18.

Listing 3.18. Nowe metody klasy Punkt

```
using System;

class Punkt
{
    int x;
    int y;

    int PobierzX()
    {
        return x;
    }
    int PobierzY()
    {
        return y;
    }
    void UstawX(int wspX)
    {
        x = wspX;
    }
    void UstawY(int wspY)
    {
        y = wspY;
    }
    void UstawXY(Punkt punkt)
    {
        x = punkt.x;
        y = punkt.y;
    }
    Punkt PobierzXY()
    {
        Punkt punkt = new Punkt();
        punkt.x = x;
        punkt.y = y;
        return punkt;
    }
    void WyswietlWspolrzedne()
    {
        Console.WriteLine("współrzędna x = " + x);
        Console.WriteLine("współrzędna y = " + y);
    }
}
```

Aby lepiej urozmyslić sobie sposób działania wymienionych metod, napiszemy teraz kod metody `Main`, który będzie je wykorzystywał. Kod ten został zaprezentowany na listingu 3.19. Należy go dodać do klasy najnowszej wersji klasy Punkt z listingu 3.18.

Listing 3.19. Kod metody Main

```
public static void Main()
{
    Punkt pierwszyPunkt = new Punkt();
    Punkt drugiPunkt;

    pierwszyPunkt.UstawX(100);
    pierwszyPunkt.UstawY(100);

    Console.WriteLine("Obiekt pierwszyPunkt ma współrzędne:");
    pierwszyPunkt.WyswietlWspolrzedne();
    Console.WriteLine("");

    drugiPunkt = pierwszyPunkt.PobierzXY();

    Console.WriteLine("Obiekt drugiPunkt ma współrzędne:");
    drugiPunkt.WyswietlWspolrzedne();
    Console.WriteLine("");

    Punkt trzeciPunkt = new Punkt();
    trzeciPunkt.UstawXY(drugiPunkt);

    Console.WriteLine("Obiekt trzeciPunkt ma współrzędne:");
    trzeciPunkt.WyswietlWspolrzedne();
    Console.WriteLine("");
}
```

Na początku deklarujemy zmienne `pierwszyPunkt` oraz `drugiPunkt`. Zmiennej `pierwszyPunkt` przypisujemy nowo utworzony obiekt klasy `Punkt` (rysunek 3.7 A). Następnie wykorzystujemy znane nam dobrze metody `UstawX` i `UstawY` do przypisania polom `x` i `y` wartości 100 oraz wyświetlamy te dane na ekranie, korzystając z metody `wyswietlWspolrzedne`.

W kolejnym kroku zmiennej `drugiPunkt`, która jak pamiętamy, nie została wcześniej zainicjowana, przypisujemy obiekt zwrócony przez metodę `PobierzWspolrzedne` wywołaną na rzecz obiektu `pierwszyPunkt`. A zatem zapis:

```
drugiPunkt = pierwszyPunkt.PobierzWspolrzedne();
```

oznacza, że wywoływana jest metoda `PobierzWspolrzedne` obiektu `pierwszyPunkt`, a zwrócony przez nią wynik jest przypisywany zmiennej `drugiPunkt`. Jak wiemy, wynikiem działania tej metody będzie obiekt klasy `Punkt` będący kopią obiektu `pierwszyPunkt`, czyli zawierający w polach `x` i `y` takie same wartości, jakie są zapisane w polach obiektu `pierwszyPunkt`. To znaczy, że po wykonaniu tej instrukcji zmienna `drugiPunkt` zawiera referencję do obiektu, w którym pola `x` i `y` mają wartość 100 (rysunek 3.7 B). Obie wartości wyświetlamy na ekranie za pomocą instrukcji `WyswietlWspolrzedne`.

W trzeciej części programu tworzymy obiekt `trzeciPunkt` (`Punkt trzeciPunkt = new Punkt();`) i wywołujemy jego metodę `ustawXY`, aby wypełnić pola `x` i `y` danymi. Metoda ta jako argument przyjmuje obiekt klasy `Punkt`, w tym przypadku obiekt `drugiPunkt`. Zatem po wykonaniu instrukcji wartości pól `x` i `y` obiektu `trzeciPunkt` będą takie same jak pól `x` i `y` obiektu `drugiPunkt` (rysunek 3.7 C). Nic zatem dziwnego, że wynik

działania programu z listingu 3.19 jest taki jak zaprezentowany na rysunku 3.6. Z kolei na rysunku 3.7 przedstawione zostały schematyczne zależności pomiędzy zmien- nymi i obiektami występującymi w metodzie Main.

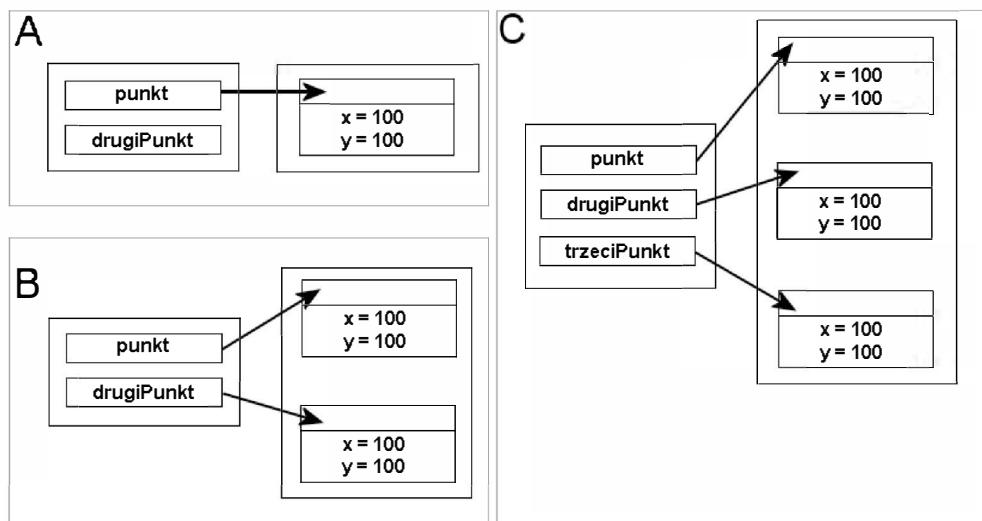
Rysunek 3.6.
Utworzenie trzech takich samych obiektów różnymi metodami

```
C:\>Punkt.exe
Obiekt pierwszyPunkt ma współrzędne:
współrzędna x = 100
współrzędna y = 100

Obiekt drugiPunkt ma współrzędne:
współrzędna x = 100
współrzędna y = 100

Obiekt trzeciPunkt ma współrzędne:
współrzędna x = 100
współrzędna y = 100

C:\>
```



Rysunek 3.7. Kolejne etapy powstawania zmiennych i obiektów w programie z listingu 3.19

W fazie pierwszej, na samym początku programu, mamy jedynie dwie zmienne: `pierwszyPunkt` i `drugiPunkt`. Tylko pierwszej z nich jest przypisany obiekt, druga jest po prostu pusta (zawiera wartość null). Przedstawiono to na rysunku 3.7 A. W części drugiej przypisujemy zmiennej `drugiPunkt` obiekt, który jest kopią obiektu `pierwszyPunkt` (rysunek 3.7 B), a w trzeciej tworzymy obiekt `trzeciPunkt` i wypełniamy go danymi pochodzącyimi z obiektu `drugiPunkt`. Tym samym ostatecznie otrzymujemy trzy zmienne i trzy obiekty (rysunek 3.7 C).

Przeciążanie metod

W trakcie pracy nad kodem klasy `Punkt` powstały dwie metody o takiej samej nazwie, ale różnym kodzie. Chodzi oczywiście o metody `ustawXY`. Pierwsza wersja przyjmowała jako argumenty dwie wartości typu `int`, a druga miała tylko jeden argument, którym był obiekt klasy `Punkt`. Okazuje się, że takie dwie metody mogą wspólnie istnieć w klasie `Punkt` i z obu z nich można korzystać w kodzie programu.

Ogólnie rzecz ujmując, w każdej klasie może istnieć dowolna liczba metod, które mają takie same nazwy, o ile tylko różnią się argumentami. Mogą one — ale nie muszą — również różnić się typem zwracanego wyniku. Taka funkcjonalność nosi nazwę **przeciążania metod** (ang. *methods overloading*). Skonstruujmy zatem taką klasę `Punkt`, w której znajdą się obie wersje metody `ustawXY`. Kod tej klasy został przedstawiony na listingu 3.20.

Listing 3.20. Przeciążone metody `UstawXY` w klasie `Punkt`

```
class Punkt
{
    int x;
    int y;

    void ustawXY(int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
    }

    void ustawXY(Punkt punkt)
    {
        x = punkt.x;
        y = punkt.y;
    }
}
```

Klasa ta zawiera w tej chwili dwie przeciążone metody o nazwie `ustawXY`. Jest to możliwe, ponieważ przyjmują one różne argumenty: pierwsza metoda — dwie liczby typu `int`, druga — jeden obiekt klasy `Punkt`. Obie metody realizują takie samo zadanie, tzn. ustawiają nowe wartości w polach `x` i `y`. Możemy przetestować ich działanie, dopisując do klasy `Punkt` metodę `Main` w postaci widocznej na listingu 3.21.

Listing 3.21. Metoda `Main` do klasy `Punkt` z listingu 3.20

```
public static void Main()
{
    Punkt punkt1 = new Punkt();
    Punkt punkt2 = new Punkt();

    punkt1.ustawXY(100, 100);
    punkt2.ustawXY(200, 200);

    System.Console.WriteLine("Po pierwszym ustawieniu współrzędnych:");
    System.Console.WriteLine("x = " + punkt1.x);
```

```
System.Console.WriteLine("y = " + punkt1.y);
System.Console.WriteLine("");
punkt1.ustawXY(punkt2);

System.Console.WriteLine("Po drugim ustawieniu współrzędnych:");
System.Console.WriteLine("x = " + punkt1.x);
System.Console.WriteLine("y = " + punkt1.y);
}
```

Działanie tej metody jest proste i nie wymaga wielu wyjaśnień. Na początku tworzymy dwa obiekty klasy `Punkt` i przypisujemy je zmiennym `punkt1` oraz `punkt2`. Następnie korzystamy z pierwszej wersji przeciążonej metody `ustawXY`, aby przypisać polom `x` i `y` pierwszego obiektu wartość 100, a polom `x` i `y` drugiego obiektu — 200. Dalej wyświetlamy zawartość obiektu `punkt1` na ekranie. Potem wykorzystujemy drugą wersję metody `ustawXY` w celu zmiany wartości pól obiektu `punkt1`, tak aby zawierały wartości zapisane w obiekcie `punkt2`. Następnie ponownie wyświetlamy wartości pól obiektu `punkt1` na ekranie.

Argumenty metody Main

Każdy program musi zawierać punkt startowy, czyli miejsce, od którego zacznie się jego wykonywanie. W C# takim miejscem jest metoda o nazwie `Main` i następującej deklaracji:

```
public static void Main()
{
    //treść metody Main
}
```

Jeśli w danej klasie znajdzie się metoda w takiej postaci, od niej właśnie zacznie się wykonywanie kodu programu. Teraz powinno być już jasne, dlaczego dotychczas prezentowane przykładowe programy miały schematyczną konstrukcję:

```
class Program
{
    public static void Main()
    {
        //tutaj instrukcje do wykonania
    }
}
```

Ta konstrukcja może mieć również nieco inną postać. Otóż metoda `Main` może przyjmować argument, którym jest tablica ciągów znaków. Zatem istnieje również jej przeciążona wersja o schematycznej postaci:

```
public static void Main(String[] args)
{
    //treść metody Main
}
```

Tablica `args` zawiera parametry wywołania programu, czyli argumenty przekazane z wiersza poleceń. O tym, że tak jest w istocie, można się przekonać, uruchamiając

program widoczny na listingu 3.22. Wykorzystuje on pętlę `for` do przejrzenia i wyświetlenia na ekranie zawartości wszystkich komórek tablicy `args` (można byłoby też z powodzeniem użyć pętli `foreach`). Przykładowy wynik jego działania jest widoczny na rysunku 3.8.

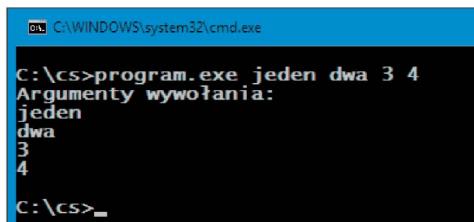
Listing 3.22. Odczytanie argumentów podanych z wiersza poleceń

```
using System;

public class Program
{
    public static void Main(String[] args)
    {
        Console.WriteLine("Argumenty wywołania:");
        for(int i = 0; i < args.Length; i++)
        {
            Console.WriteLine(args[i]);
        }
    }
}
```

Rysunek 3.8.

Program wyświetlający parametry jego wywołania



Sposoby przekazywania argumentów

Argumenty metod domyślnie przekazywane są **przez wartość** (ang. *by value*). To oznacza, że wewnątrz metody dostępna jest tylko kopia argumentu, a w związku z tym jakiejkolwiek zmiany jego wartości będą wykonywane na tej kopii i obowiązywały wyłącznie wewnątrz metody. Jeśli mamy na przykład metodę `Zwieksz` o postaci:

```
public void Zwieksz(int arg)
{
    arg++;
}
```

i w którymś miejscu programu wywołamy ją, przekazując jako argument zmienną `liczba`, np. w następujący sposób:

```
int liczba = 100;
Zwieksz(liczba);
Console.WriteLine(liczba);
```

to metoda `Zwieksz` otrzyma do dyspozycji kopię wartości zmiennej `liczba` i zwiększenie wykonywane przez instrukcję `arg++`; będzie obowiązywało tylko w obrębie tej metody. Instrukcja `Console.WriteLine(liczba)`; spowoduje więc wyświetlenie wartości 100.

To zachowanie można zmienić — argumenty mogą być również przekazywane **przez referencję** (ang. *by reference*). Metoda otrzyma wtedy w postaci argumentu referencję do zmiennej i będzie mogła bezpośrednio operować na tej zmiennej (a nie na jej kopii). W takiej sytuacji należy zastosować słowa `ref` lub `out`. Różnica jest taka, że w pierwszym przypadku przekazywana zmienna musi być zainicjowana przed przekazaniem jej jako argument, a w przypadku drugim musi być zainicjowana wewnętrz metodą. Metoda `Zwieksz` mogłaby mieć zatem postać:

```
public void Zwieksz(ref int arg)
{
    arg++;
}
```

Wtedy fragment kodu:

```
int liczba = 100;
Zwieksz(ref liczba);
Console.WriteLine(liczba);
```

spowodowałby faktyczne zwiększenie zmiennej `liczba` o 1 i na ekranie, dzięki działaniu instrukcji `Console.WriteLine(liczba);`, pojawiłaby się wartość 101. Należy przy tym zwrócić uwagę, że słowo `ref` (a także `out`) musi być użyte również w wywołaniu metody (a nie tylko przy jej deklaracji). Praktyczne różnice w opisanych sposobach przekazywania argumentów zostały zobrazowane w przykładzie widocznym na lистingu 3.23.

Listing 3.23. Różnice w sposobach przekazywania argumentów

```
using System;

public class Program
{
    public void Zwieksz1(int arg)
    {
        arg++;
    }
    public void Zwieksz2(ref int arg)
    {
        arg++;
    }
    public void Zwieksz3(out int arg)
    {
        //int wartosc = arg;
        //arg++;
        arg = 10;
        arg++;
    }
    public static void Main(String[] args)
    {
        int liczba1 = 100, liczba2;
        Program pg = new Program();

        pg.Zwieksz1(liczba1);
        Console.WriteLine("Po wywołaniu Zwieksz1(liczba1):");
        Console.WriteLine(liczba1);
```

```
    pg.Zwieksz2(ref liczba1);
    Console.WriteLine("Po wywołaniu Zwieksz2(ref liczba1):");
    Console.WriteLine(liczba1);

    //pg.Zwieksz2(ref liczba2);

    pg.Zwieksz3(out liczba1);
    Console.WriteLine("Po wywołaniu Zwieksz3(out liczba1):");
    Console.WriteLine(liczba1);

    pg.Zwieksz3(out liczba2);
    Console.WriteLine("Po wywołaniu Zwieksz3(out liczba2):");
    Console.WriteLine(liczba2);
}
}
```

W kodzie zostały zdefiniowane trzy metody przyjmujące jeden argument typu int, zajmujące się zwiększeniem jego wartości. Pierwsza z nich (Zwieksz1) jest standar-dowa — argument nie zawiera żadnych modyfikatorów, a jego wartość jest zwiększa o jeden za pomocą operatora ++. Druga (Zwieksz2) ma identyczną konstrukcję, ale przed argumentem został zastosowany modyfikator ref. To oznacza, że zmieniona przekazywana jako argument będzie musiała być zainicjowana. W trzeciej metodzie (Zwieksz3) użyty został modyfikator out. To oznacza, że jej pierwsze dwie instrukcje są nieprawidłowe (dlatego zostały ujęte w komentarz). Otóż taki argument musi zostać zainicjowany wewnątrz metody przed wykonaniem jakiejkolwiek operacji na nim. Prawidłowa jest zatem dopiero trzecia instrukcja przypisująca argumentowi wartość 10, a także czwarta — zwiększająca tę wartość o jeden (po pierwszym przypisaniu wartości można już wykonywać dowolne operacje).

Działanie metod Zwieksz jest testowane w metodzie main, od której zaczyna się wykonywanie kodu programu. Zostały w niej zadeklarowane dwie zmienne: liczba1 i liczba2, pierwsza z nich została też od razu zainicjowana wartością 100, natomiast druga pozostała niezainicjowana. Następnie powstał obiekt pg klasy Program. Jest to konieczne, ponieważ aby korzystać z metod zdefiniowanych w klasie Program, niezbędny jest obiekt tej klasy⁷. Dalsze bloki kodu to wywołania kolejnych wersji metod Zwieksz i wyświetlanie bieżącego stanu zmiennej użytej w wywołaniu.

W pierwszym bloku używana jest metoda Zwieksz1, której w tradycyjny sposób przekazywany jest argument w postaci zmiennej liczba1. To oznacza, że metoda otrzymuje w istocie kopię zmiennej i operuje na tej kopii. A zatem zwiększenie wartości argumentu (arg++) obowiązuje wyłącznie w obrębie metody. Wartość zmiennej liczba1 w metodzie Main się nie zmieni (będzie równa 100).

W drugim bloku kodu używana jest metoda Zwieksz2, której przekazywany jest przez referencję z użyciem słowa ref argument w postaci zmiennej liczba1. Jest to prawidłowe wywołanie, gdyż liczba1 została zainicjowana w metodzie Main i w związku z tym ma określoną wartość. Takie wywołanie oznacza jednak, że we wnętrzu metody Zwieksz2 operacje wykonywane są na zmiennej liczba1 (a nie na jej kopii, jak miało

⁷ Inaczej metody musiałyby być zadeklarowane jako statyczne. Ta kwestia zostanie wyjaśniona w lekcji 19.

to miejsce w przypadku metody `Zwieksz1`). W efekcie po wywołaniu `pg.Zwieksz2(ref liczba1)` zmienna `liczba1` w metodzie `Main` będzie miała wartość 101 (została zwiększa przez instrukcję `arg++` znajdująca się w metodzie `Zwieksz2`).

Trzeci blok kodu zawiera tylko jedną instrukcję: `pg.Zwieksz2(ref liczba2);`, która została ujęta w komentarz, gdyż jest nieprawidłowa. Z użyciem słowa `ref` nie można przekazać metodzie `Zwieksz2` argumentu w postaci zmiennej `liczba2`, ponieważ ta zmienna nie została zainicjowana. Tymczasem słowo `ref` oznacza, że taka inicjalizacja jest wymagana. Usunięcie komentarza z tej instrukcji spowoduje więc błąd komplikacji.

W czwartym bloku kodu używana jest metoda `Zwieksz3`, której przekazywany jest przez referencję z użyciem słowa `out` argument w postaci zmiennej `liczba1`. Takie wywołanie jest prawidłowe, zmienna przekazywana z użyciem słowa `out` może być wcześniej zainicjalizowana, należy jednak pamiętać, że jej pierwotna wartość zostanie zawsze zmieniona w wywoływanej metodzie. Dlatego po tym wywołaniu zmienna `liczba1` będzie miała wartość 11 (wartość wynikającą z operacji wykonywanych w metodzie `Zwieksz3`).

W czwartym bloku kodu używana jest metoda `Zwieksz3`, której przekazywany jest przez referencję z użyciem słowa `out` argument w postaci zmiennej `liczba2`. To wywołanie jest również właściwe — słowo `out` wskazuje, że zmienna `liczba2` nie musi być zainicjowana przed przekazaniem do metody, ponieważ ta operacja zostanie wykonana właśnie w metodzie. W efekcie po wykonaniu metody `Zwieksz3` zmienna `liczba2` otrzyma wartość 11.

Ostatecznie zatem po skompilowaniu i uruchomieniu programu z listingu 3.23 na ekranie pojawi się widok przedstawiony na rysunku 3.9.

Rysunek 3.9.
Stan zmiennych
przy różnych
wywołaniach metod

C:\>Program.exe
Po wywołaniu Zwieksz1(liczba1):
100
Po wywołaniu Zwieksz2(ref liczba1):
101
Po wywołaniu Zwieksz3(out liczba1):
11
Po wywołaniu Zwieksz3(out liczba2):
11
C:\>-

Definicje metod za pomocą wyrażeń lambda

W C# 6.0 treść prostych metod może być definiowana w sposób skrócony, za pomocą tzw. **wyrażeń lambda** (ang. *lambda expressions*). Chociaż głównym celem tych konstrukcji programistycznych jest tworzenie bardziej zaawansowanych funkcjonalności takich jak funkcje lokalne (anonimowe), ich działanie można pokazać na uproszczenym przykładzie dotyczącym właśnie utworzenia metody.

Jeżeli przyjmiemy, że w programie ma powstać metoda przyjmująca dwa argumenty i zwracająca wynik ich dodawania, to stosując znane do tej pory techniki należałoby użyć konstrukcji o następującej postaci (zakładając, że metoda ma się nazywać Dodaj i przyjmować dwa argumenty typu Double):

```
public Double Dodaj(Double arg1, Double arg2)
{
    return arg1 + arg2;
}
```

Zamiast tego można użyć operatora lambda, który ma postać =>. Po lewej stronie tego operatora występuje pewien parametr, a po prawej stronie wyrażenie lub blok instrukcji. Oznacza to, że nasza metoda może też wyglądać tak:

```
public Double Dodaj(Double arg1, Double arg2) => arg1 + arg2;
```

Pełny program korzystający z takiej metody mógłby wtedy mieć postać przedstawioną na listingu 3.24 (w przykładzie użyto również dostępnej w C# 6.0 interpolacji łańcuchów znakowych).

Listing 3.24. Definicja metody za pomocą wyrażenia lambda

```
using System;

public class Program
{
    public Double Dodaj(Double arg1, Double arg2) => arg1 + arg2;
    public static void Main()
    {
        int liczba1 = 100, liczba2 = 200;
        Program pg = new Program();

        Double wynik = pg.Dodaj(liczba1, liczba2);
        Console.WriteLine($"wynik = {wynik}");
    }
}
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 15.1

Do klasy Punkt z listingów 3.16 i 3.17 dopisz metody UstawX i UstawY, które jako argument będą przyjmowały obiekt klasy Punkt.

Ćwiczenie 15.2

W klasie Punkt z listingu 3.16 zmień kod metod UstawX i UstawY, tak aby zwracały one poprzednią wartość zapisywanych pól. Zadaniem metody UstawX jest więc zmiana wartości pola x i zwrócenie jego poprzedniej wartości. Metoda UstawY ma wykonywać analogiczne zadania w stosunku do pola y.

Ćwiczenie 15.3

Do klasy Punkt z ćwiczenia 15.2 dopisz metodę `UstawXY` przyjmującą jako argument obiekt klasy Punkt. Polom `x` i `y` należy przypisać wartości pól `x` i `y` przekazanego obiektu. Metoda ma natomiast zwrócić obiekt klasy Punkt zawierający stare wartości `x` i `y`.

Ćwiczenie 15.4

Napisz kod przykładowej klasy o nazwie `Dzialania`. Umieść w niej metody `Dodaj` i `Odejmij` oraz pole o nazwie `wynik` (w deklaracji pola użyj słowa `public`, podobnie jak na listingu 3.8). Metoda `Dodaj` powinna przyjmować dwa argumenty oraz zapisywać wynik ich dodawania w polu `wynik`. Metoda `Odejmij` powinna działać analogicznie, z tą różnicą, że rezultatem jej wykonania powinna być różnica przekazanych argumentów.

Ćwiczenie 15.5

W oparciu o kod z ćwiczenia 15.4 napisz taką wersję klasy `Dzialania`, która wynik wykonywanych operacji będzie zapisywała w pierwszym argumencie, a jego pierwotna zawartość znajdzie się w polu `wynik`. Pamiętaj o takim sposobie przekazywania argumentów, aby wynik operacji dodawania lub odejmowania był dostępny po wywołaniu dowolnej z metod.

Ćwiczenie 15.6

Napisz przykładowy program ilustrujący działanie klas z ćwiczeń 15.4 i 15.5. Zastanów się, jakie modyfikacje trzeba wprowadzić, aby móc skorzystać z tych klas w jednym programie.

Lekcja 16. Konstruktory i destruktory

Lekcja 16. w większej części jest poświęcona **konstruktorom**, czyli specjalnym metodom wykonywanym podczas tworzenia obiektu. Można się z niej dowiedzieć, jak powstaje konstruktor, jak umieścić go w klasie, a także czy może przyjmować argumenty. Nie zostaną też pominięte informacje o sposobach przeciążania konstruktorów oraz o wykorzystaniu słowa kluczowego `this`. Na zakończenie przedstawiony zostanie też temat **destruktów**, które są wykonywane, kiedy obiekt jest usuwany z pamięci.

Czym jest konstruktor?

Po utworzeniu obiektu w pamięci wszystkie jego pola zawierają wartości domyślne. Wartości te dla poszczególnych typów danych zostały przedstawione w tabeli 3.1.

Tabela 3.1. Wartości domyślne niezainicjowanych pól obiektu

Typ	Wartość domyślna
byte	0
sbyte	0
short	0
ushort	0
int	0
uint	0
long	0
ulong	0
decimal	0.0
float	0.0
double	0.0
char	\0
bool	false
obiektowy	null

Najczęściej jednak chcemy, aby pola te zawierały jakieś konkretne wartości. Przykładowo moglibyśmy życzyć sobie, aby każdy obiekt klasy `Punkt` powstałej w lekcji 14. (listing 3.1) otrzymywał współrzędne: `x = 1` i `y = 1`. Oczywiście można po każdym utworzeniu obiektu przypisywać wartości tym polem, np.:

```
Punkt punkt1 = new Punkt();
punkt1.x = 1;
punkt1.y = 1;
```

Mogą też dopisać do klasy `Punkt` dodatkową metodę, na przykład o nazwie `inicjuj` (albo `init`, `initialize` lub podobnej), w postaci:

```
void init()
{
    x = 1;
    y = 1;
}
```

i wywoływać ją po każdym utworzeniu obiektu. Widać jednak od razu, że żadna z tych metod nie jest wygodna. Przede wszystkim wymagają one, aby programista zawsze pamiętał o ich stosowaniu, a jak pokazuje praktyka, jest to zwykle zbyt optymistyczne założenie. Na szczęście obiektowe języki programowania udostępniają dużo wygodniejszy mechanizm konstruktorów. Otóż konstruktor jest to specjalna metoda, która jest wywoływana zawsze w trakcie tworzenia obiektu w pamięci. Nadaje się więc doskonale do jego zainicjowania.

Metoda będąca konstruktorem nigdy nie zwraca wyniku i musi mieć nazwę zgodną z nazwą klasy, czyli schematycznie wygląda to następująco:

```
class nazwa_klasy
{
    nazwa_klasy()
    {
        //kod konstruktora
    }
}
```

Jak widać, przed definicją nie umieszcza się nawet słowa `void`, tak jak miałyby to miejsce w przypadku zwykłej metody. To, co będzie robił konstruktor, czyli jakie wykona zadania, zależy już tylko od programisty.

Dopisany zatem do klasy `Punkt` z listingu 3.1 (czyli jej najprostszej wersji) konstruktor, który będzie przypisywał polom `x` i `y` każdego obiektu wartość 1. Wygląd takiej klasy zaprezentowano na listingu 3.25.

Listing 3.25. Prosty konstruktor dla klasy `Punkt`

```
class Punkt
{
    int x;
    int y;
    Punkt()
    {
        x = 1;
        y = 1;
    }
}
```

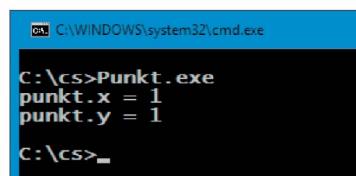
Jak widać, wszystko jest tu zgodne z podanym wyżej schematem. Konstruktor nie zwraca żadnej wartości i ma nazwę zgodną z nazwą klasy. Przed nazwą nie występuje słowo `void`. W jego ciele następuje proste przypisanie wartości polom obiektu. O tym, że konstruktor faktycznie działa, można się przekonać, pisząc dodatkowo metodę `Main`, w której skorzystamy z obiektu nowej klasy `Punkt`. Taka przykładowa metoda `Main` jest widoczna na listingu 3.26.

Listing 3.26. Metoda `Main` testująca konstruktor klasy `Punkt`

```
public static void Main()
{
    Punkt punkt1 = new Punkt();
    System.Console.WriteLine("punkt.x = " + punkt1.x);
    System.Console.WriteLine("punkt.y = " + punkt1.y);
}
```

Metoda ta ma wyjątkowo prostą konstrukcję, jedyne jej zadania to utworzenie obiektu klasy `Punkt` i przypisanie odniesienia do niego zmiennej `punkt1` oraz wyświetlenie zawartości jego pól na ekranie. Dzięki temu przekonamy się, że konstruktor faktycznie został wykonany, zobaczymy bowiem widok zaprezentowany na rysunku 3.10.

Rysunek 3.10.
Konstruktor klasy
Punkt faktycznie
został wykonany



Argumenty konstruktorów

Konstruktor nie musi być bezargumentowy, może również przyjmować argumenty, które zostaną wykorzystane, bezpośrednio lub pośrednio, na przykład do zainicjowania pól obiektu. Argumenty przekazuje się dokładnie tak samo jak w przypadku zwykłych metod (lekcja 15.). Schemat takiego konstruktora byłby więc następujący:

```
class nazwa_klasy
{
    nazwa_klasy(typ1 argument1, typ2 argument2,..., typN argumentN)
    {
    }
}
```

Jeśli konstruktor przyjmuje argumenty, to przy tworzeniu obiektu należy je podać, czyli zamiast stosowanej do tej pory konstrukcji:

nazwa_klasy zmienna = new nazwa_klasy()
trzeba zastosować wywołanie:

```
nazwa_klasy zmienna = new nazwa_klasy(argumenty_konstruktora)
```

W przypadku naszej przykładowej klasy Punkt byłby przydatny np. konstruktor przyjmujący dwa argumenty, które oznaczałyby współrzędne punktu. Jego definicja, co nie jest z pewnością żadnym zaskoczeniem, wyglądać będzie następująco:

```
Punkt(int wspX, int wspY)
{
    x = wspX;
    y = wspY;
}
```

Kiedy zostanie umieszczony w klasie Punkt, przyjmie ona postać widoczną na lистingu 3.27.

Listing 3.27. Konstruktor przyjmujący argumenty

```
class Punkt
{
    int x;
    int y;
    Punkt(int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
    }
}
```

Teraz podczas każdej próby utworzenia obiektu klasy `Punkt` będziemy musieli podawać jego współrzędne. Jeśli na przykład początkowa współrzędna `x` ma mieć wartość 100, a początkowa współrzędna `y` — 200, powinniśmy zastosować konstrukcję:

```
Punkt punkt = new Punkt(100, 200);
```

Taka instrukcja może być umieszczona w metodzie `Main` (analogicznie do przykładu z listingu 3.26) testującej zachowanie tej wersji konstruktora, której przykładowa postać została zaprezentowana na listingu 3.28.

Listing 3.28. Testowanie konstruktora przyjmującego argumenty

```
public static void Main()
{
    Punkt punkt1 = new Punkt(100, 200);
    System.Console.WriteLine("punkt.x = " + punkt1.x);
    System.Console.WriteLine("punkt.y = " + punkt1.y);
}
```

Przeciążanie konstruktorów

Konstruktory, tak jak zwykłe metody, mogą być przeciążane, tzn. każda klasa może mieć kilka konstruktorów, o ile tylko różnią się one przyjmowanymi argumentami. Do tej pory powstały dwa konstruktory klasy `Punkt`: pierwszy bezargumentowy i drugi przyjmujący dwa argumenty typu `int`. Dopiszmy zatem jeszcze trzeci, który jako argument będzie przyjmował obiekt klasy `Punkt`. Jego postać będzie zatem następująca:

```
Punkt(Punkt punkt)
{
    x = punkt.x;
    y = punkt.y;
}
```

Zasada działania jest prosta: polu `x` jest przypisywana wartość pola `x` obiektu przekazanego jako argument, natomiast polu `y` — wartość pola `y` tego obiektu. Można teraz zebrać wszystkie trzy napisane dotychczas konstruktory i umieścić je w klasie `Punkt`. Będzie ona wtedy miała postać przedstawioną na listingu 3.29.

Listing 3.29. Trzy konstruktory w klasie `Punkt`

```
class Punkt
{
    int x;
    int y;
    Punkt()
    {
        x = 1;
        y = 1;
    }
    Punkt(int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
```

```
        }
        Punkt(Punkt punkt)
        {
            x = punkt.x;
            y = punkt.y;
        }
    }
```

Taka budowa klasy `Punkt` pozwala na osobne wywoływanie każdego z trzech konstruktów, w zależności od tego, który z nich jest najbardziej odpowiedni w danej sytuacji. Warto teraz na konkretnym przykładzie przekonać się, że tak jest w istocie. Dopiszemy więc do klasy `Punkt` metodę `Main`, w której utworzymy trzy obiekty typu `Punkt`, a każdy z nich będzie tworzony za pomocą innego konstruktora. Taka przykładowa metoda jest widoczna na listingu 3.30.

Listing 3.30. Użycie przeciążonych konstruktorów

```
public static void Main()
{
    Punkt punkt1 = new Punkt();

    System.Console.WriteLine("punkt1:");
    System.Console.WriteLine("x = " + punkt1.x);
    System.Console.WriteLine("y = " + punkt1.y);
    System.Console.WriteLine("");

    Punkt punkt2 = new Punkt(100, 100);

    System.Console.WriteLine("punkt2:");
    System.Console.WriteLine("x = " + punkt2.x);
    System.Console.WriteLine("y = " + punkt2.y);
    System.Console.WriteLine("");

    Punkt punkt3 = new Punkt(punkt1);

    System.Console.WriteLine("punkt3:");
    System.Console.WriteLine("x = " + punkt3.x);
    System.Console.WriteLine("y = " + punkt3.y);
    System.Console.WriteLine("");
}
```

Pierwszy obiekt — `punkt1` — jest tworzony za pomocą konstruktora bezargumentowego, który przypisuje polom `x` i `y` wartość 1. Obiekt drugi — `punkt2` — jest tworzony poprzez wywołanie drugiego ze znanych nam konstruktów, który przyjmuje dwa argumenty odzwierciedlające wartości `x` i `y`. Oba pola otrzymują wartość 100. Konstruktor trzeci, zastosowany wobec obiektu `punkt3`, to nasza najnowsza konstrukcja. Jako argument przyjmuje on obiekt klasy `Punkt`, w tym przypadku obiekt wskazywany przez `punkt1`. Ponieważ w tym obiekcie oba pola mają wartość 1, również pola obiektu `punkt3` przyjmą wartość 1. W efekcie działania programu zobaczymy na ekranie widok zaprezentowany na rysunku 3.11.

Rysunek 3.11.

Wykorzystanie
trzech różnych
konstruktorów
klasy Punkt

```
C:\>Punkt.exe
punkt1:
x = 1
y = 1

punkt2:
x = 100
y = 100

punkt3:
x = 1
y = 1

C:\>
```

Słowo kluczowe this

Słowo kluczowe `this` to nic innego jak odwołanie do obiektu bieżącego. Można je traktować jako referencję do aktualnego obiektu. Najłatwiej pokazać to na przykładzie. Założymy, że mamy konstruktor klasy `Punkt`, taki jak na listingu 3.27, czyli przyjmujący dwa argumenty, którymi są liczby typu `int`. Nazwami tych argumentów były `wspX` i `wspY`. Co by się jednak stało, gdyby ich nazwami były `x` i `y`, czyli gdyby jego deklaracja wyglądała jak poniżej?

```
Punkt(int x, int y)
{
    //treść konstruktora
}
```

Co należy wpisać w jego treści, aby spełniał swoje zadanie? Gdybyśmy postępowali w sposób podobny jak w przypadku klasy z listingu 3.27, otrzymalibyśmy konstrukcję:

```
Punkt(int x, int y)
{
    x = x;
    y = x;
}
```

Oczywiście, nie ma to najmniejszego sensu⁸. W jaki bowiem sposób kompilator ma ustalić, kiedy chodzi nam o argument konstruktora, a kiedy o pole klasy, jeśli ich nazwy są takie same? Oczywiście sam sobie nie poradzi i tu właśnie z pomocą przychodzi nam słowo `this`. Otóż jeśli chcemy zaznaczyć, że chodzi nam o składową klasy (np. pole, metodę), korzystamy z odwołania w postaci:

```
this.nazwa_pola
```

lub:

```
this.nazwa_metody(argumenty)
```

Wynika z tego, że poprawna postać opisywanego konstruktora powinna wyglądać następująco:

⁸ Chociaż formalnie taki zapis jest w pełni poprawny.

```
Punkt(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

Instrukcję `this.x = x` rozumiemy jako: „Przypisz polu `x` wartość przekazaną jako argument o nazwie `x`”, a instrukcję `this.y = y` analogicznie jako: „Przypisz polu `y` wartość przekazaną jako argument o nazwie `y`”.

Słowo `this` pozwala również na inną ciekawą konstrukcję. Umożliwia mianowicie wywołanie konstruktora z wnętrza innego konstruktora. Może to być przydatne w sytuacji, kiedy w klasie mamy kilka przeciążonych konstruktorów, a zakres wykonywanego przez nie kodu się pokrywa. Nie zawsze takie wywołanie jest możliwe i niezbędne, niemniej taka możliwość istnieje, trzeba więc wiedzieć, jak takie zadanie zrealizować.

Jeżeli za jednym z konstruktorów umieścimy dwukropek, a za nim słowo `this` i listę argumentów umieszczonych w nawiasie okrągłym, czyli zastosujemy konstrukcję o schemacie:

```
class nazwa_klasy
{
    nazwa_klasy(argumenty):this(argument1, argument2, ... , argumentN)
    {
    }
    //pozostałe konstruktory
}
```

to przed widocznym konstruktorem zostanie wywołany ten, którego argumenty pasują do wymienionych w nawiasie po `this`. Jest to tak zwane zastosowanie **inicjalizatora** lub **listy inicjalizacyjnej**. Przykład kodu wykorzystującego taką technikę jest widoczny na listingu 3.31.

Listing 3.31. Wywołanie konstruktora z wnętrza innego konstruktora

```
class Punkt
{
    int x;
    int y;
    Punkt(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    Punkt():this(1, 1)
    {
    }
    public static void Main()
    {
        Punkt punkt1 = new Punkt(100, 200);
        Punkt punkt2 = new Punkt();
        System.Console.WriteLine("punkt1.x = " + punkt1.x);
        System.Console.WriteLine("punkt1.y = " + punkt1.y);
        System.Console.WriteLine("");
        System.Console.WriteLine("punkt2.x = " + punkt2.x);
```

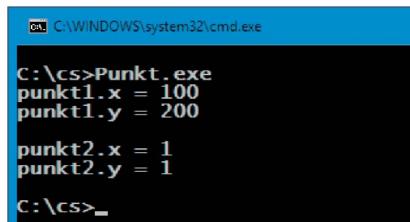
```
        System.Console.WriteLine("punkt2.y = " + punkt2.y);
    }
}
```

Klasa Punkt ma teraz dwa konstruktory. Jeden z nich ma postać standardową — przyjmuje po prostu dwa argumenty typu int i przypisuje ich wartości polom x i y. Drugi z konstruktorów jest natomiast bezargumentowy, a jego zadaniem jest przypisanie polom x i y wartości 1. Nie dzieje się to jednak w sposób znany z dotychczasowych przykładów. Wnętrze tego konstruktora jest puste⁹, a wykorzystywana jest lista inicjalizacyjna — Punkt():this(1, 1). Dzięki temu jest wywoływany konstruktor, którego argumenty są zgodne z podanymi na liście, a więc konstruktor przyjmujący dwa argumenty typu int.

Jak więc zadziała kod metody Main? Najpierw za pomocą konstruktora dwuargumentowego jest tworzony obiekt punkt1. Tu nie dzieje się nic nowego, pola otrzymują zatem wartości 100 i 200. Następnie powstaje obiekt punkt2, a do jego utworzenia jest wykorzystywany konstruktor bezargumentowy. Ponieważ korzysta on z listy inicjalizacyjnej, najpierw zostanie wywołany konstruktor dwuargumentowy, któremu w postaci argumentów zostaną przekazane dwie wartości 1. A zatem oba pola obiektu punkt2 przyjmą wartość 1. Przekonujemy się o tym, wyświetlając wartości pól obu obiektów na ekranie (rysunek 3.12).

Rysunek 3.12.

Efekt użycia konstruktora korzystającego z listy inicjalizacyjnej



```
C:\Windows\system32\cmd.exe
C:\cs>Punkt.exe
punkt1.x = 100
punkt1.y = 200
punkt2.x = 1
punkt2.y = 1
C:\cs>-
```

Argumentem przekazywanym na liście inicjalizacyjnej może też być argument konstruktora (patrz też zadanie 16.4 w podrozdziale „Ćwiczenia do samodzielnego wykonania”). Schematycznie można taką sytuację przedstawić następująco:

```
class nazwa_klasy
{
    nazwa_klasy(argument)
    {
    }
    nazwa_klasy(argument1, argument2):this(argument1)
    {
    }
    //pozostałe konstruktory
}
```

W takim przypadku argument o nazwie *argument1* zostanie użyty zarówno w konstruktorze jednoargumentowym, jak i dwuargumentowym.

⁹ Oczywiście nie jest to obligatoryjne. Konstruktor korzystający z listy inicjalizacyjnej może również zawierać instrukcje wykonujące inne czynności.

Niszczenie obiektu

Osoby, które programowały w językach obiektowych, takich jak np. C++ czy Object Pascal, zastanawiają się zapewne, jak w C# wygląda destruktor i kiedy zwalniamy pamięć zarezerwowaną dla obiektów. Skoro bowiem operator new pozwala na utworzenie obiektu, a tym samym na zarezerwowanie dla niego pamięci operacyjnej, logicznym założeniem jest, że po jego wykorzystaniu pamięć tę należy zwolnić. Ponieważ jednak takie podejście, tzn. zrzucenie na barki programistów konieczności zwalniania przydzielonej obiektom pamięci, powodowało powstawanie wielu błędów, w nowoczesnych językach programowania stosuje się inne rozwiązanie. Otóż za zwalnianie pamięci odpowiada środowisko uruchomieniowe, a programista praktycznie nie ma nad tym procesem kontroli¹⁰.

Zajmuje się tym tak zwany **odśmieczacz** (ang. *garbage collector*), który czuwa nad optymalnym wykorzystaniem pamięci i uruchamia proces jej odzyskiwania w momencie, kiedy wolna ilość oddana do dyspozycji programu zbytnio się zmniejszy. Jest to wyjątkowo wygodne podejście dla programisty, zwalnia go bowiem z obowiązku zarządzania pamięcią. Zwiększa jednak narzuty czasowe związane z wykonaniem programu, wszak sam proces odśmieciania musi zająć czas procesora. Niemniej dzisiejsze środowiska uruchomieniowe są na tyle dopracowane, że w większości przypadków nie ma najmniejszej potrzeby zaprzątania myśli tym problemem.

Trzeba jednak zdawać sobie sprawę, że środowisko .NET jest w stanie automatycznie zarządzać wykorzystywaniem pamięci, ale tylko tej, która jest alokowana standardowo, czyli za pomocą operatora new. W nielicznych przypadkach, np. w sytuacji, gdyby stworzony przez nas obiekt wykorzystywał jakieś specyficzne zasoby, które nie mogą być zwolnione automatycznie, o posprzątanie systemu trzeba zadbać samodzielnie.

W C# w tym celu wykorzystuje się **destruktory**¹¹, które są wykonywane zawsze, kiedy obiekt jest niszczony, usuwany z pamięci. Wystarczy więc, jeśli klasa będzie zawierała taki destruktor, a przy niszczeniu jej obiektu zostanie on wykonany. W ciele destruktora można wykonać dowolne instrukcje sprzątające. Destruktor deklaruje się tak jak konstruktor, z tą różnicą, że nazwę poprzedzamy znakiem tyldy, ogólnie:

```
class nazwa_klasy
{
    ~nazwa_klasy()
    {
        //kod destruktora
    }
}
```

¹⁰ Aczkolwiek wywołując metodę System.GC.Collect(), można wymusić zainicjowanie procesu odzyskiwania pamięci. Nie należy jednak tego wywołania nadużywać.

¹¹ W rzeczywistości destruktor jest tłumaczony wewnętrznie (przez kompilator) na wywołanie metody Finalize (co dodatkowo jest obejmowane blokiem obsługi sytuacji wyjątkowych), można więc z równie dobrym skutkiem umieścić zamiast niego w klasie taką metodę. Użycie destruktora wydaje się jednak czytelniejsze.

Destruktora należy jednak używać tylko i wyłącznie w sytuacji, kiedy faktycznie niezbędne jest zwolnienie alokowanych niestandardowo zasobów. Nie należy natomiast umieszczać w kodzie pustych destruktatorów, gdyż obniży to wydajność aplikacji¹².

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 16.1

Napisz klasę, której zadaniem będzie przechowywanie wartości typu `int`. Dołącz jedenargumentowy konstruktor przyjmujący argument typu `int`. Polu klasy nadaj nazwę `liczba`, tak samo nazwij argument konstruktora.

Ćwiczenie 16.2

Do klasy powstalej w ćwiczeniu 16.1 dopisz przeciążony konstruktor bezargumentowy ustawiający jej pole na wartość `-1`.

Ćwiczenie 16.3

Napisz klasę zawierającą dwa pola: pierwsze typu `double` i drugie typu `char`. Dopisz cztery przeciążone konstruktory: pierwszy przyjmujący jeden argument typu `double`, drugi przyjmujący jeden argument typu `char`, trzeci przyjmujący dwa argumenty — pierwszy typu `double`, drugi typu `char` — i czwarty przyjmujący również dwa argumenty — pierwszy typu `char`, drugi typu `double`.

Ćwiczenie 16.4

Zmień kod klasy powstalej w ćwiczeniu 16.3 tak, aby w konstruktorach dwuargumentowych były wykorzystywane konstruktory jednoargumentowe.

Ćwiczenie 16.5

Napisz kod klasy przechowującej dane określające prostokąt na płaszczyźnie; zapamiętywane mają być współrzędne lewego górnego rogu oraz prawego dolnego rogu. Do klasy dodaj jeden konstruktor przyjmujący cztery argumenty liczbowe, które będą określały współrzędne lewego górnego rogu oraz szerokość i wysokość prostokąta.

Ćwiczenie 16.6

Napisz klasę `Kwadrat` przechowującą informację o kwadracie. Klasa powinna mieć konstruktory umożliwiające przekazanie parametrów o następujących znaczeniach: a) współrzędne lewego górnego rogu oraz prawego dolnego, b) współrzędne lewego górnego rogu oraz długość boku, c) współrzędne środka i długość boku, d) współrzędne środka i pole.

¹² Ze względu na specjalne traktowanie takich obiektów przez środowisko uruchomieniowe.

Ćwiczenie 16.7.

Do klasy `Kwadrat` z ćwiczenia 16.6 dopisz metodę wyświetlającą dane kwadratu (współrzędne lewego górnego rogu i długość boku) oraz metodę `Main` testującą działanie wszystkich konstruktorów (na cztery sposoby powinien powstać taki sam kwadrat — o identycznych współrzędnych).

Dziedziczenie

Dziedziczenie (ang. *inheritance*) to jeden z fundamentów programowania obiektowego. Umożliwia sprawne i łatwe wykorzystywanie już raz napisanego kodu czy budowanie hierarchii klas przejmujących swoje właściwości. Ten podrozdział zawiera trzy lekcje przybliżające temat dziedziczenia. W lekcji 17. zaprezentowane są podstawy, czyli sposoby tworzenia klas potomnych oraz zachowania konstruktorów klasy bazowej i potomnej. W lekcji 18. poruszony został temat modyfikatorów dostępu pozwalających na ustalanie praw dostępu do składowych klas. W lekcji 19. przedstawiono techniki przesłaniania pól i metod w klasach potomnych oraz składowe statyczne.

Lekcja 17. Klasy potomne

W lekcji 17. przedstawione zostały podstawy dziedziczenia, czyli budowania nowych klas na bazie już istniejących. Każda taka nowa klasa przejmuje zachowanie i właściwości klasy bazowej. Zobaczmy, jak tworzy się klasy potomne, jakie podstawowe zależności występują między klasą bazową a potomną oraz jak zachowują się konstruktory w przypadku dziedziczenia.

Dziedziczenie

Na początku lekcji 14. utworzyliśmy klasę `Punkt`, która przechowywała informację o współrzędnych punktu na płaszczyźnie. W trakcie dalszych ćwiczeń rozbudowaliśmy ją o dodatkowe metody, które pozwalały na ustawianie i pobieranie tych współrzędnych. Zastanówmy się teraz, co byśmy zrobili, gdybyśmy chcieli określić położenie punktu nie w dwóch, ale w trzech wymiarach, czyli gdyby do współrzędnych `x` i `y` trzeba było dodać współrzędną `z`. Pomyślem, który się od razu nasuwa, jest napisanie dodatkowej klasy, np. o nazwie `Punkt3D`, w postaci:

```
class Punkt3D
{
    int x;
    int y;
    int z;
}
```

Do tej klasy należałoby dalej dopisać pełny zestaw metod, które znajdowały się w klasie Punkt, takich jak `PobierzX`, `PobierzY`, `UstawX`, `UstawY` itd., oraz dodatkowe metody operujące na współrzędnej `z`. Zauważmy jednak, że w takiej sytuacji w dużej części po prostu powtarzamy już raz napisany kod. Czym bowiem będzie się różniła metoda `UstawX` klasy Punkt od metody `UstawX` klasy `Punkt3D`? Oczywiście niczym. Po prostu `Punkt3D` jest pewnego rodzaju rozszerzeniem klasy Punkt. Rozszerza ją o dodatkowe możliwości (pola, metody), pozostawiając stare właściwości bez zmian. Zamiast więc pisać całkiem od nowa klasę `Punkt3D`, lepiej spowodować, aby przejęła ona wszystkie możliwości klasy Punkt, wprowadzając dodatkowo swoje własne. Jest to tak zwane dziedziczenie, czyli jeden z fundamentów programowania obiektowego. Powiemy, że klasa `Punkt3D` dziedziczy po klasie Punkt, czyli przejmuje jej składowe oraz dodaje swoje własne.

W C# dziedziczenie jest wyrażane za pomocą symbolu dwukropka, a cała definicja schematycznie wygląda następująco:

```
class klasa_potomna : klasa_bazowa
{
    //wnętrze klasy
}
```

Zapis taki oznacza, że klasa potomna (inaczej: podrzędna, pochodna, ang. *subclass*, *child class*) dziedziczy po klasie bazowej (inaczej: nadrzędnej, nadklasie, ang. *base class*, *superclass*, *parent class*)¹³. Zobaczmy, jak taka deklaracja będzie wyglądała w praktyce dla wspomnianych klas `Punkt` i `Punkt3D`. Jest to bardzo proste:

```
class Punkt3D : Punkt
{
    int z;
}
```

Taki zapis oznacza, że klasa `Punkt3D` przejęła wszystkie właściwości klasy `Punkt`, a dodatkowo otrzymała pole typu `int` o nazwie `z`. Przekonajmy się, że tak jest w istocie. Niech klasy `Punkt` i `Punkt3D` wyglądają tak, jak na listingu 3.32.

Listing 3.32. Dziedziczenie pomiędzy klasami

```
class Punkt
{
    public int x;
    public int y;

    public int PobierzX()
    {
        return x;
    }
    public int PobierzY()
    {
        return y;
    }
    public void UstawX(int wspX)
    {
```

¹³ Często spotykany jest też termin „dziedziczyć z klasy”.

```
        x = wspX;
    }
    public void UstawY(int wspY)
    {
        y = wspY;
    }
    public void UstawXY(int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
    }
    public void WyświetlWspółrzędne()
    {
        System.Console.WriteLine("współrzędna x = " + x);
        System.Console.WriteLine("współrzędna y = " + y);
    }
}

class Punkt3D : Punkt
{
    public int z;
}
```

Klasa Punkt ma tu postać znana z wcześniejszych przykładów. Zawiera dwa pola, x i y, oraz sześć metod: PobierzX i PobierzY (zwracające współrzędne x i y), UstawX, UstawY i UstawXY (ustawiające współrzędne) oraz WyświetlWspółrzędne (wyświetlającą wartości pól x i y na ekranie). Ponieważ klasa Punkt3D dziedziczy po klasie Punkt, również zawiera wymienione pola i metody oraz dodatkowo pole o nazwie z. Przed każdą składową (polem lub metodą) zostało umieszczone słowo public. Oznacza ono, że składowe są dostępne publicznie, a więc można się do nich bezpośrednio odwoływać. Ta kwestia zostanie dokładniej wyjaśniona w kolejnej lekcji.

Kod z listingu 3.32 można zapisać w jednym pliku, np. o nazwie *Punkt.cs*, lub też w dwóch. Skorzystajmy z tej drugiej możliwości i zapiszmy kod klasy Punkt w pliku *Punkt.cs*, a klasy Punkt3D w pliku *Punkt3D.cs*. Napiszemy też teraz dodatkową klasę Program, widoczną na listingu 3.33, testującą obiekt klasy Punkt3D. Pozwoli to naoczyć się, że na takim obiekcie zadziałają wszystkie metody, które znajdowały się w klasie Punkt.

Listing 3.33. Testowanie klasy *Punkt3D*

```
using System;

public class Program
{
    public static void Main()
    {
        Punkt3D punkt = new Punkt3D();

        Console.WriteLine("x = " + punkt.x);
        Console.WriteLine("y = " + punkt.y);
        Console.WriteLine("z = " + punkt.z);
        Console.WriteLine("");
    }
}
```

```
punkt.UstawX(100);
punkt.UstawY(200);

Console.WriteLine("x = " + punkt.x);
Console.WriteLine("y = " + punkt.y);
Console.WriteLine("z = " + punkt.z);
Console.WriteLine("");

punkt.UstawXY(300, 400);

Console.WriteLine("x = " + punkt.x);
Console.WriteLine("y = " + punkt.y);
Console.WriteLine("z = " + punkt.z);
}
}
```

Na początku definiujemy zmienną klasy `Punkt3D` o nazwie `punkt` i przypisujemy jej referencję do nowo utworzonego obiektu klasy `Punkt3D`. Wykorzystujemy oczywiście dobrze nam znany operator `new`. Następnie wyświetlamy na ekranie wartości wszystkich pól tego obiektu. Wiemy, że są to trzy pola, `x`, `y`, `z`, oraz że powinny onetrzymać wartości domyślne równe 0 (tabela 3.1). Następnie wykorzystujemy metody `UstawX` oraz `UstawY`, aby przypisać polom `x` i `y` wartości 100 oraz 200. W kolejnym kroku ponownie wyświetlamy zawartość wszystkich pól na ekranie. W dalszej części kodu wykorzystujemy metodę `UstawXY` do przypisania polu `x` wartości 300, a polu `y` wartości 400 i jeszcze jeden raz wyświetlamy zawartość wszystkich pól na ekranie.

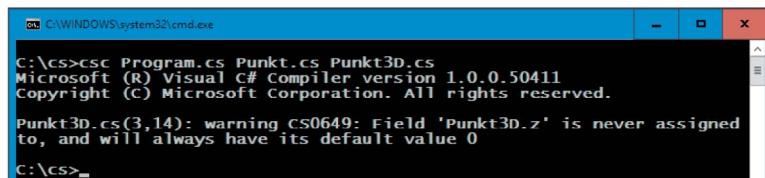
Możemy więc skompilować program. Ponieważ składa się on z trzech plików: `Program.cs`, `Punkt.cs` i `Punkt3D.cs`, w wierszu poleceń trzeba wydać komendę:

```
csc Program.cs Punkt.cs Punkt3D.cs
```

Kompilator wyświetli ostrzeżenie widoczne na rysunku 3.13. Jest to informacja o tym, że nie wykorzystujemy pola `z` i że będzie ono miało cały czas wartość 0, czym oczywiście nie musimy się przejmować — jest to prawda; faktycznie nigdzie nie ustawiliśmy wartości pola `z`.

Rysunek 3.13.

Ostrzeżenie kompilatora o niezainicjalizowanym polu z



Po uruchomieniu zobaczymy widok zaprezentowany na rysunku 3.14. Jest to też najlepszy dowód, że faktycznie klasa `Punkt3D` odziedziczyła wszystkie pola i metody klasy `Punkt`.

Klasa `Punkt3D` nie jest jednak w takiej postaci w pełni funkcjonalna, należałoby przecież dopisać metody operujące na nowym polu `z`. Na pewno przydatne będą: `UstawZ`, `PobierzZ` oraz `UstawXYZ`. Oczywiście metoda `UstawZ` będzie przyjmowała jeden argument typu `int` i przypisywała jego wartość polu `z`, metoda `pobierzZ` będzie zwracała

Rysunek 3.14.

Klasa *Punkt3D*
przejęła pola
i metody
klasy *Punkt*

```
C:\>Program.exe
x = 0
y = 0
z = 0

x = 100
y = 200
z = 0

x = 300
y = 400
z = 0
C:\>
```

wartość pola *z*, natomiast *ustawXYZ* będzie przyjmowała trzy argumenty typu *int* i przypisywała je polom *x*, *y* i *z*. Z pewnością nie jest żadnym zaskoczeniem, że metody te będą wyglądały tak, jak jest to zaprezentowane na listingu 3.34. Można się również zastanowić nad dopisaniem metod analogicznych do *ustawXY*, czyli metod *ustawXZ* oraz *ustawYZ*, to jednak będzie dobrym ćwiczeniem do samodzielnego wykonania.

Listing 3.34. Metody operujące na polu *z*

```
class Punkt3D : Punkt
{
    public int z;
    public void UstawZ(int wspZ)
    {
        z = wspZ;
    }
    public int PobierzZ()
    {
        return z;
    }
    public void UstawXYZ(int wspX, int wspY, int wspZ)
    {
        x = wspX;
        y = wspY;
        z = wspZ;
    }
}
```

Konstruktory klasy bazowej i potomnej

Klasom widocznym na listingach 3.32 i 3.34 brakuje konstruktorów. Przypomnijmy sobie, że w trakcie prac nad klasą *Punkt* powstały aż trzy konstruktory (listing 3.29 z lekcji 16.):

- ◆ bezargumentowy, ustawiający wartość wszystkich pól na 1;
- ◆ dwuargumentowy, przyjmujący dwie wartości typu *int*;
- ◆ jednoargumentowy, przyjmujący obiekt klasy *Punkt*.

Można je z powodzeniem dopisać do kodu widocznego na listingu 3.32. Niestety, żaden z nich nie zajmuje się polem `z`, którego w klasie `Punkt` po prostu nie ma. Konstruktory dla klasy `Punkt3D` musimy więc napisać osobno. Nie jest to skomplikowane zadanie, zostały one zaprezentowane na listingu 3.35.

Listing 3.35. Konstruktory dla klasy `Punkt3D`

```
class Punkt3D : Punkt
{
    public int z;
    public Punkt3D()
    {
        x = 1;
        y = 1;
        z = 1;
    }
    public Punkt3D(int wspX, int wspY, int wspZ)
    {
        x = wspX;
        y = wspY;
        z = wspZ;
    }
    public Punkt3D(Punkt3D punkt)
    {
        x = punkt.x;
        y = punkt.y;
        z = punkt.z;
    }
    /*
    ...pozostałe metody klasy Punkt3D...
    */
}
```

Jak widać, pierwszy konstruktor nie przyjmuje żadnych argumentów i przypisuje wszystkim polom wartość 1. Konstruktor drugi przyjmuje trzy argumenty: `wspX`, `wspY` oraz `wspZ`, wszystkie typu `int`, i przypisuje otrzymywane wartości polom `x`, `y` i `z`. Konstruktor trzeci otrzymuje jako argument obiekt klasy `Punkt3D` i kopiuje `z` niego wartości pól. Pozostałe metody klasy `Punkt3D` pozostają bez zmian, nie zostały one uwzględnione na listingu, aby niepotrzebnie nie powielać prezentowanego już kodu (są one natomiast uwzględnione na listingach dostępnych na serwerze FTP).

Jeśli przyjrzymy się dokładnie napisanym właśnie konstruktorom, zauważymy z pewnością, że w znacznej części ich kod dubluje się z kodem konstruktorów klasy `Punkt`. Dokładniej są to te same instrukcje, uzupełnione dodatkowo o instrukcje operujące na wartościach pola `z`. Spójrzmy, konstruktory:

```
Punkt3D(int wspX, int wspY, int wspZ)
{
    x = wspX;
    y = wspY;
    z = wspZ;
}
```

oraz:

```
Punkt(int wspX, int wspY)
{
    x = wspX;
    y = wspY;
}
```

są przecież prawie identyczne! Jedyna różnica to dodatkowy argument i dodatkowa instrukcja przypisująca jego wartość polu `z`. Czy nie lepiej byłoby zatem wykorzystać konstruktor klasy `Punkt` w klasie `Punkt3D` lub ogólniej — konstruktor klasy bazowej w konstruktorze klasy potomnej? Oczywiście, że tak. Nie można jednak wywołać konstruktora tak jak zwyczajnej metody — do tego celu służy specjalna konstrukcja ze słowem `base`, o której postaci:

```
class klasa_potomna : klasa_bazowa
{
    klasa_potomna(argumenty):base(argumenty)
    {
        /*
        ...kod konstruktora...
        */
    }
}
```

Zauważmy, że bardzo przypomina to opisaną wcześniej składnię ze słowem `this`. Różnica jest taka, że `this` służy do wywoływania konstruktorów w ramach jednej klasy, a `base` do wywoływania konstruktorów klasy bazowej. Jeśli zatem w klasie `Punkt` będą istniały konstruktory takie jak widoczne na listingu 3.36, to będzie można je wywoływać w klasie `Punkt3D` w sposób zaprezentowany na listingu 3.37.

Listing 3.36. Konstruktory w klasie `Punkt`

```
class Punkt
{
    public int x;
    public int y;

    public Punkt()
    {
        x = 1;
        y = 1;
    }
    public Punkt(int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
    }
    public Punkt(Punkt punkt)
    {
        x = punkt.x;
        y = punkt.y;
    }
    /*
    ...dalszy kod klasy Punkt...
    */
}
```

Listing 3.37. Wywołanie konstruktorów klasy bazowej

```
class Punkt3D : Punkt
{
    public int z;
    public Punkt3D():base()
    {
        z = 1;
    }
    public Punkt3D(int wspX, int wspY, int wspZ):base(wspX, wspY)
    {
        z = wspZ;
    }
    public Punkt3D(Punkt3D punkt):base(punkt)
    {
        z = punkt.z;
    }
    /*
    ...pozostale metody klasy Punkt3D...
    */
}
```

W pierwszym konstruktorze występuje ciąg `base()`, co powoduje wywołanie bezargumentowego konstruktora klasy bazowej. Taki konstruktor (bezargumentowy) istnieje w klasie `Punkt`, konstrukcja ta nie budzi więc żadnych wątpliwości. W konstruktorze drugim w nawiasie za `base` występują dwa argumenty typu `int` (`base(wspX, wspY)`). Ponieważ w klasie `Punkt` istnieje konstruktor dwuargumentowy, przyjmujący dwie wartości typu `int`, również i ta konstrukcja jest jasna — zostanie on wywołany i będą mu przekazane wartości `wspX` i `wspY` przekazane w wywołaniu konstruktora klasy `Punkt3D`.

Konstruktor trzeci przyjmuje jeden argument typu (klasy) `Punkt3D` i przekazuje go jako argument w wywołaniu `base (base(punkt))`. W klasie `Punkt` istnieje konstruktor przyjmujący jeden argument klasy... no właśnie, w klasie `Punkt` przecież wcale nie ma konstruktora, który przyjmowałby argument tego typu (`Punkt3D`)! Jest co prawda konstruktor:

```
Punkt(Punkt punkt)
{
    //instrukcje konstruktora
}
```

ale przecież przyjmuje on argument typu `Punkt`, a nie `Punkt3D`. Tymczasem klasa z listingu 3.37 skompiluje się bez żadnych problemów! Jak to możliwe? Przecież nie zgadzają się typy argumentów! Otóż okazuje się, że jeśli oczekujemy argumentu klasy `X`, a podany zostanie argument klasy `Y`, która jest klasą potomną dla `X`, błędu nie będzie. W takiej sytuacji nastąpi tak zwane rzutowanie typu obiektu, czym jednak zajmiemy się dokładniej dopiero w rozdziale 6. Na razie wystarczy zapamiętać zasadę: w miejscu, gdzie powinien być zastosowany obiekt pewnej klasy `X`, można zastosować również obiekt klasy potomnej dla `X`.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 17.1

Zmodyfikuj kod klasy `Punkt` z listingu 3.32 w taki sposób, aby nazwy parametrów w metodach `UstawX`, `UstawY` oraz `UstawXY` miały takie same nazwy jak nazwy pól, czyli `x` i `y`. Zatem nagłówki tych metod mają wyglądać następująco:

```
void UstawX(int x)
void UstawY(int y)
void UstawXY(int x, int y)
```

Ćwiczenie 17.2

Dopisz do klasy `Punkt3D` zaprezentowanej na listingu 3.34 metodę `UstawXZ` oraz `UstawYZ`.

Ćwiczenie 17.3

Napisz przykładową klasę `Program` wykorzystującą wszystkie trzy konstruktory klasy `Punkt3D` z listingu 3.35.

Ćwiczenie 17.4

Zmodyfikuj kod z listingu 3.35 w taki sposób, aby w żadnym z konstruktorów nie występowało bezpośrednie przypisanie wartości do pól klasy. Możesz użyć metody `UstawXYZ`.

Ćwiczenie 17.5

Napisz kod klasy `KolorowyPunkt` będącej rozszerzeniem klasy `Punkt` o informację o kolorze. Kolor ma być określany dodatkowym polem o nazwie `kolor` i typie `int`. Dopisz metody `UstawKolor` i `PobierzKolor`, a także odpowiednie konstruktory.

Ćwiczenie 17.6

Dopisz do klasy `Punkt3D` z listingu 3.37 konstruktor, który jako argument będzie przyjmował obiekt klasy `Punkt`. Wykorzystaj w tym konstruktorze wywołanie base.

Lekcja 18. Modyfikatory dostępu

Modyfikatory dostępu (nazywane również **specyfikatorami dostępu**, ang. *access modifiers*) pełnią ważną funkcję w C#, pozwalając bowiem na określenie praw dostępu do składowych klas, a także do samych klas. Występują one w kilku rodzajach, które zostaną przedstawione właśnie w lekcji 18.

Określanie reguł dostępu

W dotychczasowych naszych programach zarówno przed słowem `class`, jak i przed niektórymi składowymi, pojawiało się czasem słowo `public`. Jest to tak zwany modyfikator lub specyfikator dostępu i oznacza, że dana klasa jest publiczna, czyli że mogą z niej korzystać (mogą się do niej odwoływać) wszystkie inne klasy. Każda klasa, pole oraz metoda¹⁴ mogą być:

- ◆ publiczne (`public`),
- ◆ chronione (`protected`),
- ◆ wewnętrzne (`internal`),
- ◆ wewnętrzne chronione (`protected internal`),
- ◆ prywatne (`private`).

Typowa klasa, czyli o takiej postaci jak dotychczas stosowana, np.:

```
class Punkt  
{  
}
```

może być albo publiczna (`public`), albo wewnętrzna (`internal`)¹⁵. Domyślnie jest wewnętrzna, czyli dostęp do niej jest możliwy w obrębie jednego zestawu (lekcja 14.). Dopuszczalna jest zmiana sposobu dostępu na publiczny przez użycie słowa `public`:

```
public class Punkt  
{  
}
```

Użycie słowa `public` oznacza zniesienie wszelkich ograniczeń w dostępie do klasy (ale już nie do jej składowych, dostęp do składowych klasy definiuje się osobno). W tej fazie nauki różnice nie są jednak istotne, gdyż i tak korzystamy zawsze z jednego zestawu tworzącego konkretny program, a więc użycie bądź nieużycie słowa `public` przy klasie nie wywoła żadnych negatywnych konsekwencji.

W przypadku składowych klas obowiązują następujące zasady. Publiczne składowe określa się słowem `public`, co oznacza, że wszyscy mają do nich dostęp oraz że są dziedziczone przez klasy pochodne. Do składowych prywatnych (`private`) można dostać się tylko z wnętrza danej klasy, natomiast do składowych chronionych (`protected`) można uzyskać dostęp z wnętrza danej klasy oraz klas potomnych. Znaczenie tych specyfikatorów dostępu jest praktycznie takie samo jak w innych językach obiektowych, na przykład w Javie.

¹⁴ Dotyczy to także struktur, interfejsów, wyliczeń i delegacji. Te zagadnienia będą omawiane w dalszej części książki.

¹⁵ Stosowanie pozostałych modyfikatorów jest możliwe w przypadku klas wewnętrznych (zagnieżdżonych), które zostaną omówione w rozdziale 6.

W C# do dyspozycji są jednak dodatkowo specyfikatory `internal` i `protected internal`. Słowo `internal` oznacza, że dana składowa klasy będzie dostępna dla wszystkich klas z danego zestawu. Z kolei `protected internal`, jak łatwo się domyślić, jest kombinacją `protected` oraz `internal` i oznacza, że dostęp do składowej mają zarówno klasy potomne, jak i klasy z danego zestawu. Niemniej tymi dwoma specyfikatorami nie będziemy się zajmować, przyjrzymy się za to bliżej modyfikatorom `public`, `private` i `protected`.

Jeśli przed daną składową nie wystąpi żaden modyfikator, to będzie ona domyślnie prywatna. To właśnie dlatego w niektórych dotychczasowych przykładach poziom dostępu był zmieniany na publiczny, tak aby do składowych można się było odwoływać z innych klas.

Specyfikator dostępu należy umieścić przed nazwą typu, co schematycznie wygląda następująco:

specyfikator_dostępu nazwa_typu nazwa_pola

Podobnie jest z metodami — specyfikator dostępu powinien być pierwszym elementem deklaracji, czyli ogólnie napiszemy:

specyfikator_dostępu typ_zwracany nazwa_metody(argumenty)

Znaczenie modyfikatorów w przypadku określania reguł dostępu do całych klas jest podobne, z tym zastrzeżeniem, że modyfikatory `protected` i `private` mogą być stosowane tylko w przypadku klas zagnieżdżonych (lekcja 32.). Domyślnym poziomem dostępu (czyli gdy przed jej nazwą nie występuje żadne określenie reguł dostępu) do klasy jest `internal`.

Dostęp publiczny — `public`

Jeżeli dana składowa klasy jest publiczna, oznacza to, że mają do niej dostęp wszystkie inne klasy, czyli dostęp ten nie jest w żaden sposób ograniczony. Weźmy np. pierwotną wersję klasy `Punkt` z listingu 3.1 (lekcja 14.). Gdyby pola `x` i `y` tej klasy miały być publiczne, musiałaby ona wyglądać tak, jak na listingu 3.38.

Listing 3.38. Publiczne składowe klasy `Punkt`

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

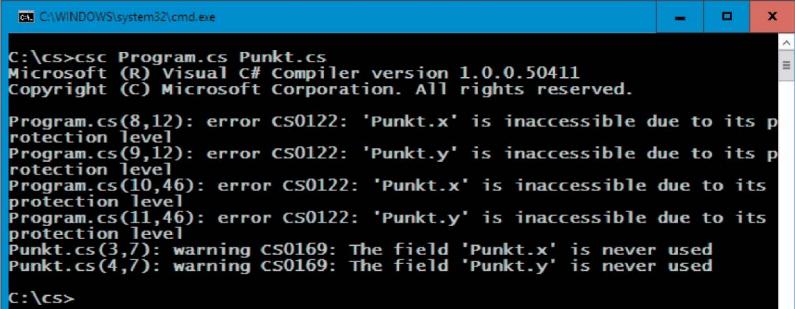
O tym, że poziom dostępu do pól tej klasy zmienił się, można się przekonać w prosty sposób. Użyjmy klasy `Program` z listingu 3.9 i klasy `Punkt` z listingu 3.1. Tworzony jest tam obiekt klasy `Punkt`, jego polom `x` i `y` są przypisywane wartości 100 i 200, a następnie są one odczytywane i wyświetlane na ekranie. Próba komilacji takiego zestawu klas się nie uda. Po wydaniu w wierszu poleceń komendy:

```
csc Program.cs Punkt.cs
```

zakończy się błędem kompilacji widocznym na rysunku 3.15. Nic w tym dziwnego, skoro domyślny poziom dostępu nie pozwala klasie Program na bezpośrednie odwoływanie się do składowych klasy Punkt (zgodnie z podanym wyżej opisem domyślnie składowe klasy są prywatne).

Rysunek 3.15.

Próba dostępu
do prywatnych
składowych kończy
się błędami
kompilacji



C:\>csc Program.cs Punkt.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

Program.cs(8,12): error CS0122: 'Punkt.x' is inaccessible due to its p
rotection level
Program.cs(9,12): error CS0122: 'Punkt.y' is inaccessible due to its p
rotection level
Program.cs(10,46): error CS0122: 'Punkt.x' is inaccessible due to its p
rotection level
Program.cs(11,46): error CS0122: 'Punkt.y' is inaccessible due to its p
rotection level
Punkt.cs(3,7): warning CS0169: The field 'Punkt.x' is never used
Punkt.cs(4,7): warning CS0169: The field 'Punkt.y' is never used

C:\>

Zupełnie inaczej będzie w przypadku tej samej klasy Program oraz klasy Punkt z listingu 3.38. Ponieważ w takim przypadku dostęp do pól x i y będzie publiczny, program uda się skompilować bez problemów.

Warto przy tym wspomnieć, że zaleca się, aby dostęp do pól klasy nie był publiczny, a ich odczyt i modyfikacja odbywały się poprzez odpowiednio zdefiniowane metody. Dlaczego tak jest, zostanie pokazane w dalszej części lekcji. Gdybyśmy chcieli dopisać do klasy Punkt z listingu 3.38 publiczne wersje metod PobierzX, PobierzY, UstawX i UstawY, przyjęłaby ona postać widoczną na listingu 3.39.

Listing 3.39. Publiczne pola i metody klasy Punkt

```
class Punkt  
{  
    public int x;  
    public int y;  
    public int PobierzX()  
    {  
        return x;  
    }  
    public int PobierzY()  
    {  
        return y;  
    }  
    public void UstawX(int wspX)  
    {  
        x = wspX;  
    }  
    public void UstawY(int wspY)  
    {  
        y = wspY;  
    }  
}
```

Gdyby natomiast klasa `Punkt` z listingu 3.38 miała być publiczna, to wyglądałaby tak jak na listingu 3.40. Z reguły główne klasy określane są jako publiczne, tak aby można było się do nich dowolnie odwoływać, natomiast klasy pomocnicze, usługowe wobec klasy głównej, określane są jako wewnętrzne (`internal`), tak aby dostęp do nich był jedynie z wnętrza danego zestawu.

Listing 3.40. Publiczna klasa `Punkt`

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

Dostęp prywatny — private

Składowe oznaczone słowem `private` to takie, które są dostępne jedynie z wnętrza danej klasy. To znaczy, że wszystkie metody danej klasy mogą je dowolnie odczytywać i zapisywać, natomiast dostęp z zewnątrz jest zabroniony zarówno dla zapisu, jak i odczytu. Jeżeli zatem w klasie `Punkt` z listingu 3.38 zechcemy jawnie ustawić oba pola jako prywatne, będzie ona miała postać widoczną na listingu 3.41.

Listing 3.41. Klasa `Punkt` z prywatnymi polami

```
class Punkt
{
    private int x;
    private int y;
}
```

O tym, że dostęp spoza klasy został zabroniony, przekonamy się, próbując dokonać komplikacji podobnej do tej w poprzednim podpunkcie, tzn. używając klasy `Program` z listingu 3.9 i klasy `Punkt` z listingu 3.41. Efekt będzie taki sam jak na rysunku 3.15. Tak więc do składowych prywatnych na pewno nie można się odwołać spoza klasy, w której zostały zdefiniowane. Ta uwaga dotyczy również klas potomnych.

W jaki zatem sposób odwołać się do pola prywatnego? Przypomnijmy opis prywatnej składowej klasy: jest to taka składowa, która jest dostępna z wnętrza danej klasy, czyli dostęp do niej mają wszystkie metody klasy. Wystarczy zatem, jeśli napiszemy publiczne metody pobierające i ustawiające pola prywatne, a będziemy mogli wykonywać na nich operacje. W przypadku klasy `Punkt` z listingu 3.41 niezbędne byłyby metody `UstawX`, `UstawY`, `PobierzX` i `PobierzY`. Klasa `Punkt` zawierająca prywatne pola `x` i `y` oraz wymienione metody o dostępie publicznym została przedstawiona na listingu 3.42.

Listing 3.42. Dostęp do prywatnych pól za pomocą metod publicznych

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

```
public int PobierzX()
{
    return x;
}
public int PobierzY()
{
    return y;
}
public void UstawX(int wspX)
{
    x = wspX;
}
public void UstawY(int wspY)
{
    y = wspY;
}
```

Takie metody pozwolą nam już na bezproblemowe odwoływanie się do obu prywatnych pól. Teraz program z listingu 3.9 trzeba by poprawić tak, aby wykorzystywał nowe metody, czyli zamiast:

```
punkt.x = 100;
```

napiszemy:

```
punkt.UstawX(100);
```

a zamiast:

```
Console.WriteLine("punkt.x = " + punkt.x);
```

napiszemy:

```
Console.WriteLine("punkt.x = " + punkt.PobierzX());
```

Podobne zmiany trzeba będzie wprowadzić w przypadku dostępu do pola y.

Dostęp chroniony — protected

Składowe klasy oznaczone słowem `protected` to składowe chronione. Są one dostępne jedynie dla metod danej klasy oraz klas potomnych. Oznacza to, że jeśli mamy przykładową klasę `Punkt`, w której znajdzie się chronione pole o nazwie `x`, to w klasie `Punkt3D`, o ile jest ona klasą pochodną od `Punkt`, również będziemy mogli odwoływać się do pola `x`. Jednak dla każdej innej klasy, która nie dziedziczy po `Punkt`, pole `x` będzie niedostępne. W praktyce klasa `Punkt` — z polami `x` i `y` zadeklarowanymi jako chronione — będzie wyglądała tak, jak na listingu 3.43.

Listing 3.43. Chronione pola w klasie `Punkt`

```
class Punkt
{
    protected int x;
    protected int y;
}
```

Jeśli teraz z klasy Punkt wyprowadzimy klasę Punkt3D w postaci widocznej na listingu 3.44, to będzie ona miała (odmiennie, niż byłoby to w przypadku składowych prywatnych) pełny dostęp do składowych x i y klasy Punkt.

Listing 3.44. Klasa dziedzicząca po Punkt

```
class Punkt3D : Punkt
{
    protected int z;
}
```

Dlaczego ukrywamy wnętrze klasy?

W tym miejscu pojawi się zapewne pytanie: dlaczego chcemy zabraniać bezpośredniego dostępu do niektórych składowych klas, stosując modyfikatory `private` i `protected`? Otóż chodzi o ukrycie implementacji wnętrza klasy. Programista, projektując daną klasę, udostępnia na zewnątrz (innym programistom) pewien interfejs służący do posługiwania się jej obiektami. Określa więc sposób, w jaki można korzystać z danego obiektu. To, co znajduje się we wnętrzu, jest ukryte; dzięki temu można całkowicie zmienić wewnętrzną konstrukcję klasy, nie zmieniając zupełnie sposobu korzystania z niej.

To, że takie podejście może nam się przydać, można pokazać nawet na przykładzie tak prostej klasy, jaką jest nieśmiertelna klasa Punkt. Założmy, że ma ona postać widoczną na listingu 3.42. Pola x i y są prywatne i zabezpieczone przed dostępem z zewnątrz, operacje na współrzędnych możemy wykonywać wyłącznie dzięki publicznym metodom: `PobierzX`, `PobierzY`, `UstawX`, `UstawY`. Program przedstawiony na listingu 3.45 będzie zatem działał poprawnie.

Listing 3.45. Program korzystający z klasy Punkt

```
using System;

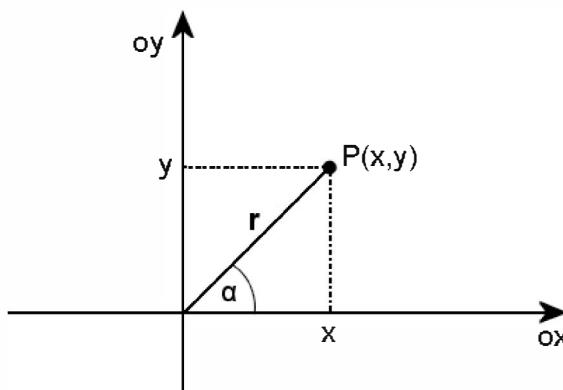
public class Program
{
    public static void Main()
    {
        Punkt punkt1 = new Punkt();
        punkt1.UstawX(100);
        punkt1.UstawY(200);
        Console.WriteLine("punkt1.x = " + punkt1.PobierzX());
        Console.WriteLine("punkt1.y = " + punkt1.PobierzY());
    }
}
```

Założmy teraz, że zostaliśmy zmuszeni (obojętnie, z jakiego powodu) do zmiany sposobu reprezentacji współrzędnych na tak zwany układ biegunowy, w którym położenie punktu jest opisywane za pomocą dwóch parametrów: kąta α oraz odległości od początku układu współrzędnych (rysunek 3.16). W klasie Punkt nie będzie już zatem pól x i y, przestaną mieć więc sens wszelkie odwołania do nich. Gdyby pola te były zadeklarowane jako publiczne, spowodowałoby to spory problem. Nie dość, że we

wszystkich programach wykorzystujących klasę `Punkt` trzeba by zmieniać odwołania, to dodatkowo należałoby w każdym takim miejscu dokonywać przeliczania współrzędnych. Wymagałoby to wykonania ogromnej pracy, a ponadto pojawiłoby się w ten sposób sporo możliwości powstania niepotrzebnych błędów.

Rysunek 3.16.

Położenie punktu reprezentowane za pomocą współrzędnych biegunkowych



Jednak dzięki temu, że pola `x` i `y` są prywatne, a dostęp do nich odbywa się przez publiczne metody, wystarczy, że tylko odpowiednio zmienimy te metody. Jak się za chwilę okaże, można całkowicie tę klasę przebudować, a korzystający z niej program z listingu 3.45 nie będzie wymagał nawet najmniejszej poprawki!

Najpierw trzeba zamienić pola `x` i `y` typu `int` na pola reprezentujące kąt i odległość. Kąt najlepiej reprezentować za pomocą jego funkcji trygonometrycznej — wybierzemy np. sinus. Nowe pola nazwiemy więc `sinusalfa` oraz `r` (będzie reprezentowało odległość punktu od początku układu współrzędnych). Zatem podstawowa wersja nowej klasy `Punkt` będzie miała postać:

```
public class Punkt
{
    private double sinusalfa;
    private double r;
}
```

Dopisać należy teraz wszystkie cztery metody pierwotnie operujące na polach `x` i `y`. Aby to zrobić, musimy znać wzory przekształcające wartości współrzędnych kartezjańskich (tzn. współrzędne (x, y)) na układ biegunowy (czyli kąt i moduł) oraz wzory odwrotne, czyli przekształcające współrzędne biegunowe na kartezjańskie. Wyrowadzenie tych wzorów nie jest skomplikowane, wystarczy znajomość podstawowych funkcji trygonometrycznych oraz twierdzenia Pitagorasa. Jednak książka ta to kurs programowania, a nie lekcja matematyki, wzory zostaną więc przedstawione już w gotowej postaci¹⁶. I tak (dla oznaczeń jak na rysunku 3.16):

¹⁶ W celu uniknięcia umieszczenia w kodzie klasy dodatkowych instrukcji warunkowych, zaciemniających sedno zagadnienia, przedstawiony kod i wzory są poprawne dla dodatnich współrzędnych `x`. Uzupełnienie klasy `Punkt` w taki sposób, aby możliwe było także korzystanie z ujemnych wartości `x`, można potraktować jako ćwiczenie do samodzielnego wykonania.

$$x = r \times \sqrt{1 - \sin^2(\alpha)}$$

$$y = r \times \sin(\alpha)$$

oraz:

$$r = \sqrt{x^2 + y^2}$$

$$\sin(\alpha) = \frac{y}{r}$$

Mając te dane, możemy przystąpić do napisania odpowiednich metod. Zaczniemy od metody `PobierzY`. Jej postać będzie następująca:

```
public int PobierzY()
{
    double y = r * sinusalfa;
    return (int) y;
}
```

Deklarujemy zmiennej pomocniczą `y` typu `double` i przypisujemy jej wynik mnożenia wartości pól `r` oraz `sinusalfa` — zgodnie z podanymi wyżej wzorami. Ponieważ metoda ma zwrócić wartość `int`, a wynikiem obliczeń jest wartość `double`, przed zwróceniem wyniku dokonujemy konwersji na typ `int`. Odpowiada za to konstrukcja `(int) y`¹⁷. (W tej instrukcji jest wykonywane tzw. rzutowanie typu; temat ten zostanie jednak omówiony dokładnie dopiero w lekcji 27., w rozdziale 6.). W analogiczny sposób napiszemy metodę `PobierzX`, choć będziemy musieli oczywiście wykonać nieco więcej obliczeń. Metoda ta wygląda następująco:

```
public int PobierzX()
{
    double x = r * Math.Sqrt(1 - sinusalfa * sinusalfa);
    return (int) x;
}
```

Tym razem deklarujemy, podobnie jak w poprzednim przypadku, pomocniczą zmieniąną `x` typu `double` oraz przypisujemy jej wynik działania: `r * Math.Sqrt(1 - sinusalfa * sinusalfa)`. `Math.Sqrt` — to standardowa metoda obliczająca pierwiastek kwadratowy z przekazanego jej argumentu (czyli np. wykonanie instrukcji `Math.sqrt(4)` da w wyniku 2) — wykorzystywaliśmy ją już w programach rozwiązujących równania kwadratowe. W tym przypadku ten argument to `1 - sinusalfa * sinusalfa`, czyli `1 - sinusalfa2`, zgodnie z podanym wzorem na współrzędną `x`. Wykonujemy mnożenie zamiast potęgowania, gdyż jest ono po prostu szybsze i wygodniejsze.

¹⁷ Nie jest to sposób w pełni poprawny, gdyż pozbywamy się zupełnie części ułamkowej, zamiast wykonać prawidłowe zaokrąglenie, a w związku z tym w wynikach mogą się pojawić pewne nieścisłości. Żeby jednak nie zaciemniać przedstawianego zagadnienia dodatkowymi instrukcjami, musimy się z tą drobną niedogodnością pogodzić.

Pozostały jeszcze do napisania metody `UstawX` i `UstawY`. Pierwsza z nich będzie mieć następującą postać:

```
public void UstawX(int wspX)
{
    int x = wspX;
    int y = PobierzY();

    r = Math.Sqrt(x * x + y * y);
    sinusalfa = y / r;
}
```

Ponieważ zarówno parametr `r`, jak i `sinusalfa` zależą od obu współrzędnych, trzeba je najpierw uzyskać. Współrzędna `x` jest oczywiście przekazywana jako argument, natomiast `y` uzyskujemy, wywołując napisaną przed chwilą metodę `PobierzY`. Dalsza część metody `UstawX` to wykonanie działań zgodnych z podanymi wzorami¹⁸. Podobnie jak w przypadku `PobierzY`, zamiast potęgowania wykonujemy zwykłe mnożenie `x * x` i `y * y`. Metoda `UstawY` przyjmie prawie identyczną postać, z tą różnicą, że skoro będzie jej przekazywana wartość współrzędnej `y`, to musimy uzyskać jedynie wartość `x`, czyli początkowe instrukcje będą następujące:

```
int x = PobierzX();
int y = wspY;
```

Kiedy złożymy wszystkie napisane do tej pory elementy w jedną całość, uzyskamy klasę `Punkt` w postaci widocznej na listingu 3.46 (na początku została dodana dyrektywa `using`, tak aby można było swobodnie odwoływać się do klasy `Math` zdefiniowanej w przestrzeni nazw `System`). Jeśli teraz uruchomimy program z listingu 3.45, przekonamy się, że wynik jego działania z nową klasą `Punkt` jest taki sam jak w przypadku jej poprzedniej postaci¹⁹. Mimo całkowitej wymiany wnętrza klasy `Punkt` program zadziała tak, jakby nic się nie zmieniło.

Listing 3.46. Nowa wersja klasy `Punkt`

```
using System;

class Punkt
{
    private double sinusalfa;
    private double r;

    public int PobierzX()
    {
        double x = r * Math.Sqrt(1 - sinusalfa * sinusalfa);
        return (int) x;
    }
}
```

¹⁸ Jak można zauważyć, taki kod nie będzie działał poprawnie dla punktu o współrzędnych (0,0). Niezbędne byłoby wprowadzenie dodatkowych instrukcji warunkowych.

¹⁹ Uwaga! W praktyce należałoby dopisać również dwuargumentowy konstruktor. Inaczej, ze względu na konstrukcję kodu, jeżeli choć jedna współrzędna będzie zerem, druga zostanie automatycznie wyzerowana (uzyskamy wtedy zawsze punkt o współrzędnych 0,0).

```
public int PobierzY()
{
    double y = r * sinusalfa;
    return (int) y;
}

public void UstawX(int wspX)
{
    int x = wspX;
    int y = PobierzY();

    r = Math.Sqrt(x * x + y * y);
    sinusalfa = y / r;
}

public void UstawY(int wspY)
{
    int x = PobierzX();
    int y = wspY;

    r = Math.Sqrt(x * x + y * y);
    sinusalfa = y / r;
}
```

Jak zatrzymać dziedziczenia?

W praktyce programistycznej można spotkać się z sytuacjami, kiedy konieczne jest zatrzymanie dziedziczenia. Innymi słowy będziemy chcieli spowodować, aby z naszej klasy nie można było wyprowadzać klas potomnych. Służy do tego słowo kluczowe `sealed`, które należy umieścić przed nazwą klasy zgodnie ze schematem:

```
specyfikator_dostępu sealed class nazwa_klasy
{
    //składowe klasy
}
```

Nie ma przy tym formalnego znaczenia to, czy słowo `sealed` będzie przed, czy za specyfikatorem dostępu, czyli czy napiszemy np. `public sealed class`, czy `sealed public class`, niemniej dla przejrzystości i ujednolicenia notacji najlepiej konsekwentnie stosować tylko jeden przedstawionych sposobów.

Przykładowa klasa `Wartosc` tego typu została przedstawiona na listingu 3.47.

Listing 3.47. Zastosowanie modyfikatora `sealed`

```
public sealed class Wartosc
{
    public int liczba;
    public void Wyswietl()
    {
        System.Console.WriteLine(liczba);
    }
}
```

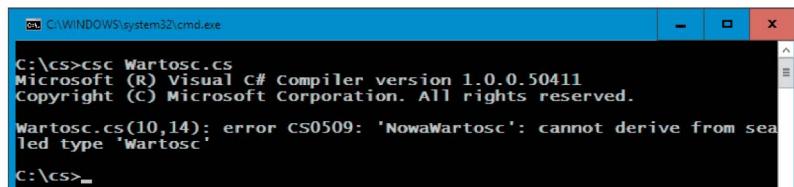
Z takiej klasy nie można wyprowadzić żadnej innej. Tak więc przedstawiona na listingu 3.48 klasa NowaWartosc dziedzicząca po Wartosc jest niepoprawna. Kompilator C# nie dopuści do komplikacji takiego kodu i zgłosi komunikat o błędzie zaprezentowany na rysunku 3.17.

Listing 3.48. Niepoprawne dziedziczenie

```
public class NowaWartosc : Wartosc
{
    public int drugaLiczba;
    /*
    ... dalsze składowe klasy ...
    */
}
```

Rysunek 3.17.

Próba nieprawidłowego dziedziczenia kończy się błędem kompilacji



The screenshot shows a command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command entered is 'C:\cs>csc Wartosc.cs'. The output shows the Microsoft (R) Visual C# Compiler version 1.0.0.50411 and Copyright (C) Microsoft Corporation. All rights reserved. Below this, an error message is displayed: 'Wartosc.cs(10,14): error CS0509: 'NowaWartosc': cannot derive from sealed type 'Wartosc''. The prompt then returns to 'C:\cs>-'.

Tylko do odczytu

W C# można zadeklarować pole klasy jako tylko do odczytu, co oznacza, że przypisanej mu wartości nie można będzie zmieniać. Takie pola oznacza się modyfikatorem `readonly`, który musi wystąpić przed nazwą typu, schematycznie:

specyfikator_dostępu readonly typ_pola nazwa_pola;

lub

`readonly specyfikator_dostępu typ_pola nazwa_pola;`

Tak więc poprawne będą poniższe przykładowe deklaracje:

```
readonly int liczba;
readonly public int liczba;
public readonly int liczba;
```

Wartość takiego pola może być przypisana albo w momencie deklaracji, albo też w konstruktorze klasy i nie może być później zmieniana.

Pola `readonly` typów prostych

Przykładowa klasa zawierająca pola tylko do odczytu została przedstawiona na listingu 3.49.

Listing 3.49. Klasa zawierająca pola tylko do odczytu

```
public class Wartosci
{
    public readonly int liczba1 = 100;
    public readonly int liczba2;
    public int liczba3;
    public Wartosci()
    {
        //prawidłowo inicjalizacja pola liczba2
        liczba2 = 200;

        //prawidłowo można zmienić wartość pola w konstruktorze
        liczba1 = 150;
    }
    public void Obliczenia()
    {
        //prawidłowo odczyt pola liczba1, zapis pola liczba3
        liczba3 = 2 * liczba1;

        //prawidłowo odczyt pól liczba1 i liczba2, zapis pola liczba3
        liczba3 = liczba2 + liczba1;

        //nieprawidłowo niedozwolony zapis pola liczba1
        //liczba1 = liczba2 / 2;

        System.Console.WriteLine(liczba1);
        System.Console.WriteLine(liczba2);
        System.Console.WriteLine(liczba3);
    }
    public static void Main()
    {
        Wartosci w = new Wartosci();
        w.Obliczenia();
    }
}
```

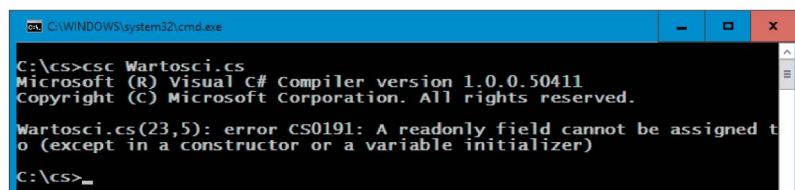
Zostały tu zadeklarowane trzy pola, liczba1, liczba2 i liczba3, wszystkie publiczne o typie int. Dwa pierwsze są również polami tylko do odczytu, a zatem przypisanych im wartości nie wolno modyfikować poza konstruktorem. W klasie znalazł się również konstruktor oraz metoda Obliczenia, która wykonuje działania, wykorzystując wartości przypisane zadeklarowanym polom. W konstruktorze polu liczba2 została przypisana wartość 200, a polu liczba1 wartość 150. Oba przypisania są prawidłowe, mimo że liczba1 miało już ustaloną wcześniej wartość. W konstruktorze można bowiem przypisać nową wartość polu tylko do odczytu i jest to jedyne miejsce, w którym taka operacja jest prawidłowa.

W metodzie Obliczenia najpierw zmiennej liczba3 przypisujemy wynik mnożenia 2 * liczba1. Jest to poprawna instrukcja, gdyż wolno odczytywać wartość pola tylko do odczytu liczba1 oraz przypisywać wartości zwykłemu polu liczba3. Podobną sytuację mamy w przypadku drugiego działania. Trzecia instrukcja przypisania została ujęta w komentarz, gdyż jest nieprawidłowa i spowodowałaby błąd komplikacji wiadoczny na rysunku 3.18. Występuje tu bowiem próba przyporządkowania wyniku

działania `liczba2 / 2` polu `liczba1`, w stosunku do którego został użyty modyfikator `readonly`. Takiej operacji nie wolno wykonywać, zatem po usunięciu znaków komentarza z tej instrukcji kompilator zaprotestuje. Do klasy `Wartosci` dopisana została też metoda `Main` (por. lekcja 15.), w której tworzymy nowy obiekt klasy `Wartosci` i wywołujemy jego metodę `Oblizenia`.

Rysunek 3.18.

Próba przypisania wartości zmiennej typu `readonly`



```
C:\cs>csc Wartosci.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

Wartosci.cs(23,5): error CS0191: A readonly field cannot be assigned to
(except in a constructor or a variable initializer)

C:\cs>-
```

Pola `readonly` typów odnośnikowych

Zachowanie pól z modyfikatorem `readonly` w przypadku typów prostych jest jasne — nie wolno zmieniać ich wartości. To znaczy wartość przypisana polu pozostaje niezmienna przez cały czas działania programu. W przypadku typów odnośnikowych jest oczywiście tak samo, trzeba jednak dobrze uświadomić sobie, co to wówczas oznacza. Otóż pisząc:

```
nazwa_klasy nazwa_pola = new nazwa_klasy(argumenty_konstruktora)
```

polu `nazwa_pola` przypisujemy referencję do nowo powstałego obiektu klasy `nazwa_klasy`. Przykładowo w przypadku klasy `Punkt`, którą przedstawiono na początku rozdziału, deklaracja:

```
Punkt punkt = new Punkt()
```

oznacza przypisanie zmiennej `punkt` referencji do powstałego na stercie obiektu klasy `Punkt` (lekcja 14.).

Gdyby pole to było typu `readonly`, tej referencji nie byłoby wolno zmieniać, jednak nic nie stałoby na przeszkodzie, aby modyfikować pola obiektu, na który ta referencja wskazuje. Czyli po wykonaniu instrukcji:

```
readonly Punkt punkt = new Punkt();
```

możliwe byłoby odwołanie w postaci (zakładając publiczny dostęp do pola `x`):

```
punkt.x = 100;
```

Aby lepiej to zrozumieć, spójrzmy na kod przedstawiony na listingu 3.50.

Listing 3.50. Odwołania do pól typu `readonly`

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

public class Program
{
```

```
public readonly Punkt punkt = new Punkt();
public void UzyjPunktu()
{
    //prawidłowo, można modyfikować pola obiektu punkt
    punkt.x = 100;
    punkt.y = 200;

    //nieprawidłowo, nie można zmieniać referencji typu readonly
    //punkt = new Punkt();
}
```

Są tu widoczne dwie publiczne klasy: Program i Punkt. Klasa Punkt zawiera dwa publiczne pola typu int o nazwach x i y. Klasa Program zawiera jedno publiczne pole tylko do odczytu o nazwie Punkt, któremu została przypisana referencja do obiektu klasy Punkt. Ponieważ pole jest publiczne, mają do niego dostęp wszystkie inne klasy; ponieważ jest typu readonly, nie wolno zmieniać jego wartości. Ale uwaga: zgodnie z tym, co zostało napisane we wcześniejszych akapitach, nie wolno zmienić referencji, ale nic nie stoi na przeszkodzie, aby modyfikować pola obiektu, na który ona wskazuje. Dlatego też pierwsze dwa odwołania w metodzie UzyjPunktu są poprawne. Wolno przypisać dowolne wartości polom x i y obiektu wskazywanego przez pole punkt. Nie wolno natomiast zmieniać samej referencji, zatem ujęta w komentarz instrukcja punkt = new Punkt() jest nieprawidłowa.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 18.1

Zmień kod z listingu 3.9 tak, aby poprawnie współpracował z klasą Punkt z listingu 3.42.

Ćwiczenie 18.2

Zmodyfikuj kod z listingu 3.46 tak, aby dawał prawidłowe wyniki również dla ujemnych współrzędnych x oraz by poprawnie obsługiwany był punkt o współrzędnych (0,0). Nie zmieniaj zastosowanych wzorów.

Ćwiczenie 18.3

Dopisz do klasy Punkt z ćwiczenia 18.2 konstruktor przyjmujący współrzędne x i y, tak aby poprawnie obsługiwana była również i taka sytuacja, gdy tylko jedna współrzędna jest równa 0. Przetestuj otrzymany kod.

Ćwiczenie 18.4

Napisz kod klasy realizującej zadanie odwrotne do przykładu z listingu 3.46. Dane wewnętrzne powinny być przechowywane w postaci pól x i y, natomiast metody powinny obsługiwać dane w układzie biegunowym (pobierzR, ustawR, pobierzSinusalfa, ustawSinusalfa).

Lekcja 19. Przesłanianie metod i składowe statyczne

W lekcji 15. został wyjaśniony termin przeciążania metod; teraz będzie wyjaśnione, co się dzieje ze składowymi klasy (w tym metodami, ale również polami), gdy w grę wchodzi dziedziczenie — zostanie przybliżona technika tzw. przesłaniania pól i metod. Technika ta pozwala na bardzo ciekawy efekt umieszczenia składowych o identycznych nazwach zarówno w klasie bazowej, jak i potomnej. Drugim poruszonym tematem będą z kolei składowe statyczne, czyli takie, które mogą istnieć nawet wtedy, kiedy nie istnieją obiekty danej klasy.

Przesłanianie metod

Zastanówmy się, co się stanie, kiedy w klasie potomnej ponownie zdefiniujemy metodę o takiej samej nazwie i takich samych argumentach jak w klasie bazowej. Albo inaczej: jakiego zachowania metod mamy się spodziewać w przypadku klas przedstawionych na listingu 3.51.

Listing 3.51. Przesłanianie metod

```
public class A
{
    public void f()
    {
        System.Console.WriteLine("Metoda f z klasy A.");
    }
}

public class B : A
{
    public void f()
    {
        System.Console.WriteLine("Metoda f z klasy B.");
    }
}
```

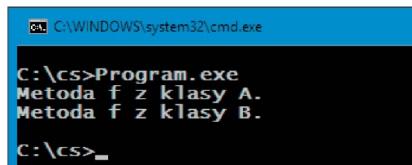
W klasie A znajduje się bezargumentowa metoda o nazwie `f`, wyświetlająca na ekranie informację o nazwie klasy, w której została zdefiniowana. Klasa B dziedziczy po klasie A, zgodnie z zasadami dziedziczenia przejmuje więc metodę `f` z klasy A. Tymczasem w klasie B została ponownie zadeklarowana bezargumentowa metoda `f` (również wyświetlająca nazwę klasy, w której została zdefiniowana, czyli tym razem klasy B). Wydawać by się mogło, że w takim wypadku wystąpi konflikt nazw (dwukrotne zadeklarowanie metody `f`). Jednak próba komplikacji wykaże, że kompilator nie zgłasza żadnych błędów — pojawi się jedynie ostrzeżenie (o tym za chwilę). Dlaczego konflikt nazw nie występuje? Otóż zasada jest następująca: jeśli w klasie bazowej i pochodnej występuje metoda o tej samej nazwie i argumentach, metoda z klasy bazowej jest przesłaniana (przykrywana, ang. *override*) i mamy do czynienia z tzw. **przesłanianiem metod** (ang. *methods overriding*). A zatem w obiektach klasy bazowej będzie obo-

wiązywała metoda z klasy bazowej, a w obiektach klasy pochodnej — metoda z klasy pochodnej.

Sprawdzimy to. Co pojawi się na ekranie po uruchomieniu klasy `Program` z listingu 3.52, która korzysta z obiektów klas `A` i `B` z listingu 3.51? Oczywiście najpierw tekst Metoda f z klasy A., a następnie tekst Metoda f z klasy B. (rysunek 3.19). Skoro bowiem obiekt `a` jest klasy `A`, to wywołanie `a.f()` powoduje wywołanie metody `f` z klasy `A`. Z kolei obiekt `b` jest klasy `B`, zatem wywołanie `b.f()` powoduje uruchomienie metody `f` z klasy `B`.

Rysunek 3.19.

Efekt wywołania
przesłoniętej
metody



Listing 3.52. Użycie obiektów klas A i B

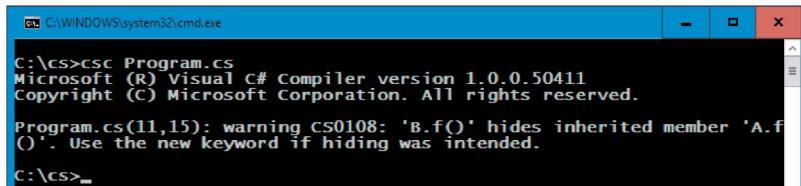
```
public class Program
{
    public static void Main()
    {
        A a = new A();
        B b = new B();

        a.f();
        b.f();
    }
}
```

Wróćmy teraz do ostrzeżenia wygenerowanego przez kompilator przy komplikacji współpracujących ze sobą klas z listingów 3.51 i 3.52. Jest ono widoczne na rysunku 3.20. Otóż kompilator oczywiście wykrył istnienie metody o takiej samej deklaracji w klasach bazowej (`A`) i potomnej (`B`) i poinformował nas o tym. Formalnie należy bowiem określić sposób zachowania takich metod. Zostanie to dokładniej wyjaśnione w rozdziale 6., omawiającym zaawansowane zagadnienia programowania obiektowego.

Rysunek 3.20.

Ostrzeżenie
generowane
przez kompilator



Na razie przyjmijmy, że w prezentowanej sytuacji, czyli wtedy, gdy w klasie potomnej ma zostać zdefiniowana nowa metoda o takiej samej nazwie, argumentach i typie zwracanym, do jej deklaracji należy użyć słowa kluczowego `new`, które umieszcza się przed typem zwracanym. To właśnie sugeruje komunikat kompilatora z rysunku 3.20.

Tak więc schematyczna deklaracja takiej metody powinna mieć postać:

```
specyfikator_dostępu new typ_zwracany nazwa_metody(argumenty)
{
    //wewnętrze metody
}
```

lub:

```
new specyfikator_dostępu typ_zwracany nazwa_metody(argumenty)
{
    //wewnętrze metody
}
```

W naszym przypadku klasa B powinna więc wyglądać tak jak na listingu 3.53.

Listing 3.53. Użycie modyfikatora new

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

Może się w tym miejscu pojawić pytanie, czy jest w takim razie możliwe wywołanie przesłoniętej metody z klasy bazowej. Jeśli pytanie to brzmi zbyt zawile, to — na przykładzie klas z listingu 3.51 — chodzi o to, czy w klasie B można wywołać metodę f z klasy A. Nie jest to zagadnienie czysto teoretyczne, gdyż w praktyce programistycznej takie odwołania często upraszczają kod i ułatwiają tworzenie spójnych hierarchii klas. Skoro tak, to odwołanie takie oczywiście jest możliwe. Jak pamiętamy z lekcji 17., jeśli trzeba było wywołać konstruktor klasy bazowej, używało się słowa base. W tym przypadku jest podobnie. Odwołanie do przesłoniętej metody klasy bazowej uzyskujemy dzięki wywołaniu w schematycznej postaci:

```
base.nazwa_metody(argumenty);
```

Wywołanie takie najczęściej stosuje się w metodzie przesłaniającej (np. metodzie f klasy B), ale możliwe jest ono również w dowolnej innej metodzie klasy pochodnej. Gdyby więc metoda f klasy B z listingu 3.51 miała wywoływać metodę klasy bazowej, kod klasy B powinien przyjąć postać widoczną na listingu 3.54.

Listing 3.54. Wywołanie przesłoniętej metody z klasy bazowej

```
public class B : A
{
    public new void f()
    {
        base.f();
        System.Console.WriteLine("Metoda f z klasy B.");
    }
}
```

Przesłanianie pól

Pola klas bazowych są przesłaniane w sposób analogiczny do metod. Jeśli więc w klasie pochodnej zdefiniujemy pole o takiej samej nazwie jak w klasie bazowej, bezpośrednio dostępne będzie tylko to z klasy pochodnej. Przy deklaracji należy użyć modyfikatora new. Taka sytuacja jest zobrazowana na listingu 3.55.

Listing 3.55. Przesłonięte pola

```
public class A
{
    public int liczba;
}

public class B : A
{
    public new int liczba;
}
```

W klasie A zostało zdefiniowane pole o nazwie `liczba` i typie `int`. W klasie B, która dziedziczy po A, ponownie zostało zadeklarowane pole o takiej samej nazwie i typie. Trzeba sobie jednak dobrze uświadomić, że każdy obiekt klasy B będzie w takiej sytuacji zawierał dwa pola o nazwie `liczba` — jedno pochodzące z klasy A, drugie z B. Co więcej, tym polom można przypisywać różne wartości. Zilustrowano to w programie widocznym na listingu 3.56.

Listing 3.56. Odwołania do przesloniętych pól

```
using System;

public class Program
{
    public static void Main()
    {
        B b = new B();
        b.liczba = 10;
        ((A)b).liczba = 20;

        Console.WriteLine("Wartość pola liczba z klasy B: ");
        Console.WriteLine(b.liczba);
        Console.WriteLine("Wartość pola liczba odziedziczonego po klasie A: ");
        Console.WriteLine(((A)b).liczba);
    }
}
```

Tworzymy najpierw obiekt `b` klasy B, odbywa się to w standardowy sposób. Podobnie pierwsza instrukcja przypisania ma dobrze nam już znaną postać:

```
b.liczba = 10;
```

W ten sposób ustalona została wartość pola `liczba` zdefiniowanego w klasie B. Dzieje się tak dlatego, że to pole przesłania (przykrywa) pole o tej samej nazwie, pochodzące z klasy A. Klasyczne odwołanie powoduje więc dostęp do pola zdefiniowanego w klasie,

która jest typem obiektu (w tym przypadku obiekt `b` jest typu `B`). W obiekcie `b` istnieje jednak również drugie pole o nazwie `liczba`, odziedziczone po klasie `A`. Do niego również istnieje możliwość dostępu. W tym celu jest wykorzystywana tak zwana technika rzutowania, która zostanie zaprezentowana w dalszej części książki. Na razie przyjmijmy jedynie, że konstrukcja:

`((A)b)`

oznacza: „Potraktuj obiekt klasy `B` tak, jakby był obiektem klasy `A`”. Tak więc odwołanie:

`((A)b).liczba = 20;`

to nic innego jak przypisanie wartości 20 polu `liczba` pochodząemu z klasy `A`.

O tym, że obiekt `b` faktycznie przechowuje dwie różne wartości, przekonujemy się, wyświetlając je na ekranie za pomocą metody `WriteLine` z klasy `Console`. Po komplikacji i uruchomieniu programu zobaczymy widok taki jak przedstawiony na rysunku 3.21.

Rysunek 3.21.

Odwołania do dwóch pól zawartych w obiekcie klasy `B`

```
C:\Windows\system32\cmd.exe
C:\cs>Program.exe
Wartość pola liczba z klasy B: 10
Wartość pola liczba odziedziczonego po klasie A: 20
C:\cs>_
```

Składowe statyczne

Składowe statyczne (ang. *static members*) to takie, które istnieją nawet wtedy, gdy nie istnieje żaden obiekt danej klasy. Każda taka składowa jest wspólna dla wszystkich obiektów klasy. Składowe te są oznaczane słowem `static`. W dotychczasowych przykładach wykorzystywaliśmy jedną metodę tego typu — `Main`, od której rozpoczyna się wykonywanie programu.

Metody statyczne

Metodę statyczną oznaczamy słowem `static`, które powinno znaleźć się przed typem zwracanym. Zwyczajowo umieszcza się je zaraz za specyfikatorem dostępu²⁰, czyli schematycznie deklaracja metody statycznej będzie wyglądała następująco:

```
specyfikator_dostępu static typ_zwracany nazwa_metody(argumenty)
{
    //treść metody
}
```

Przykładowa klasa z zadeklarowaną metodą statyczną może wyglądać tak, jak zostało to przedstawione na listingu 3.57.

²⁰ W rzeczywistości słowo kluczowe `static` może pojawić się również przed specyfikatorem dostępu, ta kolejność nie jest bowiem istotna z punktu widzenia kompilatora. Przyjmuje się jednak, że — ze względu na ujednolicenie notacji — o ile występuje specyfikator dostępu metody, słowo `static` powinno znaleźć się za nim; na przykład: `public static void main`, a nie `static public void main`.

Listing 3.57. Klasa zawierająca metodę statyczną

```
public class A
{
    public static void f()
    {
        System.Console.WriteLine("Metoda f klasy A");
    }
}
```

Tak napisaną metodę można wywołać tylko przez zastosowanie konstrukcji o ogólnej postaci:

nazwa_klasy.nazwa_metody(argumenty_metody);

W przypadku klasy A wywołanie tego typu miałoby następującą postać:

A.f();

Nie można natomiast zastosować odwołania poprzez obiekt, a więc instrukcje:

```
A a = new A();
a.f();
```

są nieprawidłowe i spowodują błąd komplikacji.

Na listingu 3.58 jest przedstawiona przykładowa klasa Program, która korzysta z takiego wywołania. Uruchomienie tego kodu pozwoli przekonać się, że faktycznie w przypadku metody statycznej nie trzeba tworzyć obiektu.

Listing 3.58. Wywołanie metody statycznej

```
public class Program
{
    public static void Main()
    {
        A.f();
    }
}
```

Dlatego też metoda Main, od której rozpoczyna się wykonywanie kodu programu, jest metodą statyczną, może bowiem zostać wykonana, mimo że w trakcie uruchamiania aplikacji nie powstały jeszcze żadne obiekty.

Musimy jednak zdawać sobie sprawę, że metoda statyczna jest umieszczana w specjalnie zarezerwowanym do tego celu obszarze pamięci i jeśli powstaną obiekty danej klasy, to będzie ona dla nich wspólna. To znaczy, że dla każdego obiektu klasy nie tworzy się kopii metody statycznej.

Statyczne pola

Do pól oznaczonych jako statyczne można się odwoływać podobnie jak w przypadku statycznych metod, czyli nawet wtedy, gdy nie istnieje żaden obiekt danej klasy. Pola takie deklaruje się, umieszczając przed typem słowo static. Schematycznie deklaracja taka wygląda następująco:

```
static typ_pola nazwa_pola;
```

lub:

```
specyfikator_dostępu static typ_pola nazwa_pola;
```

Jeśli zatem w naszej przykładowej klasie A ma się pojawić statyczne pole o nazwie liczba typu int o dostępie publicznym, klasa taka będzie miała postać widoczną na listingu 3.59²¹.

Listing 3.59. Umieszczenie w klasie pola statycznego

```
public class A
{
    public static int liczba;
}
```

Do pól statycznych nie można odwołać się w sposób klasyczny, tak jak do innych pól klasy — poprzedzając je nazwą obiektu (oczywiście, jeśli wcześniej utworzymy dany obiekt). W celu zapisu lub odczytu należy zastosować konstrukcję:

```
nazwa_klasy.nazwa_pola
```

Podobnie jak metody statyczne, również i pola tego typu znajdują się w wyznaczonym obszarze pamięci i są wspólne dla wszystkich obiektów danej klasy. Tak więc niezależnie od liczby obiektów danej klasy pole statyczne o danej nazwie będzie tylko jedno. Przypisanie i odczytanie zawartości pola statycznego klasy A z listingu 3.59 może zostać zrealizowane w sposób przedstawiony na listingu 3.60.

Listing 3.60. Użycie pola statycznego

```
public class Program
{
    public static void Main()
    {
        A.liczba = 100;
        System.Console.WriteLine("Pole liczba klasy A ma wartość {0}.",
            A.liczba);
    }
}
```

Odwołanie do pola statycznego może też mieć miejsce wewnątrz klasy. Nie trzeba wtedy stosować przedstawionej konstrukcji — przecież pole to jest częścią klasy. Dlatego też do klasy A można by dopisać przykładową metodę f o postaci:

```
public void f(int wartosc)
{
    liczba = wartosc;
}
```

której zadaniem jest zmiana wartości pola statycznego.

²¹ Podobnie jak w przypadku metod statycznych, z formalnego punktu widzenia słowo static może się znaleźć przed specyfikatorem dostępu, czyli na przykład: static public int liczba. Jednak dla ujednolicenia notacji oraz zachowania zwyczajowej konwencji zapisu będzie konsekwentnie stosowana forma zaprezentowana w powyższym akapicie, czyli: public static int liczba.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 19.1

Napisz klasę `Punkt` przechowującą współrzędne punktów na płaszczyźnie oraz klasę `Punkt3D` przechowującą współrzędne punktów w przestrzeni trójwymiarowej. W obu przypadkach przygotuj metodę `odleglosc`, której zadaniem będzie zwrócenie odległości punktu od początku układu współrzędnych.

Ćwiczenie 19.2

Napisz klasę `Punkt3D` dziedziczącą po klasie `Punkt` zaprezentowanej na listingu 3.8. Umieść w niej pole typu `int` o nazwie `z`. W obu klasach zdefiniuj publiczną metodę `WyswietlWspolrzedne` wyświetlającą wartości współrzędnych na ekranie. W metodzie `WyswietlWspolrzedne` z klasy `Punkt3D` nie używaj odwołań do pól `x` i `y`.

Ćwiczenie 19.3

Napisz klasę `Dodawanie`, która będzie zawierała statyczną metodę `Dodaj` przyjmującą dwa argumenty typu `int`. Metoda ta powinna zwrócić wartość będącą wynikiem dodawania obu argumentów.

Ćwiczenie 19.4

Napisz klasę `Przechowalnia`, która będzie zawierała statyczną metodę o nazwie `Przechowaj` przyjmującą jeden argument typu `int`. Klasa ta ma zapamiętywać argument przekazany metodzie `Przechowaj` w taki sposób, że każde wywołanie tej metody spowoduje zwrócenie poprzednio zapisanej wartości i zapamiętanie aktualnie przekazanej.

Ćwiczenie 19.5

Napisz kod przykładowej klasy (o dowolnej nazwie) i umieść w niej statyczną prywatną metodę `Wyswietl`, wyświetlającą na ekranie dowolny napis. Przygotuj też osobną klasę `Program` i spraw, aby metoda `Wyswietl` została wywołana, tak aby efekt jej działania pojawił się na ekranie.

Lekcja 20. Właściwości i struktury

Lekcja 20. poświęcona jest dwóm różnym zagadnieniom — właściwościom oraz strukturom. Zostanie w niej pokazane, czym są te konstrukcje programistyczne oraz jak i kiedy się nimi posługiwać. Nie będą też pominięte informacje o tym, czym są tzw. akcesory `get` i `set` oraz jak tworzyć właściwości tylko do zapisu lub tylko do odczytu.

Właściwości

Struktura właściwości

Opisanymi dotychczas składowymi klas były pola i metody. W C# uważa się, że pola z reguły powinny być prywatne, a dostęp do nich realizowany za pomocą innych konstrukcji, np. metod. To dlatego we wcześniejszych przykładach, np. w klasie `Punkt`, stosowane były metody takie jak `UstawX` czy `PobierzY`. Istnieje jednak jeszcze jeden, i to bardzo wygodny, sposób dostępu, jakim są właściwości (ang. *properties*). Otóż właściwość (ang. *property*) to jakby połączenie możliwości, jakie dają pola i metody. Dostęp bowiem wygląda tak samo jak w przypadku pól, ale w rzeczywistości wykonywane są specjalne metody dostępowe zwane **akcesorami** (ang. *accessors*). Ogólny schemat takiej konstrukcji jest następujący:

```
[modyfikator_dostępu] typ_właściwości nazwa_właściwości
{
    get
    {
        //instrukcje wykonywane podczas pobierania wartości
    }
    set
    {
        //instrukcje wykonywane podczas ustawiania wartości
    }
}
```

Akcesory `get` (tzw. getter) i `set` (tzw. setter) są przy tym niezależne od siebie. Akcesor `get` powinien w wyniku swojego działania zwracać (za pomocą instrukcji `return`) wartość takiego typu, jakiego jest właściwość, natomiast `set` otrzymuje przypisywaną mu wartość w postaci argumentu o nazwie `value`.

Założymy więc, że w klasie `Kontener` umieściliśmy prywatne pole o nazwie `_wartosc` i typie `int`. Do takiego pola, jak już wiadomo z lekcji 18., nie można się bezpośrednio odwoływać spoza klasy. Do jego odczytu i zapisu można więc użyć albo metod, albo właśnie właściwości. Jak to zrobić, zobrazowano w przykładzie widocznym na listingu 3.61.

Listing 3.61. Użycie prostej właściwości

```
public class Kontener
{
    private int _wartosc;
    public int wartosc
    {
        get
        {
            return _wartosc;
        }
        set
        {
            _wartosc = value;
        }
    }
}
```

Klasa zawiera prywatne pole `_wartosc` oraz publiczną właściwość `wartosc`. Wewnątrz definicji właściwości znalazły się akcesory `get` i `set`. Oba mają bardzo prostą konstrukcję: `get` za pomocą instrukcji `return` zwraca po prostu wartość zapisaną w polu `_wartosc`, natomiast `set` ustawia wartość tego pola za pomocą prostej instrukcji przypisania. Słowo `value` oznacza tutaj wartość przekazaną akcesorowi w instrukcji przypisania. Zobaczmy, jak będzie wyglądało wykorzystanie obiektu typu `Kontener` w działającym programie. Jest on widoczny na listingu 3.62.

Listing 3.62. Użycie klasy `Kontener`

```
using System;

public class Program
{
    public static void Main()
    {
        Kontener obj = new Kontener();
        obj.wartosc = 100;
        Console.WriteLine(obj.wartosc);
    }
}
```

W metodzie `Main` jest tworzony i przypisywany zmiennej `obj` nowy obiekt klasy `Kontener`. Następnie właściwości `wartosc` tego obiektu jest przypisywana wartość 100. Jak widać, odbywa się to dokładnie w taki sam sposób jak w przypadku pól. Odwołanie do właściwości następuje za pomocą operatora oznaczanego symbolem kropki, a przypisanie — za pomocą operatora `=`. Jednak wykonanie instrukcji:

```
obj.wartosc = 100;
```

oznacza w rzeczywistości przekazanie wartości 100 akcesorowi `set` związanemu z właściwością `wartosc`. Wartość ta jest dostępna wewnątrz akcesora poprzez słowo `value`. Tym samym wymieniona instrukcja powoduje zapamiętanie w obiekcie wartości 100. Przekonujemy się o tym, odczytując zawartość właściwości w trzeciej instrukcji metody `Main` i wyświetlając ją na ekranie. Oczywiście odczytanie właściwości to nic innego jak wywołanie akcesora `get`.

Właściwości a sprawdzanie poprawności danych

Właściwości doskonale nadają się do sprawdzania poprawności danych przypisywanych prywatnym polom. Założny, że mamy do czynienia z klasą o nazwie `Data` zawierającą pole typu `byte` określające dzień tygodnia, takie że 1 to niedziela, 2 — poniedziałek itd. Jeśli dostęp do tego pola będzie się odbywał przez właściwość, to łatwo będzie można sprawdzać, czy aby na pewno przypisywana mu wartość nie przekracza dopuszczalnego zakresu 1 – 7. Napiszmy więc treść takiej klasy; jest ona widoczna na listingu 3.63.

Listing 3.63. Sprawdzanie poprawności przypisywanych danych

```
public class Data
{
    private byte _dzień;
    public byte DzieńTygodnia
```

```
{  
    get  
    {  
        return _dzien;  
    }  
    set  
    {  
        if(value > 0 && value < 8)  
        {  
            _dzien = value;  
        }  
    }  
}
```

Ogólna struktura klasy jest podobna do tej zaprezentowanej na listingu 3.61 i omówionej w poprzednim podpunkcie. Inaczej wygląda jedynie akcesor set, w którym znalazła się instrukcja warunkowa if. Bada ona, czy wartość value (czyli ta przekazana podczas operacji przypisania) jest większa od 0 i mniejsza od 8, czyli czy zawiera się w przedziale 1 – 7. Jeśli tak, jest przypisywana polu _dzien, a więc przechowywana w obiekcie; jeśli nie, nie dzieje się nic. Spróbujmy więc zobaczyć, jak w praktyce zachowa się obiekt takiej klasy przy przypisywaniu różnych wartości właściwości DzienTygodnia. Odpowiedni przykład jest widoczny na listingu 3.64.

Listing 3.64. Użycie klasy Data

```
using System;  
  
public class Program  
{  
    public static void Main()  
    {  
        Data pierwszaData = new Data();  
        Data drugaData = new Data();  
        pierwszaData.DzienTygodnia = 8;  
        drugaData.DzienTygodnia = 2;  
  
        Console.WriteLine("\n--- po pierwszym przypisaniu ---");  
        Console.Write("1. numer dnia tygodnia to ");  
        Console.WriteLine("{0}.". pierwszaData.DzienTygodnia);  
        Console.Write("2. numer dnia tygodnia to ");  
        Console.WriteLine("{0}.". drugaData.DzienTygodnia);  
  
        drugaData.DzienTygodnia = 9;  
        Console.WriteLine("\n--- po drugim przypisaniu ---");  
        Console.Write("2. numer dnia tygodnia to ");  
        Console.WriteLine("{0}.". drugaData.DzienTygodnia);  
    }  
}
```

Najpierw tworzone są dwa obiekty typu Data. Pierwszy z nich jest przypisywany zmiennej pierwszaData, a drugi zmiennej drugaData. Następnie właściwości DzienTygodnia obiektu pierwszaData jest przypisywana wartość 8, a tej samej właściwości obiektu drugaData — wartość 2. Jak już wiadomo, pierwsza z tych operacji nie może

zostać poprawnie wykonana, gdyż dzień tygodnia musi zawierać się w przedziale 1 – 7. W związku z tym wartość właściwości (oraz związanego z nią pola `_dzień`) pozostanie niezmieniona, a więc będzie to wartość przypisywana niezainicjowanym polom typu `byte`, czyli 0. W drugim przypadku operacja przypisania może zostać wykonana, a więc wartością właściwości `DzienTygodnia` obiektu `drugaData` będzie 2.

O tym, że oba przypisania działają zgodnie z powyższym opisem, przekonujemy się, wyświetlając wartości właściwości obu obiektów za pomocą instrukcji `Console.WriteLine`. Później wykonujemy jednak kolejne przypisanie, o postaci:

```
drugaData.DzienTygodnia = 9;
```

Ono oczywiście również nie może zostać poprawnie wykonane, więc instrukcja ta nie zmieni stanu obiektu `drugaData`. Sprawdzamy to, ponownie odczytując i wyświetlając wartość właściwości `DzienTygodnia` tego obiektu. Ostatecznie po komplikacji i uruchomieniu na ekranie zobaczymy widok zaprezentowany na rysunku 3.22.

Rysunek 3.22.

Wynik testowania właściwości `DzienTygodnia`

```
C:\>Program.exe
--- po pierwszym przypisaniu ---
1. numer dnia tygodnia to 0.
2. numer dnia tygodnia to 2.

--- po drugim przypisaniu ---
2. numer dnia tygodnia to 2.

C:\><input>
```

Sygnalizacja błędów

Przykład z poprzedniego podpunktu pokazywał, w jaki sposób sprawdzać poprawność danych przypisywanych właściwości. Nie uwzględniał on jednak sygnalizacji błędnych danych. W przypadku zwykłej metody ustawiającej wartość pola informacja o błędzie mogłaby być zwracana jako rezultat działania. W przypadku właściwości takiej możliwości jednak nie ma. Akcesor nie może przecież zwracać żadnej wartości. Można jednak w tym celu wykorzystać technikę tzw. wyjątków. Wyjątki zostaną omówione dopiero w kolejnym rozdziale, a zatem Czytelnicy nieobeznani z tą tematyką powinni pominąć ten punkt i powrócić dopiero po zapoznaniu się z materiałem przedstawionym w lekcjach z rozdziału 4.

Poprawienie kodu z listingu 3.63 w taki sposób, aby w przypadku wykrycia przekroczenia dopuszczalnego zakresu danych był generowany wyjątek, nie jest skomplikowane. Kod realizujący takie zadanie został przedstawiony na listingu 3.65.

Listing 3.65. Sygnalizacja błędu za pomocą wyjątku

```
using System;

public class ValueOutOfRangeException : Exception
{
}
```

```
public class Data
{
    private byte _dzien;
    public byte DzienTygodnia
    {
        get
        {
            return _dzien;
        }
        set
        {
            if(value > 0 && value < 8)
            {
                _dzien = value;
            }
            else
            {
                throw new ValueOutOfRangeException();
            }
        }
    }
}
```

Na początku została dodana klasa wyjątku `ValueOutOfRangeException` dziedzicząca bezpośrednio po `Exception`. Jest to nasz własny wyjątek, który będzie zgłoszany po ustaleniu, że wartość przekazana akcesorowi `set` jest poza dopuszczalnym zakresem. Treść klasy `Data` nie wymagała wielkich zmian. Instrukcja `if` akcesora `set` została zmieniona na instrukcję warunkową `if..else`. W bloku `else`, wykonywanym, kiedy wartość wskazywana przez `value` jest mniejsza od 1 lub większa od 7, za pomocą instrukcji `throw` zgłoszany jest wyjątek typu `ValueOutOfRangeException`. Obiekt wyjątku tworzony jest za pomocą operatora `new`. W jaki sposób można obsłużyć błąd zgłoszany przez tę wersję klasy `Data`, zobrazowano w programie widocznym na listingu 3.66.

Listing 3.66. Obsługa błędu zgłoszonego przez akcesor `set`

```
using System;

public class Program
{
    public static void Main()
    {
        Data pierwszaData = new Data();
        try
        {
            pierwszaData.DzienTygodnia = 8;
        }
        catch(ValueOutOfRangeException)
        {
            Console.WriteLine("Wartość poza zakresem.");
        }
    }
}
```

Utworzenie obiektu jest realizowane w taki sam sposób jak w poprzednich przykładach, natomiast instrukcja przypisująca wartość 8 właściwości `DzienTygodnia` została ujęta w blok `try`. Dzięki temu, jeśli ta instrukcja spowoduje zgłoszenie wyjątku, zostaną wykonane instrukcje znajdujące się w bloku `catch`. Oczywiście w tym przypadku mamy pewność, że wyjątek zostanie zgłoszony, wartość 8 przekracza bowiem dopuszczalny zakres. Dlatego też po uruchomieniu programu na ekranie ukaże się napis `Wartość poza zakresem..`

Właściwości tylko do odczytu

We wszystkich dotychczasowych przykładach właściwości miały przypisane akcesory `get` i `set`. Nie jest to jednak obligatoryjne. Otóż jeśli pominiemy `set`, to otrzymamy właściwość tylko do odczytu. Próba przypisania jej jakiekolwiek wartości skończy się błędem komilacji. Przykład obrazujący to zagadnienie jest widoczny na listingu 3.67.

Listing 3.67. Właściwość tylko do odczytu

```
using System;

public class Dane
{
    private string _nazwa = "Klasa Dane";
    public string nazwa
    {
        get
        {
            return _nazwa;
        }
    }
}

public class Program
{
    public static void Main()
    {
        Dane dane1 = new Dane();
        string napis = dane1.nazwa;
        Console.WriteLine(napis);
        //dane1.nazwa = "Klasa Data";
    }
}
```

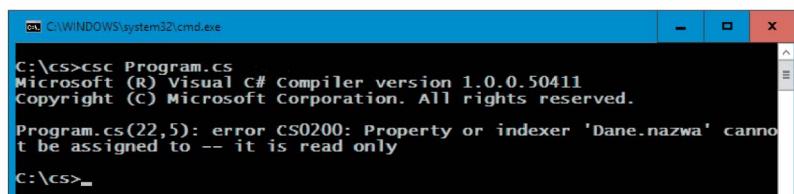
Klasa `Dane` ma jedno prywatne pole typu `string`, któremu został przypisany łańcuch znaków `Klasa Dane`. Oprócz pola znajduje się w niej również właściwość `nazwa`, w której został zdefiniowany jedynie akcesor `get`, a jego zadaniem jest zwrócenie zawartości pola `_nazwa`. Akcesora `set` po prostu nie ma, co oznacza, że właściwość można jedynie odczytywać. W klasie `Program` został utworzony nowy obiekt typu `Dane`, a następnie została odczytana jego właściwość `nazwa`. Odczytana wartość została przypisana zmiennej `napis` i wyświetlona na ekranie za pomocą instrukcji `Console.WriteLine`. Te wszystkie operacje niewątpliwie są prawidłowe, natomiast oznaczona komentarzem:

```
dane1.nazwa = "Klasa Data";
```

— już nie. Ponieważ nie został zdefiniowany akcesor set, nie można przypisywać żadnych wartości właściwości nazwa. Dlatego też po usunięciu komentarza i próbie komilacji zostanie zgłoszony błąd widoczny na rysunku 3.23.

Rysunek 3.23.

Próba przypisania
wartości
właściwości tylko
do odczytu kończy
się błędem
kompilacji



```
C:\cs>csc Program.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

Program.cs(22,5): error CS0200: Property or indexer 'Dane.nazwa' cannot be assigned to -- it is read only

C:\cs>_
```

Właściwości tylko do zapisu

Skoro, jak zostało to opisane w poprzedniej części lekcji, usunięcie akcesora set sprawiało, że właściwość można było tylko odczytywać, logika podpowiada, że usunięcie akcesora get spowoduje, iż właściwość będzie mogła tylko zapisywać. Taka możliwość jest rzadziej wykorzystywana, niemniej istnieje. Jak utworzyć właściwość tylko do zapisu, zobrazowano na listingu 3.68.

Listing 3.68. Właściwość tylko do zapisu

```
using System;

public class Dane
{
    private string _nazwa = "";
    public string nazwa
    {
        set
        {
            _nazwa = value;
        }
    }
}

public class Program
{
    public static void Main()
    {
        Dane dane1 = new Dane();
        dane1.nazwa = "Klasa Dane";
        //string napis = dane1.nazwa;
    }
}
```

Klasa Dane zawiera teraz takie samo pole jak w przypadku przykładu z listingu 3.67, zmienił się natomiast akcesor właściwości nazwa. Tym razem zamiast get jest set. Skoro nie ma get, oznacza to, że właściwość będzie mogła być tylko zapisywana. Tak też dzieje się w metodzie Main w klasie Program. Po utworzeniu obiektu typu Dane i przypisaniu odniesienia do niego zmiennej dane1 jest przypisywany właściwości nazwa ciąg znaków Klasa Dane. Taka instrukcja zostanie wykonana prawidłowo. Inaczej jest w przypadku ujętej w komentarz instrukcji:

```
string napis = dane1.nazwa;
```

Nie może być ona poprawnie wykonana, właściwość nazwa jest bowiem właściwością tylko do zapisu. W związku z tym usunięcie komentarza spowoduje błąd komplikacji widoczny na rysunku 3.24.

Rysunek 3.24.

Błąd związany
z próbą odczytania
właściwości tylko
do zapisu

Właściwości niezwiązane z polami

W dotychczasowych przykładach z tego rozdziału właściwości były powiązane z prywatnymi polami klasy i pośredniczyły w zapisie i odczytce ich wartości. Nie jest to jednak obligatoryjne; właściwości mogą być całkowicie niezależne od pól. Można sobie wyobrazić różne sytuacje, kiedy zapis czy odczyt właściwości powoduje dużo bardziej złożoną reakcję niż tylko przypisanie wartości jakiemuś polu; mogą to być np. operacje na bazach danych czy plikach. Te zagadnienia wykraczają poza ramy niniejszej publikacji, można jednak wykonać jeszcze jeden prosty przykład, który pokaże właściwość tylko do odczytu zawsze zwracającą taką samą wartość. Jest on widoczny na listingu 3.69.

Listing 3.69. Właściwość niezwiązana z polem

```
using System;

public class Dane
{
    public string nazwa
    {
        get
        {
            return "Klasa Dane";
        }
    }
}

public class Program
{
    public static void Main()
    {
        Dane dane1 = new Dane();
        Console.WriteLine(dane1.nazwa);
        Console.WriteLine(dane1.nazwa);
    }
}
```

Klasa Dane zawiera wyłącznie właściwość nazwa, nie ma w niej żadnego pola. Istnieje także tylko jeden akcesor, którym jest get. Z każdym wywołaniem zwraca on wartość typu string, którą jest ciąg znaków Klasa Dane. Ten ciąg jest niezmienny. W metodzie

Main klasy Program został utworzony nowy obiekt typu Dane, a wartość jego właściwości nazwa została dwukrotnie wyświetlona na ekranie za pomocą instrukcji Console.
→WriteLine. Oczywiście, ponieważ wartość zdefiniowana w get jest niezmienna, każdy odczyt właściwości nazwa będzie dawał ten sam wynik.

Właściwości implementowane automatycznie

Jeżeli właściwość ma być tylko interfejsem zapisu i odczytu prywatnego pola klasy i nie wymaga żadnej złożonej logiki przy wykonywaniu tych operacji, może być implementowana automatycznie. Jeśli mamy na przykład klasę Kontener w wersji z listingu 3.61 i nie będziemy jej rozbudowywać (np. o sprawdzanie poprawności przekazywanych danych jak miało to miejsce w kolejnych podanych wyżej), kod można znaczco uproszczyć. W takim wypadku stosuje się konstrukcję o ogólnej postaci:

```
[modyfikator_dostępu] typ_właściwości nazwa_właściwości {get; set;}
```

Cały kod klasy mógłby wyglądać więc tak jak na listingu 3.70.

Listing 3.70. Uproszczony kod właściwości

```
public class Kontener
{
    public int wartosc {get; set;}
}
```

Będzie to dokładny funkcjonalny odpowiednik kodu z listingu 3.61 i będzie bez problemów współpracował np. z klasą Program z listingu 3.62 (jak i z każdą inną). Komplator po prostu sam zadba zarówno o umieszczenie w klasie prywatnego pola powiązanego z właściwością, jak i odpowiednią implementację i settera.

Często zdarza się jednak, że właściwość musi mieć jakąś wartość początkową. Inaczej mówiąc — musi zostać zainicjalizowana. Przykładowo właściwość wartosc z klasy Kontener ma mieć początkową wartość 1 (obecnie jest to 0). W C# do wersji 5. wyłącznie trzeba było w tym celu napisać konstruktor i to w nim dokonać odpowiedniego przypisania. Kod klasy Kontener z listingu 3.70 musiałby więc wyglądać tak jak na listingu 3.71.

Listing 3.71. Konstruktor inicjalizujący właściwość

```
public class Kontener
{
    public Kontener()
    {
        wartosc = 1;
    }
    public int wartosc {get; set;}
}
```

W C# 6.0 pojawiła się natomiast nowa możliwość, tzw. automatyczna inicjalizacja właściwości. W tej wersji języka, i nowszych, przypisanie wartości początkowej może odbywać się tuż za akcesorem set (przy czym dotyczy to tylko opisywanych w tym

podpunkcie właściwości implementowanych automatycznie). Schematycznie wygląda to tak:

```
[modyfikator_dostępu] typ_właściwości nazwa_właściwości {get; set;} = wartość;
```

W C# 6.0 i nowszych klasa Kontener z listingu 3.71 mogłaby zatem być pozbawiona konstruktora i wyglądać tak jak zaprezentowano to na listingu 3.72. Działanie obu wersji będzie takie samo.

Listing 3.72. Inicjalizacja właściwości bezpośrednio w akcesorze set

```
public class Kontener
{
    public int wartosc {get; set;} = 1;
}
```

Struktury

Tworzenie struktur

W C# oprócz klas mamy do dyspozycji również struktury. Składnia obu tych konstrukcji programistycznych jest podobna, choć zachowują się one inaczej. Struktury najlepiej sprawują się przy reprezentacji niewielkich obiektów zawierających po kilka pól i ewentualnie niewielką liczbę innych składowych (metod, właściwości itp.). Ogólna definicja struktury jest następująca:

```
[modyfikator_dostępu] struct nazwa_struktury
{
    //składowe struktury
}
```

Składowe struktury definiuje się tak samo jak składowe klasy. Gdybyśmy na przykład chcieli utworzyć strukturę o nazwie Punkt przechowującą całkowite współrzędne x i y punktów na płaszczyźnie, powinniśmy zastosować konstrukcję przedstawioną na listingu 3.73.

Listing 3.73. Prosta struktura

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

Jak skorzystać z takiej struktury? Tu właśnie ujawni się pierwsza różnica między klasą a strukturą. Otóż ta druga jest traktowana jak typ wartościowy (taki jak int, byte itp.), co oznacza, że po pierwsze, nie ma konieczności jawnego tworzenia obiektu, a po drugie, obiekty będące strukturami są tworzone na stosie, a nie na stercie. Tak więc zmieniąca przechowującą strukturę zawiera sam obiekt struktury, a nie jak w przypadku typów klasowych — referencję. Spójrzmy zatem na listingu 3.74. Zawiera on prosty program korzystający ze struktury Punkt z listingu 3.73.

Listing 3.74. Użycie struktury Punkt

```
using System;

public class Program
{
    public static void Main()
    {
        Punkt punkt;
        punkt.x = 100;
        punkt.y = 200;
        Console.WriteLine("punkt.x = {0}", punkt.x);
        Console.WriteLine("punkt.y = {0}", punkt.y);
    }
}
```

W metodzie `Main` w klasie `Program` została utworzona zmienna `punkt` typu `Punkt`. Jest to równoznaczne z powstaniem instancji tej struktury, obiektu typu `Punkt`. Zwróćmy uwagę, że nie został użyty operator `new`, a więc zachowanie jest podobne jak w przypadku typów prostych. Kiedy pisaliśmy np.:

```
int liczba;
```

od razu powstawała gotowa do użycia zmienna `liczba`. O tym, że faktycznie tak samo jest w przypadku struktur, przekonujemy się, przypisując polom `x` i `y` wartości 100 i 200, a następnie wyświetlając je na ekranie za pomocą instrukcji `Console.WriteLine`.

Nie oznacza to jednak, że do tworzenia struktur nie można użyć operatora `new`. Otóż instrukcja w postaci:

```
Punkt punkt = new Punkt;
```

również jest prawidłowa. Trzeba jednak wiedzieć, że nie oznacza to tego samego. Otóż jeśli stosujemy konstrukcję o schematycznej postaci:

```
nazwa_struktury zmienna;
```

polu struktury pozostają niezainicjowane i dopóki nie zostaną zainicjowane, nie można z nich korzystać. Jeśli natomiast użyjemy konstrukcji o postaci:

```
nazwa_struktury zmienna = new nazwa_struktury();
```

to zostanie wywołany konstruktor domyślny (bezargumentowy, utworzony przez kompilator) i wszystkie pola zostaną zainicjowane wartościami domyślnymi dla danego typu (tabela 3.1 z lekcji 16.). Te różnice zostały zobrazowane w przykładzie z listingu 3.75.

Listing 3.75. Różne sposoby tworzenia struktur

```
using System;

public class Program
{
    public static void Main()
    {
        Punkt punkt1 = new Punkt();
        Punkt punkt2;
```

```

punkt1.x = 100;
punkt2.x = 100;

Console.WriteLine("punkt1.x = {0}", punkt1.x);
Console.WriteLine("punkt1.y = {0}", punkt1.y);

Console.WriteLine("punkt2.x = {0}", punkt2.x);
//Console.WriteLine("punkt2.y = {0}", punkt2.y);
}
}

```

Powstały tu dwie zmienne, a więc i struktury typu Punkt: punkt1 i punkt2. Pierwsza z nich została utworzona za pomocą operatora new, a druga tak jak zwykła zmieniona typu prostego. W związku z tym ich zachowanie będzie nieco inne. Po utworzeniu struktur zostały zainicjowane ich pola x, w obu przypadkach przypisano wartość 100. Następnie za pomocą dwóch instrukcji Console.WriteLine na ekranie zostały wyświetlane wartości pól x i y struktury punkt1. Te operacje są prawidłowe. Ponieważ do utworzenia struktury punkt1 został użyty operator new, został też wywołany konstruktor domyślny, a pola otrzymały wartość początkową równą 0. Niezmienione w dalszej części kodu pole y będzie więc miało wartość 0, która może być bez problemu odczytana.

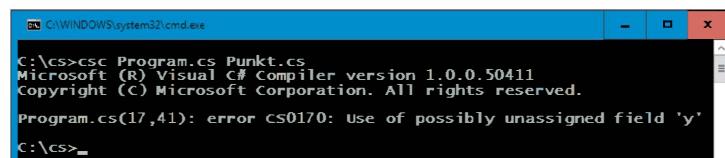
Inaczej jest w przypadku drugiej zmiennej. O ile polu x została przypisana wartość i instrukcja:

```
Console.WriteLine("punkt2.x = {0}", punkt2.x);
```

może zostać wykonana, to pole y pozostało niezainicjowane i nie można go odczytywać. W związku z tym instrukcja ujęta w komentarz jest nieprawidłowa, a próba jej wykonania spowodowałaby błąd komplikacji przedstawiony na rysunku 3.25.

Rysunek 3.25.

Próba odwołania do niezainicjowanego pola struktury



Konstruktory i inicjalizacja pól

Składowe struktur nie mogą być inicjalizowane w trakcie deklaracji²². Przypisanie wartości może odbywać się albo w konstruktorze, albo po utworzeniu struktury przez zwykłe operacje przypisania. Oznacza to, że przykładowy kod widoczny na listingu 3.76 jest nieprawidłowy i spowoduje błąd komplikacji.

Listing 3.76. Nieprawidłowa inicjalizacja pól struktury

```

public struct Punkt
{
    public int x = 100;
    public int y = 200;
}

```

²² Chyba że zostały zadeklarowane jako stałe (const) lub statyczne (static).

Struktury mogą zawierać konstruktory, z tym zastrzeżeniem, że nie można definiować domyślnego konstruktora bezargumentowego. Taki konstruktor jest tworzony automatycznie przez kompilator i nie może być redefiniowany. Jeśli chcielibyśmy wyposażyć strukturę Punkt w dwuargumentowy konstruktor ustawiający wartości pól x i y, powinniśmy zastosować kod widoczny na listingu 3.77.

Listing 3.77. Konstruktor struktury Punkt

```
public struct Punkt
{
    public int x;
    public int y;
    public Punkt(int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
    }
}
```

Użycie takiego konstruktora mogłoby wyglądać na przykład następująco:

```
Punkt punkt1 = new Punkt(100, 200);
```

Należy też zwrócić uwagę, że inaczej niż w przypadku klas wprowadzenie konstruktora przyjmującego argumenty nie powoduje pominięcia przez kompilator bezargumentowego konstruktora domyślnego. Jak zostało wspomniane wcześniej, do struktur konstruktor domyślny jest dodawany zawsze. Tak więc używając wersji struktury Punkt widocznej na listingu 3.77, nadal można tworzyć zmienne za pomocą konstrukcji typu:

```
Punkt punkt2 = new Punkt();
```

Struktury a dziedziczenie

Struktury nie podlegają dziedziczeniu względem klas i struktur. Oznacza to, że struktura nie może dziedziczyć ani po klasie, ani po innej strukturze, a także że klasa nie może dziedziczyć po strukturze. Struktury mogą natomiast dziedziczyć po interfejsach (czy też dokładniej: mogą implementować interfejsy). Temat interfejsów zostanie omówiony dopiero w rozdziale 6., tam też został opublikowany kod interfejsu IPunkt, który jest wykorzystany w poniższym przykładzie. Tak więc Czytelnicy, którzy nie mieli do tej pory do czynienia z tymi konstrukcjami programistycznymi, mogą na razie pominąć tę część lekcji.

Dziedziczenie struktury po interfejsie wygląda tak samo jak w przypadku klas. Stosowana jest konstrukcja o ogólnej postaci:

```
[modyfikator_dostępu] struct nazwa_struktury : nazwa_interfejsu
{
    //wewnętrze struktury
}
```

Gdyby więc miała powstać struktura Punkt dziedzicząca po interfejsie IPunkt (implementująca interfejs IPunkt; rozdział 6., lekcja 30., listing 6.24), to mogłaby ona przyjąć postać widoczną na listingu 3.78.

Listing 3.78. Dziedziczenie po interfejsie

```
public struct Punkt : IPunkt
{
    private int _x;
    private int _y;
    public int x
    {
        get
        {
            return _x;
        }
        set
        {
            _x = value;
        }
    }
    public int y
    {
        get
        {
            return _y;
        }
        set
        {
            _y = value;
        }
    }
}
```

W interfejsie IPunkt zdefiniowane zostały dwie publiczne właściwości: x i y, obie z akcesorami get i set. W związku z tym takie elementy muszą się też pojawić w strukturze. Wartości x i y muszą być jednak gdzieś przechowywane, dlatego struktura zawiera również prywatne pola _x i _y (dopuszczalne byłoby także użycie struktur implementowanych automatycznie). Budowa akcesorów jest tu bardzo prosta. Akcesor get zwraca w przypadku właściwości x — wartość pola _x, a w przypadku właściwości y — wartość pola _y. Zadanie akcesora set jest oczywiście odwrotne, w przypadku właściwości x ustawia on pole _x, a w przypadku właściwości y — pole _y.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 20.1

Napisz kod klasy Punkt zawierającej właściwości x i y oraz klasy Punkt3D dziedziczącej po Punkt, zawierającej właściwość z.

Ćwiczenie 20.2

Napisz kod klasy Punkt zawierającej właściwości x i y. Dane o współrzędnych x i y mają być przechowywane w tablicy liczb typu int.

Ćwiczenie 20.3

Napisz kod klasy zawierającej właściwość `liczba` typu rzeczywistego. Kod powinien działać w taki sposób, aby przypisanie wartości właściwości `liczba` powodowało zapisanie jedynie połowy przypisywanej liczby, a odczyt powodował zwrócenie po-dwojonej zapisanej wartości.

Ćwiczenie 20.4

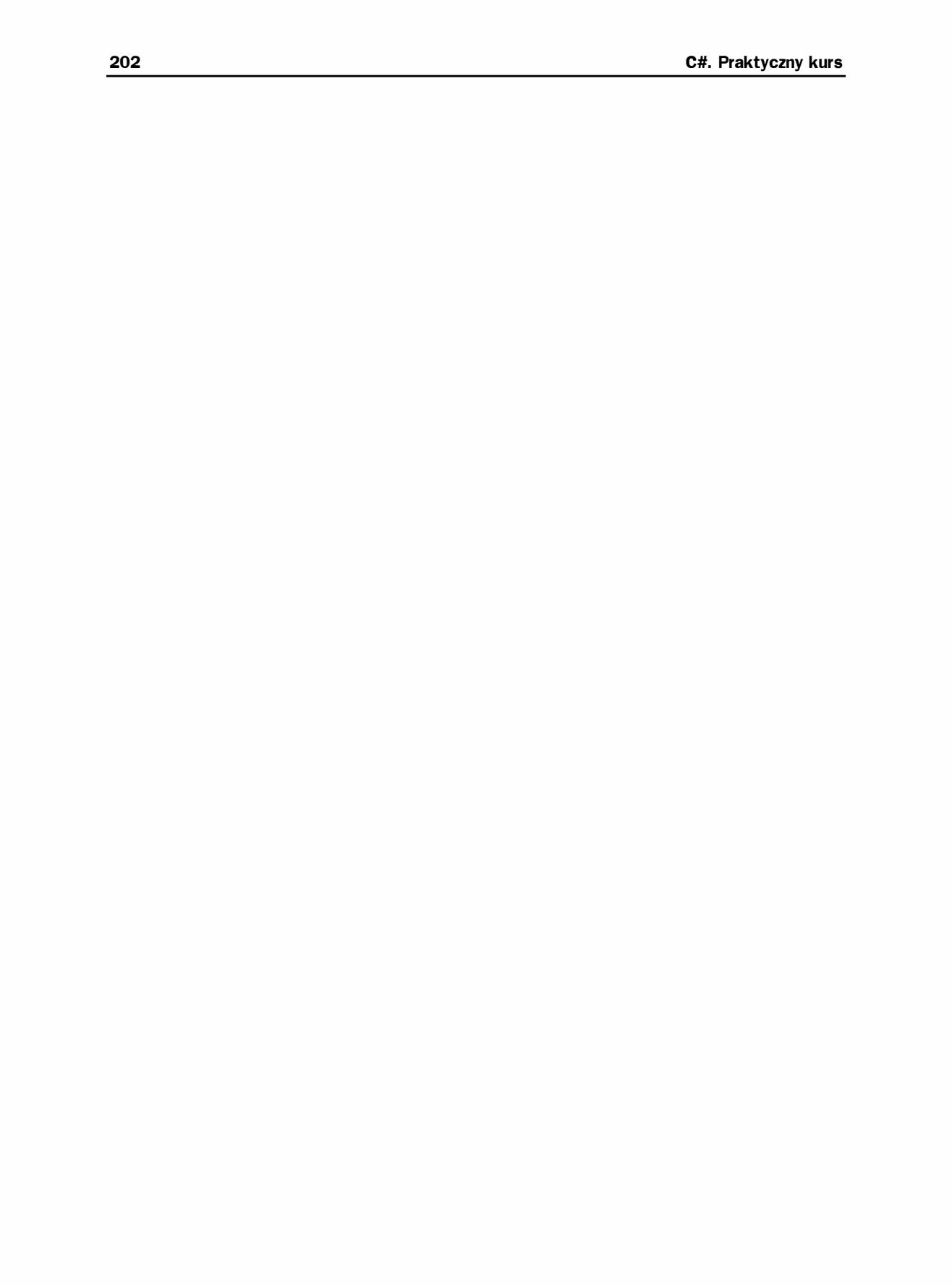
Napisz kod klasy zawierającej właściwość przechowującą wartość całkowitą. Każdy odczyt tej właściwości powinien powodować zwrócenie kolejnego wyrazu ciągu opisanego wzorem $a_{n+1} = 2 \times (a_n - 1) - 2$.

Ćwiczenie 20.5

Do struktury z listingu 3.78 dopisz dwuargumentowy konstruktor ustawiający wartość jej pól. Zastanów się, czy modyfikacja pól może się odbywać poprzez właściwości `x` i `y`.

Ćwiczenie 20.6

Zmodyfikuj kod z listingu 3.78 w taki sposób, aby użyta została automatyczna implementacja właściwości.



Rozdział 4.

Wyjątki i obsługa błędów

Praktycznie w każdym większym programie powstają jakieś błędy. Powodów tego stanu rzeczy jest bardzo wiele — może to być skutek niefrasobliwości programisty, przyjęcia założenia, że wprowadzone dane są zawsze poprawne, niedokładnej specyfikacji poszczególnych modułów aplikacji, użycia niesprawdzonych bibliotek czy nawet zwykłego zapomnienia o zainicjowaniu jednej tylko zmiennej. Na szczęście w C#, tak jak i w większości współczesnych obiektowych języków programowania, istnieje mechanizm tzw. wyjątków, który pozwala na przechwytywanie błędów. Ta właśnie tematyka zostanie przedstawiona w kolejnych trzech lekcjach.

Lekcja 21. Blok try...catch

Lekcja 21. jest wprowadzeniem do tematyki wyjątków. Zobaczmy, jakie są sposoby zapobiegania powstawaniu niektórych typów błędów w programach, a także jak stosować przechwytyujący błąd blok instrukcji `try...catch`. Przedstawiony zostanie też bliżej wyjątek o nieco egzotycznej dla początkujących programistów nazwie `IndexOutOfRangeException`, dzięki któremu można uniknąć błędów związanych z przekroczeniem dopuszczalnego zakresu indeksów tablic.

Badanie poprawności danych

Powróćmy na chwilę do rozdziału 2. i lekcji 12. Został tam przedstawiony przykład, w którym następowało odwołanie do nieistniejącego elementu tablicy (listing 2.40). Występowała w nim sekwencja instrukcji:

```
int tab[] = new int[10];
tab[10] = 1;
```

Doświadczony programista od razu zauważy, że instrukcje te są błędne, jako że zadeklarowana została tablica 10-elementowa, więc — ponieważ indeksowanie tablicyaczyna się od 0 — ostatni element tablicy ma indeks 9. Tak więc instrukcja `tab[10] = 1`

powoduje próbę odwołania się do nieistniejącego jedenastego elementu tablicy. Ten błąd jest jednak stosunkowo prosty do wychwycenia, nawet gdyby pomiędzy deklaracją tablicy a nieprawidłowym odwołaniem były umieszczone inne instrukcje.

Dużo więcej kłopotów mogłyby nam sprawić sytuacja, w której np. tablica byłaby deklarowana w jednej klasie, a odwołanie do niej następowałoby w innej. Taka przykładowa sytuacja została przedstawiona na listingu 4.1.

Listing 4.1. Odwołanie do nieistniejącego elementu tablicy

```
using System;

public class Tablica
{
    private int[] tablica = new int[10];
    public int pobierzElement(int indeks)
    {
        return tablica[indeks];
    }
    public void ustawElement(int indeks, int wartosc)
    {
        tablica[indeks] = wartosc;
    }
}

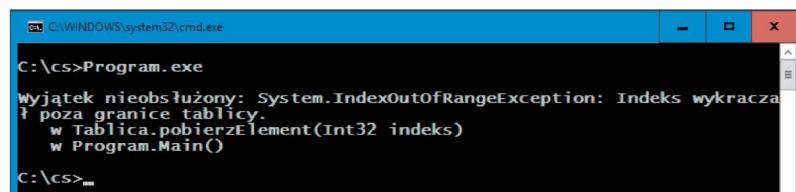
public class Program
{
    public static void Main()
    {
        Tablica tablica = new Tablica();
        tablica.ustawElement(5, 10);
        int liczba = tablica.pobierzElement(10);
        Console.WriteLine(liczba);
    }
}
```

Powstały tu dwie klasy: `Tablica` oraz `Program`. W klasie `Tablica` zostało zadeklarowane prywatne pole typu tablicowego o nazwie `tablica`, któremu została przypisana 10-elementowa tablica liczb całkowitych. Ponieważ pole to jest prywatne (por. lekcja 18.), dostęp do niego mają jedynie inne składowe klasy `Tablica`. Dlatego też powstały dwie metody, `pobierzElement` oraz `ustawElement`, operujące na elementach tablicy. Metoda `pobierzElement` zwraca wartość zapisaną w komórce o indeksie przekazanym jako argument, natomiast `ustawElement` zapisuje wartość drugiego argumentu w komórce o indeksie wskazywanym przez argument pierwszy.

W klasie `Program` tworzymy obiekt klasy `Tablica` i wykorzystujemy metodę `ustawElement` do zapisania w komórce o indeksie 5 wartości 10. W kolejnej linii popełniamy drobny błąd. W metodzie `pobierzElement` odwołujemy się do nieistniejącego elementu o indeksie 10. Musi to spowodować wystąpienie błędu w trakcie działania aplikacji (rysunek 4.1). Błąd tego typu bardzo łatwo popełnić, gdyż w klasie `Program` nie widzimy rozmiarów tablicy, nietrudno więc o pomyłkę (klasa `Tablica` mogłaby być przecież zapisana w osobnym pliku).

Rysunek 4.1.

Efekt odwołania do nieistniejącego elementu w klasie Tablica



Jak poradzić sobie z takim problemem? Pierwszym nasuwającym się sposobem jest sprawdzenie w metodach pobierzElement i ustawElement, czy przekazane argumenty nie przekraczają dopuszczalnych wartości. Jeśli takie przekroczenie nastąpi, należy zasygnalizować błąd. To jednak prowokuje pytanie, w jaki sposób ten błąd sygnalizować. Jednym z pomysłów jest zwracanie przez metodę (funkcję) wartości -1 w przypadku błędu oraz wartości nieujemnej (najczęściej 0 lub 1), jeśli błąd nie wystąpił. Ten sposób będzie dobry w przypadku metody ustawElement, która wyglądałaby wtedy następująco:

```
public int ustawElement(int indeks, int wartosc)
{
    if(indeks >= tablica.length || indeks < 0)
    {
        return -1;
    }
    else
    {
        tablica[indeks] = wartosc;
        return 0;
    }
}
```

Wystarczyłoby teraz w klasie Main testować wartość zwróconą przez ustawElement, aby sprawdzić, czy nie przekroczymy dopuszczalnego indeksu tablicy. Niestety, tej techniki nie można zastosować w przypadku metody pobierzElement — przecież zwraca ona wartość zapisaną w jednej z komórek tablicy. A zatem -1 i 0 użyte przed chwilą do zasygnalizowania, czy operacja zakończyła się błędem, mogą być wartościami odczytanymi z tablicy. Trzeba więc wymyślić inną metodę. Może to być np. użycie w klasie Tablica dodatkowego pola sygnalizującego. Pole to byłoby typu bool i dostępne przez odpowiednią właściwość. Ustawiona na true oznaczałaby, że ostatnia operacja zakończyła się błędem, natomiast ustawiona na false, że zakończyła się sukcesem. Klasa Tablica miałaby wtedy postać jak na listingu 4.2.

Listing 4.2. Zastosowanie dodatkowego pola sygnalizującego stan operacji

```
public class Tablica
{
    private int[] tablica = new int[10];
    private bool _blad = false;
    public bool wystapilBlad
    {
        get
        {
            return _blad;
        }
    }
}
```

```
public int pobierzElement(int indeks)
{
    if(indeks >= tablica.Length || indeks < 0)
    {
        _blad = true;
        return 0;
    }
    else
    {
        _blad = false;
        return tablica[indeks];
    }
}
public void ustawElement(int indeks, int wartosc)
{
    if(indeks >= tablica.Length || indeks < 0)
    {
        _blad = true;
    }
    else
    {
        tablica[indeks] = wartosc;
        _blad = false;
    }
}
```

Do klasy dodaliśmy pole typu `bool` o nazwie `_blad` (początkowa wartość to `false`), a także właściwość `wystapiłBlad` umożliwiającą odczyt jego stanu. W metodzie `pobierzElement` sprawdzamy najpierw, czy przekazany indeks przekracza dopuszczalny zakres. Jeśli przekracza, ustawiamy pole `_blad` na `true` oraz zwracamy wartość 0. Oczywiście w tym przypadku zwrocona wartość nie ma żadnego praktycznego znaczenia (przecież wystąpił błąd), niemniej jednak coś musimy zwrócić. Użycie instrukcji `return` i zwrócenie wartości typu `int` jest konieczne, inaczej kompilator zgłosi błąd. Jeżeli jednak argument przekazany metodzie nie przekracza dopuszczalnego indeksu tablicy, pole `_blad` ustawiamy na `false` oraz zwracamy wartość znajdująca się pod wskazanym indeksem.

W metodzie `ustawElement` postępujemy podobnie. Sprawdzamy, czy przekazany indeks przekracza dozwolone wartości. Jeśli tak jest, pole `_blad` ustawiamy na `true`, w przeciwnym wypadku przypisujemy wskazanej komórce tablicy wartość przekazaną przez argument `wartosc` i ustawiamy pole `_blad` na `false`. Po takiej modyfikacji obu metod w klasie `Program` można już bez problemów stwierdzić, czy operacje wykonywane na klasie `Tablica` zakończyły się sukcesem. Przykładowe wykorzystanie możliwości, jakie daje nowe pole wraz z właściwością `wystapiłBlad`, zostało przedstawione na listingu 4.3.

Listing 4.3. Wykorzystanie pola sygnalizującego stan operacji

```
using System;

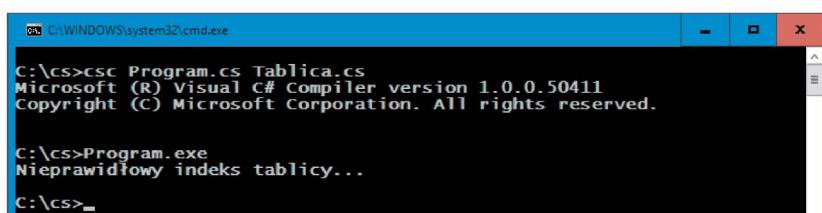
public class Program
{
```

```
public static void Main()
{
    Tablica tablica = new Tablica();
    tablica.ustawElement(5, 10);
    int liczba = tablica.pobierzElement(10);
    if (tablica.wystapilBlad)
    {
        Console.WriteLine("Nieprawidłowy indeks tablicy...");
    }
    else
    {
        Console.WriteLine(liczba);
    }
}
```

Podstawowe wykonywane operacje są takie same jak w przypadku klasy z listingu 4.1. Po pobraniu elementu sprawdzamy jednak, czy operacja ta zakończyła się sukcesem, i wyświetlamy odpowiedni komunikat na ekranie. Identyczne sprawdzenie można byłoby wykonać również po wywołaniu metody `ustawElement`. Wykonanie kodu z listingu 4.3 spowoduje oczywiście wyświetlenie napisu `Nieprawidłowy indeks tablicy...` (rysunek 4.2).

Rysunek 4.2.

Informacja o błędzie generowana przez program z listingu 4.3



Zamiast jednak wymyślać coraz to nowe sposoby radzenia sobie z takimi błędami, w C# można zastosować mechanizm obsługi sytuacji wyjątkowych. Jak okaże się za chwilę, pozwala on w bardzo wygodny i przejrzysty sposób radzić sobie z błędami w aplikacjach.

Wyjątki w C#

Wyjątek (ang. *exception*) jest to typ programistyczny, który powstaje w razie wystąpienia sytuacji wyjątkowej — najczęściej jakiegoś błędu. Z powstaniem wyjątku spotkaliśmy się już w rozdziale 2., w lekcji 12. Był on spowodowany przekroczeniem dopuszczalnego zakresu tablicy. Został wtedy wygenerowany właśnie wyjątek o nazwie `IndexOutOfRangeException` (jest on zdefiniowany w przestrzeni nazw `System`), oznaczający, że indeks tablicy znajduje się poza dopuszczalnymi granicami. Środowisko uruchomieniowe wygenerowało więc odpowiedni komunikat (taki sam jak na rysunku 4.1) i zakończyło działanie aplikacji.

Oczywiście, gdyby możliwości tego mechanizmu kończyły się na wyświetlaniu informacji na ekranie i przerywaniu działania programu, jego przydatność byłaby mocno ograniczona. Na szczęście wygenerowany wyjątek można przechwycić i wykonać

własny kod obsługi błędu. Do takiego przechwycenia służy blok instrukcji `try...catch`. W najprostszej postaci wygląda on następująco:

```
try
{
    //instrukcje mogące spowodować wyjątek
}
catch(TypWyjątku [identyfikatorWyjątku])
{
    //obsługa wyjątku
}
```

W nawiasie klamrowym występującym po słowie `try` umieszczamy instrukcję, która może spowodować wystąpienie wyjątku. W bloku występującym po `catch` umieszczamy kod, który ma zostać wykonany, kiedy wystąpi wyjątek. W nawiasie okrągłym znajdująącym się po słowie `catch` podaje się typ wyjątku oraz opcjonalny identyfikator (opcjonalność tego elementu została zaznaczona nawiasem kwadratowym). W praktyce mogłoby to wyglądać tak jak na listingu 4.4.

Listing 4.4. Użycie bloku `try...catch`

```
using System;

public class Program
{
    public static void Main()
    {
        int[] tab = new int[10];
        try
        {
            tab[10] = 100;
        }
        catch(IndexOutOfRangeException e)
        {
            Console.WriteLine("Nieprawidłowy indeks tablicy!");
        }
    }
}
```

Jak widać, wszystko odbywa się tu zgodnie z wcześniejszym ogólnym opisem. W bloku `try` znalazła się instrukcja `tab[10] = 100`, która — jak wiemy — spowoduje wygenerowanie wyjątku. W nawiasie okrągłym występującym po instrukcji `catch` został wymieniony rodzaj (typ) wyjątku, który będzie wygenerowany: `IndexOutOfRangeException`, oraz jego identyfikator: `e`. Identyfikator to nazwa¹, która pozwala na wykonywanie operacji związanych z wyjątkiem, tym jednak zajmiemy się w kolejnej lekcji. Ze względu na to, że identyfikator `e` nie został użyty w bloku `catch`, kompilator zgłosi ostrzeżenie o braku odwołań do zmiennej `e`. W bloku po `catch` znajduje się instrukcja `Console.WriteLine` wyświetlająca odpowiedni komunikat na ekranie. Tym razem po uruchomieniu kodu zobaczymy widok podobny do zaprezentowanego na rysunku 4.2.

¹ Dokładniej jest to nazwa zmiennej obiektowej, co zostanie bliżej wyjaśnione w lekcji 22.

Ponieważ w tym przypadku identyfikator nie jest używany w bloku `catch`, w praktyce można by go również pominąć (dzięki temu kompilator nie będzie wyświetlał ostrzeżeń, a funkcjonalność kodu się nie zmieni), czyli blok ten mógłby również wyglądać następująco:

```
catch(IndexOutOfRangeException)
{
    Console.WriteLine("Nieprawidłowy indeks tablicy!");
}
```

Blok `try...catch` nie musi jednak obejmować tylko jednej instrukcji ani też tylko instrukcji mogących wygenerować wyjątek. Blokiem tym można objąć więcej instrukcji, tak jak zostało to zaprezentowane na listingu 4.5. Dzięki temu kod programu może być bardziej zwięzły, a za pomocą jednej instrukcji `try...catch` da się obsłużyć wiele wyjątków (zostanie to dokładniej pokazane w lekcji 22.).

Listing 4.5. Korzystanie z bloku `try`

```
using System;

public class Program
{
    public static void Main()
    {
        try
        {
            int[] tab = new int[10];
            tab[10] = 5;
            Console.WriteLine(
                "Dziesiąty element tablicy ma wartość: " + tab[10]);
        }
        catch(IndexOutOfRangeException)
        {
            Console.WriteLine("Nieprawidłowy indeks tablicy!");
        }
    }
}
```

Nie trzeba również obejmować blokiem `try` instrukcji bezpośrednio generujących wyjątek, tak jak miało to miejsce w dotychczasowych przykładach. Wyjątek wygenerowany przez obiekt klasy `Y` może być bowiem przechwytywany w klasie `X`, która korzysta z obiektów klasy `Y`. Łatwo to pokazać na przykładzie klas z listingu 4.1. Klasa `Tablica` pozostanie bez zmian, natomiast klasę `Program` zmodyfikujemy tak, aby miała wygląd zaprezentowany na listingu 4.6.

Listing 4.6. Przechwycenie wyjątku generowanego w innej klasie

```
public class Program
{
    public static void Main()
    {
        Tablica tablica = new Tablica();
        try
        {
```

```
tablica.ustawElement(5, 10);
int liczba = tablica.pobierzElement(10);
Console.WriteLine(liczba);
}
catch(IndexOutOfRangeException)
{
    Console.WriteLine("Nieprawidłowy indeks tablicy!");
}
}
```

Spójrzmy: w bloku try mamy trzy instrukcje, z których jedna, `int liczba = tablica.pobierzElement(10)`, jest odpowiedzialna za wygenerowanie wyjątku. Czy ten blok jest zatem prawidłowy? Wyjątek powstaje przecież we wnętrzu metody `pobierzElement` w klasie `Tablica`, a nie w klasie `Program`. Zostanie on jednak przekazany do metody `Main` w klasie `Program`, jako że wywołuje ona metodę `pobierzElement` klasy `Tablica` (czyli tę, która generuje wyjątek). Tym samym w metodzie `Main` z powodzeniem możemy zastosować blok `try...catch`.

Z identyczną sytuacją będziemy mieć do czynienia w przypadku hierarchicznego wywołania metod jednej klasy, czyli na przykład kiedy metoda `f` wywołuje metodę `g`, która wywołuje metodę `h` generującą z kolei wyjątek. W każdej z wymienionych metod można zastosować blok `try...catch` do przechwycenia tego wyjątku. Dokładnie taki przykład jest widoczny na listingu 4.7.

Listing 4.7. Propagacja wyjątku

```
using System;

public class Program
{
    public void f()
    {
        try
        {
            g();
        }
        catch(IndexOutOfRangeException)
        {
            Console.WriteLine("Wyjątek: metoda f");
        }
    }
    public void g()
    {
        try
        {
            h();
        }
        catch(IndexOutOfRangeException)
        {
            Console.WriteLine("Wyjątek: metoda g");
        }
    }
    public void h()
```

```
{  
    int[] tab = new int[0];  
    try  
    {  
        tab[0] = 1;  
    }  
    catch(IndexOutOfRangeException)  
    {  
        Console.WriteLine("Wyjątek: metoda h");  
    }  
}  
public static void Main()  
{  
    Program pr = new Program();  
    try  
    {  
        pr.f();  
    }  
    catch(IndexOutOfRangeException)  
    {  
        Console.WriteLine("Wyjątek: metoda main");  
    }  
}
```

Taką klasę skompilujemy bez żadnych problemów. Musimy jednak dobrze zdawać sobie sprawę, jak taki kod będzie wykonywany. Pytanie bowiem dotyczy tego, które bloki try zostaną wykonane. Zasada jest następująca: zostanie wykonany blok najbliższy instrukcji powodującej wyjątek. Tak więc w przypadku przedstawionym na listingu 4.7 będzie to jedynie blok obejmujący instrukcję `tab[0] = 1;` w metodzie `h`. Jeśli jednak będziemy usuwać kolejne bloki try najpierw z metody `h`, następnie `g`, `f` i ostatecznie z `Main`, zobaczymy, że faktycznie wykonywany jest zawsze blok najbliższy miejsca wystąpienia błędu. Po usunięciu wszystkich instrukcji try wyjątek nie zostanie obsłużony w naszej klasie i obsłuży go środowisko uruchomieniowe, co spowoduje zakończenie pracy aplikacji i pojawi się znany nam już komunikat na konsoli.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 21.1

Przygotuj taką wersję klasy `Tablica` z listingu 4.2, w której będzie możliwe rozpoznanie, czy błąd powstał na skutek przekroczenia dolnego, czy też górnego zakresu indeksu.

Ćwiczenie 21.2

Napisz przykładowy program ilustrujący zachowanie klasy `Tablica` z ćwiczenia 21.1.

Ćwiczenie 21.3

Zmień kod klasy `Program` z listingu 4.3 w taki sposób, aby było również sprawdzane, czy wywołanie metody `ustawElement` zakończyło się sukcesem.

Ćwiczenie 21.4

Popraw kod z listingu 4.2 tak, aby do wychwytywania błędów był wykorzystywany mechanizm wyjątków, a nie instrukcja warunkowa `if`.

Ćwiczenie 21.5

Napisz klasę `Example`, w której będzie się znajdowała metoda o nazwie `a`, wywoływaną z kolei przez metodę o nazwie `b`. W metodzie `a` wygeneruj wyjątek `IndexOutOfRangeException`. Napisz następnie klasę `Program` zawierającą metodę `Main`, w której zostanie utworzony obiekt klasy `Example` i zostaną wywołane metody `a` oraz `b` tego obiektu. W metodzie `Main` zastosuj bloki `try...catch` przechwytyjące powstałe wyjątki.

Lekcja 22. Wyjątki to obiekty

W lekcji 21. przedstawiony został wyjątek sygnalizujący przekroczenie dopuszczalnego zakresu tablicy. To nie jest oczywiście jedyny dostępny typ — czas omówić również inne. W lekcji 22. okaże się, że wyjątki są tak naprawdę obiektami, a także że tworzą hierarchiczną strukturę. Zostanie pokazane, jak przechwytywać wiele wyjątków w jednym bloku `try` oraz że bloki `try...catch` mogą być zagnieżdżane. Stanie się też jasne, że jeden wyjątek ogólny może obsłużyć wiele błędnych sytuacji.

Dzielenie przez zero

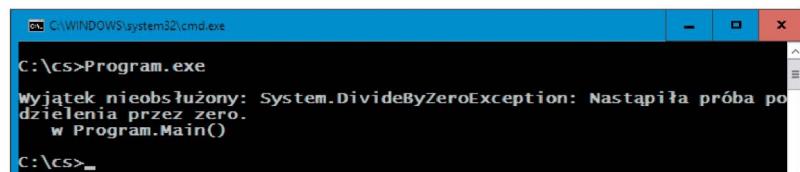
Rodzajów wyjątków jest bardzo wiele. Powiedziano już, jak reagować na przekroczenie zakresu tablicy. Przeanalizujmy zatem inny typ wyjątku, powstający, kiedy jest podejmowana próba wykonania dzielenia przez zero. W tym celu trzeba spreparować odpowiedni fragment kodu. Wystarczy, że w metodzie `Main` umieścimy przykładowe instrukcje:

```
int liczba1 = 10, liczba2 = 0;
liczba1 = liczba1 / liczba2;
```

Kompilacja takiego kodu przebiegnie bez problemu, jednak próba wykonania musi skończyć się komunikatem o błędzie, widocznym na rysunku 4.3 — przecież nie można dzielić przez 0. Widać wyraźnie, że tym razem został zgłoszony wyjątek `DivideByZeroException`.

Rysunek 4.3.

Próba wykonania dzielenia przez zero



Wykorzystując wiedzę z lekcji 21., nie powinniśmy mieć żadnych problemów z napisaniem kodu, który przechwyci taki wyjątek. Trzeba wykorzystać dobrze już znaną instrukcję try...catch w postaci:

```
try
{
    int liczba1 = 10, liczba2 = 0;
    liczba1 = liczba1 / liczba2;
}
catch(DivideByZeroException)
{
    //instrukcje do wykonania, kiedy wystąpi wyjątek
}
```

Intuicja podpowiada, że rodzajów wyjątków może być bardzo, bardzo dużo. Aby sprawnie się nimi posługiwać, trzeba wiedzieć, czym tak naprawdę są.

Wyjątek jest obiektem

Wyjątek, który do tej pory określany był jako typ programistyczny, to nic innego jak obiekt powstający, kiedy w programie wystąpi sytuacja wyjątkowa. Skoro wyjątek jest obiektem, to wspominany wcześniej typ wyjątku (`IndexOutOfRangeException`, `DivideByZeroException`) to nic innego jak klasa opisująca tenże obiekt. Jeśli teraz spojrzymy ponownie na ogólną postać instrukcji try...catch:

```
try
{
    //instrukcje mogące spowodować wyjątek
}
catch(TypWyjątku [identyfikatorWyjątku])
{
    //obsługa wyjątku
}
```

stanie się jasne, że w takim razie opcjonalny *identyfikatorWyjątku* to zmienna obiektowa wskazująca na obiekt wyjątku. Na tym obiekcie można wykonywać operacje zdefiniowane w klasie wyjątku. Do tej pory identyfikator nie był używany, czas więc sprawdzić, jak można go użyć. Zobaczmy to na przykładzie wyjątku generowanego podczas próby wykonania dzielenia przez zero. Przykład jest widoczny na listingu 4.8.

Listing 4.8. Użycie właściwości Message obiektu wyjątku

```
using System;

public class Program
{
    public static void Main()
    {
        try
        {
            int liczba1 = 10, liczba2 = 0;
            liczba1 = liczba1 / liczba2;
        }
        catch(DivideByZeroException e)
```

```

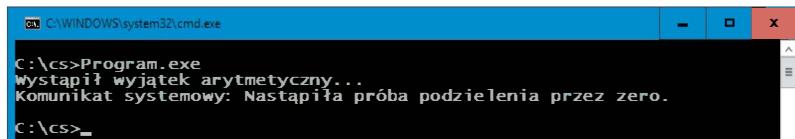
    {
        Console.WriteLine("Wystąpił wyjątek arytmetyczny...");
        Console.Write("Komunikat systemowy: ");
        Console.WriteLine(e.Message);
    }
}

```

Wykonujemy tutaj próbę niedozwolonego dzielenia przez zero oraz przechwytyujemy wyjątek klasy `DivideByZeroException`. W bloku catch najpierw wyświetlamy nasze własne komunikaty o błędzie, a następnie komunikat systemowy. Po uruchomieniu kodu na ekranie zobaczymy widok zaprezentowany na rysunku 4.4.

Rysunek 4.4.

Wyświetlenie systemowego komunikatu o błędzie



Istnieje jeszcze jedna możliwość uzyskania komunikatu o błędzie. Jest nią użycie metody `ToString` obiektu wyjątku, czyli zamiast pisać:

`e.Message`

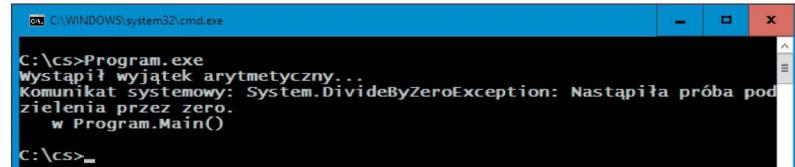
można użyć konstrukcji:

`e.ToString()`

Komunikat będzie wtedy pełniejszy, jest on widoczny na rysunku 4.5.

Rysunek 4.5.

Komunikat o błędzie uzyskany za pomocą metody ToString

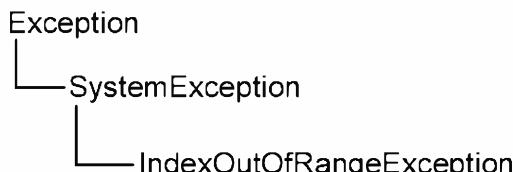


Hierarchia wyjątków

Każdy wyjątek jest obiektem pewnej klasy. Klasy podlegają z kolei regułom dziedziczenia, zgodnie z którymi powstaje ich hierarchia. Kiedy zatem pracujemy z wyjątkami, musimy tę kwestię wziąć pod uwagę. Na przykład dla znanego nam już wyjątku o nazwie `IndexOutOfRangeException` hierarchia wygląda jak na rysunku 4.6.

Rysunek 4.6.

Hierarchia klas dla wyjątku IndexOutOfRangeException



Wynika z tego kilka własności. Przede wszystkim, jeśli spodziewamy się, że dana instrukcja może wygenerować wyjątek typu X , możemy zawsze przechwycić wyjątek ogólniejszy, czyli taki, którego typem będzie jedna z klas nadrzędnych do X . Jest to bardzo wygodna technika. Przykładowo z klasy `SystemException` dziedziczy bardzo wiele klas wyjątków odpowiadających najróżniejszym błędom. Jedną z nich jest `ArithmeticeException` (z niej z kolei dziedziczy wiele innych klas, np. `DivideByZeroException`). Jeśli instrukcje, które obejmujemy blokiem `try...catch`, mogą spowodować wiele różnych wyjątków, zamiast stosować wiele oddzielnych instrukcji przechwytyujących konkretne typy błędów, często lepiej jest wykorzystać jedną przechwytyującą wyjątek ogólniejszy. Spójrzmy na listing 4.9.

Listing 4.9. Przechwytywanie wyjątku ogólnego

```
using System;

public class Program
{
    public static void Main()
    {
        try
        {
            int liczba1 = 10, liczba2 = 0;
            liczba1 = liczba1 / liczba2;
        }
        catch(SystemException e)
        {
            Console.WriteLine("Wystąpił wyjątek systemowy...");
            Console.Write("Komunikat systemowy: ");
            Console.WriteLine(e.ToString());
        }
    }
}
```

Jest to znany nam już program generujący błąd polegający na próbie wykonania niedozwolonego dzielenia przez zero. Tym razem jednak zamiast wyjątku klasy `DivideByZeroException` przechwytyujemy wyjątek klasy ogólniejszej — `SystemException`. Co więcej, nic nie stoi na przeszkodzie, aby przechwycić wyjątek jeszcze ogólniejszy, czyli klasy nadrzędnej do `SystemException`. Byłaby to klasa `Exception`.

Przechwytywanie wielu wyjątków

W jednym bloku `try` można przechwytywać wiele wyjątków. Konstrukcja taka zawiera wtedy jeden blok `try` i wiele bloków `catch`. Schematycznie wygląda ona następująco:

```
try
{
    //instrukcje mogące spowodować wyjątek
}
catch(KlasaWyjątku1 identyfikatorWyjątku1)
{
    //obsługa wyjątku 1
}
catch(KlasaWyjątku2 identyfikatorWyjątku2)
```

```
{  
    //obsługa wyjątku 2  
}  
/*  
... dalsze bloki catch ...  
*/  
catch(KlasaWyjątkuN identyfikatorWyjątkuN)  
{  
    //obsługa wyjątku N  
}
```

Po wygenerowaniu wyjątku sprawdzane jest, czy jest on klasy *KlasaWyjątku1* (inaczej: czy jego typem jest *KlasaWyjątku1*), jeśli tak — są wykonywane instrukcje obsługi tego wyjątku i blok `try...catch` jest opuszczany. Jeżeli jednak klasą wyjątku nie jest *KlasaWyjątku1*, następuje sprawdzenie, czy jest to klasa *KlasaWyjątku2* itd.

Przy tego typu konstrukcjach należy jednak pamiętać o hierarchii wyjątków, nie jest bowiem obojętne, w jakiej kolejności będą one przechwytywane. Ogólna zasada jest taka, że nie ma znaczenia kolejność, o ile wszystkie wyjątki są na jednym poziomie hierarchii. Jeśli jednak przechwytyujemy wyjątki z różnych poziomów, najpierw muszą to być te bardziej szczegółowe, czyli stojące niżej w hierarchii, a dopiero po nich ogólniejsze, czyli stojące wyżej w hierarchii.

Nie można zatem najpierw przechwycić wyjątku `SystemException`, a dopiero po nim `ArithmeticException`, gdyż skończy się to błędem komilacji (`ArithmeticException` dziedziczy po `SystemException`). Jeśli więc dokonamy próby komilacji przykładowego programu przedstawionego na listingu 4.10, efektem będą komunikaty widoczne na rysunku 4.7.

Listing 4.10. Nieprawidłowa kolejność przechwytywania wyjątków

```
using System;  
  
public class Program  
{  
    public static void Main()  
    {  
        try  
        {  
            int liczba1 = 10, liczba2 = 0;  
            liczba1 = liczba1 / liczba2;  
        }  
        catch(SystemException e)  
        {  
            Console.WriteLine(e.Message);  
        }  
        catch(DivideByZeroException e)  
        {  
            Console.WriteLine(e.Message);  
        }  
    }  
}
```

Rysunek 4.7.

Błędna hierarchia wyjątków powoduje błąd kompilacji

```
C:\>csc Program.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

Program.cs(16,11): error CS0160: A previous catch clause already catches all exceptions of this or of a super type ('SystemException')

C:\>
```

Dzieje się tak dlatego, że (można powiedzieć) błąd ogólniejszy zawiera już w sobie błąd bardziej szczegółowy. Jeśli zatem przechwytyujemy najpierw wyjątek `SystemException`, to jest tak, jakbyśmy przechwycili już wyjątki wszystkich klas dziedziczących po `SystemException`. Dlatego też kompilator protestuje.

Kiedy jednak może przydać się sytuacja, gdy najpierw przechwytyujemy wyjątek szczegółowy, a dopiero potem ogólny? Otóż wtedy, kiedy chcemy w specyficzny sposób zareagować na konkretny typ wyjątku, a wszystkie pozostałe z danego poziomu hierarchii obsłużyć w identyczny, standardowy sposób. Taka przykładowa sytuacja jest przedstawiona na listingu 4.11.

Listing 4.11. Przechwytywanie różnych wyjątków

```
using System;

public class Program
{
    public static void Main()
    {
        Punkt punkt = null;
        int liczba1 = 10, liczba2 = 0;
        try
        {
            liczba1 = liczba1 / liczba2;
            punkt.x = liczba1;
        }
        catch(ArithmeticException e)
        {
            Console.WriteLine("Nieprawidłowa operacja arytmetyczna");
            Console.WriteLine(e.ToString());
        }
        catch(Exception e)
        {
            Console.WriteLine("Błąd ogólny");
            Console.WriteLine(e.ToString());
        }
    }
}
```

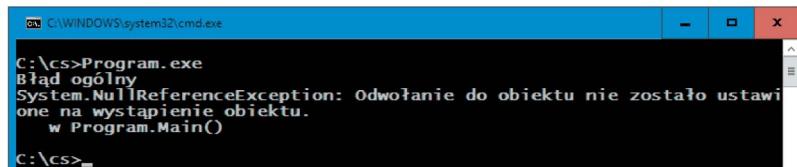
Zostały zadeklarowane trzy zmienne: pierwsza typu `Punkt` o nazwie `punkt` oraz dwie typu `int`: `liczba1` i `liczba2`. Zmiennej `punkt` została przypisana wartość pusta `null`, nie został zatem utworzony żaden obiekt klasy `Punkt`. W bloku `try` są wykonywane dwie błędne instrukcje. Pierwsza z nich to znane z poprzednich przykładów dzielenie przez zero. Druga instrukcja z bloku `try` to z kolei próba odwołania się do pola `x` nieistniejącego obiektu klasy `Punkt` (przecież zmenna `punkt` zawiera wartość `null`). Ponieważ

chcemy w sposób niestandardowy zareagować na błąd arytmetyczny, najpierw przechwytyjemy błąd typu `ArithmetiException` (jest to klasa nadziedzona dla `DivideByZeroException`) i w przypadku, kiedy wystąpi, wyświetlamy na ekranie napis Nieprawidłowa operacja arytmetyczna. W drugim bloku `catch` przechwytyjemy wszystkie inne możliwe wyjątki, w tym także `NullReferenceException`, występujący, kiedy próbujemy wykonać operacje na zmiennej obiektowej, która zawiera wartość `null`.

Po komplikacji (należy użyć dodatkowo jednej z zaprezentowanych w rozdziale 3. wersji klasy `Punkt` o publicznym dostępie do składowych `x` i `y`) i uruchomieniu kodu pojawi się na ekranie zgłoszenie tylko pierwszego błędu (efekt będzie podobny do przedstawionego na rysunku 4.5 w poprzedniej części lekcji). Dzieje się tak dlatego, że po jego wystąpieniu blok `try` został przerwany, a sterowanie zostało przekazane blokowi `catch`. Jeśli więc w bloku `try` któraś z instrukcji spowoduje wygenerowanie wyjątku, dalsze instrukcje z tego bloku nie zostaną wykonane. Nie miała więc szansy zostać wykonana nieprawidłowa instrukcja `punkt.x = liczba;`. Jeśli jednak usuniemy wcześniejsze dzielenie przez zero, przekonamy się, że i ten błąd zostanie przechwycony przez drugi blok `catch`, a na ekranie pojawi się stosowny komunikat (rysunek 4.8).

Rysunek 4.8.

Odwolanie do pustej zmiennej obiektowej zostało wychwycone przez drugi blok `catch`



Zagnieżdżanie bloków `try...catch`

Bloki `try...catch` można zagnieździć. To znaczy, że w jednym bloku przechwytyjącym wyjątek X może istnieć drugi blok, który będzie przechwytywał wyjątek Y (moga to być wyjątki tego samego typu). Schematycznie taka konstrukcja wygląda następująco:

```
try
{
    //instrukcje mogące spowodować wyjątek 1
    try
    {
        //instrukcje mogące spowodować wyjątek 2
    }
    catch (TypWystąpienia2 identyfikatorWystąpienia2)
    {
        //obsługa wyjątku 2
    }
}
catch (TypWystąpienia1 identyfikatorWystąpienia1)
{
    //obsługa wyjątku 1
}
```

Takie zagnieżdżenie może być wielopoziomowe, czyli w już zagnieżdżonym bloku `try` można umieścić kolejny taki blok. W praktyce tego rodzaju piętrowych konstrukcji zazwyczaj się nie stosuje, zwykle nie ma bowiem takiej potrzeby, a maksymalny

poziom bezpośredniego zagnieżdżenia z reguły nie przekracza dwóch poziomów (nie jest to jednak ograniczenie formalne — liczba zagnieżdżeń może być nieograniczona). Aby na praktycznym przykładzie pokazać taką dwupoziomową konstrukcję, zmodyfikujemy przykład z listingu 4.11. Zamiast obejmowania jednym blokiem try dwóch instrukcji powodujących błędów zastosujemy zagnieżdżenie, tak jak jest to widoczne na listingu 4.12.

Listing 4.12. Zagnieżdżone bloki try...catch

```
using System;

public class Program
{
    public static void Main()
    {
        Punkt punkt = null;
        int liczba1 = 10, liczba2 = 0;
        try
        {
            try
            {
                liczba1 = liczba1 / liczba2;
            }
            catch(ArithmeticeException)
            {
                Console.WriteLine("Nieprawidłowa operacja arytmetyczna");
                Console.WriteLine("Przypisuję zmiennej liczba1 wartość 10.");
                liczba1 = 10;
            }
            punkt.x = liczba1;
        }
        catch(Exception e)
        {
            Console.Write("Błąd ogólny: ");
            Console.WriteLine(e.Message);
        }
    }
}
```

Podobnie jak w poprzednim przypadku, deklarujemy trzy zmienne: punkt klasy Punkt oraz liczba1 i liczba2 typu int. Zmienna punkt otrzymuje też wartość pustą null. W wewnętrznym bloku try próbujemy wykonać nieprawidłowe dzielenie przez zero i przechwytyujemy wyjątek ArithmeticeException. Jeśli on wystąpi, zmiennej liczba1 przypisujemy wartość domyślną równą 10, dzięki czemu można wykonać kolejną operację, czyli próbę przypisania wartości zmiennej liczba1 polu x obiektu reprezentowanego przez punkt. Rzecz jasna, przypisanie takie nie może zostać wykonane, gdyż zmienna punkt jest pusta, jest zatem generowany wyjątek NullReferenceException, przechwytywany przez zewnętrzny blok try. Widac więc, że zagnieżdżanie bloków try może być przydatne, choć warto zauważyć, że identyczny efekt można osiągnąć, korzystając również z niezagnieżdzonej postaci instrukcji try...catch (ćwiczenie 21.3).

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 22.1

Popraw kod z listingu 4.10 tak, aby przechwytywanie wyjątków odbywało się w prawidłowej kolejności.

Ćwiczenie 22.2

Zmodyfikuj kod z listingu 4.11 tak, aby zgłoszone zostały oba typy błędów: `Arithmeti
cException` oraz `NullReferenceException`.

Ćwiczenie 22.3

Zmodyfikuj kod z listingu 4.12 w taki sposób, aby usunąć zagnieżdżenie bloków `try...catch`, nie zmieniając jednak efektów działania programu.

Ćwiczenie 22.4

Napisz przykładowy program, w którym zostaną wygenerowane dwa różne wyjątki. Wyświetl na ekranie systemowe komunikaty, ale w odwrotnej kolejności (najpierw powinien pojawić się komunikat dotyczący drugiego wyjątku, a dopiero potem ten dotyczący pierwszego).

Ćwiczenie 22.5

Napisz program zawierający taką metodę, aby pewne wartości przekazanych jej argumentów mogły powodować powstanie co najmniej dwóch różnych wyjątków. Wynikiem działania tej metody powinien być pusty ciąg znaków, jeśli wyjątki nie wystąpiły, lub też ciąg znaków zawierający wszystkie komunikaty systemowe wygenerowanych wyjątków. Przetestuj działanie metody, wywołując ją z różnymi argumentami.

Lekcja 23. Własne wyjątki

Wyjątki można przechwytywać, aby zapobiec niekontrolowanemu zakończeniu programu w przypadku wystąpienia błędu. Ta technika została pokazana w lekcjach 21. i 22. To jednak nie wszystko. Wyjątki można również samemu zgłaszać, a także tworzyć nowe, nieistniejące wcześniej ich rodzaje. Tej właśnie tematyce jest poświęcona bieżąca, 23. lekcja. Okaże się w niej również, że raz zgłoszony wyjątek może być zgłoszony ponownie.

Zgłaszanie wyjątków

Dzięki lekcji 22. wiadomo, że wyjątki są obiektami. Zgłoszenie (potocznie „wyrzucenie”, ang. *throw* — rzucać, wyrzucać) własnego wyjątku będzie polegało na utworzeniu nowego obiektu klasy opisującej wyjątek oraz użyciu instrukcji *throw*. Dokładniej, za pomocą instrukcji *new* należy utworzyć nowy obiekt klasy *Exception* lub dziedziczącej, bezpośrednio lub pośrednio, po *Exception*. Tak utworzony obiekt powinien stać się argumentem instrukcji *throw*. Jeśli zatem gdziekolwiek w pisany przez nas kodzie chcemy zgłosić wyjątek ogólny, wystarczy, że napiszemy:

```
throw new Exception();
```

Zobaczmy, jak to wygląda w praktyce. Założymy, że mamy klasę *Program*, a w niej metodę *Main*. Jednym zadaniem tej metody będzie zgłoszenie wyjątku klasy *Exception*. Taka klasa jest widoczna na listingu 4.13.

Listing 4.13. Zgłoszenie wyjątku za pomocą instrukcji *throw*

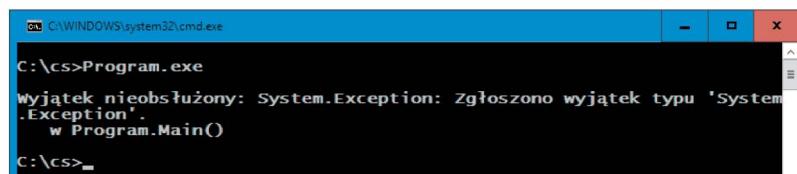
```
using System;

public class Program
{
    public static void Main()
    {
        throw new Exception();
    }
}
```

Wewnątrz metody *Main* została wykorzystana instrukcja *throw*, która jako argument otrzymała nowy obiekt klasy *Exception*. Po uruchomieniu takiego programu na ekranie zobaczymy widok zaprezentowany na rysunku 4.9. Jest to najlepszy dowód, że faktycznie udało nam się zgłosić wyjątek.

Rysunek 4.9.

Zgłoszenie wyjątku klasy *Exception*



Utworzenie obiektu wyjątku nie musi mieć miejsca bezpośrednio w instrukcji *throw*, można go utworzyć wcześniej, przypisać zmiennej obiektowej i dopiero tę zmienną wykorzystać jako argument dla *throw*. Zamiast więc pisać:

```
throw new Exception();
```

można również dobrze zastosować konstrukcję:

```
Exception exception = new Exception();
throw exception;
```

W obu przedstawionych przypadkach efekt będzie identyczny, najczęściej korzysta się jednak z pierwszego zaprezentowanego sposobu.

Jeśli chcemy, aby zgłoszonym wyjątkowi został przypisany komunikat, należy przekazać go jako argument konstruktora klasy `Exception`, a więc użyć instrukcji w postaci:

```
throw new Exception("komunikat");
```

lub:

```
Exception exception = new Exception("komunikat");
throw exception;
```

Oczywiście, można tworzyć obiekty wyjątków klas dziedziczących po `Exception`. Jeśli na przykład sami wykryjemy próbę dzielenia przez zero, być może zechcemy wygenerować nasz wyjątek, nie czekając, aż zgłosi go środowisko uruchomieniowe. Spójrzmy na listing 4.14.

Listing 4.14. Samodzielne zgłoszenie wyjątku `DivideByZeroException`

```
using System;

public class Dzielenie
{
    public static double Podziel(int liczba1, int liczba2)
    {
        if(liczba2 == 0)
            throw new DivideByZeroException(
                "Dzielenie przez zero: " + liczba1 + "/" + liczba2
            );
        return liczba1 / liczba2;
    }

    public class Program
    {
        public static void Main()
        {
            double wynik = Dzielenie.Podziel(20, 10);
            Console.WriteLine("Wynik pierwszego dzielenia: " + wynik);
            wynik = Dzielenie.Podziel (20, 0);
            Console.WriteLine("Wynik drugiego dzielenia: " + wynik);
        }
    }
}
```

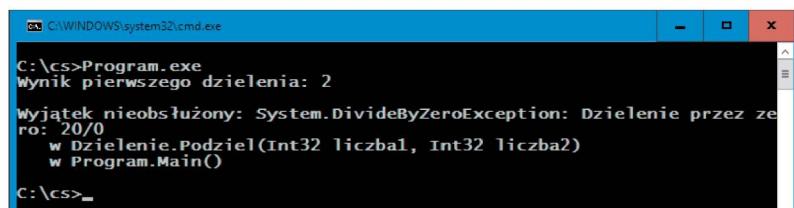
W klasie `Dzielenie` jest zdefiniowana statyczna metoda `Podziel`, która przyjmuje dwa argumenty typu `int`. Ma ona zwracać wynik dzielenia wartości przekazanej w argumencie `liczba1` przez wartość przekazaną w argumencie `liczba2`. Jest zatem jasne, że `liczba2` nie może mieć wartości 0. Sprawdzamy to, wykorzystując instrukcję warunkową `if`. Jeśli okaże się, że `liczba2` ma jednak wartość 0, za pomocą instrukcji `throw` zgłaszamy nowy wyjątek klasy `DivideByZeroException`. W konstruktorze klasy przekazujemy komunikat informujący o dzieleniu przez zero. Podajemy w nim wartości argumentów metody `Podziel`, tak by łatwo można było stwierdzić, jakie parametry spowodowały błąd.

Działanie metody `Podziel` jest testowane w metodzie `Main` z klasy `Program` (nie ma przy tym potrzeby tworzenia nowego obiektu klasy `Dzielenie`, gdyż `Podziel` jest metodą statyczną). Dwukrotnie wywołujemy metodę `Podziel`, raz przekazując jej argu-

menty równe 20 i 10, drugi raz równe 20 i 0. Spodziewamy się, że w drugim przypadku program zgłosi wyjątek `DivideByZeroException` ze zdefiniowanym przez nas komunikatem. Faktycznie program zachowa się właśnie tak, co jest widoczne na rysunku 4.10.

Rysunek 4.10.

Zgłoszenie wyjątku klasy `DivideByZeroException`



```
C:\WINDOWS\system32\cmd.exe
C:\cs>Program.exe
Wynik pierwszego dzielenia: 2
Wystąpił nieobsłużony: System.DivideByZeroException: Dzielenie przez zero: 20/0
w Dzielenie.Podziel(Int32 liczba1, Int32 liczba2)
w Program.Main()

C:\cs>-
```

Ponowne zgłoszenie przechwyconego wyjątku

Wiadomo już, jak przechwytywać wyjątki oraz jak je samemu zgłaszać. To pozwoli zapoznać się z techniką ponownego zgłaszania (potocznie: wyrzucania) już przechwyconego wyjątku. Jak pamiętamy, bloki `try...catch` można zagnieżdżać bezpośrednio, a także stosować je w przypadku kaskadowo wywoływanych metod. Jeśli jednak na którymkolwiek poziomie przechwytywaliśmy wyjątek, jego obsługa ulegała zakończeniu. Nie zawsze jest to korzystne zachowanie, czasami istnieje potrzeba, aby po wykonaniu naszego bloku obsługi obiekt wyjątku nie był niszczony, ale by był przekazywany dalej. Aby doprowadzić do takiego zachowania, musimy zastosować instrukcję `throw`. Schematycznie wyglądałoby to następująco:

```
try
{
    //instrukcje mogące spowodować wyjątek
}
catch(TypWyjątku identyfikatorWyjątku)
{
    //instrukcje obsługujące sytuację wyjątkową
    throw identyfikatorWyjątku
}
```

Na listingu 4.15 zostało przedstawione, jak taka sytuacja wygląda w praktyce. W bloku `try` jest wykonywana niedozwolona instrukcja dzielenia przez zero. W bloku `catch` najpierw wyświetlamy na ekranie informację o przechyceniu wyjątku, a następnie za pomocą instrukcji `throw` ponownie wyrzucamy (zgłaszymy) przechwycony już wyjątek. Ponieważ w programie nie ma już innego bloku `try...catch`, który mógłby przechwycić ten wyjątek, zostanie on obsłużony standardowo przez maszynę wirtualną. Dlatego też na ekranie zobaczymy widok zaprezentowany na rysunku 4.11.

Listing 4.15. Ponowne zgłoszenie wyjątku

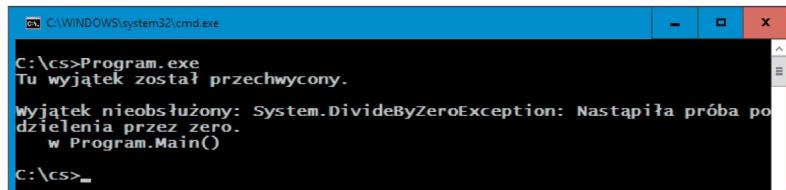
```
using System;

public class Program
{
    public static void Main()
    {
        int liczba1 = 10, liczba2 = 0;
        try
```

```
        {
            liczba1 = liczba1 / liczba2;
        }
    catch(DivideByZeroException e)
    {
        Console.WriteLine("Tu wyjątek został przechwycony.");
        throw e;
    }
}
```

Rysunek 4.11.

Ponowne zgłoszenie raz przechwyconego wyjątku



W przypadku zagnieżdżonych bloków try sytuacja wygląda analogicznie. Wyjątek przechwycony w bloku wewnętrznym i ponownie zgłoszony może być obsłużony w bloku zewnętrznym, w którym może być oczywiście zgłoszony kolejny raz itd. Zostało to zobrazowane w kodzie widocznym na listingu 4.16.

Listing 4.16. Wielokrotne zgłaszanie wyjątku

```
using System;

public class Program
{
    public static void Main()
    {
        int liczba1 = 10, liczba2 = 0;
        //tutaj dowolne instrukcje
        try
        {
            //tutaj dowolne instrukcje
            try
            {
                liczba1 = liczba1 / liczba2;
            }
            catch(ArithmeticeException e)
            {
                Console.WriteLine(
                    "Tu wyjątek został przechwycony pierwszy raz.");
                throw e;
            }
        }
        catch(ArithmeticeException e)
        {
            Console.WriteLine(
                "Tu wyjątek został przechwycony drugi raz.");
            throw e;
        }
    }
}
```

Mamy tu dwa zagnieżdżone bloki try. W bloku wewnętrznym zostaje wykonana nieprawidłowa instrukcja dzielenia przez zero. Zostaje ona w tym bloku przechwycona, a na ekranie wyświetlany jest komunikat o pierwszym przechwyceniu wyjątku. Następnie wyjątek jest ponownie zgłoszany. W bloku zewnętrznym następuje drugie przechwycenie, wyświetlenie drugiego komunikatu oraz kolejne zgłoszenie wyjątku. Ponieważ nie istnieje trzeci blok try...catch, ostatecznie wyjątek jest obsługiwany przez maszynę wirtualną, a po uruchomieniu programu zobaczymy widok zaprezentowany na rysunku 4.12.

Rysunek 4.12.

Przechwytywanie
i ponowne
zgłoszanie
wyjątków

The screenshot shows a Windows Command Prompt window titled 'cmd.exe' with the path 'C:\WINDOWS\system32'. The window contains the following text:

```
C:\cs>Program.exe
Tu wyjątek został przechwycony pierwszy raz.
Tu wyjątek został przechwycony drugi raz.
Wyjątek nieobsłużony: System.DivideByZeroException: Nastąpiła próba po
dzielenia przez zero.
  w Program.Main()
C:\cs>_
```

Tworzenie własnych wyjątków

Programując w C#, nie musimy zdawać się na wyjątki zdefiniowane w klasach .NET. Nic bowiem nie stoi na przeszkodzie, aby tworzyć własne. Wystarczy, że napiszemy klasę pochodną pośrednio lub bezpośrednio od `Exception`. Klasa taka w najprostszej postaci będzie wyglądać tak:

```
public class nazwa_klasy : Exception
{
    //treść klasy
}
```

Przykładowo możemy utworzyć bardzo prostą klasę o nazwie `GeneralException` (ang. *general exception* — wyjątek ogólny) w postaci:

```
public class GeneralException : Exception
{
}
```

To w zupełności wystarczy. Nie musimy dodawać żadnych nowych pól i metod. Jest to pełnoprawna klasa obsługująca wyjątki, z której możemy korzystać w taki sam sposób jak ze wszystkich innych klas opisujących wyjątki. Na listingu 4.17 jest widoczna przykładowa klasa `Program` z metodą `Main` generującą wyjątek `GeneralException`.

Listing 4.17. Użycie własnej klasy do zgłoszenia wyjątku

```
using System;

public class GeneralException : Exception
{
}

public class Program
{
    public static void Main()
```

```

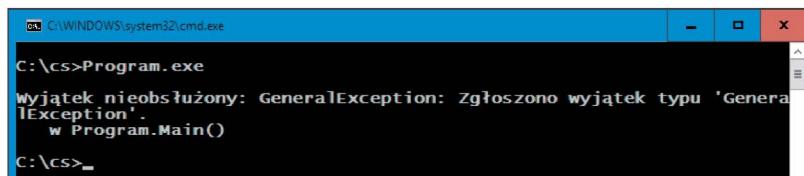
    {
        throw new GeneralException();
    }
}

```

Wyjątek jest tu zgłoszany za pomocą instrukcji `throw` dokładnie w taki sam sposób jak we wcześniejszych przykładach. Na rysunku 4.13 jest widoczny efekt działania takiego programu; widać, że faktycznie zgłoszony został wyjątek nowej klasy — `GeneralException`. Nic też nie stoi na przeszkodzie, aby obiekowi naszego wyjątku przekazać komunikat. Nie da się tego jednak zrobić, używając zaprezentowanej wersji klasy `GeneralException`. Odpowiednia modyfikacja będzie jednak dobrym ćwiczeniem do samodzielnego wykonania.

Rysunek 4.13.

Zgłaszenie własnych wyjątków



Wyjątki warunkowe

W C# 6.0 przechwytywanie wyjątków zostało uzupełnione o dodatkową możliwość, mianowicie przechwytywanie warunkowe. Dzięki temu można zdecydować, aby blok `catch` był wykonywany wtedy, gdy wystąpi wyjątek i jednocześnie spełniony jest pewien warunek. Jeżeli ten warunek nie będzie spełniony, dany blok `catch` zostanie pominięty (a więc kod zachowa się tak, jakby go nie było). Schemat takiej konstrukcji jest następujący:

```

try{
    //instrukcje mogące spowodować wyjątek
}
catch(TypWyjątku [id]) when (warunek){
    //instrukcje wykonywane, gdy wystąpi wyjątek i prawdziwy jest warunek
}

```

Przyjmijmy przykładowo, że napisaliśmy naszą własną klasę obsługi wyjątków, w której znalazło się pole określające wyjątek czy też jego status. Zakładając, że takich statusew mogłyby być najwyżej 256, mogłyby być one typu `byte`. Wtedy kod miałby postać przedstawioną na listing 4.18.

Listing 4.18. Klasa wyjątku z polem statusowym

```

using System;

public class ExceptionWithStatus : Exception{
    public byte status = 0;
    public ExceptionWithStatus(String msg, byte status) : base(msg){
        this.status = status;
    }
}

```

Klasa `ExceptionWithStatus` dziedziczy bezpośrednio po `Exception` oraz zawiera jedno publiczne pole typu `byte` oraz konstruktor. Konstruktor przyjmuje dwa argumenty: pierwszy to komunikat związany z wyjątkiem, a drugi to kod określający hipotetyczny status wyjątku (liczba od 0 do 255). Komunikat przekazywany jest do klasy bazowej, a wartość parametru `status` trafia do pola o takiej samej nazwie (korzystamy przy tym ze składni ze słowem `this`).

Teraz skoro każdy wyjątek typu `ExceptionWithStatus` będzie zawierał pole statusowe o konkretnej wartości przekazywanej w konstruktorze, to wyjątki takie będzie można łatwo filtrować w blokach `catch`. Dany blok może być wykonany w zależności od tego czy pole `status` ma zadaną wartość. Program ilustrujący to zagadnienie został przedstawiony na listingu 4.19.

Listing 4.19. Wykonanie bloku `catch` uzależnione od warunku

```
using System;

public class Program
{
    public void dajWyjatek(String msg, short status){
        throw new ExceptionWithStatus(msg, status);
    }
    public static void Main()
    {
        Program pr = new Program();
        try
        {
            pr.dajWyjatek("Mój wyjątek 1", 0);
        }
        catch(ExceptionWithStatus e) when (e.status == 0)
        {
            Console.WriteLine("Wygenerowano wyjątek: " + e.Message);
        }
        Console.WriteLine("Po pierwszym wyjątku.");
        try
        {
            pr.dajWyjatek("Mój wyjątek 2", 1);
        }
        catch(ExceptionWithStatus e) when (e.status == 0)
        {
            Console.WriteLine("Wygenerowano wyjątek: " + e.Message);
        }
        Console.WriteLine("Po drugim wyjątku.");
    }
}
```

W klasie `Program` została zdefiniowana metoda `dajWyjatek`. Jedynym jej zadaniem jest wygenerowanie wyjątku typu `ExceptionWithStatus`. Przyjmuje ona dwa argumenty: komunikat oraz `status`, które zostaną użyte w konstruktorze klasy `ExceptionWithStatus`. W metodzie `Main` powstaje jeden obiekt typu `Program`, a następnie dwukrotnie wywoływana jest jego metoda `dajWyjatek`. W obu przypadkach wywołanie jest objęte blokiem `try` wraz z warunkowym blokiem `catch`, przy czym blok `catch` jest wywoływany w sytuacji, gdy wartość pola `status` obiektu wyjątku jest równa zero. W innych przypadkach wyjątek nie będzie obsłużony.

Dlatego też pierwsze wywołanie (`pr.dajWyjatek("Mój wyjątek 1", 0)`) jest obsługiwane i nie powoduje przerwania wykonywania programu (status wyjątku jest bowiem równy 0), ale drugie wywołanie (`pr.dajWyjatek("Mój wyjątek 2", 1)`) powoduje powstanie nieprzechwyconego wyjątku (bo tym razem status jest równy 1). A zatem po skompilowaniu i uruchomieniu programu w konsoli pojawi się widok zaprezentowany na rysunku 4.14.

Rysunek 4.14.
Efekt warunkowego
przechwytywania
wyjątków

Sekcja finally

Do bloku `try` można dołączyć sekcję `finally`, która będzie wykonana zawsze, niezależnie od tego, co będzie się działo w bloku `try`. Schematycznie taka konstrukcja wygląda następująco:

```
try
{
    //instrukcje mogące spowodować wyjątek
}
catch(TypWyjątku)
{
    //instrukcje sekcji catch
}
finally
{
    //instrukcje sekcji finally
}
```

O tym, że instrukcje sekcji `finally` są wykonywane zawsze, niezależnie od tego, czy w bloku `try` wystąpi wyjątek, czy nie, można przekonać się dzięki przykładowi wiadocznemu na listingu 4.20.

Listing 4.20. Użycie sekcji `finally`

```
using System;

public class Dzielenie
{
    public static double Podziel(int liczba1, int liczba2)
    {
        if(liczba2 == 0)
            throw new DivideByZeroException(
                "Dzielenie przez zero: " + liczba1 + "/" + liczba2
            );
        return liczba1 / liczba2;
    }
}
```

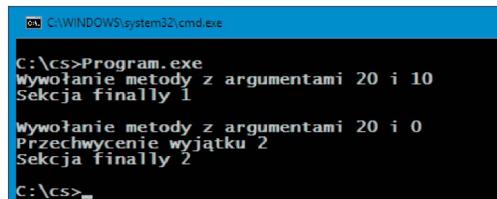
```
public class Program
{
    public static void Main()
    {
        double wynik;
        try
        {
            Console.WriteLine("Wywołanie metody z argumentami 20 i 10");
            wynik = Dzielenie.Podziel(20, 10);
        }
        catch(DivideByZeroException)
        {
            Console.WriteLine("Przechwycenie wyjątku 1");
        }
        finally
        {
            Console.WriteLine("Sekcja finally 1");
        }
        try
        {
            Console.WriteLine("\nWywołanie metody z argumentami 20 i 0");
            wynik = Dzielenie.Podziel(20, 0);
        }
        catch(DivideByZeroException){
            Console.WriteLine("Przechwycenie wyjątku 2");
        }
        finally
        {
            Console.WriteLine("Sekcja finally 2");
        }
    }
}
```

Jest to znana nam klasa `Dzielenie` ze statyczną metodą `Podziel`, wykonującą dzielenie przekazanych jej argumentów. Tym razem metoda `Podziel` pozostała bez zmian w stosunku do wersji z listingu 4.14, czyli zgłasza błąd `DivideByZeroException`. Zmodyfikowana została natomiast metoda `Main` z klasy `Program`. Oba wywołania metody zostały ujęte w bloki `try...catch...finally`. Pierwsze wywołanie nie powoduje powstania wyjątku, nie jest więc wykonywany pierwszy blok `catch`, ale jest wykonywany pierwszy blok `finally`. Tym samym na ekranie pojawi się napis `Sekcja finally 1`.

Drugie wywołanie metody `Podziel` powoduje wygenerowanie wyjątku, zostaną zatem wykonane zarówno instrukcje bloku `catch`, jak i `finally`. Na ekranie pojawią się więc dwa napisy: `Przechwycenie wyjątku 2` oraz `Sekcja finally 2`. Ostatecznie wynik działania całego programu będzie taki jak ten zaprezentowany na rysunku 4.15.

Rysunek 4.15.

Blok `finally` jest wykonywany niezależnie od tego, czy pojawi się wyjątek, czy nie



Sekcję `finally` można zastosować również w przypadku instrukcji, które nie powodują wygenerowania wyjątku. Stosuje się wtedy instrukcję `try...finally` w postaci:

```
try
{
    //instrukcje
}
finally
{
    //instrukcje
}
```

Działanie jest takie samo jak w przypadku bloku `try...catch...finally`, to znaczy kod z bloku `finally` będzie wykonany zawsze, niezależnie od tego, jakie instrukcje znajdują się w bloku `try`. Na przykład nawet jeśli w bloku `try` znajdzie się instrukcja `return` lub zostanie wygenerowany wyjątek, blok `finally` i tak będzie wykonany. Zobrazowano to w przykładzie pokazanym na listingu 4.21.

Listing 4.21. Zastosowanie sekcji `try...finally`

```
using System;

public class Program
{
    public int f1()
    {
        try
        {
            return 0;
        }
        finally
        {
            Console.WriteLine("Sekcja finally f1");
        }
    }
    public void f2()
    {
        try
        {
            int liczba1 = 10, liczba2 = 0;
            liczba1 = liczba1 / liczba2;
        }
        finally
        {
            Console.WriteLine("Sekcja finally f2");
        }
    }
    public static void Main()
    {
        Program pr = new Program();
        pr.f1();
        pr.f2();
    }
}
```

W metodzie f1 znajduje się instrukcja return zwracająca wartość 0. Wiadomo, że powoduje ona zakończenie działania metody. Ponieważ jednak instrukcja została ujęta w blok try...finally, zostanie również wykonany kod znajdujący się w bloku finally. Podobną konstrukcję ma metoda f2. W bloku try zawarte są instrukcje, które powodują powstanie dzielenia przez 0. Jest to równoznaczne z wygenerowaniem wyjątku i przerwaniem wykonywania kodu metody. Ponieważ jednak w sekcji finally znajduje się instrukcja wyświetlająca napis na ekranie, to zostanie ona wykonana niezależnie od tego, czy wyjątek wystąpi, czy nie.

W metodzie main tworzony jest nowy obiekt klasy Program, a następnie wywoływane są jego metody f1 i f2. Spowoduje to wyświetlenie na ekranie napisów Sekcja finally f1 i Sekcja finally f2. Dzięki temu można się przekonać, że instrukcje bloku finally faktycznie są wykonywane zawsze, niezależnie od tego, co zdarzy się w bloku try.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 23.1

Napisz klasę Program, w której zostaną zadeklarowane metody f i Main. W metodzie f napisz dowolną instrukcję generującą wyjątek NullReferenceException. W Main wywołaj metodę f i przechwyć wyjątek za pomocą bloku try...catch.

Ćwiczenie 23.2

Zmodyfikuj kod z listingu 4.16 tak, aby był generowany, przechwytywany i ponownie zgłoszany wyjątek IndexOutOfRangeException.

Ćwiczenie 23.3

Napisz klasę wyjątku o nazwie NegativeValueException oraz klasę Program, która będzie z niego korzystać. W klasie Program napisz metodę o nazwie Odejmij przyjmującą dwa argumenty typu int. Metoda f powinna zwracać wartość będącą wynikiem odejmowania argumentu pierwszego od drugiego. Jednak w przypadku, gdyby wynik ten był ujemny, powinien zostać zgłoszony wyjątek NegativeValueException. Dopisz metodę Main, która przetestuje działanie metody Odejmij.

Ćwiczenie 23.4

Napisz taką wersję klasy GeneralException, aby obiekowi wyjątku można było przekazać dowolny komunikat. Następnie zmodyfikuj program z listingu 4.17, tak aby korzystał z tej możliwości.

Ćwiczenie 23.5

Przygotuj taką wersję ćwiczenia 18.4 z rozdziału 3. (lekcia 18.), w której do sygnalizacji błędnych parametrów używana jest technika wyjątku. Osobny wyjątek powinien

być generowany, gdy wartość sinusa wykracza poza dopuszczalny zakres $<-1, 1>$, a osobny, gdy podana odległość jest ujemna. Wyjątki powinny zawierać stosowne komunikaty informujące o wartości błędnych argumentów.

Ćwiczenie 23.6

Napisz prosty program ilustrujący działanie klas z ćwiczenia 23.5.

Rozdział 5.

System wejścia-wyjścia

Do tworzenia aplikacji w C# niezbędna jest znajomość przynajmniej podstaw obsługi systemu wejścia-wyjścia. Właśnie tej tematyce jest poświęcony rozdział 6. W czterech kolejnych lekcjach zostanie wyjaśnione, jak obsługiwać standardowe wejście, czyli odczytywać dane wprowadzane z klawiatury, jak wykonywać operacje na systemie plików oraz jak zapisywać i odczytywać zawartość plików. Będzie omówione wprowadzanie do aplikacji tekstu i liczb, tworzenie i usuwanie katalogów, pobieranie informacji o plikach, takich jak długość czy czas utworzenia, a także zapisywanie w plikach danych binarnych i tekstowych.

Lekcja 24. Ciągi znaków

Lekcja 24. poświęcona jest obiektom typu `string` reprezentującym ciągi znaków. Przedstawione zostaną m.in. różnice między znakiem a ciągiem znakowym, sposoby wyświetlania takich danych na ekranie, a także jakie znaczenie ma w tych przypadkach operator dodawania. Pokazany będzie sposób traktowania sekwencji specjalnych oraz konwersje napisów na wartości liczbowe. Nie będą też pominięte sposoby formatowania ciągów tak, by przyjmowały pożądaną postać. Na końcu lekcji znajdą się informacje o metodach przetwarzających dane typu `string`, w tym o wyszukiwaniu i wyodrębnianiu fragmentów ciągów.

Znaki i łańcuchy znakowe

W rozdziale 2., w lekcji 4., przedstawione zostały typy danych dostępne standardowo w C#. Wśród nich znalazły się `char` oraz `string`. Pierwszy z nich służy do reprezentowania znaków, a drugi — ciągów znaków, inaczej mówiąc, łańcuchów znakowych. Ciąg czy też łańcuch znakowy to po prostu uporządkowana sekwencja znaków. Zwykle jest to napis, których chcemy w jakiś sposób zaprezentować na ekranie. Takie napisy były używane już wielokrotnie w rozmaitych przykładach.

Jeżeli w kodzie programu chcemy umieścić ciąg znaków, np. przypisać go zmiennej, ujmujemy go w cudzysłów prosty:

"To jest napis"

Taki ciąg może być przypisany zmiennej, np.:

```
string napis = "To jest napis";
```

To oznacza, że jeśli chcemy coś wyświetlić na ekranie, nie musimy umieszczać napisu bezpośrednio w wywołaniu metody `WriteLine` klasy `Console`, tak jak miało to miejsce w dotychczas prezentowanych przykładach. Można posłużyć się też zmienną (zmiennymi) pomocniczą, np. w taki sposób, jaki został zaprezentowany na listingu 5.1.

Listing 5.1. Ciąg znaków umieszczony w zmiennej

```
using System;

public class Program
{
    public static void Main()
    {
        string napis1 = "To jest ";
        string napis2 = "przykładowy napis.";
        Console.WriteLine(napis1);
        Console.WriteLine(napis2);
        Console.WriteLine(napis1 + napis2);
    }
}
```

W kodzie znajdują się dwie zmienne typu `string`: `napis1` i `napis2`. Każdej z nich przypisano osobny łańcuch znaków. Następnie za pomocą metod `Write` i `WriteLine` zawartość obu zmiennych została wyświetlona na ekranie w jednym wierszu, dzięki czemu powstało pełne zdanie. Ostatnia instrukcja również powoduje wyświetlenie jednego wiersza tekstu składającego się z zawartości zmiennych `napis1` i `napis2`, ale do połączenia łańcuchów znakowych został w niej użyty operator `+`.

W programach można też umieszczać pojedyncze znaki, czyli tworzyć dane typu `char`. Zgodnie z opisem podanym w lekcji 4. w takim przypadku symbol znaku należy ująć w znaki apostrofu prostego, np. zapis:

```
'a'
```

oznacza małą literę `a`. Może być ona przypisana zmiennej znakowej typu `char`, np.:

```
char znak = 'a';
```

Pojedyncze znaki zapisane w zmiennych również mogą być wyświetlane na ekranie w standardowy sposób. Przykład został zaprezentowany na listingu 5.2.

Listing 5.2. Wyświetlanie pojedynczych znaków

```
using System;

public class Program
{
    public static void Main()
    {
        char znak1 = 'Z';
        char znak2 = 'n';
        char znak3 = 'a';
        char znak4 = 'k';
        Console.Write(znak1);Console.Write(znak2);
        Console.Write(znak3);Console.Write(znak4);
    }
}
```

Kod jest bardzo prosty. Powstały cztery zmienne typu char, którym przypisano cztery różne znaki. Następnie zawartość zmiennych została wyświetlona na ekranie za pomocą metody Write. Dzięki temu poszczególne znaki znajdą się obok siebie, tworząc tekst Znak.

W tym miejscu warto się zastanowić, czy można by użyć konstrukcji z operatorem +, analogicznej do przedstawionej na listingu 5.1. Co by się stało, gdyby w kodzie pojawiła się instrukcja w postaci:

```
Console.WriteLine(znak1 + znak2 + znak3 + znak4);
```

W pierwszej chwili może się wydawać, że pojawi się również napis Znak. To jednak nieprawda. Efektem działania byłaby wartość 404. Można się o tym łatwo przekonać, umieszczając powyższą instrukcję w programie z listingu 5.2. Dlaczego tak by się stało i skąd wzięłaby się ta liczba? Trzeba najpierw przypomnieć sobie, czym tak naprawdę są dane typu char (zostało to wyjaśnione w lekcji 4. przy opisie tego typu). Są to po prostu 16-bitowe kody liczbowe określające znaki. Znak Z ma kod 90, znak n — 110, znak a — 97, znak k — 107. W sumie daje to wartość 404. A zatem w opisywanej instrukcji najpierw zostały wykonane dodawanie całkowitoliczbowe, a następnie uzyskana wartość została wyświetlona na ekranie.

Takie dodawanie mogłoby też zostać wykonane bezpośrednio, np.:

```
Console.WriteLine('Z' + 'n' + 'a' + 'k');
```

Co więcej, jego wynik można zapisać w zmiennej typu int, np.:

```
int liczba = 'Z' + 'n' + 'a' + 'k';
```

Wbrew pozorom jest to logiczne. Skoro pojedyncza dana typu char jest tak naprawdę liczbą (kodem) pewnego znaku, to dodawanie tych danych jest w istocie dodawaniem liczb. Oczywiście to kwestia interpretacji i decyzji twórców danego języka programowania. Można sobie wyobrazić również inne rozwiązanie tej kwestii, np. automatyczne tworzenie łańcucha znakowego z tak dodawanych znaków, niemniej w C# (a także w wielu innych językach programowania) stosowane jest dodawanie arytmetyczne.

Zupełnie inaczej będzie, jeśli pojedynczy znak ujmiemy w cudzysłów. Cudzysłów oznacza ciąg (łańcuch) znaków, nie ma przy tym znaczenia ich liczba. Pisząc:

"a"

tworzymy ciąg znaków zawierający jeden znak a. Z kolei dodawanie ciągów (z użyciem operatora +) znaków powoduje ich łączenie (czyli konkatenację). W rezultacie powstanie ciąg wynikowy będący złączeniem ciągów składowych. A zatem efektem działania:

"Z" + "n" + "a" + "k"

będzie ciąg znaków Znak. Różnice między dodawaniem znaków a dodawaniem ciągów znaków łatwo można zauważać, uruchamiając program z listingu 5.3. Na ekranie pojawią się wtedy dwa wiersze. W pierwszym znajdzie się wartość 404 (wynik dodawania znaków, a dokładniej ich kodów), a w drugim — napis Znak (wynik dodawania łańcuchów znakowych).

Listing 5.3. Dodawanie znaków i ciągów znaków

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine('Z' + 'n' + 'a' + 'k');
        Console.WriteLine("Z" + "n" + "a" + "k");
    }
}
```

W tym miejscu trzeba jeszcze dodatkowo zwrócić uwagę na kwestię, która została już wyżej wspomniana. Otóż ciąg znaków powstaje przy użyciu cudzysłowu, niezależnie od tego, ile znaków zostanie w nim faktycznie umieszczone. Dlatego w przykładzie z listingu 5.3 można było użyć ciągów znaków zawierających jeden znak. Skoro jednak liczba nie ma znaczenia, to można skonstruować ciąg znaków niezawierający żadnych znaków — zawierający 0 znaków. Choć może się to wydawać dziwną konstrukcją, w praktyce programistycznej jest to często stosowane. Mówimy wtedy o **pustym ciągu znaków**, który zapisuje się w następujący sposób:

""

Taki ciąg może być przypisany dowolnej zmiennej typu **string**, np.:

```
string str = "";
```

Widząc taką instrukcję, powiemy, że zmiennej **str** został przypisany pusty ciąg znaków i że zmieniona ta zawiera pusty ciąg znaków.

Znaki specjalne

Dana typu `char` musi przechowywać dokładnie jeden znak, nie oznacza to jednak, że między znakami apostrofu wolno umieścić tylko jeden symbol. Określenie znaku może składać się z kilku symboli — są to sekwencje specjalne przedstawione w tabeli 2.3, w lekcji 4. (rozdział 2.), rozpoczynające się od lewego ukośnika `\`. Można zatem użyć np. następującej instrukcji:

```
char znak = '\n';
```

Spowoduje ona przypisanie znaku nowego wiersza zmiennej `znak`. Z kolei efektem działania instrukcji:

```
char znak = '\x0061';
```

będzie zapisanie w zmiennej `znak` małej litery `a` (0061 to szesnastkowy kod tej litery).

Sekwencje specjalne mogą być też używane w łańcuchach znakowych. Warto w tym miejscu przypomnieć, że skorzystanie z apostrofu w zmiennej typu `char` lub cudzysłów w zmiennej typu `string` jest możliwe tylko dzięki takim sekwencjom. Niedopuszczalny jest zapis typu:

...

lub:

...

gdyż kompilator nie mógłby ustalić, które symbole tworzą znaki, a które wyznaczają początek i koniec danych. Sposób użycia sekwencji specjalnych do zbudowania napisów został zilustrowany w programie zaprezentowanym na listingu 5.4. W wyniku jego działania na ekranie pojawi się widok zaprezentowany na rysunku 5.1¹.

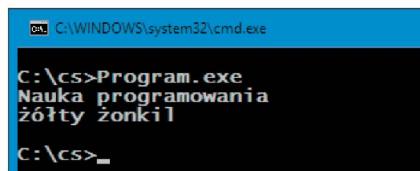
Listing 5.4. Zastosowanie sekwencji specjalnych

```
using System;

public class Program
{
    public static void Main()
    {
        string str1 = "\x004e\x0061\x0075\x006b\x0061\x0020";
        string str2 = "\x0070\x0072\x006f\x0067\x0072\x0061";
        string str3 = "\x006d\x006f\x0077\x0061\x006e\x0069\x0061";
        string str4 = "\u017c\u00f3\u0142\u0074\u0079\u0020";
        string str5 = "\u017c\u006f\u006e\u006b\u0069\u006c";
        Console.WriteLine(str1 + str2 + str3);
        Console.WriteLine(str4 + str5);
    }
}
```

¹ Przy założeniu, że aktywną stroną kodową konsoli jest 852, 1250 lub 65001. To ustawienie można zmienić za pomocą systemowego polecenia `chcp`, pisząc np. `chcp 1250`.

Rysunek 5.1.
Efekt działania programu z listingu 5.4



W kodzie zostało zadeklarowanych pięć zmiennych typu `string`. Trzy pierwsze zawierają kody znaków ASCII w postaci szesnastkowej, natomiast czwarta i piąta — kody znaków w standardzie Unicode. Pierwsza instrukcja `Console.WriteLine` powoduje wyświetlenie połączonej zawartości zmiennych `str1`, `str2` i `str3`, natomiast druga — zawartości zmiennych `str4` i `str5`. Tym samym po uruchomieniu aplikacji na ekranie pojawią się dwa wiersze tekstu, takie jak na rysunku 5.1. Użyte kody znaków składają się bowiem na dwa przykładowe napisy: `Nauka programowania` oraz `żółty żonkil`.

Zamiana ciągów na wartości

Ciągi znaków mogą reprezentować różne wartości innych typów, np. liczby całkowite lub rzeczywiste zapisywane w różnych notacjach. Czasem niezbędne jest więc przetworzenie ciągu znaków reprezentującego daną liczbę na wartość konkretnego typu, np. `int` lub `double`. W tym celu można użyć klasy `Convert` i udostępnianych przez nią metod. Metody te zostały zebrane w tabeli 5.1.

Tabela 5.1. Wybrane metody klasy `Convert`

Metoda	Opis
<code>ToBoolean</code>	Konwersja na typ <code>bool</code>
<code>ToByte</code>	Konwersja na typ <code>byte</code>
<code>ToChar</code>	Konwersja na typ <code>char</code>
<code>ToDecimal</code>	Konwersja na typ <code>decimal</code>
<code>ToDouble</code>	Konwersja na typ <code>double</code>
<code>ToInt16</code>	Konwersja na typ <code>short</code>
<code>ToInt32</code>	Konwersja na typ <code>int</code>
<code>ToInt64</code>	Konwersja na typ <code>long</code>
<code>ToSByte</code>	Konwersja na typ <code>sbyte</code>
<code>ToUInt16</code>	Konwersja na typ <code>ushort</code>
<code>ToUInt32</code>	Konwersja na typ <code>uint</code>
<code>ToUInt64</code>	Konwersja na typ <code>ulong</code>

Ciąg podlegający konwersji należy umieścić w argumencie wywołania, np.:

```
int liczba = Convert.ToInt32("20");
```

W przypadku konwersji na typy całkowitoliczbowe dopuszczalne jest użycie drugiego argumentu określającego podstawę systemu liczbowego, np. dla systemu szesnastkowego:

```
int liczba = Convert.ToInt32("20", 16);
```

Rozpoznawane podstawy systemów liczbowych to 2 (dwójkowy, binarny), 8 (ósemkowy, oktalny), 10 (dziesiętny, decymalny), 16 (szesnastkowy, heksadecymalny). Użycie innej podstawy spowoduje wygenerowanie wyjątku `ArgumentException`.

Jeżeli przekazany ciąg znaków nie będzie zawierał wartości we właściwym formacie (np. będzie zawierał same litery, a konwersja będzie się miała odbywać dla systemu dziesiętnego), powstanie wyjątek `FormatException`. Jeśli natomiast konwertowana wartość będzie wykraczała poza dopuszczalny zakres dla danego typu, będzie wygenerowany wyjątek `OverflowException`. Przykłady kilku konwersji zostały przedstawione w kodzie widocznym na listingu 5.5.

Listing 5.5. Przykłady konwersji przy użyciu klasy `Convert`

```
using System;

public class Program
{
    public static void Main()
    {
        int liczba1 = Convert.ToInt32("10", 2);
        int liczba2 = Convert.ToInt32("10", 8);
        int liczba3 = Convert.ToInt32("10", 10);
        int liczba4 = Convert.ToInt32("10", 16);

        double liczba5 = Convert.ToDouble("1,4e1");

        Console.Write("10 w różnych systemach liczbowych: ");
        Console.WriteLine("{0}, {1}, {2}, {3}",
            liczba1, liczba2, liczba3, liczba4);
        Console.WriteLine("liczba5 (1.4e1) = " + liczba5);

        try
        {
            int liczba6 = Convert.ToByte("-10");
        }
        catch(OverflowException)
        {
            Console.Write("Convert.ToByte(\"-10\"): ");
            Console.WriteLine("przekroczyły zakres danych");
        }
        try
        {
            double liczba7 = Convert.ToDouble("abc");
        }
        catchFormatException)
        {
            Console.Write("Convert.ToDouble(\"abc\"): ");
            Console.WriteLine("nieprawidłowy format danych");
        }
    }
}
```

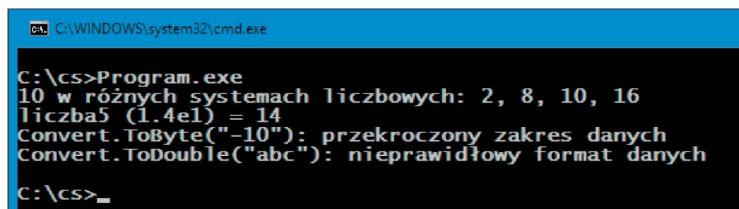
Na początku tworzymy cztery zmienne typu `int`, którym przypisujemy wynik działania metody `ToInt32` przetwarzającej串 znaków 10 na liczbę typu `int`. Przy każdym wywołaniu stosowany jest inny drugi argument, dzięki czemu konwersja odbywa się

na podstawie różnych systemów liczbowych (dwójkowego, ósemkowego, dziesiętnego i szesnastkowego). Dzięki temu będzie można się przekonać, jak wartość reprezentowana przez ciąg 10 wygląda w każdym z systemów.

Wykonywana jest również konwersja ciągu 1,4e1 na wartość typu `double`. Ponieważ taki ciąg oznacza liczbę opisaną działaniem $1,4 \times 10^1$, powstanie w ten sposób wartość 14 (przypisywana zmiennej `liczba5`). Wszystkie te konwersje są prawidłowe, a otrzymane wartość zostaną wyświetcone na ekranie za pomocą metod `Write` i `WriteLine`.

W dalszej części kodu znalazły się instrukcje nieprawidłowe, generujące wyjątki przechwytywane w blokach `try...catch`. Pierwsza z nich to próba dokonania konwersji ciągu -10 do wartości typu `byte`. Nie jest to możliwe, gdyż typ `byte` pozwala na reprezentację liczb od 0 do 255. Dlatego też zgodnie z opisem podanym wyżej wywołanie metody `ToByte` spowoduje wygenerowanie wyjątku `OverflowException`. W drugiej instrukcji podejmowana jest próba konwersji ciągu abc do wartości typu `double`. Ponieważ jednak taki ciąg nie reprezentuje żadnej wartości liczbowej (w systemie dziesiętnym), w tym przypadku powstanie wyjątku `FormatException`. Ostatecznie po komplikacji i uruchomieniu programu zostaną wyświetcone na ekranie komunikaty przedstawione na rysunku 5.2.

Rysunek 5.2.
Efekty działania programu konwertującego ciągi znaków na wartości liczbowe



```
C:\>Program.exe
10 w różnych systemach liczbowych: 2, 8, 10, 16
liczba5 (1.4e1) = 14
Convert.ToByte("-10"): przekroczony zakres danych
Convert.ToDouble("abc"): nieprawidłowy format danych
C:\>
```

Formatowanie danych

W lekcji 6. z rozdziału 2. podany był sposób na umieszczanie w wyświetlonym napisie wartości wstawianych w konkretne miejsca ciągu znakowego. Numery poszczególnych parametrów należało ująć w nawias klamrowy. Schemat takiej konstrukcji był następujący:

```
Console.WriteLine("zm1 = {0}, zm2 = {1}", zm1, zm2);
```

Liczba stosowanych parametrów nie była przy tym ograniczona, można było stosować ich dowolnie wiele. Taki zapis może być jednak uzupełniony o specyfikatory formatów. Wtedy numer parametru uzupełnia się o ustalenie formatu określającego sposób wyświetlania (interpretacji) danych, schematycznie:

```
{numer_parametru[,[-]wypełnienie]:specyfikator_formatu[precyza]}
```

Dostępne specyfikatory zostały przedstawione w tabeli 5.2.

Tabela 5.2. Specyfikatory formatów dostępne w C#

Specyfikator	Znaczenie	Obsługiwane typy danych	Przykład
C lub c	Traktowanie wartości jako walutowej	Wszystkie numeryczne	10,02 zł
D lub d	Traktowanie wartości jako dziesiętnej	Tylko całkowite	10
E lub e	Traktowanie wartości jako rzeczywistej w notacji wykładniczej z domyślną precyzją 6 znaków	Wszystkie numeryczne	1,25e+002
F lub f	Traktowanie wartości jako rzeczywistej (z separatorem dziesiętnym)	Wszystkie numeryczne	3,14
G lub g	Zapis rzeczywisty lub wykładniczy, w zależności od tego, który będzie krótszy	Wszystkie numeryczne	3,14
N lub n	Format numeryczny z separatorami grup dziesiętnych	Wszystkie numeryczne	1 200,33
P lub p	Format procentowy	Wszystkie numeryczne	12,00%
R lub r	Tworzy ciąg, który może być ponownie przetworzony na daną wartość	float, double, BigInteger	12,123456789
X lub x	Wartość będzie wyświetlona jako szesnastkowa	Tylko całkowite	7A

Numer parametru określa to, która dana ma być podstawiona pod dany parametr; wypełnienie specyfikuje preferowaną długość ciągu wynikowego dla bieżącej danej — brakujące miejsca zostaną wypełnione spacjami. Domyślnie spacje dodawane są z prawej strony; jeżeli mają być dodane z lewej, należy dodatkowo użyć znaku –. Opcje dotyczących wypełnienia nie trzeba jednak stosować, są opcjonalne. Opcjonalna jest również precyzja, czyli określenie całkowitej liczby znaków, które mają być użyte do wyświetlenia wartości. Jeżeli w wartości występuje mniej cyfr, niż określa to parametr *precyzja*, do wynikowego ciągu zostaną dodane zera.

Ponieważ sam opis może nie być do końca jasny, najlepiej w praktyce zobaczyć, jak zachowują się rozmaite specyfikatory formatów. Odpowiedni przykład został przedstawiony na listingu 5.6, a efekt jego działania — na rysunku 5.3.

Listing 5.6. Korzystanie ze specyfikatorów formatów

```
using System;

public class Program
{
    public static void Main()
    {
        int liczba1 = 12;
        double liczba2 = 254.28;

        Console.WriteLine("|{0:D}|", liczba1);
        Console.WriteLine("|{0,4:D}|", liczba1);
        Console.WriteLine("|{0,-4:D}|", liczba1);
        Console.WriteLine("|{0,-6:D4}|", liczba1);
```

```

Console.WriteLine("|{0:F}|", liczba2);
Console.WriteLine("|{0,8:F}|", liczba2);
Console.WriteLine("|{0,-8:F}|", liczba2);
Console.WriteLine("|{0,-10:F4}|", liczba2);

Console.WriteLine("|{0:E3}|", liczba2);
Console.WriteLine("|{0:P}|", liczba1);
Console.WriteLine("|{0,12:C}|", liczba2);
}
}

```

Rysunek 5.3.

*Wyświetlanie liczb
w różnych formatach*

```

C:\>Program.exe
[112]
| 12 |
|12 |
|0012 |
|254,28|
| 254,28 |
|254,28 |
|254,2800 |
|2,543E+002 |
| 200,00% |
|    254,28 z{}|
C:\>_

```

Przetwarzanie ciągów

Ciąg znaków umieszczony w kodzie programu jest obiektem typu `string`. A zatem zapis:

```
string str = "abc";
```

oznacza powstanie obiektu typu `string` zawierającego sekwencję znaków abc i przypisanie odniesienia do tego obiektu zmiennej `str`. Konsekwencją tego jest możliwość używania metod i właściwości dostępnych dla typu `string`. Dotyczy to zarówno zmiennych typu `string`, jak i bezpośrednio ciągów ujętych w znaki cudzysłowni. Bezpośrednio dostępna jest jedna właściwość: `Length`. Określa ona całkowitą długość ciągu (liczbę znaków). A zatem przy założeniu, że istnieje zmienna `str` zdefiniowana jak wyżej, użycie przykładowej instrukcji:

```
int ile = str.Length;
```

spowoduje przypisanie zmiennej `ile` wartości 3 (zmienna `str` zawiera bowiem串 skladający się z trzech znaków). Możliwe jest także odczytanie dowolnego znaku w ciągu. W tym celu używany jest tak zwany indekser. Wystarczy za zmienią lub literałem typu `string` w nawiasie prostokątnym (kwadratowym) umieścić indeks poszukiwanego znaku. Aby zatem uzyskać drugi znak zapisany w ciągu reprezentowanym przez `str` i umieścić go w zmiennej typu `char`, można napisać:

```
char znak = str[1];
```

Spowoduje to zapisanie w zmiennej `znak` znaku b (znaki są numerowane od 0, zatem aby uzyskać drugi z nich, należało użyć indeksu 1). Ponieważ literaly określające ciągi znaków stają się również obiektami, prawidłowe będą również następujące instrukcje:

```
int ile = "abc".Length;
char znak = "abc"[1];
```

Należy pamiętać, że w ten sposób można jedynie odczytywać znaki z ciągu. Zapis jest zabroniony (w C# ciągi znaków po utworzeniu są niezmienialne, ang. *immutable*). Na listingu 5.7 został przedstawiony prosty program korzystający z pętli for i wymienionych właściwości klasy `string` do odczytu pojedynczych znaków łańcucha i wyświetlenia ich na ekranie w osobnych wierszach.

Listing 5.7. Odczyt pojedynczych znaków łańcucha znakowego

```
using System;

public class Program
{
    public static void Main()
    {
        string str = "Przykładowy tekst";
        for(int i = 0; i < str.Length; i++)
        {
            Console.WriteLine(str[i]);
        }
    }
}
```

Metody klasy `string` pozwalają na wykonywanie na tekstach wielu różnorodnych operacji, takich jak przeszukiwanie, kopiowanie, łączenie, dzielenie, pobieranie podciągów i wiele innych. Pełną listę wraz z wszystkimi wariantami (wiele z metod ma po kilka przeciążonych wersji) można znaleźć w dokumentacji technicznej języka. Najważniejsze z nich zostały natomiast wymienione w tabeli 5.3.

Tabela 5.3. Wybrane metody dostępne dla typu `string`

Typ	Metoda	Opis
public static int	Compare(string strA, string strB)	Porównuje ciągi znaków strA i strB. Zwraca wartość mniejszą od 0, gdy strA < strB, wartość większą od zera, gdy strA > strB, oraz 0, jeśli strA jest równe strB.
public static string	Concat(string str0, string str1)	Zwraca串 będący połączeniem (konkatenacją) ciągów str0 i str1. Istnieją wersje przyjmujące trzy i cztery argumenty typu <code>string</code> , a także argumenty innych typów.
public bool	Contains(string str)	Sprawdza, czy w ciągu bieżącym występuje串 str. Jeśli występuje, zwraca true, jeśli nie — false.
public bool	EndsWith(string str)	Zwraca true, jeśli łańcuch kończy się ciągiem wskazanym przez argument str. W przeciwnym razie zwraca false.
public bool	Equals(string str)	Zwraca true, jeśli串 bieżący i串 wskazany przez argument str są takie same. W przeciwnym razie zwraca false.

Tabela 5.3. Wybrane metody dostępne dla typu *string* — ciąg dalszy

Typ	Metoda	Opis
public int	IndexOf(string str)	Zwraca indeks pierwszego wystąpienia w łańcuchu ciągu wskazanego przez argument <i>str</i> lub wartość <i>-1</i> , jeśli taki ciąg nie występuje w łańcuchu. Jeżeli zostanie użyty argument <i>indeks</i> , przeszukiwanie rozpocznie się od znaku o wskazanym indeksie.
	IndexOf(string str, int indeks)	
public string	Insert(int indeks, string str)	Wstawia do łańcucha ciąg <i>str</i> w miejscu wskazywanym przez argument <i>indeks</i> . Zwraca串 wynikowy.
public static bool	IsNullOrEmpty(string str)	Zwraca <i>true</i> , jeśli <i>str</i> zawiera pusty串 znaków lub wartość <i>null</i> .
public static string	Join(string separator, string[] arr)	Łączy ciągi pobrane z tablicy <i>arr</i> , wstawiając między poszczególne elementy znaki separatora. Zwraca串 wynikowy.
public int	LastIndexOf(string str)	Zwraca indeks ostatniego wystąpienia ciągu <i>str</i> w bieżącym łańcuchu lub wartość <i>-1</i> , jeżeli ciąg <i>str</i> nie zostanie znaleziony.
public string	Replace(string old, string new)	Zwraca串, w którym wszystkie wystąpienia ciągu <i>old</i> zostały zamienione na串 <i>new</i> .
public string[]	Split(char[] separator)	Zwraca tablicę podciągów bieżącego łańcucha wyznaczanych przez znaki zawarte w tablicy <i>separator</i> . Jeżeli zostanie użyty argument <i>ile</i> , zwrocona liczba podciągów będzie ograniczona do wskazywanej przez niego wartości.
public bool	StartsWith(string value)	Zwraca <i>true</i> , jeśli bieżący łańcuch zaczyna się od ciągu wskazywanego przez argument <i>str</i> . W przeciwnym razie zwraca <i>false</i> .
public string	Substring(int indeks, int ile)	Zwraca podciąg rozpoczynający się od znaku wskazywanego przez argument <i>indeks</i> o liczbie znaków określonej przez argument <i>ile</i> .
public string	ToLower()	Zwraca串, w którym wszystkie litery zostały zamienione na małe.
public string	ToUpper()	Zwraca串, w którym wszystkie litery zostały zamienione na wielkie.
public string	Trim()	Zwraca串, w którym z początku i końca zostały usunięte białe znaki (spacje, tabulatory itp.).

Warto zwrócić uwagę, że żadna z metod nie zmienia oryginalnego ciągu. Jeżeli w wyniku działania metody ma powstać modyfikacja łańcucha znakowego, zawsze tworzony jest nowy łańcuch (zawierający modyfikację) i jest on zwracany jako rezultat działania metody. Przyjrzyjmy się więc bliżej działaniu niektórych, często używanych metod z tabeli 5.3.

Metoda `concat` jest statyczna i zwraca ciąg będący połączeniem wszystkich ciągów przekazanych w postaci argumentów. Może przyjmować od dwóch do czterech takich argumentów. Czyli przykładowe wywołania:

```
string str1 = string.Concat("abc", "123");
string str2 = string.Concat("abc", "123", "def");
```

Spowodują przypisanie zmiennej `str1` ciągu `abc123`, a zmiennej `str2` ciągu `abc123def`.

Metoda `IndexOf` pozwala ustalić, czy w ciągu istnieje dany podciąg, a zatem sprawdza, czy w ciągu podstawowym istnieje inny ciąg, wskazany za pomocą argumentu. Jeżeli istnieje, zwracany jest indeks wystąpienia, jeśli nie — wartość `-1`. To oznacza, że instrukcja:

```
int indeks = "piękna Łąka".IndexOf("na");
```

spowoduje przypisanie zmiennej `indeks` wartości `4` — ponieważ ciąg `na` w ciągu `piękna Łąka` zaczyna się w pozycji o indeksie `4` (`p` — `0`, `i` — `1`, `ę` — `2`, `k` — `3`, `n` — `4`). Z kolei instrukcja:

```
int indeks = "piękna Łąka".IndexOf("one");
```

spowoduje przypisanie zmiennej `indeks` wartości `-1`, gdyż w ciągu `piękna Łąka` nie występuje ciąg `one`.

Omawiana metoda umożliwia użycie drugiego argumentu. Określa on indeks znaku, od którego ma się zacząć przeszukiwanie ciągu podstawowego. Znaczy to, że przykładowe wywołanie:

```
int indeks = "piękna Łąka".IndexOf("ą", 4);
```

spowoduje przypisanie zmiennej `indeks` wartości `8` — ponieważ ciąg `ą` zaczyna się na `8`. pozycji, a przeszukiwanie rozpoczyna od `4`. Natomiast wywołanie:

```
int indeks = "piękna Łąka".IndexOf("ą", 9);
```

spowoduje przypisanie zmiennej `indeks` wartości `-1` — gdyż ciąg `ą` zaczyna się na `8`. pozycji, a przeszukiwanie zaczyna się od pozycji `9`.

Metoda `LastIndexOf` działa na tej samej zasadzie co `IndexOf`, ale przeszukuje ciąg od końca. Jeśli więc wykonamy serię instrukcji:

```
int i1 = "błękitne niebo".LastIndexOf("ne");
int i2 = "błękitne niebo".LastIndexOf("na");
int i3 = "błękitne niebo".LastIndexOf("ki", 6);
int i4 = "błękitne niebo".LastIndexOf("ki", 2);
```

okazuje się, że:

- ◆ zmienna `i1` zawiera wartość `6`, ponieważ ciąg `ne` rozpoczyna się w indeksie `6`;
- ◆ zmienna `i2` zawiera wartość `-1`, ponieważ ciąg `na` nie występuje w ciągu `błękitne niebo`;
- ◆ zmienna `i3` zawiera wartość `3`, ponieważ przeszukiwanie rozpoczyna się w indeksie `6` (licząc od początku), a ciąg `ki` rozpoczyna się w indeksie `3`;
- ◆ zmienna `i4` zawiera wartość `-1`, ponieważ ciąg `ki` rozpoczyna się w indeksie `3`, a przeszukiwanie rozpoczyna się w indeksie `2` (i dąży do indeksu `0`).

Metoda `replace` zamienia wszystkie podciagi podane jako pierwszy argument na ciągi przekazane jako drugi argument. Przykładowo po wykonaniu instrukcji:

```
string str = "Cześć, %IMIE%. Miło Cię spotkać.".Replace("%IMIE%", "Adam");
```

zmienna str będzie zawierała ciąg znaków:

Cześć, Adam. Miło Cię spotkać.

gdzie ciąg `%IMIE%` zostanie zamieniony na ciąg `Adam`.

Metoda `split` umożliwia podzielenie ciągu względem znaków separatora przekazanych jako pierwszy argument (tablica elementów typu `char`). Podzielony ciąg jest zwracany w postaci tablicy obiektów typu `string`. Użycie drugiego argumentu pozwala określić maksymalną liczbę ciągów wynikowych (a tym samym rozmiar tablicy wynikowej). Wykonanie przykładowych instrukcji:

```
string[] tab1 = "a b c".Split(new char[] {' '});  
string[] tab2 = "a,b,c".Split(new char[] {','}, 2);  
string[] tab3 = "a, b, c".Split(new char[] {' ', ','});
```

spowoduje utworzenie następujących tablic:

- ◆ `tab1` — zawierającej trzy komórki z ciągami `a`, `b` i `c` — znakiem separatora jest bowiem spacja, a liczba ciągów wynikowych nie jest ograniczona,
- ◆ `tab2` — zawierającej dwie komórki, pierwszą z ciągiem `a` i drugą z ciągiem `b,c` — znakiem separatora jest bowiem przecinek, a liczba ciągów wynikowych jest ograniczona do dwóch,
- ◆ `tab3` — zawierającej pięć komórek odpowiadających poszczególnym elementom ciągu, komórki 0, 2 i 4 będą zawierały znaki `a`, `b` i `c`, natomiast komórki 1 i 3 — puste ciągi znaków (separatorami są bowiem znaki przecinka i spacji).

Metoda `Substring` pozwala wyodrębnić fragment ciągu. Pierwszy argument określa indeks początkowy, a drugi — liczbę znaków do pobrania. Drugi argument można pominąć — wtedy pobierany fragment rozpocznie się od indeksu wskazywanego przez pierwszy argument, a skończy w końcu ciągu głównego. Znaczy to, że przykładowe wywołanie:

```
string str1 = "wspaniały świat".Substring(2, 4);
```

spowoduje przypisanie zmiennej `str1` ciągu `pani` (4 znaki, począwszy od znaku o indeksie 2 w ciągu głównym), a wywołanie:

```
string str2 = "wspaniały świat".Substring(10);
```

spowoduje przypisanie zmiennej `str2` ciągu `świat` (wszystkie znaki, począwszy od tego o indeksie 10, aż do końca ciągu głównego).

Działanie metod `ToLower` i `ToUpper` jest analogiczne, choć działają przeciwnie. Pierwsza zamienia wszystkie litery ciągu na małe, a druga — na wielkie. Dzieje się to niezależnie od tego, jaka była wielkość liter w ciągu oryginalnym. Zwracany jest ciąg przetworzony, a ciąg oryginalny nie jest zmieniany. Znaczy to, że instrukcja:

```
string str1 = "Wielki Zderzacz Hadronów".ToLower();
```

spowoduje przypisanie zmiennej str1 ciągu wielki zderzacz hadronów, a instrukcja

```
string str2 = "Wielki Zderzacz Hadronów".ToUpper();
```

przypisanie zmiennej str2 ciągu WIELKI ZDERZACZ HADRONÓW.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 24.1

Napisz program wyświetlający na ekranie napis składający się z kilku słów. Nie używaj jednak ani zmennych, ani literalów, ani innych obiektów typu string.

Ćwiczenie 24.2

Napisz program wyświetlający na ekranie napis Język C#. Do tworzenia tekstu użyj wyłącznie sekwencji specjalnych.

Ćwiczenie 24.3

Zmień przykład z listingu 5.7, tak aby tekst został wyświetlony od końca, a w każdym wierszu znajdowały się dwa znaki.

Ćwiczenie 24.4

Napisz program, który wyświetli szesnastkowe kody wszystkich liter alfabetu, zarówno małych, jak i wielkich.

Ćwiczenie 24.5

Umieść w kodzie programu metodę przyjmującą jako argument ciąg znaków, która przetworzy go w taki sposób, że każda litera a, b i c, przed którą nie znajduje się litera k, l lub j, zostanie zamieniona na spację. Rezultatem działania metody powinien być przetworzony ciąg. Przetestuj działanie metody na kilku różnych ciągach znaków.

Lekcja 25. Standardowe wejście i wyjście

Z podstawowymi operacjami wyjściowymi, czyli wyświetlaniem informacji na ekranie konsoli, mieliśmy już wielokrotnie do czynienia. W tej lekcji skupimy się więc na operacji odwrotnej, czyli na odczytywaniu danych wprowadzanych przez użytkownika z klawiatury. Sprawdzimy, jak pobierać pojedyncze znaki, całe wiersze tekstu, a także dane liczbowe oraz jak przetwarzać tak otrzymane informacje w aplikacji. Zostanie bliżej omówiona wykonujące te i wiele innych operacji klasa Console.

Klasa Console i odczyt znaków

Podstawowe operacje wejścia-wyjścia na konsoli, takie jak wyświetlanie tekstu oraz pobieranie danych wprowadzanych przez użytkownika z klawiatury, mogą być wykonywane za pomocą klasy `Console`. Ma ona szereg właściwości i metod odpowiadających za realizację różnych zadań. Wielokrotnie używaliśmy np. metod `Write` i `WriteLine` do wyświetlania na ekranie wyników działania przykładowych programów. Właściwości udostępniane przez klasę `Console` zostały zebrane w tabeli 5.4 (wszystkie są publiczne i statyczne), natomiast metody — w tabeli 5.5.

Tabela 5.4. Właściwości klasy `Console`

Typ	Nazwa	Opis
<code>ConsoleColor</code>	<code>BackgroundColor</code>	Określa kolor tła konsoli.
<code>int</code>	<code>BufferSize</code>	Określa wysokość obszaru bufora.
<code>int</code>	<code>BufferSize</code>	Określa szerokość obszaru bufora.
<code>bool</code>	<code>CapsLock</code>	Określa, czy jest aktywny klawisz <i>Caps Lock</i> .
<code>int</code>	<code>CursorLeft</code>	Określa kolumnę, w której znajduje się kursor.
<code>int</code>	<code>CursorPosition</code>	Określa wysokość kurSORA (w procentach wysokości komórki znakowej od 1 do 100).
<code>int</code>	<code>CursorPosition</code>	Określa wiersz, w którym znajduje się kursor.
<code>bool</code>	<code>CursorVisible</code>	Określa, czy kurSOR jest widoczny.
<code>TextWriter</code>	<code>Error</code>	Pobiera (właściwość tylko do odczytu) standardowy strumień obsługi błędów.
<code>ConsoleColor</code>	<code>ForegroundColor</code>	Określa kolor tekstu (kolor pierwszoplanowy) konsoli.
<code>TextReader</code>	<code>In</code>	Pobiera (właściwość tylko do odczytu) standardowy strumień wejściowy.
<code>Encoding</code>	<code>InputEncoding</code>	Określa standard kodowania znaków przy odczytce z konsoli.
<code>bool</code>	<code>KeyAvailable</code>	Określa, czy w strumieniu wejściowym dostępny jest kod naciśniętego klawisza.
<code>int</code>	<code>LargestWindowHeight</code>	Pobiera maksymalną liczbę wierszy konsoli (dla bieżącej rozdzielcości ekranu i wielkości fontu).
<code>int</code>	<code>LargestWindowWidth</code>	Pobiera maksymalną liczbę kolumn konsoli (dla bieżącej rozdzielcości ekranu i wielkości fontu).
<code>bool</code>	<code>NumberLock</code>	Określa, czy jest aktywny klawisz <i>Num Lock</i> .
<code>TextWriter</code>	<code>Out</code>	Pobiera (właściwość tylko do odczytu) standardowy strumień wyjściowy.
<code>Encoding</code>	<code>OutputEncoding</code>	Określa standard kodowania znaków przy wyświetleniu (zapisie) na konsoli.
<code>String</code>	<code>Title</code>	Określa tekst wyświetlany na pasku tytułu okna konsoli.
<code>bool</code>	<code>TreatControlCAsInput</code>	Określa, czy kombinacja klawiszy <i>Ctrl+C</i> ma być traktowana jako zwykła kombinacja klawiszy, czy też jako sygnał przerwania obsługiwany przez system operacyjny.

Tabela 5.4. Właściwości klasy Console — ciąg dalszy

Typ	Nazwa	Opis
Int	WindowHeight	Określa wysokość okna konsoli (w wierszach).
int	WindowLeft	Określa położenie w poziomie lewego górnego rogu okna konsoli.
int	WindowTop	Określa położenie w pionie lewego górnego rogu okna konsoli.
int	WindowWidth	Określa szerokość okna konsoli (w kolumnach).

Tabela 5.5. Publiczne metody klasy Console

Typ zwracany	Nazwa	Opis
void	Beep	Powoduje wydanie dźwięku.
void	Clear	Czyści bufor i ekran konsoli.
void	MoveBufferArea	Kopiuje część bufora w inne miejsce.
Stream	OpenStandardError	Pobiera odwołanie do standardowego strumienia błędów.
Stream	OpenStandardInput	Pobiera odwołanie do standardowego strumienia wejściowego.
Stream	OpenStandardOutput	Pobiera odwołanie do standardowego strumienia wyjściowego.
int	Read	Odczytuje kolejny znak ze standardowego strumienia wejściowego.
ConsoleKeyInfo	.ReadKey	Pobiera kolejną wartość (określenie naciśniętego klawisza) ze standardowego strumienia wejściowego.
string	ReadLine	Odczytuje kolejną linię tekstu ze standardowego strumienia wejściowego.
void	ResetColor	Ustawia kolory tekstu i tła na domyślne.
void	SetBufferSize	Określa wysokość i szerokość bufora tekstu.
void	SetCursorPosition	Ustala pozycję kurSORA.
void	SetError	Ustawia właściwość Error.
void	SetIn	Ustawia właściwość In.
void	SetOut	Ustawia właściwość Out.
void	SetWindowPosition	Ustawia pozycję okna konsoli.
void	SetWindowSize	Ustawia rozmiary okna konsoli.
void	Write	Wysyła do standardowego wyjścia tekstową reprezentację przekazanych wartości.
void	WriteLine	Wysyła do standardowego wyjścia tekstową reprezentację przekazanych wartości zakończoną znakiem końca linii.

Spróbujmy więc napisać teraz program, który odczyta znak wprowadzony z klawiatury i wyświetli na ekranie jego kod. Jeśli zajrzymy do tabeli 5.5, znajdziemy w niej metodę Read, która wykonuje pierwszą część takiego zadania, czyli zwraca kod znaku odpowiadającego naciśniętemu klawiszowi (lub kombinacji klawiszy). Jeśli zapamiętamy

ten kod w zmiennej i wyświetlimy jej zawartość za pomocą metody `WriteLine`, to otrzymamy dokładnie to, o co nam chodziło. Pełny kod programu jest widoczny na listingu 5.8.

Listing 5.8. Wczytanie pojedynczego znaku

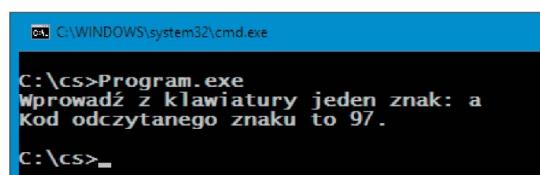
```
using System;

public class Program
{
    public static void Main()
    {
        Console.Write("Wprowadź z klawiatury jeden znak: ");
        int kodZnaku = Console.Read();
        Console.WriteLine("Kod odczytanego znaku to {0}.", kodZnaku);
    }
}
```

Wynik wywołania `Console.Read()` jest przypisywany zmiennej typu `int` o nazwie `kodZnaku`. Ta zmienna jest następnie używana w instrukcji `Console.WriteLine` wprowadzającej tekst na konsolę. Jeśli teraz skompilujemy i uruchomimy program, po czym naciśniemy dowolny klawisz (np. *a*) oraz *Enter*, zobaczymy widok taki, jak zaprezentowany na rysunku 5.4. Można zauważyć, że malej literze *a* jest przyporządkowany kod 97. Gdybyśmy zastosowali kombinację *Shift+A* (co odpowiada dużej literze *A*), otrzymalibyśmy wartość 65.

Rysunek 5.4.

Efekt działania programu odczytującego kod znaku



O wiele więcej informacji niesie ze sobą metoda `ReadKey`. Otóż w wyniku jej działania zwracany jest obiekt typu `Console.ReadKey`. Zawiera on trzy publiczne właściwości pozwalające na ustalenie, który klawisz został naciśnięty, jaki jest jego kod Unicode oraz czy zostały również naciśnięte klawisze funkcyjne *Alt*, *Ctrl* lub *Shift*. Właściwości te zostały zebrane w tabeli 5.6.

Tabela 5.6. Właściwości struktury `Console.ReadKey`

Typ	Nazwa	Opis
<code>ConsoleKey</code>	<code>Key</code>	Zawiera określenie naciśniętego klawisza.
<code>Char</code>	<code>KeyChar</code>	Zawiera kod odczytanego znaku.
<code>ConsoleModifiers</code>	<code>Modifiers</code>	Zawiera określenie, które klawisze funkcyjne (<i>Alt</i> , <i>Ctrl</i> lub <i>Shift</i>) zostały naciśnięte.

Właściwość `Key` jest typu wyliczeniowego `ConsoleKey`. Zawiera on określenie naciśniętego klawisza. W przypadku liter te określenia to `ConsoleKey.A`, `ConsoleKey.B` itd. W przypadku klawiszy funkcyjnych *F1*, *F2* itd. to `ConsoleKey.F1`, `ConsoleKey.F2` itd. W przypadku cyfr — `ConsoleKey.D0`, `ConsoleKey.D1` itd. Oprócz tego istnieje także

wiele innych określeń (np. dla klawiatury numerycznej, kurSORów i wszelkich innych klawiszy), których pełną listę można znaleźć w dokumentacji platformy .NET na stronach <http://msdn.microsoft.com>².

Napiszmy więc krótki program, którego zadaniem będzie oczekiwanie na naciśnięcie przez użytkownika konkretnego klawisza. Niech będzie to klawisz z literą *Q*. Kod takiej aplikacji jest widoczny na listingu 5.9.

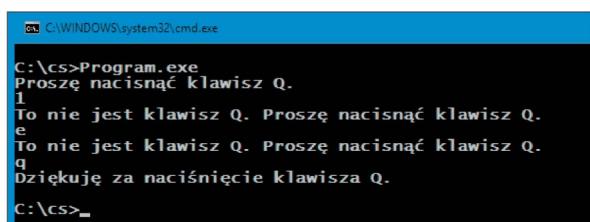
Listing 5.9. Oczekiwanie na naciśnięcie konkretnego klawisza

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Proszę nacisnąć klawisz Q.");
        ConsoleKeyInfo keyInfo = Console.ReadKey();
        while(keyInfo.Key != ConsoleKey.Q)
        {
            Console.WriteLine(
                "\nTo nie jest klawisz Q. Proszę nacisnąć klawisz Q.");
            keyInfo = Console.ReadKey();
        }
        Console.WriteLine("\nDziękuję za naciśnięcie klawisza Q.");
    }
}
```

Kod rozpoczyna się od wyświetlenia prośby o naciśnięcie klawisza *Q*. Następnie wywołana jest metoda *ReadKey* z klasy *Console*, a wynik jej działania przypisuje się pomocniczej zmiennej *keyInfo* typu *ConsoleKeyInfo*. Dalej w pętli *while* następuje badanie, czy właściwość *Key* obiektu *keyInfo* jest równa wartości *ConsoleKey.Q*, a zatem czy faktycznie użytkownik aplikacji nacisnął klawisz *Q*. Jeśli tak, pętla jest opuszczana i jest wyświetlone podziękowanie; jeśli nie, jest wyświetlana ponowna prośba o naciśnięcie właściwego klawisza i ponownie jest wywoływana metoda *ReadKey*, której wynik działania trafia do zmiennej *keyInfo*. Dzięki temu dopóki użytkownik nie naciśnie klawisza *Q*, prośba będzie ponawiana, tak jak jest to widoczne na rysunku 5.5. Warto też zauważyć, że można uniknąć dwukrotnego wywoływanego metody *ReadKey* (raz przed pętlą i raz w jej wnętrzu), jeśli tylko zmieni się typ pętli na *do...while*, co jednak będzie dobrym ćwiczeniem do samodzielnego wykonania.

Rysunek 5.5.
Oczekiwanie
na naciśnięcie
konkretnego
klawisza



² W trakcie powstawania książki aktywnym adresem był <http://msdn.microsoft.com/en-us/library/system.consolekey.aspx>.

Struktura `ConsoleKeyInfo` zawiera również informacje o stanie klawiszy specjalnych *Alt*, *Ctrl* i *Shift*. W prosty sposób można się więc dowiedzieć, czy któryś z nich był naciśnięty z jakimś innym klawiszem. Odpowiada za to właściwość `Modifiers`. Aby ustalić, czy któryś z wymienionych klawiszy był naciśnięty, należy wykonać iloczyn bitowy tej właściwości oraz jednej z wartości:

- ◆ `ConsoleModifiers.Alt` — dla klawisza *Alt*,
- ◆ `ConsoleModifiers.Control` — dla klawisza *Ctrl*,
- ◆ `ConsoleModifiers.Shift` — dla klawisza *Shift*.

Jeśli wynik takiej operacji będzie różny od 0, będzie to znaczyło, że dany klawisz był naciśnięty. Jak dokonać tego w praktyce, zobrazowano w przykładzie z listingu 5.10.

Listing 5.10. Rozpoznawanie klawiszy specjalnych

```
using System;

public class Program
{
    public static void Main()
    {
        Console.Write("Proszę naciskać dowolne klawisze. ");
        Console.WriteLine("Klawisz Esc kończy działanie programu.");
        Console.TreatControlCAsInput = true;
        ConsoleKeyInfo keyInfo;
        do
        {
            keyInfo = Console.ReadKey(true);
            String str = keyInfo.Key.ToString();
            if((keyInfo.Modifiers & ConsoleModifiers.Alt) != 0)
            {
                str += " [ALT]";
            }
            if((keyInfo.Modifiers & ConsoleModifiers.Control) != 0)
            {
                str += " [CTRL]";
            }
            if((keyInfo.Modifiers & ConsoleModifiers.Shift) != 0)
            {
                str += " [SHIFT]";
            }
            Console.Write("Zastosowano kombinację " + str);
            Console.WriteLine(", czyli znak " + keyInfo.KeyChar);
        }
        while(keyInfo.Key != ConsoleKey.Escape);
    }
}
```

Na początku kodu wyświetlana jest informacja o sposobie działania aplikacji, a przez przypisanie wartości `true` właściwości `TreatControlCAsInput` klasy `Console` zmieniany jest sposób traktowania kombinacji *Ctrl+C* — nie będzie ona powodowała przerwania działania programu. Główne instrukcje są wykonywane w pętli `do...while`. Działa ona tak długo, aż właściwość `Key` struktury `keyInfo` otrzyma wartość `ConsoleKey.Escape`.

co jest równoznaczne z naciśnięciem przez użytkownika klawisza *Esc*. Zmienna `keyInfo` jest deklarowana tuż przed pętlą, a w pierwszej instrukcji pętli jest jej przypisywana wartość zwrocona przez wywołanie `ReadKey`. Tym razem wykorzystywana jest inna wersja tej metody niż w przypadku listingu 5.9. Przyjmuje ona bowiem argument typu `bool`. Jeśli jest on równy `true` (tak jak w kodzie programu), oznacza to, że znak odpowiadający naciśniętemu klawiszowi nie ma się pojawiać na ekranie, o jego wyświetlanie należy zadbać samemu.

Po odczytaniu danych konstruowany jest ciąg `str` zawierający napis, który ma się pojawić na ekranie. Na początku temu ciągowi przypisywana jest wartość uzyskana za pomocą wywołania metody `ToString` struktury `Key` obiektu `keyInfo`:

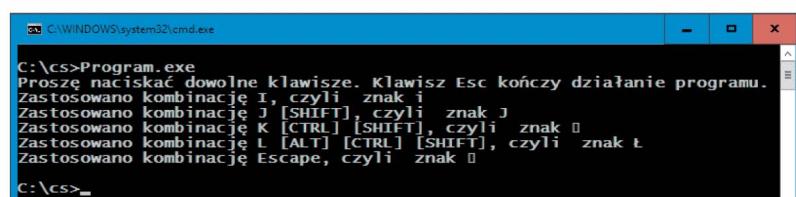
```
String str = keyInfo.Key.ToString();
```

Będzie to nazwa naciśniętego klawisza (np. *A*, *B*, *Delete*, *Esc*, *Page Up* itp.). Następnie w serii instrukcji warunkowych `if` badany jest stan klawiszy specjalnych *Alt*, *Ctrl* i *Shift*. W przypadku wykrycia, że któryś z nich był naciśnięty razem z klawiszem głównym, nazwa klawisza specjalnego ujęta w nawias kwadratowy jest dodawana do ciągu `str`. Ostatecznie konstruowany jest pełny ciąg o postaci:

Naciśnięto kombinację *kombinacja*, czyli znak *znak*.

Jest on wyświetlany na ekranie za pomocą instrukcji `Console.Write` i `Console.WriteLine`. Znak odpowiadający wykorzystanej kombinacji klawiszy jest uzyskiwany poprzez odwołanie się do właściwości `KeyChar` struktury `keyInfo`. Przykładowy efekt działania programu został zaprezentowany na rysunku 5.6.

Rysunek 5.6.
Efekt działania programu obsługującego klawisze specjalne



Powracając do tabeli 5.4, znajdziemy także właściwości `BackgroundColor` i `ForegroundColor`. Pierwsza określa kolor tła, a druga kolor tekstu wyświetlanyego na konsoli. Obie są typu `ConsoleColor`. Jest to typ wyliczeniowy, którego składowe określają kolory możliwe do zastosowania na konsoli. Zostały one zebrane w tabeli 5.7. W prosty sposób można więc manipulować kolorami, a przykład tego został przedstawiony na listingu 5.11.

Tabela 5.7. Kolory zdefiniowane w wyliczeniu `ConsoleColor`

Składowa wyliczenia	Kolor
<code>Black</code>	Czarny
<code>Blue</code>	Niebieski
<code>Cyan</code>	Niebieskozielony
<code>DarkBlue</code>	Ciemnoniebieski
<code>DarkCyan</code>	Ciemny niebieskozielony

Tabela 5.7. Kolory zdefiniowane w wyliczeniu ConsoleColor — ciąg dalszy

Składowa wyliczenia	Kolor
DarkGray	Ciemnoszary
DarkGreen	Ciemnozielony
DarkMagenta	Ciemna fuksja (ciemny purpurowoczerwony)
DarkRed	Ciemnoczerwony
DarkYellow	Ciemnożółty (ochra)
Gray	Szary
Green	Zielony
Magenta	Fuksja (purpurowoczerwony)
Red	Czerwony
White	Biały
Yellow	Żółty

Listing 5.11. Zmiana kolorów na konsoli

```
using System;

public class Program
{
    public static void Main()
    {
        Console.BackgroundColor = ConsoleColor.Blue;
        Console.ForegroundColor = ConsoleColor.Yellow;
        Console.Write("abcd");

        Console.BackgroundColor = ConsoleColor.Green;
        Console.ForegroundColor = ConsoleColor.DarkBlue;
        Console.Write("efgh");

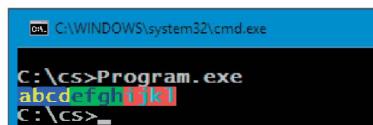
        Console.BackgroundColor = ConsoleColor.Red;
        Console.ForegroundColor = ConsoleColor.Cyan;
        Console.Write("ijkl");

        Console.ResetColor();
    }
}
```

Program wyświetla trzy łańcuchy tekstowe, a przed każdym wyświetlaniem zmieniane są kolory tekstu oraz tła. Kolor tła jest modyfikowany przez przypisania odpowiedniego elementu wyliczenia `ConsoleColor` właściwości `BackgroundColor`, a kolor tekstu — właściwości `ForegroundColor`. Na zakończenie przywracane są kolory domyślne, za co odpowiada wywołanie metody `ResetColor`. Efekt działania aplikacji został przedstawiony na rysunku 5.7.

Rysunek 5.7.

Efekt działania programu zmieniającego kolory na konsoli



Wczytywanie tekstu z klawiatury

Wiadomo już, jak odczytać jeden znak. Co jednak zrobić, kiedy chcemy wprowadzić całą linię tekstu? Przecież taka sytuacja jest o wiele częstsza. Można oczywiście odczytywać pojedyncze znaki w pętli tak dugo, aż zostanie osiągnięty znak końca linii, oraz połączyć je w obiekt typu `String`. Najprościej jednak użyć metody `ReadLine`, która wykona to zadanie automatycznie. Po jej wywołaniu program zaczeka, aż zostanie wprowadzony ciąg znaków zakończony znakiem końca linii (co odpowiada naciśnięciu klawisza `Enter`); ciąg ten zostanie zwrócony w postaci obiektu typu `String`. Na listingu 5.12 jest widoczny przykład odczytujący z klawiatury kolejne linie tekstu i wyświetlający je z powrotem na ekranie.

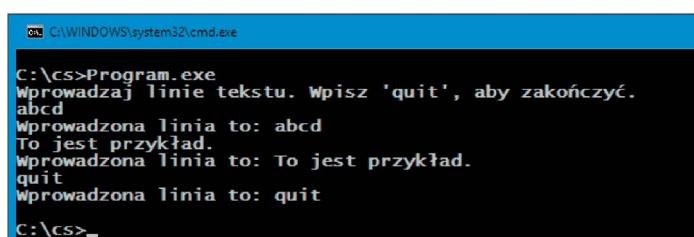
Listing 5.12. Pobieranie linii tekstu

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine(
            "Wprowadzaj linie tekstu. Wpisz 'quit', aby zakończyć.");
        String line;
        do
        {
            line = Console.ReadLine();
            Console.WriteLine("Wprowadzona linia to: {0}", line);
        }
        while(line != "quit");
    }
}
```

Na początku jest wyświetlana prośba o wprowadzanie linii tekstu oraz deklarowana zmienna `line` — będzie ona przechowywała wprowadzane przez użytkownika ciągi znaków. W pętli `do...while` jest wywoływana metoda `ReadLine`, a wynik jej działania przypisywany zmiennej `line`. Następnie odczytana treść jest ponownie wyświetlana na ekranie za pomocą instrukcji `Console.WriteLine`. Pętla kończy swoje działanie, kiedy użytkownik wprowadzi z klawiatury ciąg znaków `quit`, tak więc warunkiem jej zakończenia jest `line != "quit"`. Przykładowy efekt działania programu jest widoczny na rysunku 5.8.

Rysunek 5.8.
Efekt działania programu odczytującego linie tekstu



Wprowadzanie liczb

Wiadomo już jak odczytywać w aplikacji linie tekstu wprowadzanego z klawiatury. Równie ważnym zadaniem jest jednak wprowadzanie liczb. Jak to zrobić? Trzeba sobie uzmysłowić, że z klawiatury zawsze wprowadzany jest tekst. Jeśli próbujemy wprowadzić do aplikacji wartość 123, to w rzeczywistości wprowadzimy trzy znaki: 1, 2 i 3 o kodach ASCII 61, 62, 63. Mogą one zostać przedstawione w postaci ciągu "123", ale to dopiero aplikacja musi przetworzyć ten ciąg na wartość 123. Takiej konwersji w przypadku wartości całkowitej można dokonać np. za pomocą metody Parse struktury Int32. Jest to metoda statyczna, możemy ją więc wywołać, nie tworząc obiektu typu Int32 (lekcia 19.). Przykładowe wywołanie może wyglądać następująco:

```
int liczba = Int32.Parse("ciąg_znaków");
```

Zmiennej liczba zostanie przypisana wartość typu int zawarta w ciągu znaków *ciąg_znaków*. W przypadku gdyby ciąg znaków przekazany jako argument metody Parse nie zawierał poprawnej wartości całkowitej, zostanie wygenerowany jeden z wyjątków:

- ◆ ArgumentNullException — jeśli argumentem jest wartość null;
- ◆ FormatException — jeśli argument nie może być przekonwertowany na liczbę całkowitą (zawiera znaki nietworzące liczby);
- ◆ OverflowException — jeśli argument zawiera prawidłową wartość całkowitą, ale przekracza ona dopuszczalny zakres dla typu Int32.

Aby zatem wprowadzić do aplikacji wartość całkowitą, można odczytać linię tekstu, korzystając z metody ReadLine, a następnie wywołać metodę Parse. Ten właśnie sposób został wykorzystany w programie z listingu 5.13. Jego zadaniem jest wczytanie liczby całkowitej oraz wyświetlenie wyniku mnożenia tej liczby przez wartość 2.

Listing 5.13. Wczytanie wartości liczbowej i pomnożenie jej przez 2

```
using System;

public class Program
{
    public static void Main()
    {
        Console.Write("Wprowadź liczbę całkowitą: ");

        String line = Console.ReadLine();
        int liczba;

        try
        {
            liczba = Int32.Parse(line);
        }
        catch(Exception)
        {
            Console.WriteLine("Wprowadzona wartość nie jest prawidłowa.");
            return;
        }
        Console.WriteLine("{0} * 2 = {1}", liczba, liczba * 2);
    }
}
```

Kod rozpoczyna się od wyświetlenia prośby o wprowadzenie liczby całkowitej. Następnie wprowadzone dane są odczytywane za pomocą metody `ReadLine` i zapisywane w zmiennej `line`. Dalej znajduje się deklaracja zmiennej `liczba` typu `int` oraz przypisanie jej wyniku działania metody `Parse`. Metodzie tej przekazujemy ciąg znaków zapisany w `line`. Jeśli wprowadzony przez użytkownika ciąg znaków nie reprezentuje poprawnej wartości liczbowej, wygenerowany zostanie jeden z opisanych wyżej wyjątków. W takim wypadku (dzięki zastosowaniu bloku `try...catch`) wyświetlamy komunikat o błędzie oraz kończymy działanie metody `Main`, a tym samym programu, wywołując instrukcję `return`. Jeśli jednak konwersja tekstu na liczbę powiedzie się, odpowiednia wartość zostanie zapisana w zmiennej `liczba`, można zatem wykonać mnożenie `liczba * 2` i wyświetlić wartość wynikającą z tego mnożenia na ekranie. Przykładowy wynik działania programu jest widoczny na rysunku 5.9.

Rysunek 5.9.

Wynik działania programu mnożącego wprowadzoną wartość przez 2

```
C:\Windows\system32\cmd.exe
C:\cs>Program.exe
Wprowadź liczbę całkowitą: abc
Wprowadzona wartość nie jest prawidłowa.
C:\cs>Program.exe
Wprowadź liczbę całkowitą: 4
4 * 2 = 8
C:\cs>
```

Gdybyśmy chcieli wczytać liczbę **zmiennoprzecinkową**, należałoby do konwersji zastosować metodę `Parse` struktury `Double`, co jest doskonałym ćwiczeniem do samodzielnego wykonania. Ogólnie rzecz ujmując, dla każdego z typów numerycznych w przestrzeni nazw `System` znajdziemy struktury (`Int16`, `Int32`, `SByte`, `Char` itd.) zawierającą metodę `Parse`, która wykonuje konwersję ciągu znaków do tego typu.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 25.1

Zmień kod z listingu 5.9 w taki sposób, aby w programie zamiast pętli `while` była używana `do..while`.

Ćwiczenie 25.2

Napisz program, który będzie realizował tzw. szyfr Cezara działający na znakach wprowadzanych bezpośrednio z klawiatury. Naciśnięcie klawisza odpowiadającego literze `a` ma powodować pojawianie się na ekranie znaku `d`, odpowiadającego literze `b` — znaku `e`, odpowiadającego literze `c` — znaku `f` itd. Możesz ograniczyć się do przekodowywania tylko małych liter z alfabetu łacińskiego.

Ćwiczenie 25.3

Zmodyfikuj program z listingu 5.12 w taki sposób, aby po wprowadzeniu przez użytkownika ciągu `quit` nie był on ponownie wyświetlany na ekranie, ale by aplikacja od razu kończyła swoje działanie.

Ćwiczenie 25.4

Napisz program, który będzie wymagał wprowadzenia dwóch liczb rzeczywistych i wyświetli wynik ich mnożenia na ekranie. W razie niepodania poprawnej wartości liczbowej program powinien ponawiać prośbę o jej wprowadzenie.

Ćwiczenie 25.5

Napisz program rozwiązuający równania kwadratowe, w którym parametry A , B i C będą wprowadzane przez użytkownika z klawiatury.

Ćwiczenie 25.6

Napisz program, który umożliwi użytkownikowi wprowadzenie wiersza tekstu zawierającego liczby całkowite oddzielone znakiem separatora (np. przecinkiem), a więc przykładowego ciągu $1,5,24,8,150,2$. Program powinien następnie wyświetlić te z uzytkowanych wartości, które są podzielne przez 2.

Lekcja 26. Operacje na systemie plików

Lekcja 26. jest poświęcona technikom pozwalającym operować na systemie plików. Znajdują się w niej informacje o sposobach tworzenia i usuwania plików oraz katalogów. Przedstawione zostaną bliżej klasy `FileSystemInfo`, `DirectoryInfo` i `FileInfo`, a także udostępniane przez nie metody. Zobaczmy, jak pobrać zawartość katalogu oraz jak usunąć katalog. Po zapoznaniu się z tymi tematami będzie można przejść do metod zapisu i odczytu plików, czym jednak zajmiemy się dopiero w kolejnej lekcji.

Klasa `FileSystemInfo`

Klasa `FileSystemInfo` jest abstrakcyjną klasą bazową dla `DirectoryInfo` i `FileInfo`, które z kolei pozwalają na wykonywanie na plikach i katalogach podstawowych operacji, takich jak ich tworzenie i usuwanie, operacje na nazwach czy pobieranie parametrów, jak np. czas utworzenia bądź modyfikacji. Obejmuje ona właściwości i metody wspólne dla plików i katalogów. Właściwości zostały zebrane w tabeli 5.8 (wszystkie są publiczne), a wybrane metody w tabeli 5.9. Wymienione klasy znajdują się w przestrzeni nazw `System.IO`, tak więc w programach przykładowych będzie stosowana dyrektywa `using` w postaci:

```
using System.IO;
```

Tabela 5.8. Publiczne właściwości klasy *FileInfo*

Typ	Właściwość	Opis
FileAttributes	Attributes	Określa atrybuty pliku lub katalogu.
DateTime	CreationTime	Określa czas utworzenia pliku lub katalogu.
DateTime	CreationTimeUtc	Określa czas utworzenia pliku lub katalogu w formacie UTC.
bool	Exists	Określa, czy plik lub katalog istnieje.
string	Extension	Zawiera rozszerzenie nazwy pliku lub katalogu (tylko do odczytu).
string	FullName	Zawiera pełną ścieżkę dostępu do pliku lub katalogu (tylko do odczytu).
DateTime	LastAccessTime	Określa czas ostatniego dostępu do pliku lub katalogu.
DateTime	LastAccessTimeUtc	Określa czas ostatniego dostępu do pliku lub katalogu w formacie UTC.
DateTime	LastWriteTime	Określa czas ostatniego zapisu w pliku lub katalogu.
DateTime	LastWriteTimeUtc	Określa czas ostatniego zapisu w pliku lub katalogu w formacie UTC.
string	Name	Podaje nazwę pliku lub katalogu.

Tabela 5.9. Wybrane metody klasy *FileInfo*

Typ zwracany	Metoda	Opis
void	Delete	Usuwa plik lub katalog.
Type	GetType	Zwraca typ obiektu.
void	Refresh	Odświeża stan obiektu (pobiera aktualne informacje przekazane przez system operacyjny).

Operacje na katalogach

Klasa DirectoryInfo

Klasa *DirectoryInfo* pozwala na wykonywanie podstawowych operacji na katalogach, takich jak ich tworzenie i usuwanie, operacje na nazwach czy pobieranie parametrów, jak np. czas utworzenia bądź modyfikacji. Większość jej właściwości jest odziedziczona po klasie *FileInfo* — w tabeli 5.10 zostały natomiast uwzględnione właściwości dodatkowe, zdefiniowane bezpośrednio w *DirectoryInfo*. Metody klasy *DirectoryInfo* zostały przedstawione w tabeli 5.11. Będziemy je wykorzystywać w dalszej części lekcji.

Tabela 5.10. Właściwości klasy *DirectoryInfo*

Typ	Właściwość	Opis
<i>DirectoryInfo</i>	Parent	Określa katalog nadzędny.
<i>DirectoryInfo</i>	Root	Określa korzeń drzewa katalogów.

Tabela 5.11. Metody klasy DirectoryInfo

Typ zwracany	Metoda	Opis
void	Create	Tworzy nowy katalog.
DirectoryInfo	CreateSubdirectory	Tworzy podkatalog lub podkatalogi.
DirectoryInfo[]	GetDirectories	Pobiera listę podkatalogów.
FileInfo[]	GetFiles	Pobiera listę plików z danego katalogu.
FileSystemInfo[]	GetFileSystemInfos	Pobiera listę podkatalogów i plików.
void	MoveTo	Przenosi katalog do innej lokalizacji.

Pobranie zawartości katalogu

W celu poznania zawartości danego katalogu należy skorzystać z metod GetDirectories i GetFiles klasy DirectoryInfo. Pierwsza zwraca tablicę obiektów typu DirectoryInfo, które zawierają informacje o katalogach, a druga tablicę obiektów typu FileInfo z informacjami o plikach. Obie klasy mają właściwość Name odziedziczoną po klasie nadzędnej FileInfo, zatem łatwo można uzyskać nazwy odczytanych elementów systemu plików. Tak więc napisanie programu, którego zadaniem będzie wyświetlenie zawartości katalogu, z pewnością nie będzie dla nikogo stanowiło problemu. Taki przykładowy kod jest widoczny na listingu 5.14.

Listing 5.14. Program wyświetlający zawartość katalogu bieżącego

```
using System;
using System.IO;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("Zawartość katalogu bieżącego:");
        DirectoryInfo di = new DirectoryInfo(".");
        DirectoryInfo[] katalogi = di.GetDirectories();
        FileInfo[] pliki = di.GetFiles();

        Console.WriteLine("--PODKATALOGI--");
        foreach(DirectoryInfo katalog in katalogi)
        {
            Console.WriteLine(katalog.Name);
        }

        Console.WriteLine("--PLIKI--");
        foreach(FileInfo plik in pliki)
        {
            Console.WriteLine(plik.Name);
        }
    }
}
```

Na początku konstruowany jest obiekt `di` klasy `DirectoryInfo`. Konstruktor otrzymuje w postaci argumentu ścieżkę dostępu do katalogu, którego zawartość ma być wypisana — to katalog bieżący oznaczony jako `..`. Następnie deklarujemy zmienne katalogi i pliki. Pierwsza z nich będzie zawierała tablicę obiektów typu `DirectoryInfo`, czyli listę podkatalogów, przypisujemy więc jej wynik działania metody `GetDirectories`:

```
 DirectoryInfo[] katalogi = di.GetDirectories();
```

Druga będzie zawierała tablicę obiektów typu `FileInfo`, czyli listę plików, przypisujemy więc jej wynik działania metody `GetFiles`:

```
 FileInfo[] pliki = di.GetFiles();
```

Ponieważ obie wymienione zmienne zawierają tablice, pozostało dwukrotne zastosowanie pętli `foreach` do odczytania ich zawartości i wyświetlenia na ekranie nazw przechowywanych obiektów. Nazwy plików i katalogów uzyskujemy przy tym przez odwołanie się do właściwości `Name`. Przykładowy efekt wykonania kodu z listingu 5.14 jest widoczny na rysunku 5.10.

Rysunek 5.10.
Wyświetlenie listy podkatalogów i plików

```
C:\cs>Program.exe
Zawartość katalogu bieżącego:
--PODKATALOGI--
biblioteki
bin
dane
PierwszaAplikacja
--PLIKI--
Data.cs
ExceptionWithStatus.cs
Program.cs
Program.exe
```

Proste wyświetlenie zawartości katalogu z pewnością nikomu nie sprawiło żadnego problemu, jednak klasa `DirectoryInfo` udostępnia również przeciążone wersje metod `GetDirectories` i `GetFiles`, które dają większe możliwości. Pozwalają bowiem na pobranie nazw tylko tych plików i katalogów, które pasują do określonego wzorca. Przyjmują one parametr typu `string`, pozwalający określić, które nazwy zaakceptować, a które odrzucić. Aby zobaczyć, jak to wygląda w praktyce, napiszemy program, który będzie wyświetlał pliki z dowolnego katalogu o nazwach pasujących do określonego wzorca. Nazwa katalogu oraz wzorzec będą wczytywane z wiersza poleceń podczas uruchamiania aplikacji. Spójrzmy zatem na kod przedstawiony na listingu 5.15.

Listing 5.15. Lista plików pasujących do określonego wzorca

```
using System;
using System.IO;

public class Program
{
    public static void Main(String[] args)
    {
        if(args.Length < 2)
        {
            Console.WriteLine(
                "Wywołanie programu: Program katalog wzorzec");
        }
    }
}
```

```
        return;
    }
    String katalog = args[0];
    String wzorzec = args[1];

    DirectoryInfo di = new DirectoryInfo(katalog);

    if(!di.Exists)
    {
        Console.WriteLine("Brak dostępu do katalogu: {0}", katalog);
        return;
    }

    FileInfo[] pliki;
    try
    {
        pliki = di.GetFiles(wzorzec);
    }
    catch(Exception)
    {
        Console.WriteLine("Wzorzec {0} jest niepoprawny.", wzorzec);
        return;
    }
    Console.WriteLine("Pliki w katalogu {0} pasujące do wzorca {1}:",
                      katalog, wzorzec);
    foreach(FileInfo plik in pliki)
    {
        Console.WriteLine(plik.Name);
    }
}
```

Zaczynamy od sprawdzenia, czy podczas wywołania zostały podane przynajmniej dwa argumenty (lekcia 15.). Jeśli nie, informujemy o tym użytkownika, wyświetlając informację na ekranie, i kończymy działanie aplikacji. Jeśli tak, zakładamy, że pierwszy z nich zawiera nazwę katalogu, którego zawartość ma zostać wyświetlona, a drugi — wzorzec, z którym ta zawartość będzie porównywana. Nazwę katalogu przypisujemy zmiennej `katalog`, a wzorca — zmiennej `wzorzec`. Następnie tworzymy nowy obiekt typu `DirectoryInfo`, przekazując w konstruktorze wartość zmiennej `katalog`, oraz badamy, czy tak określony katalog istnieje na dysku. To sprawdzenie jest wykonywane za pomocą instrukcji warunkowej `if`, badającej stan właściwości `Exists`. Jeśli właściwość ta jest równa `false`, oznacza to, że katalogu nie ma bądź z innych względów nie można otrzymać do niego praw dostępu, jest więc wyświetlana informacja o błędzie i program kończy działanie (wywołanie instrukcji `return`).

Jeśli jednak katalog istnieje, następuje próba odczytania jego zawartości przez wywołanie metody `GetFiles` i przypisanie zwróconej przez nią tablicy zmiennej `pliki`. Wykorzystujemy tu przeciążoną wersję metody, która przyjmuje argument typu `string` określający wzorzec, do którego musi pasować nazwa pliku, aby została uwzględniona w zestawieniu. Wywołanie jest ujęte w blok `try...catch`, ponieważ w przypadku gdyby wzorzec był nieprawidłowy (np. równy `null`), zostanie zgłoszony wyjątek (lekcie z rozdziału 4.). Dzięki temu blokowi wyjątek może zostać przechwycony, a stosowna infor-

macja może pojawić się na ekranie. Samo wyświetlenie listy katalogów odbywa się w taki sam sposób jak w poprzednim przykładzie. Efekt przykładowego działania programu został przedstawiony na rysunku 5.11.

Rysunek 5.11.

Efekt działania programu wyświetlającego nazwy plików pasujące do wybranego wzorca

The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command entered is 'Program.exe c:\cs\ P*.cs'. The output shows a list of files in the 'c:\cs\' directory that match the pattern 'P*.cs': 'Program.cs', 'Punkt.cs', and 'Punkt3D.cs'. The prompt 'c:\cs>-' is visible at the bottom.

Uwaga! Nazwa katalogu nie może zawierać nieprawidłowych znaków, gdyż spowoduje to powstanie wyjątku. Nie jest on przechwytywany, aby nie rozbudowywać dodatkowo kodu przykładu. Ta kwestia zostanie poruszona w kolejnej części lekcji.

Wzorzec stosowany jako filtr nazw plików może zawierać znaki specjalne * i ?. Pierwszy z nich zastępuje dowolną liczbę innych znaków, a drugi dokładnie jeden znak. Oznacza to, że do przykładowego wzorca Pro* będą pasowały ciągi Program, Programy, Promocja, Profesjonalista itp., a do wzorca Warszaw? — ciągi Warszawa, Warszawy, Warszawo itp. Jeśli więc chcemy np. uzyskać wszystkie pliki o rozszerzeniu cs, to powinniśmy zastosować wzorzec *.cs, a gdy potrzebna jest lista plików o rozszerzeniu exe rozpoczynających się od znaku P — wzorzec P*.exe.

Tworzenie katalogów

Do tworzenia katalogów służy metoda `Create` klasy `DirectoryInfo`. Jeżeli katalog istnieje już na dysku, metoda nie robi nic, jeśli natomiast nie może zostać utworzony (np. zostało użyte określenie nieistniejącego dysku), zostanie zgłoszony wyjątek `IOException`. Metoda tworzy również wszystkie brakujące podkatalogi w hierarchii. Jeśli na przykład istnieje dysk C:, a w nim katalog `dane`, to gdy argumentem będzie ciąg:

c:\dane\pliki\zrodłowe

zostanie utworzony podkatalog `pliki`, a w nim podkatalog `zrodłowe`. Należy też pamiętać, że jeśli w użytej nazwie katalogu znajdują się nieprawidłowe znaki, wywołanie konstruktora spowoduje wygenerowanie wyjątku `ArgumentException`. (Znaki, których w danym systemie operacyjnym nie można używać w ścieżkach dostępu do katalogów i plików, można odczytać z właściwości `InvalidPathChars` klasy `Path` zdefiniowanej w przestrzeni nazw `System.IO`).

Napiszmy zatem program, który w wierszu poleceń będzie przyjmował nazwę katalogu i będzie go tworzył. Kod takiej aplikacji został zaprezentowany na listingu 5.16.

Listing 5.16. Program tworzący katalog o zadanej nazwie

```
using System;
using System.IO;

public class Program
{
```

```
public static void Main(String[] args)
{
    if(args.Length < 1)
    {
        Console.WriteLine("Wywołanie programu: Program katalog");
        return;
    }

    String katalog = args[0];
    DirectoryInfo di;

    try{
        di = new DirectoryInfo(katalog);
    }
    catch(ArgumentException)
    {
        Console.WriteLine(
            "Nazwa {0} zawiera nieprawidłowe znaki.", katalog);
        return;
    }

    if(di.Exists)
    {
        Console.WriteLine("Katalog {0} już istnieje", katalog);
        return;
    }

    try
    {
        di.Create();
    }
    catch(IOException)
    {
        Console.WriteLine(
            "Katalog {0} nie może być utworzony.", katalog);
        return;
    }
    Console.WriteLine("Katalog {0} został utworzony.", katalog);
}
}
```

Zaczynamy od sprawdzenia, czy w wywołaniu programu został podany co najmniej jeden argument. Jeśli nie, czyli jeśli prawdziwy jest warunek `args.Length < 1`, wyświetlamy informacje o tym, jak powinno wyglądać wywołanie, i kończymy działanie aplikacji za pomocą instrukcji `return`. W sytuacji, kiedy argument został przekazany, przyjmujemy, że jest to nazwa katalogu do utworzenia, i zapisujemy ją w zmiennej `katalog`. Zmienna ta jest następnie używana jako argument konstruktora obiektu typu `DirectoryInfo`:

```
DirectoryInfo di = new DirectoryInfo(katalog);
```

Ta instrukcja jest ujęta w blok `try...catch`, ponieważ w przypadku gdy w argumencie konstruktora znajdą się nieprawidłowe znaki (znaki, które nie mogą być częścią nazwy katalogu), zostanie wygenerowany wyjątek `ArgumentException`. Gdyby tak się stało, na ekranie pojawiłby się komunikat informacyjny (wyświetlany w bloku `catch`), a działanie programu zostało zakończone przy użyciu instrukcji `return`.

W kolejnym kroku badamy, czy katalog o wskazanej nazwie istnieje na dysku, sprawdzając za pomocą instrukcji `if` stan właściwości `Exists`. Jeśli bowiem katalog istnieje (`Exists` ma wartość `true`), nie ma potrzeby jego tworzenia — wyświetlany więc wtedy stosowną informację i kończymy działanie programu. Jeśli katalog nie istnieje, trzeba go utworzyć, wywołując metodę `Create`:

```
di.Create();
```

Instrukcja ta jest ujęta w blok `try...catch` przechwytyujący wyjątek `IOException`. Występuje on wtedy, gdy operacja tworząca katalog zakończy się niepowodzeniem. Jeśli więc wystąpi wyjątek, wyświetlana jest informacja o niemożności utworzenia katalogu, a jeśli nie wystąpi — o tym, że katalog został utworzony.

Usuwanie katalogów

Do usuwania katalogów służy metoda `Delete` klasy `DirectoryInfo`. Usuwany katalog musi być pusty. Jeśli nie jest pusty, nie istnieje lub jest to katalog bieżący aplikacji, zostanie zgłoszony wyjątek `IOException`. Jeśli natomiast aplikacja nie będzie miała wystarczających praw dostępu, zostanie zgłoszony wyjątek `SecurityException` (klasa `SecurityException` jest zdefiniowana w przestrzeni nazw `System.Security`). Przykładowy program usuwający katalog, o nazwie przekazanej w postaci argumentu z wiersza poleceń, jest widoczny na listingu 5.17.

Listing 5.17. Program usuwający wskazany katalog

```
using System;
using System.IO;
using System.Security;

public class Program
{
    public static void Main(String[] args)
    {
        // tutaj początek kodu z listingu 5.16

        if(!di.Exists)
        {
            Console.WriteLine("Katalog {0} nie istnieje.", katalog);
            return;
        }

        try
        {
            di.Delete();
        }
        catch(IOException)
        {
            Console.WriteLine("Katalog {0} nie może zostać usunięty.", katalog);
            return;
        }
        catch(SecurityException)
        {
            Console.WriteLine("Brak uprawnień do usunięcia katalogu {0}.", katalog);
            return;
        }
    }
}
```

```
        }
        Console.WriteLine("Katalog {0} został usunięty.", katalog);
    }
}
```

Pierwsza część kodu aplikacji jest taka sama jak w przykładzie z listingu 5.16, dlatego też została pominięta. Na początku trzeba po prostu zbadać, czy został przekazany argument, oraz utworzyć nowy obiekt typu `DirectoryInfo`, uwzględniając przy tym fakt, że aplikacja może otrzymać nieprawidłowe dane. Następnie sprawdzane jest, czy istnieje katalog, który ma być usunięty. Jeśli nie (`if (!di.Exists)`), nie ma czego usuwać i program kończy działanie, wyświetlając stosowny komunikat. Jeśli natomiast istnieje, jest wykonywana metoda `Delete` usuwająca go z dysku.

Należy jednak pamiętać, że ta operacja może nie zakończyć się powodzeniem. Są dwa główne powody. Pierwszy to nieprawidłowa nazwa (nieistniejąca ścieżka dostępu), drugi to brak odpowiednich uprawnień. Dlatego też instrukcja usuwająca katalog została ujęta w blok `try...catch`. Przechwytywane są dwa typy wyjątków obsługujących opisane sytuacje: `IOException` — nieprawidłowe wskazanie katalogu bądź inny błąd wejścia-wyjścia, `SecurityException` — brak uprawnień. Wyjątek `SecurityException` jest zdefiniowany w przestrzeni nazw `System.Security`, dlatego też na początku aplikacji znajduje się odpowiednia dyrektywa `using`.

Operacje na plikach

Klasa `FileInfo`

Klasa `FileInfo` pozwala na wykonywanie podstawowych operacji na plikach, takich jak ich tworzenie i usuwanie, operacje na nazwach czy pobieranie parametrów, np. czasu utworzenia bądź modyfikacji. Jest to zatem odpowiednik `DirectoryInfo`, ale operujący na plikach. Większość jej właściwości jest odziedziczona po klasie `File` → `SystemInfo` — w tabeli 5.12 natomiast uwzględniono kilka nowych. Metody klasy `FileInfo` zostały przedstawione w tabeli 5.13. Część z nich pozwala na wykonywanie operacji związanych z odczytem i zapisem danych, jednak tymi tematami zajmiemy się dopiero w kolejnej lekcji.

Tabela 5.12. Właściwości klasy `FileInfo`

Typ	Właściwość	Opis
<code>DirectoryInfo</code>	<code>Directory</code>	Zawiera obiekt katalogu nadzawanego.
<code>string</code>	<code>DirectoryName</code>	Zawiera nazwę katalogu nadzawanego.
<code>bool</code>	<code>IsReadOnly</code>	Ustala, czy plik ma atrybut tylko do odczytu.
<code>long</code>	<code>Length</code>	Okręsła wielkość pliku w bajtach.

Tabela 5.13. Metody klasy *FileInfo*

Typ zwracany	Metoda	Opis
StreamWriter	AppendText	Tworzy obiekt typu <i>StreamWriter</i> pozwalający na dopisywanie tekstu do pliku.
FileInfo	CopyTo	Kopiuje istniejący plik do nowego.
FileStream	Create	Tworzy nowy plik.
StreamWriter	CreateText	Tworzy obiekt typu <i>StreamWriter</i> pozwalający na zapisywanie danych w pliku tekstowym.
void	Decrypt	Odszyfrowuje plik zakodowany za pomocą metody <i>Encrypt</i> .
void	Encrypt	Szyfruje plik.
void	MoveTo	Przenosi plik do wskazanej lokalizacji.
FileStream	Open	Otwiera plik.
FileStream	OpenRead	Otwiera plik w trybie tylko do odczytu.
StreamReader	OpenText	Tworzy obiekt typu <i>StreamReader</i> odczytujący dane tekstowe w kodowaniu UTF-8 z istniejącego pliku tekstowego.
FileStream	OpenWrite	Otwiera plik w trybie tylko do zapisu.
FileInfo	Replace	Zamienia zawartość wskazanego pliku na treść pliku bieżącego, tworząc jednocześnie kopię zapasową oryginalnych danych.

Tworzenie pliku

Do tworzenia plików służy metoda *Create* klasy *FileInfo*. Jeżeli plik istnieje na dysku, metoda nie robi nic; jeśli natomiast nie może zostać utworzony (np. w ścieżce dostępu występują nieprawidłowe znaki bądź określenie nieistniejącego dysku), zostanie zgłoszony jeden z wyjątków:

- ◆ *UnauthorizedAccessException* — niewystarczające prawa dostępu lub wskazany został istniejący plik z atrybutem *read-only* (tylko do odczytu);
- ◆ *ArgumentException* — ścieżka jest ciągiem o zerowej długości, zawiera jedynie białe znaki lub zawiera znaki nieprawidłowe;
- ◆ *ArgumentNullException* — jako ścieżkę dostępu przekazano wartość *null*;
- ◆ *PathTooLongException* — ścieżka dostępu zawiera zbyt wiele znaków;
- ◆ *DirectoryNotFoundException* — ścieżka dostępu wskazuje nieistniejący katalog lub plik;
- ◆ *IOException* — wystąpił błąd wejścia-wyjścia;
- ◆ *NotSupportedException* — ścieżka dostępu ma nieprawidłowy format.

Wartością zwracaną przez *Create* jest obiekt typu *FileStream* pozwalający na wykonywanie operacji na pliku, takich jak zapis i odczyt danych. Jest to jednak temat, którym zajmiemy się dopiero w lekcji 27. Na razie interesuje nas jedynie utworzenie pliku. Sposób wykonania takiej czynności został pokazany na listingu 5.18.

Listing 5.18. Utworzenie pliku

```
using System;
using System.IO;

public class Program
{
    public static void Main(String[] args)
    {
        if(args.Length < 1)
        {
            Console.WriteLine("Wywołanie programu: Program plik");
            return;
        }
        String plik = args[0];
        FileInfo fi;

        try
        {
            fi = new FileInfo(plik);
        }
        catch(ArgumentException)
        {
            Console.WriteLine(
                "Nazwa {0} zawiera nieprawidłowe znaki.", plik);
            return;
        }

        if(fi.Exists)
        {
            Console.WriteLine("Plik {0} już istnieje", plik);
            return;
        }

        FileStream fs;
        try
        {
            fs = fi.Create();
        }
        catch(Exception)
        {
            Console.WriteLine("Plik {0} nie może być utworzony.", plik);
            return;
        }
        /*
         tutaj można wykonać operacje na pliku
        */
        fs.Close();
        Console.WriteLine("Plik {0} został utworzony.", plik);
    }
}
```

Początek kodu jest bardzo podobny do przykładów operujących na katalogach. Nazwa pliku odczytana z wiersza poleceń jest zapisywana w zmiennej `plik`. Następnie jest tworzony obiekt typu `FileInfo`, a w konstruktorze jest przekazywana wartość wspomnianej zmiennej:

```
FileInfo fi = new FileInfo(plik);
```

Przechwytywany jest też wyjątek `ArgumentException`.

Dalej sprawdzane jest, czy plik o wskazanej nazwie istnieje. Jeśli tak, nie ma potrzeby jego tworzenia, więc program kończy pracę. Jeśli nie, tworzona jest zmieniąca typu `FileNotFoundException` i jest jej przypisywany rezultat działania metody `Create` obiektu `fi`. Wywołanie to jest ujęte w blok `try...catch`. Ponieważ w trakcie tworzenia pliku może wystąpić wiele wyjątków, jest przechwytywany ogólny, klasy `Exception`. A zatem niezależnie od przyczyny niepowodzenia zostanie wyświetlona jedna informacja.

Jeśli utworzenie pliku się powiedzie, jest wywoływana metoda `Close` obiektu `fs`, zamkająca strumień danych — oznacza to po prostu koniec operacji na pliku. W miejscu oznaczonym komentarzem można by natomiast dopisać instrukcje wykonujące inne operacje, jak np. zapis lub odczyt danych.

Pobieranie informacji o pliku

Zaglądając do tabel 5.8 i 5.12, znajdziemy wiele właściwości pozwalających na uzyskanie podstawowych informacji o pliku. Możemy więc pokusić się o napisanie programu, który z wiersza poleceń odczyta ścieżkę dostępu, a następnie wyświetli takie dane, jak atrybuty, czas utworzenia czy rozmiar pliku. Kod tak działającej aplikacji został umieszczony na listingu 5.19.

Listing 5.19. Uzyskanie podstawowych informacji o pliku

```
using System;
using System.IO;

public class Program
{
    public static void Main(String[] args)
    {
        // tutaj początek kodu z listingu 5.18

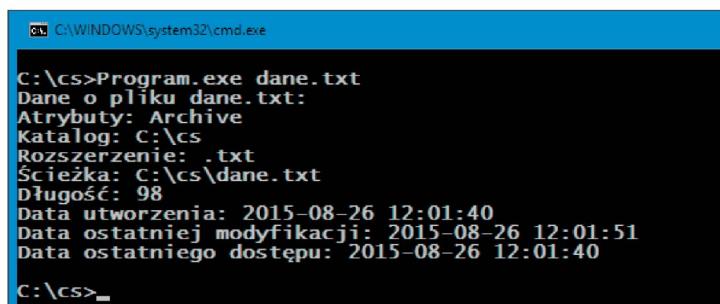
        if(!fi.Exists)
        {
            Console.WriteLine("Plik {0} nie istnieje.", plik);
            return;
        }
        Console.WriteLine("Dane o pliku {0}: ", plik);
        Console.WriteLine("Atrybuty: {0}", fi.Attributes);
        Console.WriteLine("Katalog: {0}", fi.Directory);
        Console.WriteLine("Rozszerzenie: {0}", fi.Extension);
        Console.WriteLine("Ścieżka: {0}", fi.FullName);
        Console.WriteLine("Długość: {0}", fi.Length);
        Console.WriteLine("Data utworzenia: {0}", fi.CreationTime);
        Console.WriteLine("Data ostatniej modyfikacji: {0}", fi.LastWriteTime);
        Console.WriteLine("Data ostatniego dostępu: {0}", fi.LastAccessTime);
    }
}
```

Struktura tego kodu jest na tyle prosta, że nie wymaga długich wyjaśnień. Pierwsza część jest taka sama jak w przypadku przykładu z listingu 5.18. Trzeba upewnić się, że w wierszu poleceń został przekazany przynajmniej jeden argument, a następnie

jego wartość zapisać w zmiennej plik, która zostanie użyta do utworzenia obiektu typu FileInfo. Obiekt ten, zapisany w zmiennej fi, jest najpierw używany do sprawdzenia, czy taki plik istnieje, a następnie do wyświetlenia różnych informacji. Odbywa się to przez dostęp do właściwości Attributes, Directory, Extension, FullName, Length, CreationTime, LastWriteTime i LastAccessTime. Efekt przykładowego wywołania programu został zaprezentowany na rysunku 5.12.

Rysunek 5.12.

Efekt działania aplikacji podającej informacje o wybranym pliku



A screenshot of a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command entered is 'C:\cs>Program.exe dane.txt'. The output shows details about the file 'dane.txt':
Dane o pliku dane.txt:
Atrybuty: Archive
Katalog: C:\cs
Rozszerzenie: .txt
Ścieżka: C:\cs\dane.txt
Długość: 98
Data utworzenia: 2015-08-26 12:01:40
Data ostatniej modyfikacji: 2015-08-26 12:01:51
Data ostatniego dostępu: 2015-08-26 12:01:40

Kasowanie pliku

Skoro potrafimy już tworzyć pliki oraz odczytywać informacje o nich, powinniśmy także wiedzieć, w jaki sposób je usuwać. Metodę wykonującą to zadanie znajdziemy w tabeli 5.9, ma ona nazwę Delete. Pliki usuwa się więc tak jak katalogi, z tą różnicą, że korzystamy z klasy FileInfo, a nie DirectoryInfo. Przykład programu, który usuwa plik o nazwie (ścieżce dostępu) przekazanej jako argument wywołania z wiersza poleceń, został zaprezentowany na listingu 5.20.

Listing 5.20. Usuwanie wybranego pliku

```
using System;
using System.IO;

public class Program
{
    public static void Main(String[] args)
    {
        // tutaj początek kodu z listingu 5.18

        if(!fi.Exists)
        {
            Console.WriteLine("Plik {0} nie istnieje.", plik);
            return;
        }

        try
        {
            fi.Delete();
        }
        catch(Exception)
        {
            Console.WriteLine("Plik {0} nie może zostać usunięty.", plik);
            return;
        }
    }
}
```

```
        Console.WriteLine("Plik {0} został usunięty.", plik);
    }
}
```

Struktura kodu jest podobna do programów z listingów 5.17 i 5.18. Po odczytaniu nazwy pliku z wiersza polecień tworzony jest obiekt typu `FileInfo`. Ta część jest taka sama jak w wymienionych przykładach. Następnie za pomocą wywołania metody `Exists` jest sprawdzane, czy wskazany plik istnieje. Jeśli istnieje, jest podejmowana próba jego usunięcia za pomocą metody `Delete`. Ponieważ w przypadku niemożności usunięcia pliku zostanie wygenerowany odpowiedni wyjątek, wywołanie to jest ujęte w blok `try...catch`.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 26.1

Napisz program wyświetlający podstawowe informacje o katalogu, takie jak nazwa katalogu nadzrzednego, czas jego utworzenia, atrybuty itp.

Ćwiczenie 26.2

Napisz program wyświetlający listę podkatalogów wskazanego katalogu o nazwach pasujących do określonego wzorca.

Ćwiczenie 26.3

Napisz program wyświetlający zawartość katalogu bieżącego. Do pobrania danych użyj metody `GetFileSystemInfos`.

Ćwiczenie 26.4

Napisz program usuwający plik lub katalog o nazwie przekazanej z wiersza polecień. Program powinien zapytać użytkownika o potwierdzenie chęci wykonania tej operacji.

Ćwiczenie 26.5

Napisz program wyświetlający sumaryczną wielkość plików zawartych w katalogu o nazwie przekazanej z wiersza polecień.

Lekcja 27. Zapis i odczyt plików

Lekcja 26. poświęcona była wykonywaniu operacji na systemie plików, nie obejmowała jednak tematów związanych z zapisem i odczytem danych. Tymi zagadnieniami zajmiemy się zatem w bieżącej, 26. lekcji. Sprawdzimy więc, jakie są sposoby zapisu

i odczytu danych, jak posługiwać się plikami tekstowymi i binarnymi oraz co to są strumienie. Przedstawione zostaną też bliżej takie klasy, jak: `FileStream`, `StreamReader`, `StreamWriter`, `BinaryReader` i `BinaryWriter`. Zobaczmy również, jak zapisać w pliku dane wprowadzane przez użytkownika z klawiatury.

Klasa `FileStream`

Klasa `FileStream` daje możliwość wykonywania różnych operacji na plikach. Pozwala na odczytywanie i zapisywanie danych w pliku oraz przemieszczanie się po pliku. W rzeczywistości tworzy ona strumień powiązany z plikiem (już sama nazwa na to wskazuje), jednak to pojęcie będzie wyjaśnione w dalszej części lekcji. Właściwości udostępniane przez `FileStream` są zebrane w tabeli 5.14, natomiast metody — 5.15.

Tabela 5.14. Właściwości klasy `FileStream`

Typ	Właściwość	Opis
bool	CanRead	Określa, czy ze strumienia można odczytywać dane.
bool	CanSeek	Określa, czy można przemieszczać się po strumieniu.
bool	CanTimeout	Określa, czy strumień obsługuje przekroczenie czasu żądania.
bool	CanWrite	Określa, czy do strumienia można zapisywać dane.
IntPtr	Handle	Zawiera systemowy deskryptor otwartego pliku powiązanego ze strumieniem. Właściwość przestarzała zastąpiona przez <code>SafeFileHandle</code> .
bool	IsAsync	Określa, czy strumień został otwarty w trybie synchronicznym, czy asynchronicznym.
long	Length	Określa długość strumienia w bajtach.
string	Name	Zawiera ciąg określający nazwę strumienia.
long	Position	Określa aktualną pozycję w strumieniu.
int	ReadTimeout	Określa, jak długo strumień będzie czekał na operację odczytu, zanim wystąpi przekroczenie czasu żądania.
SafeFileHandle	SafeFileHandle	Zawiera obiekt reprezentujący deskryptor otwartego pliku.
int	WriteTimeout	Określa, jak długo strumień będzie czekał na operację zapisu, zanim wystąpi przekroczenie czasu żądania.

Tabela 5.15. Wybrane metody klasy `FileStream`

Typ zwracany	Metoda	Opis
IAsyncResult	BeginRead	Rozpoczyna asynchroniczną operację odczytu.
IAsyncResult	BeginWrite	Rozpoczyna asynchroniczną operację zapisu.
void	Close	Zamyka strumień i zwalnia związane z nim zasoby.
void	CopyTo	Kopiuje zawartość bieżącego strumienia do strumienia docelowego przekazanego w postaci argumentu.
void	Dispose	Zwalnia związane ze strumieniem zasoby.

Tabela 5.15. Wybrane metody klasy *FileStream* — ciąg dalszy

Typ zwracany	Metoda	Opis
int	EndRead	Oczekuje na zakończenie asynchronicznej operacji odczytu.
void	EndWrite	Oczekuje na zakończenie asynchronicznej operacji zapisu.
void	Flush	Opróżnia bufor i zapisuje znajdujące się w nim dane.
FileSecurity	GetAccessControl	Zwraca obiekt określający prawa dostępu do pliku.
void	Lock	Blokuje innym procesom dostęp do strumienia.
int	Read	Odczytuje blok bajtów i zapisuje je we wskazanym buforze.
int	ReadByte	Odczytuje pojedynczy bajt.
long	Seek	Ustawia wskaźnik pozycji w strumieniu.
void	SetAccessControl	Ustala prawa dostępu do pliku powiązanego ze strumieniem.
void	SetLength	Ustawia długość strumienia.
void	Unlock	Usuwa blokadę nałożoną przez wywołanie metody Lock.
void	Write	Zapisuje blok bajtów w strumieniu.
void	WriteByte	Zapisuje pojedynczy bajt w strumieniu.

Aby wykonywać operacje na pliku, trzeba utworzyć obiekt typu *FileStream*. Jak to zrobić? Można bezpośrednio wywołać konstruktor lub też użyć jednej z metod klasy *FileInfo*. Jeśli spojrzymy do tabeli 5.13, zobaczymy, że metody *Create*, *Open*, *OpenRead* i *OpenWrite* zwracają właśnie obiekty typu *FileStream*. Obiektu tego typu użyliśmy też w programie z listingu 5.18.

Klasa *FileStream* udostępnia kilkanaście konstruktorów. Dla nas jednak najbardziej interesujący jest ten przyjmujący dwa argumenty: nazwę pliku oraz tryb dostępu do pliku. Jego deklaracja jest następująca:

```
public FileStream (string path, FileMode mode)
```

Tryb dostępu jest określony przez typ wyliczeniowy *FileMode*. Ma on następujące składowe:

- ◆ *Append* — otwarcie pliku, jeśli istnieje, i przesunięcie wskaźnika pozycji na jego koniec lub utworzenie pliku;
- ◆ *Create* — utworzenie nowego pliku lub nadpisanie istniejącego;
- ◆ *CreateNew* — utworzenie nowego pliku; jeśli plik istnieje, zostanie wygenerowany wyjątek *IOException*;
- ◆ *Open* — otwarcie istniejącego pliku; jeśli plik nie istnieje, zostanie wygenerowany wyjątek *FileNotFoundException*;
- ◆ *OpenOrCreate* — otwarcie lub utworzenie pliku;
- ◆ *Truncate* — otwarcie istniejącego pliku i obcięcie jego długości do 0.

Przykładowe wywołanie konstruktora może więc mieć postać:

```
FileStream fs = new FileStream ("c:\\pliki\\dane.txt", FileMode.Create);
```

Podstawowe operacje odczytu i zapisu

Omawianie operacji na plikach zaczniemy od tych wykonywanych bezpośrednio przez metody klasy `FileStream`, w dalszej części lekcji zajmiemy się natomiast dodatkowymi klasami pośredniczącymi. Zaczniemy od zapisu danych; umożliwiają to metody `Write` i `WriteByte`.

Zapis danych

Założymy, że chcemy przechować w pliku ciąg liczb wygenerowanych przez program. Niezbędne będzie zatem użycie jednej z metod zapisujących dane. Może to być `WriteByte` lub `Write`. Pierwsza zapisuje jeden bajt, który należy jej przekazać w postaci argumentu, natomiast druga — cały blok danych. My posłużymy się metodą `Write`. Ma ona deklarację:

```
public void Write (byte[] array, int offset, int count)
```

przyjmuje więc trzy argumenty:

- ◆ `array` — tablicę bajtów, które mają zostać zapisane;
- ◆ `offset` — pozycję w tablicy `array`, od której mają być pobierane bajty;
- ◆ `count` — liczbę bajtów do zapisania.

Gdyby wykonanie tej metody nie zakończyło się sukcesem, zostanie wygenerowany jeden z wyjątków:

- ◆ `ArgumentNullException` — pierwszy argument ma wartość `null`;
- ◆ `ArgumentException` — wskazany został nieprawidłowy zakres danych (wykraczający poza rozmiary tablicy);
- ◆ `ArgumentOutOfRangeException` — drugi lub trzeci argument ma wartość ujemną;
- ◆ `IOException` — wystąpił błąd wejścia-wyjścia;
- ◆ `ObjectDisposedException` — strumień został zamknięty;
- ◆ `NotSupportedException` — bieżący strumień nie umożliwia operacji zapisu.

Zatem program wykonujący postawione wyżej zadanie (zapis liczb do pliku) będzie miał postać widoczną na listingu 5.21.

Listing 5.21. *Zapis danych do pliku*

```
using System;
using System.IO;

public class Program
{
    public static void Main(String[] args)
    {
        if(args.Length < 1)
        {
```

```
Console.WriteLine("Wywołanie programu: Program plik");
    return;
}
String plik = args[0];
int ile = 100;

byte[] dane = new byte[ile];
for(int i = 0; i < ile; i++)
{
    if(i % 2 == 0)
        dane[i] = 127;
    else
        dane[i] = 255;
}

FileStream fs;
try
{
    fs = new FileStream(plik, FileMode.Create);
}
catch(Exception)
{
    Console.WriteLine(
        "Otwarcie pliku {0} się nie powiodło.", plik);
    return;
}

try
{
    fs.Write(dane, 0, ile);
}
catch(Exception)
{
    Console.WriteLine("Zapis nie został dokonany.");
    return;
}
fs.Close();
Console.WriteLine("Zapis został dokonany.");
}
```

Na początku sprawdzamy, czy podczas wywołania programu została podana nazwa pliku; jeśli tak, zapisujemy ją w zmiennej `plik` oraz deklarujemy zmenną `ile`, której wartość będzie określała, ile liczb ma być zapisanych w pliku. Następnie tworzymy nową tablicę o rozmiarze wskazywanym przez `ile` i wypełniamy ją danymi. W przykładzie przyjęto po prostu, że komórki o indeksach podzielnych przez 2 otrzymają wartość 127, a te o indeksach niepodzielnych — 255. Po wykonaniu tych czynności tworzony jest i przypisywany zmiennej `fs` nowy obiekt typu `FileStream`. Wywołanie konstruktora ujęte jest w blok `try..catch` przechwytyjący ewentualny wyjątek, powstały, gdyby operacja ta zakończyła się niepowodzeniem. Trybem dostępu jest `FileMode.Create`, co oznacza, że jeśli plik o podanej nazwie nie istnieje, to zostanie utworzony, a jeśli istnieje, zostanie otwarty, a jego dotychczasowa zawartość będzie skasowana.

Po utworzeniu obiektu `fs` wywoływana jest jego metoda `Write`. Przyjmuje ona, zgodnie z przedstawionym wyżej opisem, trzy argumenty:

- ◆ `dane` — tablica z danymi;
- ◆ `0` — indeks komórki, od której ma się zacząć pobieranie danych do zapisu;
- ◆ `ile` — całkowita liczba komórek, które mają zostać zapisane (w tym przypadku — wszystkie).

Wywołanie metody `Write` jest również ujęte w blok `try...catch` przechwytyujący wyjątek, który mógłby powstać, gdyby z jakichś powodów operacja zapisu nie mogła zostać dokonana. Na końcu kodu znajduje się wywołanie metody `Close`, która zamkniemy plik (strumień danych).

Po uruchomieniu programu otrzymamy plik o wskazanej przez nas nazwie, zawierający wygenerowane dane. O tym, że zostały one faktycznie zapisane, możemy się przekonać, odczytując jego zawartość. Warto zatem napisać program, który wykoną taką czynność. To zadanie zostanie zrealizowane w kolejnej części lekcji.

Odczyt danych

Do odczytu danych służą metody `ReadByte` i `Read`. Pierwsza odczytuje pojedynczy bajt i zwraca go w postaci wartości typu `int` (w przypadku osiągnięcia końca pliku zwracana jest wartość `-1`). Druga metoda pozwala na odczyt całego bloku bajtów, jej więc użyjemy w kolejnym przykładzie. Deklaracja jest tu następująca:

```
public override int Read (byte[] array, int offset, int count)
```

Do dyspozycji, podobnie jak w przypadku `Write`, mamy więc trzy argumenty:

- ◆ `array` — tablicę bajtów, w której zostaną zapisane odczytane dane;
- ◆ `offset` — pozycję w tablicy `array`, od której mają być zapisywane bajty;
- ◆ `count` — liczbę bajtów do odczytania.

Gdy wykonanie tej metody nie zakończy się sukcesem, zostanie wygenerowany jeden z wyjątków przedstawionych przy opisie metody `Write`.

Jeśli więc chcemy odczytać plik z danymi wygenerowany przez program z listingu 5.21, możemy zastosować kod przedstawiony na listingu 5.22.

Listing 5.22. Odczyt danych z pliku

```
using System;
using System.IO;

public class Program
{
    public static void Main(String[] args)
    {
        if(args.Length < 1)
        {
```

```
Console.WriteLine("Wywołanie programu: Program plik");
    return;
}
String plik = args[0];
int ile = 100;

byte[] dane = new byte[ile];

FileStream fs;
try
{
    fs = new FileStream(plik, FileMode.Open);
}
catch(Exception)
{
    Console.WriteLine("Otwarcie pliku {0} nie powiodło się.", plik);
    return;
}

try
{
    fs.Read(dane, 0, ile);
    fs.Close();
}
catch(Exception)
{
    Console.WriteLine("Odczyt pliku {0} nie został dokonany.", plik);
    return;
}
Console.WriteLine("Z pliku {0} Odczytano następujące dane:", plik);
for(int i = 0; i < ile; i++)
{
    Console.WriteLine("[{0}] = {1} ", i, dane[i]);
}
```

Początek kodu jest taki sam jak w poprzednim przykładzie, z tą różnicą, że tablica dane nie jest wypełniana danymi — mają być przecież odczytane z pliku. Inny jest również tryb otwarcia pliku — jest to `FileMode.Open`. Dzięki temu, jeśli plik istnieje, zostanie otwarty, jeśli nie — zostanie zgłoszony wyjątek. Odczyt jest przeprowadzany przez wywołanie metody `Read` obiektu `fs`:

```
fs.Read(dane, 0, ile);
```

Znaczenie poszczególnych argumentów jest takie samo jak w przypadku metody `Write`, to znaczy odczytywane dane będą zapisane w tablicy dane, począwszy od komórki o indeksie 0, i zostanie odczytana liczba bajtów wskazywana przez `ile`. Wywołanie metody `Write` ujęte jest w blok `try...catch`, aby przechwycić ewentualny wyjątek, który może się pojawić, jeśli operacja odczytu nie zakończy się powodzeniem.

Po odczytaniu danych są one pobierane z tablicy i wyświetlane na ekranie w pętli `for`. Jeśli uruchomimy program, przekazując w wierszu poleceń nazwę pliku z danymi wygenerowanymi przez aplikację z listingu 5.21, przekonany się, że faktycznie zostały

one prawidłowo odczytane, tak jak jest to widoczne na rysunku 5.13. Trzeba jednak zwrócić uwagę na pewien mankament przedstawionego rozwiązania. Wymaga ono bowiem informacji o tym, ile liczb zostało zapisanych w pliku. Jeśli liczba ta zostanie zmieniona, odczyt nie będzie prawidłowy. Jak rozwiązać ten problem? Otóż można dodatkowo zapisywać w pliku informacje o tym, ile zawiera on liczb — takie rozwiązanie zostanie przedstawione w dalszej części rozdziału — ale można też użyć do odczytu metody ReadByte (bądź też skorzystać z informacji zwracanej przez metodę Read). Jak? Niech pozostało to ćwiczeniem do samodzielnego wykonania.

Rysunek 5.13.

Wyświetlenie danych odczytanych z pliku

```
C:\Windows\system32\cmd.exe
C:\cs>Program.exe dane.bin
Z pliku dane.bin Odczytano następujące dane:
[0] = 127
[1] = 255
[2] = 127
[3] = 255
[4] = 127
[5] = 255
[6] = 127
```

Operacje strumieniowe

W C# operacje wejścia-wyjścia, takie jak zapis i odczyt plików, są wykonywane za pomocą strumieni. **Strumień** to abstrakcyjny ciąg danych, który działa, w uproszczeniu, w taki sposób, że dane wprowadzone w jednym jego końcu pojawiają się na drugim. Strumienie mogą być wejściowe i wyjściowe, a także dwukierunkowe — te są jednak rzadziej spotykane. W uproszczeniu można powiedzieć, że **strumienie wyjściowe** mają początek w aplikacji i koniec w innym urządzeniu, np. na ekranie czy w pliku, umożliwiają zatem wyprowadzanie danych z programu. **Strumienie wejściowe** działają odwrotnie. Ich początek znajduje się poza aplikacją (może być to np. klawiatura albo plik dyskowy), a koniec w aplikacji, czyli umożliwiają wprowadzanie danych. Co więcej, strumienie mogą umożliwiać komunikację między obiektami w obrębie jednej aplikacji, jednak w tym rozdziale będziemy zajmować się jedynie komunikacją aplikacji ze światem zewnętrznym.

Dlaczego jednak wprowadzać takie pojęcie jak „strumień”? Otóż dlatego, że upraszcza to rozwiązanie problemu transferu danych oraz ujednolica związane z tym operacje. Zamiast zastanawiać się, jak obsługiwać dane pobierane z klawiatury, jak z pliku, jak z pamięci, a jak z innych urządzeń, operujemy po prostu na abstrakcyjnym pojęciu strumienia i używamy metod zdefiniowanych w klasie `Stream` oraz klasach od niej pochodnych. Jedną z takich klas pochodnych jest stosowana już `FileStream` — będziemy z niej korzystać jeszcze w dalszej części lekcji — na razie jednak poznamy dwie inne klasy pochodne od `Stream`, pozwalające na prosty zapis i odczyt danych tekstowych.

Odczyt danych tekstowych

Do odczytu danych z plików tekstowych najlepiej użyć klasy `StreamReader`. Jeśli zajrzymy do tabeli 5.13, zobaczymy, że niektóre metody klasy `FileInfo` udostępniają obiekty typu `StreamReader` pozwalające na odczyt tekstu, można jednak również bezpośrednio użyć jednego z konstruktorów klasy `StreamReader`. Wszystkich konstruktorów

jest kilkanaście, dla nas jednak w tej chwili najbardziej interesujące są dwa. Pierwszy przyjmuje argument typu Stream, a więc można również użyć obiektu przedstawionej w tej lekcji klasy FileStream, drugi — argument typu String, który powinien zawierać nazwę pliku do odczytu. Wywołanie konstruktora może spowodować powstanie jednego z wyjątków:

- ◆ ArgumentException — gdy ścieżka dostępu (nazwa pliku) jest pustym ciągiem znaków lub też zawiera określenie urządzenia systemowego;
- ◆ ArgumentNullException — gdy argument ma wartość null;
- ◆ FileNotFoundException — gdy wskazany plik nie może zostać znaleziony;
- ◆ DirectoryNotFoundException — gdy ścieżka dostępu do pliku jest nieprawidłowa;
- ◆ IOException — gdy ścieżka dostępu ma nieprawidłowy format.

Wybrane metody klasy StreamReader zostały zebrane w tabeli 5.16, natomiast przykład programu odczytującego dane z pliku tekstowego i wyświetlającego je na ekranie znajduje się na listingu 5.23.

Tabela 5.16. Wybrane metody klasy StreamReader

Typ zwracany	Metoda	Opis
void	Close	Zamyka strumień i zwalnia związane z nim zasoby.
void	DiscardBufferData	Unieważnia dane znajdujące się w buforze.
void	Dispose	Zwalnia zasoby związane ze strumieniem.
int	Peek	Zwraca ze strumienia kolejny znak, pozostawiając go w strumieniu.
int	Read	Odczytuje ze strumienia znak lub określoną liczbę znaków.
int	ReadBlock	Odczytuje ze strumienia określoną liczbę znaków.
string	ReadLine	Odczytuje ze strumienia wiersz tekstu (ciąg znaków zakończony znakiem końca linii).
string	ReadToEnd	Odczytuje ze strumienia wszystkie dane, począwszy od bieżącej pozycji do jego końca.

Listing 5.23. Odczyt danych z pliku tekstowego

```
using System;
using System.IO;

public class Program
{
    public static void Main(String[] args)
    {
        if(args.Length < 1)
        {
            Console.WriteLine("Wywołanie programu: Program plik");
            return;
        }
        String plik = args[0];
```

```
StreamReader sr;
try
{
    sr = new StreamReader(plik);
}
catch(Exception)
{
    Console.WriteLine(
        "Otwarcie pliku {0} się nie powiodło.", plik);
    return;
}

string line;
try
{
    while ((line = sr.ReadLine()) != null)
    {
        Console.WriteLine(line);
    }
    sr.Close();
}
catch(Exception)
{
    Console.WriteLine(
        "Wystąpił błąd podczas odczytu z pliku {0}.", plik);
    return;
}
}
```

W programie pobieramy argument przekazany z wiersza poleceń, przypisujemy go zmiennej `plik` i używamy jako argumentu konstruktora klasy `StreamReader`. Utworzony obiekt jest przypisywany zmiennej `sr`. Wywołanie konstruktora jest ujęte w blok `try...catch` przechwytyujący wyjątek, który może powstać, gdy pliku wskazanego przez zmenną `plik` nie da się otworzyć (np. nie będzie go na dysku). Jeśli utworzenie obiektu typu `StreamReader` się powiedzie, jest on używany do odczytu danych. Wykorzystana została w tym celu metoda `ReadLine` odczytująca poszczególne wiersze tekstu. Każdy odczytany wiersz jest zapisywany w zmiennej pomocniczej `line` oraz wyświetlany na ekranie za pomocą instrukcji `Console.WriteLine`.

Odczyt odbywa się w pętli `while`, której warunkiem zakończenia jest:

```
(line = sr.ReadLine()) != null
```

Taka instrukcja oznacza: wywołaj metodę `ReadLine` obiektu `sr`, wynik jej działania przypisz zmiennej `line` oraz porównaj wartość tej zmiennej z wartością `null`. To porównanie jest wykonywane dla tego, że `ReadLine` zwraca `null` w sytuacji, kiedy zostanie osiągnięty koniec strumienia (w tym przypadku pliku). Na zakończenie strumień jest zamknięty za pomocą metody `Close`.

W ten sposób powstała aplikacja, która będzie wyświetlała na ekranie zawartość dowolnego pliku tekstowego o nazwie przekazanej w wierszu poleceń.

Zapis danych tekstowych

Na zapis tekstu do pliku pozwala klasa `StreamWriter`. Jej obiekty, podobnie jak w przypadku `StreamReader`, można uzyskać, wywołując odpowiednie metody klasy `FileInfo` (por. tabela 5.13) bądź też bezpośrednio wywołując jeden z konstruktorów. Istnieje kilka konstruktorów; najbardziej dla nas interesujące są dwa: przyjmujący argument typu `Stream` i przyjmujący argument typu `string`. W pierwszym przypadku można więc użyć obiektu typu `FileStream`, a w drugim ciągu znaków określającego ścieżkę dostępu do pliku. Wywołanie konstruktora może spowodować powstanie jednego z wyjątków:

- ◆ `UnauthorizedAccessException` — gdy dostęp do pliku jest zabroniony;
- ◆ `ArgumentException` — gdy ścieżka dostępu jest pustym ciągiem znaków lub też zawiera określenie urządzenia systemowego;
- ◆ `ArgumentNullException` — gdy argument ma wartość `null`;
- ◆ `DirectoryNotFoundException` — gdy ścieżka dostępu jest nieprawidłowa;
- ◆ `PathTooLongException` — gdy ścieżka dostępu lub nazwa pliku jest zbyt długa;
- ◆ `IOException` — gdy ścieżka dostępu ma nieprawidłowy format;
- ◆ `SecurityException` — gdy brak jest wystarczających uprawnień do otwarcia pliku.

Wybrane metody klasy `StreamReader` zostały zebrane w tabeli 5.17.

Tabela 5.17. Wybrane metody klasy `StreamWriter`

Typ zwracany	Metoda	Opis
<code>void</code>	<code>Close</code>	Zamyka strumień i zwalnia związane z nim zasoby.
<code>void</code>	<code>Dispose</code>	Zwalnia związane ze strumieniem zasoby.
<code>void</code>	<code>Flush</code>	Opróżnia bufor i zapisuje znajdujące się w nim dane.
<code>void</code>	<code>Write</code>	Zapisuje w pliku tekstową reprezentację wartości jednego z typów podstawowych.
<code>void</code>	<code>WriteLine</code>	Zapisuje w pliku tekstową reprezentację wartości jednego z typów podstawowych zakończoną znakiem końca wiersza.

Na uwagę zasługują metody `Write` i `WriteLine`. Otóż istnieją one w wielu przeciążonych wersjach odpowiadających poszczególnym typom podstawowym (`char`, `int`, `double`, `string` itp.) i powodują zapisanie reprezentacji tekstowej danej wartości do strumienia. Metoda `WriteLine` dodatkowo zapisuje również znak końca wiersza. Przykład programu odczytującego dane z klawiatury i zapisującego je w pliku tekstowym jest widoczny na listingu 5.24.

Listing 5.24. Program zapisujący dane w pliku tekstowym

```
using System;
using System.IO;

public class Program
{
```

```
public static void Main(String[] args)
{
    if(args.Length < 1)
    {
        Console.WriteLine("Wywołanie programu: Program plik");
        return;
    }
    String plik = args[0];

    StreamWriter sw;
    try
    {
        sw = new StreamWriter(plik);
    }
    catch(Exception)
    {
        Console.WriteLine(
            "Otwarcie pliku {0} się nie powiodło.", plik);
        return;
    }
    Console.WriteLine(
        "Wprowadzaj wiersze tekstu. Aby zakończyć, wpisz 'quit'.");
    String line;
    try
    {
        do
        {
            line = Console.ReadLine();
            sw.WriteLine(line);
        }
        while(line != "quit");
        sw.Close();
    }
    catch(Exception)
    {
        Console.WriteLine(
            "Wystąpił błąd podczas zapisu do pliku {0}.", plik);
        return;
    }
}
```

Jak działa ten program? Po standardowym sprawdzeniu, że z wiersza poleceń została przekazany argument określający nazwę pliku, jest on używany jako argument konstruktora obiektu klasy `StreamWriter`:

```
sw = new StreamWriter(plik);
```

Wywołanie konstruktora jest ujęte w blok `try...catch` przechwytyujący mogące powstać w tej sytuacji wyjątki.

Odczyt oraz zapis danych odbywa się w pętli `do...while`. Tekst wprowadzany z klawiatury jest odczytywany za pomocą metody `ReadLine` klasy `Console` (por. materiał z lekcji 25.) i zapisywany w pomocniczej zmiennej `line`:

```
line = Console.ReadLine();
```

Następnie zmieniona ta jest używana jako argument metody `WriteLine` obiektu `sr` (klasy `StreamReader`):

```
sw.WriteLine(line);
```

Pętla kończy się, kiedy `line` ma wartość `quit`, czyli kiedy z klawiatury zostanie wprowadzone słowo `quit`. Po jej zakończeniu strumień jest zamknięty za pomocą metody `Close`.

Zapis danych binarnych

Do zapisu danych binarnych służy klasa `BinaryWriter`. Udostępnia ona konstruktory zebrane w tabeli 5.18 oraz metody widoczne w tabeli 5.19. Metoda `Write` (podobnie jak w przypadku klasy `StreamWriter`) istnieje w wielu przeciążonych wersjach odpowiadających każdemu z typów podstawowych. Można jej więc bezpośrednio użyć do zapisywania takich wartości, jak `int`, `double`, `string` itp. Przy wywoływaniu konstruktora mogą wystąpić następujące wyjątki:

- ◆ `ArgumentException` — gdy strumień nie obsługuje zapisu bądź został zamknięty;
- ◆ `ArgumentNullException` — gdy dowolny z argumentów (o ile zostały przekazane) ma wartość `null`.

Tabela 5.18. Konstruktory klasy `BinaryWriter`

Konstruktor	Opis
<code>BinaryWriter()</code>	Tworzy nowy obiekt typu <code>BinaryWriter</code> .
<code>BinaryWriter(Stream)</code>	Tworzy nowy obiekt typu <code>BinaryWriter</code> powiązany ze strumieniem danych. Przy zapisie ciągów znaków będzie używane kodowanie UTF-8.
<code>BinaryWriter(Stream, Encoding)</code>	Tworzy nowy obiekt typu <code>BinaryWriter</code> powiązany ze strumieniem danych, korzystający z określonego kodowania znaków.

Tabela 5.19. Wybrane metody klasy `BinaryWriter`

Typ zwracany	Metoda	Opis
<code>void</code>	<code>Close</code>	Zamyka strumień i zwalnia związane z nim zasoby.
<code>void</code>	<code>Flush</code>	Opróżnia bufor i zapisuje znajdujące się w nim dane.
<code>void</code>	<code>Seek</code>	Ustawia wskaźnik pozycji w strumieniu.
<code>void</code>	<code>Write</code>	Zapisuje w pliku wartość jednego z typów podstawowych.

Klasy `BinaryWriter` użyjemy, aby zapisać w pliku wybraną liczbę wartości typu `int`. Przykład kodu wykonującego takie zadanie został zamieszczony na listingu 5.25. Jak pamiętamy, podobne zadanie wykonywaliśmy już przy użyciu klasy `FileStream`, wtedy występował pewien mankament, polegający na tym, że w pliku nie pojawiała się informacja o liczbie zapisanych wartości. Tym razem naprawimy to niedopatrzenie.

Listing 5.25. Zapis danych binarnych do pliku

```
using System;
using System.IO;

public class Program
{
    public static void Main(String[] args)
    {
        if(args.Length < 1)
        {
            Console.WriteLine("Wywołanie programu: Program plik");
            return;
        }
        String plik = args[0];
        int ile = 100;

        FileStream fs;
        try
        {
            fs = new FileStream(plik, FileMode.Create);
        }
        catch(Exception)
        {
            Console.WriteLine("Otwarcie pliku {0} się nie powiodło.", plik);
            return;
        }

        BinaryWriter bw = new BinaryWriter(fs);

        try
        {
            bw.Write(ile);
            for(int i = 1; i <= ile; i++)
            {
                bw.Write(i);
            }
        }
        catch(Exception)
        {
            Console.WriteLine("Wystąpił błąd w trakcie zapisu danych.");
            return;
        }
        bw.Close();
        Console.WriteLine("Zapis został dokonany.");
    }
}
```

Zadaniem tej aplikacji jest zapisanie w pliku (o nazwie odczytanej z wiersza poleceń) wartości typu `int` od 1 do liczby wskazanej przez zmienną `ile`, tak by mogły być one później odczytane przez inny program. Najpierw został utworzony obiekt typu `FileStream` powiązany z plikiem wskazanym przez argument pobrany z wiersza poleceń. Ta czynność została wykonana tak samo jak w przypadku przykładu z listingu 5.21. Obiekt został przypisany zmiennej `fs`, a następnie użyty jako argument konstruktora klasy `BinaryWriter`:

```
BinaryWriter bw = new BinaryWriter(fs);
```

Powstał więc obiekt `bw` typu `BinaryWriter` powiązany ze strumieniem `fs` typu `FileStream`, a więc pośrednio z plikiem wskazanym z wiersza poleceń.

Następnie za pomocą metody `Write` obiektu `bw` została zapisana wartość zmiennej `iLE`, wskazująca liczbę właściwych wartości, które mają się znaleźć w pliku:

```
bw.Write(iLE);
```

a później w pętli `for` kolejne liczby typu `int`:

```
bw.Write(i);
```

Wszystkie operacje dotyczące zapisu danych zostały ujęte w blok `try...catch` przechwytyujący wyjątek, który mógłby się pojawić w przypadku wystąpienia jakiegoś błędu. Po zakończeniu zapisu wartości strumień został zamknięty za pomocą metody `Close`. Jest to ważne, gdyż inaczej dane mogłyby zostać utracone.

Odczyt danych binarnych

Skoro wiadomo już, jak zapisywać dane binarne do pliku, czas zobaczyć, jak je odczytywać. Służy do tego celu klasa `BinaryReader`. Udostępnia ona konstruktory zebrane w tabeli 5.20 oraz metody widoczne w tabeli 5.21. Jak widać, w tym przypadku każdemu z typów prostych odpowiada osobna metoda czytająca, czyli `ReadByte`, `ReadChar` itp. Klasy `BinaryReader` użyjemy do odczytania danych zapisanych przez program z listingu 5.25; odpowiedni przykład jest widoczny na listingu 5.26.

Tabela 5.20. Konstruktory klasy `BinaryReader`

Konstruktor	Opis
<code>BinaryReader(Stream)</code>	Tworzy nowy obiekt typu <code>BinaryReader</code> powiązany ze strumieniem danych.
<code>BinaryReader(Stream, Encoding)</code>	Tworzy nowy obiekt typu <code>BinaryReader</code> powiązany ze strumieniem danych, korzystający z określonego kodowania znaków.

Tabela 5.21. Metody klasy `BinaryReader`

Typ zwracany	Metoda	Opis
<code>void</code>	<code>Close</code>	Zamyka strumień i zwalnia związane z nim zasoby.
<code>int</code>	<code>PeekChar</code>	Zwraca ze strumienia kolejny znak, nie pobierając go.
<code>int</code>	<code>Read</code>	Odczytuje ze strumienia kolejny bajt (lub bajty).
<code>bool</code>	<code>ReadBoolean</code>	Odczytuje ze strumienia wartość typu <code>bool</code> .
<code>byte</code>	<code>ReadByte</code>	Odczytuje ze strumienia wartość typu <code>byte</code> .
<code>byte[]</code>	<code>ReadBytes</code>	Odczytuje ze strumienia określona liczbę bajtów.
<code>char</code>	<code>ReadChar</code>	Odczytuje ze strumienia wartość typu <code>char</code> .
<code>char[]</code>	<code>ReadChars</code>	Odczytuje ze strumienia określona liczbę znaków.
<code>decimal</code>	<code>ReadDecimal</code>	Odczytuje ze strumienia wartość typu <code>decimal</code> .
<code>double</code>	<code>ReadDouble</code>	Odczytuje ze strumienia wartość typu <code>double</code> .

Tabela 5.21. Metody klasy *BinaryReader* — ciąg dalszy

Typ zwracany	Metoda	Opis
Short	ReadInt16	Odczytuje ze strumienia wartość typu short.
int	ReadInt32	Odczytuje ze strumienia wartość typu int.
long	ReadInt64	Odczytuje ze strumienia wartość typu long.
sbyte	ReadSByte	Odczytuje ze strumienia wartość typu sbyte.
float	ReadSingle	Odczytuje ze strumienia wartość typu float.
string	ReadString	Odczytuje ze strumienia wartość typu string.
ushort	ReadUInt16	Odczytuje ze strumienia wartość typu ushort.
uint	ReadUInt32	Odczytuje ze strumienia wartość typu uint.
ulong	ReadUInt64	Odczytuje ze strumienia wartość typu ulong.

Listing 5.26. Odczyt danych binarnych z pliku

```

using System;
using System.IO;

public class Program
{
    public static void Main(String[] args)
    {
        if(args.Length < 1)
        {
            Console.WriteLine("Wywołanie programu: Program plik");
            return;
        }
        String plik = args[0];

        BinaryReader br;
        try
        {
            br = new BinaryReader(new FileStream(plik, FileMode.Open));
        }
        catch(Exception)
        {
            Console.WriteLine("Otwarcie pliku {0} się nie powiodło.", plik);
            return;
        }

        try
        {
            Console.WriteLine("Wartości odczytane z pliku {0}: ", plik);
            int ile = br.ReadInt32();
            for(int i = 0; i < ile; i++)
            {
                int wartosc = br.ReadInt32();
                Console.Write(wartosc + " ");
            }
            br.Close();
        }
        catch(Exception)
        {
    
```

```
        Console.WriteLine("Wystąpił błąd w trakcie odczytu danych.");
        return;
    }
}
```

Obiekt `BinaryReader` należy utworzyć podobnie jak `BinaryWriter`, przekazując w konstruktorze obiekt typu `FileStream`. W powyższym przykładzie korzystamy jednak z inniej rozwlekłego zapisu niż w przypadku kodu z listingu 5.25. Otóż w jednej instrukcji tworzymy zarówno obiekt typu `FileStream`, jak i `BinaryReader`:

```
br = new BinaryReader(new FileStream(plik, FileMode.Open));
```

obejmując ją jednym blokiem `try...catch`. Tę instrukcję należy rozumieć następująco: utwórz obiekt typu `FileStream`, przekazując mu w postaci argumentów wartość zmiennej `plik` oraz `FileMode.Open`, następnie użyj go jako argumentu dla konstruktora obiektu typu `BinaryReader`, a referencję do tego obiektu przypisz zmiennej `br`.

Jak pamiętamy, w pliku z danymi (wygenerowanym przez program z listingu 5.25) najpierw została zapisana wartość typu `int`, określająca, ile liczb zostało w nim umieszczonych. Skoro tak, trzeba ją pobrać i zapisać w zmiennej pomocniczej:

```
int ile = br.ReadInt32();
```

Metoda `ReadInt32` odczytuje właśnie wartość typu `int`. Kiedy wiadomo, ile liczb trzeba odczytać, wystarczy użyć pętli `for`, w której na przemian będzie odczytywana wartość:

```
int wartosc = br.ReadInt32();
```

oraz wyświetlana na ekranie:

```
Console.Write(wartosc + " ");
```

Liczba przebiegów pętli jest oczywiście określana przez stan zmiennej `ile`. Tym samym po uruchomieniu programu możemy odczytać ciąg wartości zapisany przez aplikację z listingu 5.25, tak jak jest to widoczne na rysunku 5.14.

Rysunek 5.14.

Wartości odczytane z pliku

```
C:\>Program.exe dane.bin
wartosc odczytana z pliku dane.bin:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
C:\>
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 27.1

Zmodyfikuj program z listingu 5.21 w taki sposób, aby rozmiar tablicy bajtów, która ma być zapisana w pliku, był wprowadzany z wiersza poleceń.

Ćwiczenie 27.2

Zmodyfikuj program z listingu 5.21 w taki sposób, by zapis danych odbywał się za pomocą metody `WriteByte`.

Ćwiczenie 27.3

Napisz program wykonujący takie samo zadanie jak w przykładzie z listingu 5.22, który jednak poprawnie wyświetli zawartość pliku z zapisaną dowolną liczbą danych.

Ćwiczenie 27.4

Napisz program wyświetlający na ekranie dane z dowolnego pliku tekstowego, wykorzystujący do odczytu metodę `ReadToEnd` klasy `StreamReader`.

Ćwiczenie 27.5

Napisz program, który wyświetli co drugi wiersz z pliku tekstowego. Wiersze powinny zostać ponumerowane.

Ćwiczenie 27.6

Napisz program, który będzie odczytywał wprowadzane przez użytkownika z klawiatury wartości całkowite i zapisze je w postaci binarnej w pliku.

Ćwiczenie 27.7

Napisz program, który odczyta dane z pliku tekstowego o nazwie przekazanej w postaci pierwszego argumentu z wiersza poleceń i zapisze je w odwrotnej kolejności (ostatni znak stanie się pierwszym, przedostatni — drugim itd.) w pliku o nazwie wskazanej przez drugi argument wiersza poleceń. Możesz założyć, że aplikacja ma działać poprawnie tylko z plikami o niewielkich rozmiarach.

Rozdział 6.

Zaawansowane zagadnienia programowania obiektowego

W rozdziale 3. zostały omówione podstawy programowania obiektowego. Przybliżono tam najważniejsze techniki i zasady niezbędne do sprawnego programowania w C#. Rozdział 6. zawiera treści znacznie poszerzające tamte wiadomości. Jest podzielony na cztery główne tematy: polimorfizm, interfejsy, klasy wewnętrzne oraz uogólnienia. Są to zagadnienia bardziej zaawansowane niż te, z którymi stykaliśmy się do tej pory, niemniej nie można ich pominąć. Każdy, kto poważnie myśli o programowaniu w C#, powinien się z nimi zapoznać. W szczególności dotyczy to polimorfizmu, który jest jednym z filarów programowania obiektowego, oraz interfejsów, bez których nie da się sprawnie pracować z klasami zawartymi w .NET.

Polimorfizm

Lekcja 28. Konwersje typów i rzutowanie obiektów

Lekcja 28. jest poświęcona konwersji typów oraz rzutowaniu obiektów. Sprawdzimy w niej, czy można przypisać zmiennej typu `int` wartość `double`, w jaki sposób są wykonywane domyślne zmiany typów oraz jak je kontrolować. Zostanie wyjaśniona technika pozwalająca na przypisanie zmiennej referencyjnej pewnego typu, obiektu klasy

nadrzędnej lub potomnej do tego typu. Znajdzie się tu również odpowiedź na pytanie, jak to się dzieje, że jako argumentu instrukcji `Console.WriteLine` można użyć dowolnego typu obiektowego.

Konwersje typów prostych

W lekcji 7. zastanawialiśmy się, co się będzie działo w programie, jeśli np. wynikiem dzielenia dwóch liczb całkowitych będzie liczba ułamkowa i spróbujemy ją przypisać zmiennej typu całkowitoliczbowego. W jaki zatem sposób zostanie wykonana przykładowa instrukcja:

```
int liczba = 9 / 2;
```

Jak już wiadomo, wynik takiego dzielenia będzie zaokrąglony w dół, czyli zmiennej liczba zostanie w tym przypadku przypisana wartość 4. Jednak ujmując rzecz dokładniej, będzie wykonana automatyczna konwersja typów danych. Wynik, którym jest liczba zmiennoprzecinkowa typu `double` ($9/2 = 4,5$), zostanie skonwertowany na typ `int`. W wyniku tej konwersji nastąpi utrata części ułamkowej i dlatego właśnie zmenna liczba otrzyma wartość 4. Czyli najważniejsza operacja to konwersja typów danych, a zaokrąglenie w dół jest jedynie efektem ubocznym tej konwersji.

Programista nie musi zdawać się na konwersje automatyczne, a w wielu przypadkach są one wręcz niemożliwe. Często niezbędne jest więc wykonanie konwersji jawnej. Operacji takiej dokonujemy poprzez umieszczenie przed wyrażeniem nazwy typu docelowego ujętej w nawias okrągły. Schematycznie taka konstrukcja wygląda następująco:

```
(typ_docelowy) wyrażenie;
```

Jeśli chcemy na przykład dokonać konwersji wyniku dzielenia `9 / 2`, który jest wartością typu `double`, na typ `int`, musimy zastosować instrukcję¹:

```
int liczba = (int) (9 / 2);
```

Można też dokonać jawnej konwersji zmiennej typu `double` na typ `int`, np.:

```
double liczba1 = 10.5;
int liczba2 = (int) liczba1;
```

Powinniśmy sobie zdawać sprawę, że w sytuacji odwrotnej konwersja typów również występuje, choć najczęściej jej po prostu nie zauważamy. Jeśli spróbujemy przypisać zmiennej typu `double` (reprezentującej szerszy zakres wartości) wartość typu `int` lub wartość zmiennej typu `int` (reprezentującej węższy zakres wartości), konwersja również zostanie wykonana. Możemy więc wykonać instrukcję:

```
int liczba1 = 10;
double liczba2 = liczba1;
```

¹ Wyrażenie `9 / 2` zostało również ujęte w nawias, jako że operator rzutowania typów jest silniejszy niż operator dzielenia (tabela 2.17 z lekcji 7.).

Operacja taka zostanie przeprowadzona poprawnie, jednak — formalnie rzecz biorąc — kompilator potraktuje ten zapis tak, jakby miał on postać:

```
int liczba1 = 10;
double liczba2 = (double) liczba1;
```

Rzutowanie typów obiektowych

Typy obiektowe, podobnie jak proste, również podlegają konwersjom. Przypomnijmy sobie koniec lekcji 17. Pojawiło się tam stwierdzenie, że jeśli oczekujemy argumentu klasy X, a podany zostanie argument klasy Y, która jest klasą potomną dla X, błędu nie będzie. Działa się tak dlatego, że w takiej sytuacji zostanie dokonane tak zwane automatyczne rzutowanie typu obiektu.

Pamiętamy, że obiekt klasy potomnej zawiera w sobie wszystkie pola i metody zawarte w klasie bazowej. Można powiedzieć, że obiekt klasy potomnej zawiera już w sobie obiekt klasy bazowej. W związku z tym nie ma żadnych przeciwnskazań, aby w miejscu, gdzie powinien znaleźć się obiekt klasy bazowej, umieścić obiekt klasy potomnej. Weźmy dla przykładu dobrze nam znane z rozdziału 3. klasy Punkt i Punkt3D w postaci zaprezentowanej na listingu 6.1.

Listing 6.1. Podstawowe wersje klas Punkt i Punkt3D

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

public class Punkt3D : Punkt
{
    public int z;
}
```

Jeśli zadeklarujemy teraz zmienną typu Punkt, to będzie można przypisać jej odniesienie do nowego obiektu klasy Punkt3D. Poprawne zatem będą instrukcje:

```
Punkt punkt;
punkt = new Punkt3D();
```

Oczywiście, odwołując się teraz do metod i pól obiektu wskazywanego przez zmienne punkt, możemy w standardowy sposób odwoływać się jedynie do metod zdefiniowanych w klasie Punkt. A zatem niepoprawne będzie np. odwołanie:

```
punkt.z = 10;
```

Zobrazowano to w kodzie z listingu 6.2. Próba jego komplikacji zakończy się błędem widocznym na rysunku 6.1.

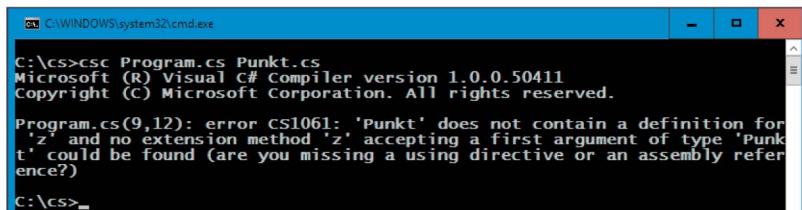
Listing 6.2. Nieprawidłowe odwołanie do składowej z

```
using System;

public class Program
{
    public static void Main()
    {
        Punkt punkt1;
        punkt1 = new Punkt3D();
        punkt1.z = 10;
    }
}
```

Rysunek 6.1.

Błąd spowodowany nieprawidłowym odwołaniem do pola z



Nie powinno to dziwić. Dla kompilatora typem zmiennej punkt1 jest Punkt, a w tej klasie nie ma pola z. Dlatego też wyświetlane są komunikaty o błędzie.

W rzeczywistości instrukcja:

```
punkt = new Punkt3D();
```

jest przez kompilator rozumiana:

```
punkt = (Punkt) new Punkt3D();
```

Zapewne bardzo przypomina nam to konwersje typów prostych. Jednak znaczenie tego zapisu jest nieco inne. Zostało tu wykonane rzutowanie wskazania do obiektu klasy Punkt3D na klasę Punkt. Jest to informacja dla kompilatora: „Traktuj zmienną punkt wskazującą obiekt klasy Punkt3D tak, jakby wskazywała ona obiekt klasy Punkt”, lub upraszczając: „Traktuj obiekt klasy Punkt3D tak, jakby był on klasy Punkt”. Obiekt Punkt3D nie zmienia się jednak ani nie traci żadnych informacji, jest po prostu inaczej traktowany.

Rzutowania można dokonać również w przypadku zmiennych już istniejących, np.:

```
Punkt3D punkt3D = new Punkt3D();
Punkt punkt = (Punkt) punkt3D;
```

Jest to też doskonały dowód, że dokonujemy tu rzutowania typów, a nie konwersji. Przypomnijmy, że w przypadku typów prostych konwersja typu bardziej ogólnego na typ bardziej szczegółowy powodowała utratę części informacji. Przykładowo wartość 4.5 typu double po konwersji na typ int zmieniała się na 4. Część ułamkowa zostanie bezpowrotnie utracona i nawet powtórna konwersja na typ double nie będzie w stanie przywrócić poprzedniej wartości. Zobrazowano to w przykładzie widocznym na li-

stingu 6.3.

Listing 6.3. Utrata pierwotnej wartości po konwersji na inny typ

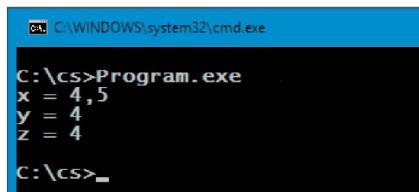
```
using System;

public class Program
{
    public static void Main()
    {
        double x = 4.5;
        int y = (int) x;
        double z = (double) y;
        Console.WriteLine("x = " + x);
        Console.WriteLine("y = " + y);
        Console.WriteLine("z = " + z);
    }
}
```

Deklarujemy zmienną `x` typu `double` i przypisujemy jej wartość 4.5. Następnie deklarujemy zmienną `y` typu `int` oraz przypisujemy jej wartość konwersji zmiennej `x` na typ `int` (`int y = (int) x;`). W kroku trzecim deklarujemy zmienną `z` typu `double`, której przypisujemy wynik konwersji zmiennej `y` na typ `double`. Na zakończenie wyświetlamy wartości wszystkich trzech zmiennych na ekranie (rysunek 6.2). Widać wyraźnie, że nie jesteśmy w stanie odzyskać informacji utraconej przez konwersję typu `double` na typ `int`.

Rysunek 6.2.

Utrata informacji
w wyniku konwersji
typów prostych



W przypadku zmiennych obiektowych będzie zupełnie inaczej, co wynika z ich właściwości. Spójrzmy na listing 6.4 obrazujący wyniki rzutowania typów obiektowych (przykład korzysta z klas `Punkt` i `Punkt3D` z listingu 6.1).

Listing 6.4. Rzutowanie typów obiektowych

```
using System;

public class Program
{
    public static void Main()
    {
        Punkt3D punkt3D1 = new Punkt3D();

        punkt3D1.x = 10;
        punkt3D1.y = 20;
        punkt3D1.z = 30;

        Console.WriteLine("punkt3D1");
        Console.WriteLine("x = " + punkt3D1.x);
        Console.WriteLine("y = " + punkt3D1.y);
```

```

Console.WriteLine("z = " + punkt3D1.z);
Console.WriteLine("");

Punkt punkt1 = (Punkt) punkt3D1;

Console.WriteLine("punkt");
Console.WriteLine("x = " + punkt1.x);
Console.WriteLine("y = " + punkt1.y);
Console.WriteLine("");

Punkt3D punkt3D2 = (Punkt3D) punkt1;

Console.WriteLine("punkt3D2");
Console.WriteLine("x = " + punkt3D2.x);
Console.WriteLine("y = " + punkt3D2.y);
Console.WriteLine("z = " + punkt3D2.z);
}
}

```

Tworzymy obiekt klasy `Punkt3D` i przypisujemy go zmiennej `punkt3D1` (`Punkt3D` `punkt3D1 = new Punkt3D();`²) oraz ustalamy wartości jego pól `x`, `y`, `z` na 10, 20 i 30. Następnie wyświetlamy te informacje na ekranie. Dalej tworzymy zmienną `punkt1` klasy `Punkt` i za pomocą techniki rzutowania przypisujemy jej obiekt, na który wskazuje `punkt3D1` (`Punkt punkt1 = (Punkt) punkt3D1;`). Zawartość pól wyświetlamy na ekranie. Oczywiście, ponieważ zmieniona `punkt1` jest klasy `Punkt`, mamy jedynie dostęp do pól `x` oraz `y`. Pole `z` nie jest dostępne, nie można modyfikować ani odczytywać jego wartości (próba odwołania do niego spowoduje błąd komilacji).

To jednak nie wszystko. W kolejnym kroku dokonujemy jeszcze jednego rzutowania. Obiekt wskazywany przez `punkt1` przypisujemy zmiennej `punkt3D2`, która jest typu `Punkt3D`. Następnie wyświetlamy zawartość wszystkich pól. Wartości te to 10, 20 i 30. Ostatecznie na ekranie zobaczymy widok zaprezentowany na rysunku 6.3.

Rysunek 6.3.

Ilustracja
rzutowania typów
obiektowych

```

C:\>Program.exe
punkt3D1
x = 10
y = 20
z = 30

punkt1
x = 10
y = 20

punkt3D2
x = 10
y = 20
z = 30
C:\>_

```

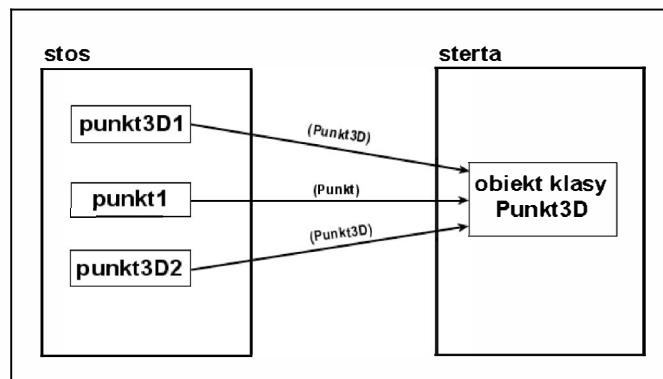
Jest już zatem jasne, że rzutowanie nie zmienia stanu obiektu. W przykładzie utworzyliśmy tylko jeden obiekt klasy `Punkt3D`. Żadna z operacji nie zmieniła typu tego obiektu, nie odbyły się żadne konwersje, które mogłyby doprowadzić do utraty części danych. Powstały natomiast trzy zmienne, przez które spoglądaliśmy na obiekt. Zmienne

² Jak pamiętamy z rozdziału 3., stwierdzenie to jest uproszczeniem. W rzeczywistości zmiennej nie jest przypisywany obiekt, ale referencja (odniesienie) do niego.

punkt3D1 i punkt3D2 traktowały go jako obiekt klasy Punkt3D, a zmienna punkt1 traktowała go jako obiekt klasy Punkt. Schematycznie tę sytuację przedstawiono na rysunku 6.4. Można zatem powiedzieć, że rzutowanie pozwala nam spojrzeć na dany obiekt z innej, szerszej lub węższej perspektywy.

Rysunek 6.4.

Schematyczne
zależności między
zmiennymi
i obiektem
z listingu 6.4



Przykład z listingu 6.4 pokazał również, że rzutowanie obiektów jest możliwe w obie strony, to znaczy obiekt klasy bazowej można rzutować na obiekt klasy potomnej, a obiekt klasy potomnej na obiekt klasy bazowej. Przypadek drugi jest oczywisty; wiemy już, że obiekt klasy potomnej zawiera w sobie obiekt klasy bazowej. Sytuacja odwrotna jest jednak bardziej skomplikowana. Jak bowiem potraktować na przykład poniższy fragment kodu?

```
Punkt3D punkt3D = (Punkt3D) new Punkt();
punkt3D.z = 10;
```

Czy może on zostać wykonany? Pozostawmy ten przykład do przemyślenia, wróćmy do niego na początku lekcji 29.

Rzutowanie na typ Object

W C# wszystkie klasy dziedziczą bezpośrednio lub pośrednio po klasie `Object`. Nie ma pod tym względem wyjątków. Nawet jeśli definicja nowej klasy bazowej wygląda tak jak w dotychczas prezentowanych przykładach, czyli:

```
public class nazwa_klasy
{
}
```

to kompilator potraktuje ten fragment kodu jako:

```
public class nazwa_klasy : Object
{
}
```

Zatem klasa `Object` jest praklasą, z której wywodzą się wszystkie inne klasy w C#. Oznacza to, że każda klasa dziedziczy wszystkie metody i pola klasy `Object`. Jedną z takich metod jest `ToString`. Często nie zdajemy sobie nawet sprawy, że jest ona faktycznie wykonywana. Przypomnijmy sobie na przykład, w jaki sposób wyświetliśmy

systemowy komunikat o typie wyjątku w lekcjach z rozdziału 4. Stosowana była między innymi konstrukcja w postaci:

```
try
{
    //instrukcje mogące spowodować wyjątek
}
catch(DivideByZeroException e)
{
    Console.WriteLine(e.ToString());
}
```

W rzeczywistości jednak można by pominąć wywołanie metody `ToString` obiektu `e` i do wyświetlenia danych zastosować instrukcję:

```
Console.WriteLine(e);
```

Zadziałałaby ona bez najmniejszego problemu. Jak to się w takim razie dzieje, że jako argument metody `WriteLine` można przekazać obiekt klasy wyjątku (w powyższym przypadku obiekt klasy `DivideByZeroException`)? Nie istnieje przecież taka wersja tej metody, która przyjmowałaby jako argument obiekt klasy `DivideByZeroException`!

Wytlumaczenie jest na szczęście proste. Istnieje przeciążona wersja metody `WriteLine` (a także metody `Write`), która przyjmuje jako argument obiekt klasy `Object`. W klasie `Object` istnieje natomiast metoda o nazwie `ToString`, która zwraca opis obiektu w postaci ciągu znaków³. Metoda `WriteLine` w celu uzyskania ciągu znaków wywołuje natomiast metodę `ToString` obiektu przekazanego jej w postaci argumentu.

Oczywiście, metoda `ToString` zdefiniowana w klasie `Object` nie jest w stanie dostarczyć komunikatu o typie zgłoszonego wyjątku `DivideByZeroException`. Jednak komunikaty z klas potomnych są uzyskiwane dzięki przesłanianiu metody `ToString` w tych klasach. Zobaczmy, jak to będzie wyglądało na konkretnym przykładzie. Spójrzmy na listing 6.5.

Listing 6.5. Przesłonięcie metody `ToString`

```
using System;

public class MojObiekt
{
    public override String ToString()
    {
        return "Jestem obiektem klasy MojObiekt.";
    }
}

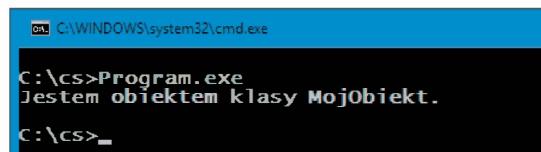
public class Program
{
    public static void Main()
    {
        MojObiekt mo = new MojObiekt();
        Console.WriteLine(mo);
    }
}
```

³ Formalnie rzecz biorąc, w postaci referencji do obiektu klasy `String`.

Powstała klasa `MojObiekt` zawierająca publiczną metodę `ToString`. Dzięki zastosowaniu słowa `override` metoda ta przesyła metodę `ToString` z klasy `Object` (bliżej zajmiemy się tym tematem w kolejnej lekcji). Jednym z jej zadaniń jest wyświetlenie na ekranie napisu widocznego na rysunku 6.5. W metodzie `Main` z klasy `Program` tworzymy nowy obiekt klasy `MojObiekt` i przekazujemy go jako argument dla instrukcji `Console.WriteLine`. Dzięki temu możemy się naoczyć przekonać, że zdefiniowana przez nas metoda `ToString` zostanie faktycznie wykonana mimo tego, że nigdzie jej nie wywołaliśmy — ta czynność została wykonana przez metodę `WriteLine`.

Rysunek 6.5.

Efekt działania
metody `ToString`



Metodę tę możemy również wywołać jawnie, stosując konstrukcję:

```
Console.WriteLine(mo.ToString());
```

Warto jednak zauważyć, że w takiej sytuacji zostanie użyta inna wersja metody `WriteLine`. W poprzednim przypadku została zastosowana wersja przyjmująca jako argument obiekt klasy `Object`, a w tym — wersja przyjmująca jako argument obiekt klasy `String`. Czyli można by tę instrukcję rozbić na dwie następujące linie:

```
String komunikat = mo.ToString();
Console.WriteLine(komunikat);
```

Typy proste też są obiektowe!

Zapewne twierdzenie zawarte w powyższym tytule jest niespodzianką, ale tak jest w istocie. Typy proste w C# w rzeczywistości są obiektowe. Dokładniej rzecz ujmując, są one aliasami (innymi nazwami) dla zdefiniowanych w przestrzeni nazw `System` typów wspólnych dla platformy .NET. Zobrazowano to w tabeli 6.1. W rzeczywistości są one zdefiniowane jako struktury.

Tabela 6.1. Typy proste i ich aliasy w .NET

Nazwa w C#	Nazwa typu wspólnego w .NET
<code>bool</code>	<code>System.Boolean</code>
<code>byte</code>	<code>System.Byte</code>
<code>sbyte</code>	<code>System.SByte</code>
<code>char</code>	<code>System.Char</code>
<code>decimal</code>	<code>System.Decimal</code>
<code>double</code>	<code>System.Double</code>
<code>float</code>	<code>System.Single</code>
<code>int</code>	<code>System.Int32</code>
<code>uint</code>	<code>System.UInt32</code>

Tabela 6.1. Typy proste i ich aliasy w .NET — ciąg dalszy

Nazwa w C#	Nazwa typu wspólnego w .NET
Long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

Jak jednak przekonaliśmy się już wielokrotnie, w praktyce, na początku nauki, zupełnie tego faktu nie zauważamy. Wszelkie operacje wykonywane są automatycznie przez kompilator, nie stosujemy więc ani konstruktorów, ani innych mechanizmów obiektowych, nie ma bowiem takiej potrzeby. Należy jednak wiedzieć, że takie możliwości istnieją. Zamiast aliasu można więc użyć właściwej nazwy typu, np. deklaracja zmiennej typu całkowitoliczbowego może mieć postać:

```
System.Int32 liczba;
```

Można jej bezpośrednio przypisać wartość:

```
System.Int32 liczba = 100;
```

Można bezpośrednio wywołać konstruktor typu prostego, i to korzystając zarówno z nazwy właściwej:

```
System.Int32 liczba = new System.Int32();
```

jak i aliasu:

```
int liczba = new int();
```

W tych dwóch przypadkach zmiennej `liczba` zostanie przypisana wartość 0. Oczywiście, jeśli użyjemy dyrektywy `using System;`, nie trzeba będzie podawać nazwy przestrzeni nazw i prawidłowe będą też konstrukcje:

```
Int32 liczba1 = 100;
Int32 liczba2 = new Int32();
```

Przydatną możliwością jest również wywoływanie metod. Przykładowo istnieje metoda `ToString` pozwalająca na uzyskanie tekstowej reprezentacji wartości dowolnego typu prostego. Jeśli więc mamy zmienną typu `int` (`Int32`), to można tę metodę wywołać w sposób następujący:

```
int liczba = 100;
string str = liczba.ToString();
```

Co jeszcze ciekawsze, metoda może być wywołana bezpośrednio w stosunku do wartości typu prostego. Prawidłowe są więc też zapisy:

```
true.ToString();
124.ToString();
2.14.ToString();
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 28.1

Napisz klasy: Główna i Pochodna dziedziczącą po Głównej oraz testową klasę Program. W klasie Program utwórz obiekt klasy Pochodna i przypisz go zmiennej typu Główna o nazwie g1. Następnie zadeklaruj dodatkową zmienną typu Pochodna i przypisz jej obiekt wskazywany przez zmienną g1.

Ćwiczenie 28.2

Zmodyfikuj kod z listingu 6.5 tak, aby metoda ToString z klasy MojObiekt zwracała również komunikat uzyskany przez wywołanie metody ToString klasy nadzędnej.

Ćwiczenie 28.3

Zmodyfikuj kod klasy Punkt z listingu 3.1 z rozdziału 3. tak, aby po użyciu obiektu tej klasy jako argumentu metody WriteLine na ekranie były wyświetlane współrzędne punktu. Dodatkowo spraw, aby klasa ta stała się klasą publiczną, oraz dopisz do niej publiczny konstruktor pozwalający na ustawienie wartości pól x i y.

Ćwiczenie 28.4

Napisz klasę Program, która przetestuje nową funkcjonalność klasy Punkt powstałą w ćwiczeniu 28.3.

Ćwiczenie 28.5

Zmodyfikuj ostatni wiersz programu z listingu 6.2 w taki sposób, aby przypisanie wartości polu z stało się możliwe. Użyj techniki rzutowania.

Lekcja 29. Późne wiązanie i wywoływanie metod klas pochodnych

W lekcji 28. omówiono pojęcia konwersji oraz rzutowania typów danych. Wiadomo już, że obiekt danej klasy można potraktować jak obiekt klasy nadzędnej lub pochodnej, nie tracąc żadnych zapisanych w nim informacji. Lekcja 29. zawiera informacje o tym, jak w C# są wywoływane metody i kiedy mamy do czynienia z rzutowaniem typów. Wyjaśnione zostanie też pojęcie polimorfizmu, jedno z głównych pojęć w programowaniu obiektowym.

Rzeczywisty typ obiektu

Wiemy, że jest możliwe rzutowanie obiektu na typ bazowy (tzw. rzutowanie w górę), czyli np. wykorzystywane w lekcji 28. rzutowanie obiektu klasy `Punkt3D` na klasę `Punkt`. Zajmiemy się tym szczegółowo w dalszej części lekcji. Powróćmy jednak do tematu rzutowania w dół, który również pojawił się w poprzedniej lekcji. Rzutowanie w dół to rzutowanie typu klasy bazowej na typ klasy pochodnej, tak jak w poniższym fragmencie kodu:

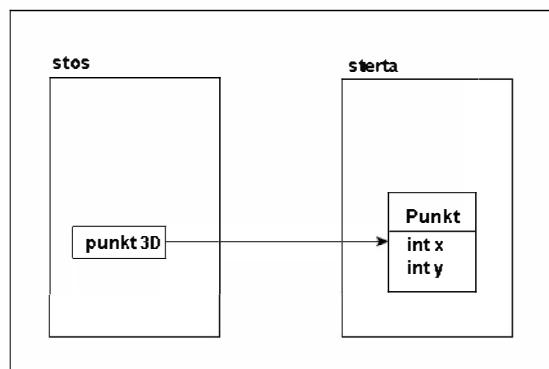
```
Punkt3D punkt3D = (Punkt3D) new Punkt();
punkt3D.z = 10;
```

Możemy taką konstrukcję zapisać również przy wykorzystaniu dodatkowej zmiennej klasy `Punkt`:

```
Punkt punkt = new Punkt();
Punkt3D punkt3D = (Punkt3D) punkt;
punkt3D.z = 10;
```

Powstaje tu obiekt klasy `Punkt`. W pierwszym przypadku jest on bezpośrednio rzutowany na klasę `Punkt3D` i przypisywany zmiennej `punkt3D`, natomiast w drugim jest najpierw przypisywany zmiennej `punkt` typu `Punkt`, a dopiero potem zawartość tej zmiennej jest rzutowana na klasę `Punkt3D` i przypisywana zmiennej `punkt3D`. W obu zatem przypadkach zmieniona `punkt3D`, która jest typu `Punkt3D`, wskazuje na obiekt klasy `Punkt`. Schematycznie zobrazowano to na rysunku 6.6.

Rysunek 6.6.
Zmienna `punkt`
klasy `Punkt3D`
wskazuje na obiekt
klasy `Punkt`



Czy można zatem wykonać instrukcję `punkt3D.z = 10`? Odpowiedź oczywiście musi brzmieć „nie”! Obiekt typu `Punkt` nie ma pola o nazwie `z`, nie ma zatem możliwości takiego przypisania. Czy uda się więc skompilować kod widoczny na listingu 6.6? Odpowiedź brzmi... „tak”.

Listing 6.6. Odwołanie do nieistniejącego w obiekcie pola `z`

```
using System;

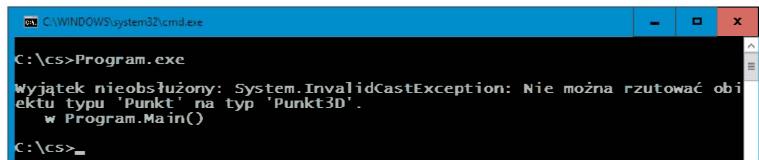
public class Program
{
    public static void Main()
```

```
{  
    Punkt punkt = new Punkt();  
    Punkt3D punkt3D = (Punkt3D) punkt;  
    punkt3D.z = 10;  
}  
}
```

Ten program jest syntaktycznie (składniowo) poprawny. Wolno dokonywać rzutowania klasy `Punkt` na klasę `Punkt3D` (por. listing 6.4), w żadnym wypadku nie zostanie on jednak poprawnie wykonany. Kompilator nie może zakładać naszej zlej woli i nie wie, że w rzeczywistości `punkt3D` wskazuje na obiekt, w którym brakuje pola `z`, czyli obiekt nieprawidłowej klasy. Jednak w trakcie wykonania programu takie sprawdzenie nastąpi i zostanie zgłoszony błąd (wyjątek) `InvalidOperationException`. Na ekranie pojawi się więc komunikat widoczny na rysunku 6.7. Oczywiście wyjątek ten można przechwycić (lekce z rozdziału 4.).

Rysunek 6.7.

Wyjątek spowodowany przez próbę odwołania do nieistniejącego pola z



Spróbujmy teraz odwołać się do jednego z istniejących w klasie `Punkt` pól `x` lub `y`, czyli wykonać instrukcje:

```
Punkt punkt = new Punkt();  
Punkt3D punkt3D = (Punkt3D) punkt;  
punkt3D.x = 10;
```

Czy tym razem program zadziała? Otóż spotka nas niespodzianka — reakcja będzie taka sama jak w poprzednim przypadku; zostanie wygenerowany wyjątek `InvalidOperationException` (rysunek 6.7). Stanie się tak, mimo że obiekt, na który wskazuje zmienna `punkt3D`, zawiera pole `x`. To jednak nie ma znaczenia, gdyż podstawowym problemem jest to, że sam obiekt jest innego typu niż `Punkt3D`. Środowisko uruchomieniowe w trakcie wykonania programu dokonuje sprawdzenia zgodności typów. Wykonywanie operacji na obiekcie jest możliwe tylko wtedy, kiedy zmienna wskazująca na ten obiekt jest zgodna co do typu z klasą tego obiektu lub klasą nadzczną — nigdy odwrotnie. Jeżeli opisywana zgodność nie następuje, w trakcie wykonania zostanie wygenerowany błąd `InvalidOperationException`, tak jak miało to miejsce w ostatnich przykładach. Jest on generowany już przy próbie wykonania instrukcji:

```
Punkt3D punkt3D = (Punkt3D) punkt;
```

nic więc dziwnego, że nie można wykonać żadnej instrukcji modyfikującej pola obiektu, niezależnie od tego, czy są one w nim zawarte, czy też nie.

Dziedziczenie a wywoływanie metod

Przedstawione przed chwilą sprawdzanie rzeczywistego (a nie deklarowanego) typu obiektu w trakcie działania programu to właśnie polimorfizm, który jest tematem nadzrzednym bieżących lekcji. **Polimorfizm** występuje pod wieloma terminami, to inaczej **późne wiązanie** (ang. *late binding*), **wiązanie czasu wykonania** (ang. *runtime binding*) czy **wiązanie dynamiczne** (ang. *dynamic binding*). W przypadku rzutowania w dół, omawianego w poprzednim podrozdziale, uniemożliwia wykonanie niedozwolonych operacji. O wiele użyteczniejszy jest jednak przy rzutowaniu w górę, czyli na klasę nadzrędną.

Przyjrzyjmy się więc dokładniej temu, jak dziedziczenie i rzutowanie wpływa na wywoływanie metod. Na początku zajmijmy się pierwszym wykorzystanym sposobem dziedziczenia metod, z użyciem słowa kluczowego new. Otóż tego słowa należy użyć wtedy, gdy nowa metoda (definiowana w klasie pochodnej) ma zastąpić metodę oryginalną (zdefiniowaną w klasie bazowej). Wtedy decyzja o tym, która z nich zostanie wywołana, jest podejmowana na etapie komplikacji i na podstawie zadeklarowanego typu obiektu. Mówimy wtedy o tak zwanym **wiązaniu statycznym** (ang. *static binding*) lub **wczesnym wiązaniu** (ang. *early binding*). Taką sytuację zobrazowano na listingu 6.7.

Listing 6.7. Ilustracja wiązania statycznego

```
using System;

public class Glowna
{
    public void Wyswietl()
    {
        Console.WriteLine("Metoda Wyswietl z klasy Glowna");
    }
}

public class Pochodna : Glowna
{
    public new void Wyswietl()
    {
        Console.WriteLine("Metoda Wyswietl z klasy Pochodna");
    }
}

public class Program
{
    public static void Main()
    {
        Glowna obj1 = new Glowna();
        Pochodna obj2 = new Pochodna();

        obj1.Wyswietl();
        obj2.Wyswietl();

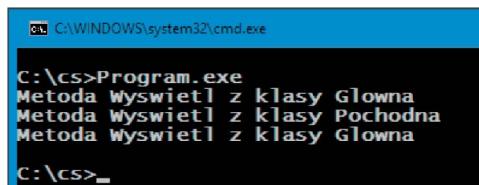
        Glowna obj3 = (Glowna) obj2;
        obj3.Wyswietl();
    }
}
```

Klasa `Główna` jest klasą bazową i zawiera publiczną metodę o nazwie `Wyswietl`, której jedynym zadaniem jest wyświetlenie komunikatu z nazwą klasy. Klasa `Pochodna` jest klasą potomną (dziedziczy po `Główna`) i również zawiera metodę `Wyswietl`, która dzięki słowu kluczowemu `new` zastępuje tę z klasy bazowej. Dlatego w wywołaniach metody `Wyswietl` zawsze będzie brany pod uwagę deklarowany podczas komplikacji typ obiektu. Aby to udowodnić, na listingu została również umieszczona klasa `Program`, korzystająca z obiektów typu `Główna` i `Pochodna`. W metodzie `Main` powstały dwa obiekty: pierwszy, typu `Główna`, przypisany zmiennej `obj1`, i drugi, typu `Pochodna`, przypisany zmiennej `obj2`. Następnie w stosunku do obu tych zmiennych została wywołana metoda `Wyswietl`.

W pierwszym przypadku obiekt jest typu `Główna` i zmienna jest typu `Główna`. W związku z tym zostanie wywołana metoda `Wyswietl` pochodząca z klasy `Główna`. W drugim przypadku obiekt jest typu `Pochodna` i zmienna jest typu `Pochodna`. W związku z tym zostanie wywołana metoda `Wyswietl` pochodząca z klasy `Pochodna`. Te zachowania nie powinny budzić żadnych wątpliwości. Przypadek trzeci jest jednak odmienny. Otóż zmienna `obj3` jest typu `Główna` i został jej przypisany — za pomocą techniki rzutowania — obiekt wskazywany przez `obj2`, czyli obiekt klasy `Pochodna`. Która z metod zostanie w tej sytuacji wywołana? Otóż zgodnie z wcześniejszymi wyjaśnieniami będzie to metoda `Wyswietl` pochodząca z klasy `Główna`, tak jak jest to widoczne na rysunku 6.8.

Rysunek 6.8.

Wynik działania kodu z listingu 6.7



```
C:\cs>Program.exe
Metoda Wyswietl z klasy Główna
Metoda Wyswietl z klasy Pochodna
Metoda Wyswietl z klasy Główna
C:\cs>
```

Jest to zachowanie poprawne — przecież kazaliśmy kompilatorowi patrzyć na obiekt klasy `Pochodna` tak, jakby był on klasy `Główna`, oraz (słowo `new`) korzystać z wiązania statycznego. Niestety, takie zachowanie w wielu sytuacjach przysporzyłoby nam wielu kłopotów. Często ważne jest, aby podczas wywoływanego metod brany był pod uwagę rzeczywisty typ obiektu, a nie ten wynikający z rzutowania. Składowe, które mają się zachowywać w taki sposób, należy w klasie bazowej poprzedzić słowem `virtual`, a w klasie pochodnej słowem `override`. Mówimy wtedy o metodach wirtualnych. Zobrazowano to w przykładzie widocznym na listingu 6.8.

Listing 6.8. Metody wirtualne

```
using System;

public class Główna
{
    public virtual void Wyswietl()
    {
        Console.WriteLine("Metoda Wyswietl z klasy Główna");
    }
}

public class Pochodna : Główna
```

```
{  
    public override void Wyswietl()  
    {  
        Console.WriteLine("Metoda Wyswietl z klasy Pochodna");  
    }  
}  
  
public class Program  
{  
    public static void Main()  
    {  
        Glowna obj1 = new Glowna();  
        Pochodna obj2 = new Pochodna();  
  
        obj1.Wyswietl();  
        obj2.Wyswietl();  
  
        Glowna obj3 = (Glowna) obj2;  
        obj3.Wyswietl();  
    }  
}
```

Kod jest bardzo podobny do poprzedniego przykładu. Różni się tylko sposobem deklaracji metod. W klasie `Glowna` metoda `Wyswietl` została zadeklarowana jako wirtualna, o czym świadczy słowo `virtual` umieszczone przed nazwą typu zwracanego (`void`). W klasie `Pochodna` również istnieje metoda `Wyswietl`, która przesyłała metodę o tej samej nazwie pochodzącej z klasy `Glowna`. Ponieważ użyte zostało słowo `override`, będziemy mieli do czynienia z wiązaniem dynamicznym i polimorficznym wywoływanym metod. Oznacza to, że pod uwagę będzie brany nie deklarowany, ale rzeczywisty typ obiektu. Przekonujemy się o tym dzięki klasie `Program` i metodzie `Main`. Mają one taką samą postać jak w przypadku listingu 6.7 i w pierwszych dwóch wywołaniach metody `Wyswietl` działanie też jest takie samo. Skoro bowiem zmieniąca `obj1` klasę `Glowna` wskazuje na obiekt klasy `Glowna`, to zostanie wywołana metoda pochodząca z tej klasy i, analogicznie, skoro zmieniąca `obj2` klasę `Pochodna` wskazuje na obiekt klasy `Pochodna`, to zostanie wywołana metoda pochodząca z klasy `Pochodna`.

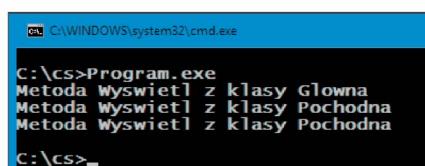
Zupełnie inaczej jest jednak w przypadku trzeciego wywołania. Otóż zmieniąca `obj3` jest klasą `Glowna`, a obiekt, na który wskazuje, jest klasą `Pochodna`. Tak więc ponieważ metoda `Wyswietl` w klasie `Glowna` jest oznaczona jako wirtualna, a w klasie `Pochodna` została przesłonięta przez zastosowanie słowa `override`, to w trakcie wykonania programu zostanie sprawdzony rzeczywisty typ obiektu i to on zostanie wzięty pod uwagę w wywołaniu. Dlatego też instrukcja:

```
obj3.Wyswietl();
```

spowoduje użycie metody `Wyswietl` z klasy `Pochodna`, a po uruchomieniu całego programu zobaczymy widok taki jak na rysunku 6.9.

Rysunek 6.9.

Efekt wywołania polimorficznego



Wykonajmy jeszcze jeden przykład. Założmy, że mamy klasę Shape, która jest nadrębna dla innych klas opisujących figury geometryczne. Wyprowadzimy z niej klasy: Circle (okrąg), Rectangle (prostokąt) oraz Triangle (trójkąt). W każdej z nich umieścimy metodę draw. W praktyce metoda ta rysowałaby zapewne daną figurę na ekranie, my ograniczymy się jedynie do wyświetlenia informacji tekstowej. Taki zestaw klas jest widoczny na listingu 6.9.

Listing 6.9. Zestaw klas zawierających metody wirtualne

```
using System;

public class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Jestem jakimś kształtem.");
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Jestem okręgiem.");
    }
}

public class Rectangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Jestem prostokątem.");
    }
}

public class Triangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Jestem trójkątem.");
    }
}
```

Struktura przedstawionych klas przypomina poprzednio omawiany przykład. Klasą główną jest Shape, po niej dziedziczą Triangle, Circle i Rectangle. Każda z wymienionych klas ma zdefiniowaną własną metodę Draw, której zadaniem, zgodnie z tym, co zostało napisane wyżej, będzie wyświetlenie na ekranie nazwy klasy. W klasie bazowej Shape metoda ta została zadeklarowana ze słowem `virtual`, a w klasach pochodnych — ze słowem `override`.

Założymy teraz, że mamy napisać klasę Program, w której znajdzie się metoda DrawShape rysująca nasze figury. To znaczy metoda taka miałaby przyjmować argument będący obiektem danej klasy i wywoływać jego metodę Draw. Bez wywołań polimorficznych

niezbędne byłoby napisanie wielu wersji przeciążonej metody DrawShape. Jedna przyjmowałaby argumenty klasy Triangle, inna Rectangle, jeszcze inna Circle. Co więcej, utworzenie nowej klasy dziedziczącej po Shape wymagałoby dopisania kolejnej przeciążonej metody DrawShape w klasie Program. Tymczasem polimorfizm pozwala napisać tylko jedną metodę DrawShape w klasie Program i będzie ona pasować do każdej kolejnej klasy wprowadzonej z Shape. Zobrazowano to w programie widocznym na listingu 6.10.

Listing 6.10. Metoda korzystająca z wywołań polimorficznych

```
public class Program
{
    public static void DrawShape(Shape shape)
    {
        shape.Draw();
    }
    public static void Main()
    {
        Circle circle = new Circle();
        Triangle triangle = new Triangle();
        Rectangle rectangle = new Rectangle();

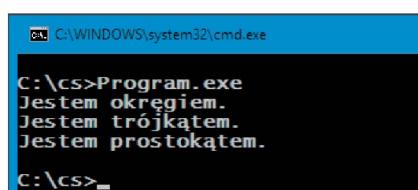
        DrawShape(circle);
        DrawShape(triangle);
        DrawShape(rectangle);
    }
}
```

W klasie Program istnieje tylko jedna metoda DrawShape (jest ona statyczna, a więc jej wywołanie nie wymaga tworzenia obiektu klasy Program), która jako argument przyjmuje obiekt klasy Shape. Będzie więc mogła również otrzymać jako argument dowolny obiekt klasy dziedziczącej po Shape. W samej metodzie następuje wywołanie metody Draw obiektu będącego argumentem. Jak wiemy, będzie to wywołanie polimorficzne, zatem zostanie wywołana metoda z rzeczywistej klasy obiektu (Triangle, Circle lub Rectangle), a nie ta pochodząca z klasy Shape.

Dalsza część programu potwierdza takie właśnie zachowanie kodu. W metodzie Main tworzymy bowiem obiekty klas Circle, Triangle oraz Rectangle i przekazujemy je jako argumenty metodzie DrawShape z klasy Program. Wynik działania, zgodny z oczekiwaniemi, jest widoczny na rysunku 6.10.

Rysunek 6.10.

Polimorficzne
wywołania metody
drawShape



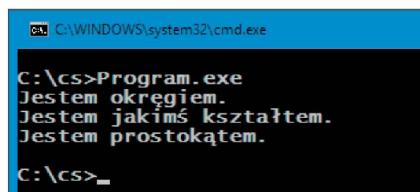
Zastanówmy się teraz, co by się stało, gdyby w jednej z klas dziedziczących po Shape nie było zdefiniowanej metody Draw. Jaki zatem będzie wynik działania programu z listingu 6.10, jeśli np. klasa Triangle z listingu 6.9 przyjmie pokazaną poniżej postać?

```
public class Triangle extends Shape  
{  
}
```

Efekt jest widoczny na rysunku 6.11. Zamiast napisu Jestem trójkątem pojawił się napis Jestem jakimś kształtem.. To nie powinno dziwić. Pamiętajmy, że metoda Draw z klasy Triangle przesłaniała metodę Draw zdefiniowaną w klasie Shape. Kiedy z definicji klasy Triangle usunęliśmy metodę Draw, została odsłonięta metoda Draw odziedziczona po klasie nadrzędnej, czyli Shape. Została zatem wykonana metoda Draw obiektu klasy Triangle, tak jak w poprzednim przypadku, tyle że była to metoda odziedziczona po klasie bazowej i stąd też na ekranie pojawił się napis Jestem jakimś kształtem..

Rysunek 6.11.

Z klasy Triangle
została usunięta
metoda draw



Taki sam efekt osiągniemy, jeśli w klasie Triangle metoda Draw zostanie zdefiniowana jako:

```
public new void Draw()  
{  
    Console.WriteLine("Jestem trójkątem.");  
}
```

Poprzez użycie słowa new poinformujemy bowiem kompilator, że ma brać pod uwagę deklarowany typ obiektu (ma korzystać z wiązania statycznego), a ponieważ deklarowanym typem argumentu metody DrawShape z klasy Program jest Shape, zostałaby wtedy użyta metoda Draw z klasy Shape.

Dziedziczenie a metody prywatne

Na zakończenie lekcji 28. zwróćmy jeszcze uwagę na kwestię związaną z dziedziczeniem i metodami prywatnymi. Jak bowiem będą się zachowywały obiekty przykładowych klas Główna i Pochodna przy komplikacji i wywoływaniu metod Wyswietl, jeśli kod będzie wyglądał tak jak na listingu 6.11?

Listing 6.11. Metoda prywatna w klasie bazowej

```
using System;  
  
public class Główna  
{  
    private void Wyswietl()  
    {  
        Console.WriteLine("Metoda Wyswietl klasy Główna");  
    }  
}
```

```

public class Pochodna : Glowna
{
    public void Wyswietl()
    {
        Console.WriteLine("Metoda Wyswietl klasy Pochodna");
    }
}

```

Takie klasy dadzą się bez problemów skompilować. Nie zobaczymy też żadnych ostrzeżeń, jak to miało np. miejsce w przypadku programu z listingu 3.49 i 3.50 w lekcji 19. (rozdział 3.), mimo że nie został użyty modyfikator new, nie ma też słów virtual czy override. Co więcej, w takiej sytuacji żadnego z tych słów nie wolno użyć⁴. Dlaczego? Otóż zauważmy, że metoda Wyswietl w klasie Glowna jest metodą prywatną, a więc nie jest dziedziczona przez klasę pochodną. Tym samym publiczna metoda Wyswietl w klasie Pochodna niczego nie przesłania i jest całkowicie niezależna! W obiektach klasy Pochodna w żaden sposób nie będzie można się odwoływać do metody Wyswietl z klasy Glowna, nawet korzystając z techniki rzutowania — taka jest przecież zasada działania modyfikatora private. Gdybyśmy więc spróbowali skompilować fragment widoczny na listingu 6.12, z pewnością próba ta zakończyłaby się błędem kompilacji (rysunek 6.12).

Listing 6.12. Nieprawidłowe odwołanie do metody Wyswietl

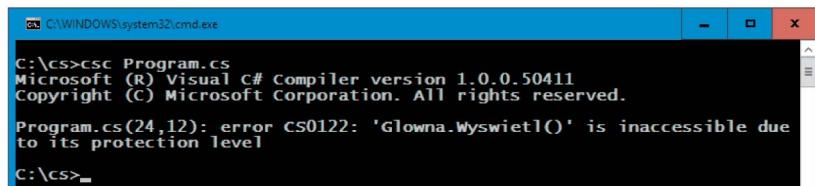
```

public class Program
{
    public static void Main()
    {
        Glowna glowna = (Glowna) new Pochodna();
        glowna.Wyswietl();
    }
}

```

Rysunek 6.12.

Nieprawidłowe odwołanie spowodowało błąd kompilacji



Ćwiczenia do samodzielnego wykonania

Ćwiczenie 29.1

Popraw kod z listingu 6.6 tak, aby po wystąpieniu wyjątku nie następowało niekontrolowane zakończenie programu. Zastosuj blok try...catch.

⁴ Dopuszczalne byłoby użycie modyfikatora new, choć nie miałby on żadnego praktycznego znaczenia. Kompilator wygenerowałby wtedy ostrzeżenie.

Ćwiczenie 29.2

Napisz kod klasy `Główna` i dziedziczącej po niej — `Pochodna`. Spraw, aby obiekty tych klas mogły być argumentami metody `WriteLine` klasy `Console` i aby w takiej sytuacji efektem działania tej metody był napis `Jestem obiektem klasy nazwa_klasy`.

Ćwiczenie 29.3

Napisz kod klasy `Program` testującej zachowanie obiektów klasy `Główna` i `Pochodna` z ćwiczenia 29.2.

Ćwiczenie 29.4

Wprowadź takie zmiany do kodu z listingu 6.9, aby klasa `Shape` dziedziczyła po innej klasie zawierającej metodę `Draw`. Efekt działania programu powstałego ze złączenia listingu 6.9 i 6.10 powinien pozostać bez zmian.

Ćwiczenie 29.5

Do klasy `Punkt` powstałej w ćwiczeniu 28.3 dopisz kod klasy pochodnej `Punkt3D`, tak aby obiektów typu `Punkt3D` można było również używać jako argumentu metody `WriteLine` (na ekranie powinny być wyświetcone trzy współrzędne). Dodatkowo napisz przykładowy kod testujący zachowanie nowej klasy.

Lekcja 30. Konstruktory oraz klasy abstrakcyjne

Lekcja 30. jest poświęcona klasom abstrakcyjnym oraz problematyce zachowań konstruktorów w specyficznych sytuacjach związanych z wywołaniami polimorficznymi. Będzie wyjaśnione, czym są klasy i metody abstrakcyjne oraz jak je stosować w praktyce. Zostanie również przedstawiona kolejność wywoływanego konstruktorów w hierarchii klas. Zostaną też omówione problemy, jakie można napotkać, kiedy w konstruktorach wywołuje się inne metody z danej klasy.

Klasy i metody abstrakcyjne

W poprzedniej lekcji wykorzystywaliśmy zestaw klas opisujących figury geometryczne. Klasy te dziedziczyły po wspólnej klasie bazowej `Shape`; zostały przedstawione na listingu 6.9. W tego typu przypadkach klasa bazowa często jest tylko atrapą, która w rzeczywistości nie wykonuje żadnych zadań, a służy jedynie do zdefiniowania zestawu metod, jakimi będą posługiwały się klasy potomne, oraz udostępnienia udogodnień, jakie niesie możliwość wywołań polimorficznych (przykłady z lekcji 29.). W tego typu sytuacjach często nie ma potrzeby lub jest wręcz niewskazane, aby były tworzone

obiekty klasy bazowej. W przypadku zwykłych klas nie można jednak nikomu zbroić tworzenia ich instancji — taką możliwość dają klasy abstrakcyjne.

Klasa abstrakcyjna (ang. *abstract class*) to taka klasa, która została zadeklarowana z użyciem słowa kluczowego `abstract`. Przy czym klasa, w której przynajmniej jedna metoda jest abstrakcyjna (oznaczona słowem kluczowym `abstract`), musi być zadeklarowana jako abstrakcyjna⁵. Schematycznie taka konstrukcja wygląda następująco:

```
[public] abstract class nazwa_klasy
{
    [specyfikator_dostępu] abstract typ_zwracany nazwa_metody(argumenty);
}
```

Metoda abstrakcyjna (ang. *abstract method*) ma jedynie deklarację zakończoną znakiem średnika, nie może zawierać żadnego kodu. Przykładowo metoda `draw` z klasy `Shape` z zestawu klas z listingu 6.9 mogłaby być z powodzeniem metodą abstrakcyjną. Oczywiście klasa `Shape` w takim wypadku również musiałaby być abstrakcyjna. Cała konstrukcja miałaby postać następującą:

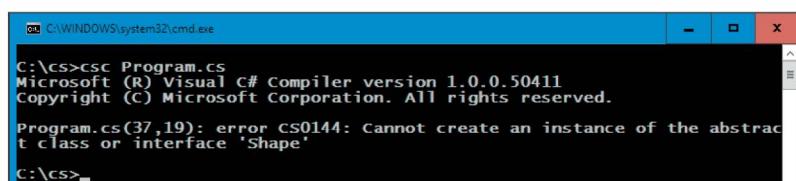
```
public abstract class Shape
{
    public abstract void Draw();
}
```

Po takiej deklaracji nie będzie możliwa tworzyć obiektów klasy `Shape`. Próba wykonania przykładowej instrukcji:

```
Shape shape = new Shape();
```

skończy się komunikatem o błędzie widocznym na rysunku 6.13.

Rysunek 6.13.
Próba utworzenia
instancji klasy
abstrakcyjnej



Co jednak ważniejsze, zadeklarowanie metody jako abstrakcyjnej wymusza jej redefinicję w klasie potomnej. Oznacza to, że każda klasa wyrowadzona z klasy `Shape` (czyli dziedzicząca po `Shape`), będzie musiała zawierać metodę `Draw`. Jeżeli w którejś z klas potomnych tej metody zabraknie, programu nie uda się skompilować. Tak więc przykładowa klasa `Triangle` w postaci:

```
public class Triangle : Shape
{
}
```

również spowoduje błąd komplikacji.

Mamy zatem pewność, że jeśli klasa bazowa zawiera metodę abstrakcyjną, to każda klasa potomna również ją zawiera. Można więc bezpiecznie stosować wywołania polimorficzne, takie jak omówione w poprzedniej lekcji.

⁵ Nie wyklucza to oczywiście istnienia klas abstrakcyjnych, w których żadna z metod nie jest abstrakcyjna.

Paść może w tym miejscu pytanie, dlaczego klasa, która zawiera metodę abstrakcyjną, również musi być zadeklarowana jako abstrakcyjna. Odpowiedź jest prosta: gdyby w zwykłej klasie została zadeklarowana metoda abstrakcyjna, a następnie utworzyliśmy obiekt tej klasy, nie byłoby przecież możliwe wywołanie tej metody, gdyż nie miałaby ona kodu wykonywalnego. Musiałaby się to skończyć błędem w trakcie wykonania aplikacji.

Pamiętajmy jednak, że zgodnie z definicją podaną wyżej klasa musi być zadeklarowana jako abstrakcyjna, jeżeli co najmniej jedna jej metoda jest abstrakcyjna. Wynika z tego, że nie ma żadnych przeciwwskazań, aby pozostałe metody nie były abstrakcyjne. Taka sytuacja została zilustrowana na listingu 6.13.

Listing 6.13. Budowa klas i metod abstrakcyjnych

```
using System;

public abstract class Shape
{
    public abstract void Draw();
    public virtual String Opis()
    {
        return "Opis kształtu";
    }
}

public class Rectangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Jestem prostokątem.");
    }
    public override String Opis()
    {
        return "Opis prostokąta";
    }
}

public class Triangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Jestem trójkątem.");
    }
}
```

Zostały tu zdefiniowane trzy klasy: klasa abstrakcyjna `Shape` oraz klasy od niej pochodne — `Rectangle` i `Triangle`. W klasie `Shape` mamy dwie metody — publiczną abstrakcyjną metodę `Draw` oraz publiczną wirtualną metodę `Opis`. Klasa `Rectangle` zawiera definicje metod `Draw` oraz `Opis`, natomiast `Triangle` — jedynie metody `Draw`. Jak już wiemy, taka sytuacja jest możliwa, gdyż metoda `Opis` klasy `Shape` nie jest abstrakcyjna, klasy pochodne nie muszą zatem zawierać jej definicji. Aby się przekonać, jak działa taki zestaw klas i metod, napiszemy dodatkowo testową klasę `Program`, która została zaprezentowana na listingu 6.14.

Listing 6.14. Sposób użycia klas z listingu 6.13

```
public class Program
{
    public static void DrawShape(Shape shape)
    {
        shape.Draw();
    }
    public static void Main()
    {
        Triangle triangle = new Triangle();
        Rectangle rectangle = new Rectangle();

        Console.WriteLine("Wywołania metod draw:");
        DrawShape(triangle);
        DrawShape(rectangle);
        Console.WriteLine("");

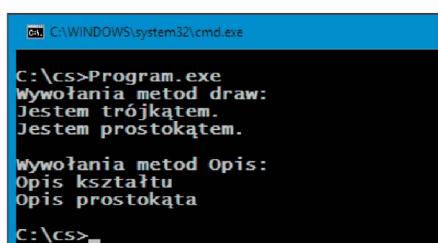
        Console.WriteLine("Wywołania metod Opis:");
        Console.WriteLine(triangle.Opis());
        Console.WriteLine(rectangle.Opis());
    }
}
```

Klasa Program zawiera metodę DrawShape przyjmującą jako argument obiekt klasy Shape. Metodzie tej przekazujemy obiekty klas Rectangle i Triangle. Jest to konstrukcja wywołania polimorficznego analogiczna do zaprezentowanej na listingu 6.10 w lekcji 29. W metodzie Main, od której rozpoczyna się wykonywanie kodu, tworzymy obiekty klas Rectangle i Triangle, a następnie wywołujemy metody DrawShape oraz Opis. Wszystkie te konstrukcje były już prezentowane wcześniej, nie wymagają więc dokładniejszego tłumaczenia.

Efekt działania całego kodu jest widoczny na rysunku 6.14. W pierwszej sekcji dzięki wywołaniom polimorficznym zostały wykonane metody DrawShape klas Triangle oraz Rectangle. Metody te musiały być w tych klasach zdefiniowane, gdyż klasy te dziedziczą po Shape, w której metoda Draw została zadeklarowana jako abstrakcyjna. W sekcji drugiej są wywoływane metody Opis obiektów triangle oraz rectangle. Ponieważ jednak w klasie Triangle nie zdefiniowano przesłoniętej metody Opis, została wywołana metoda Opis odziedziczona po klasie Shape. Jest to zachowanie jak najbardziej prawidłowe i zgodne z oczekiwaniami. Pamiętajmy więc, że w klasie abstrakcyjnej mogą znaleźć się również metody, które abstrakcyjne nie są.

Rysunek 6.14.

Efekt wywołań polimorficznych dla klas Rectangle i Triangle



Wywołania konstruktorów

Wywołania konstruktorów nie są polimorficzne, co więcej, nie są one nawet dziedziczone. Ważne jednak, abyśmy wiedzieli, jaką jest kolejność ich wywoływania, szczególnie w odniesieniu do hierarchii klas. Otóż w klasie potomnej zawsze musi zostać wywołany konstruktor klasy bazowej. Powinno to być oczywiste, jako że obiekt klasy potomnej zawiera w sobie obiekt klasy bazowej, a może przecież nie mieć dostępu do niektórych jego składowych. Skoro tak, zawsze najpierw powinien zostać wykonany konstruktor klasy bazowej, a dopiero potem klasy potomnej. Mamy wtedy pewność, że obiekt klasy bazowej został prawidłowo zainicjowany. Sytuacja taka jest zilustrowana na listingu 6.15.

Listing 6.15. Automatyczne wywołanie konstruktora klasy bazowej

```
using System;

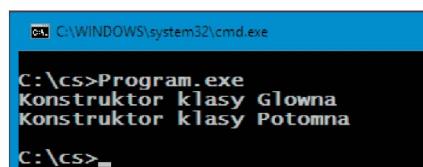
public class Glowna
{
    public Glowna()
    {
        Console.WriteLine("Konstruktor klasy Glowna");
    }
}

public class Potomna : Glowna
{
    public Potomna()
    {
        Console.WriteLine("Konstruktor klasy Potomna");
    }
    public static void Main()
    {
        Potomna obiekt = new Potomna();
    }
}
```

Klasa `Glowna` zawiera jeden publiczny konstruktor, którego zadaniem jest wyświetlenie na ekranie napisu oznajmującego, z jakiej klasy pochodzi. Klasa `Potomna`, dziedzicząca po `Glowna`, również zawiera jeden publiczny konstruktor, który wyświetla na ekranie informację, że pochodzi on z klasy `Potomna`. Konstruktory te są domyślne, jako że nie przyjmują żadnych argumentów. W klasie `Potomna` została dodatkowo zdefiniowana metoda `Main`, w której jest tworzony jeden obiekt klasy `Potomna`. Zatem zgodnie z tym, co zostało napisane powyżej, w takiej sytuacji najpierw zostanie wywołany konstruktor klasy bazowej `Glowna`, a dopiero po nim konstruktor klasy potomnej `Potomna`, co jest widoczne na rysunku 6.15.

Rysunek 6.15.

W klasie potomnej zawsze jest wywoływany konstruktor klasy bazowej



Obowiązuje zatem zasada, że jeśli w konstruktorze klasy potomnej nie zostanie jawnie wywołany żaden konstruktor klasy bazowej (nie zostanie użyta lista inicjalizacyjna, por. lekcja 16.), automatycznie zostanie wywołany domyślny konstruktor klasy bazowej (konstruktorem domyślnym jest konstruktor bezargumentowy). Co by się jednak stało, gdybyśmy w klasie `Glowna` z listingu 6.15 nie umieścili żadnego konstruktora, czyli gdyby miała ona zaprezentowaną niżej postać?

```
public class Glowna
{
}
```

Obie klasy udałoby się skompilować, a na ekranie pojawiłby się jedynie napis `Konstruktor klasy Potomna`. I choć wydawać by się mogło, że w takim razie nie został wywołany żaden konstruktor klasy `Glowna`, tak nie jest. Konstruktor musi zostać wywołany, nawet jeśli nie umieścimy go jawnie w ciele klasy. W takiej sytuacji kompilator dodaje własny pusty konstruktor domyślny, który oczywiście jest wywoływany.

Z zupełnie inną sytuacją będziemy jednak mieli do czynienia, kiedy w klasie bazowej nie umieścimy konstruktora domyślnego, ale znajdzie się w niej dowolny inny konstruktor. Sytuacja tego typu została zaprezentowana na listingu 6.16.

Listing 6.16. Klasa bez konstruktora domyślnego

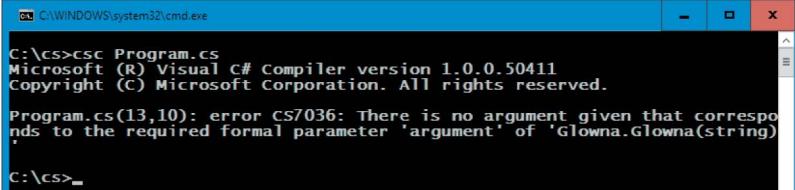
```
using System;

public class Glowna
{
    public Glowna(string argument)
    {
        Console.WriteLine("Konstruktor klasy Glowna");
    }
}

public class Potomna : Glowna
{
    public Potomna()
    {
        Console.WriteLine("Konstruktor klasy Potomna");
    }
    public static void Main()
    {
        Potomna obiekt = new Potomna();
    }
}
```

Klasa `Potomna` nie zmieniła się w stosunku do wersji zaprezentowanej na listingu 6.15, zmodyfikowana została natomiast treść klasy `Glowna`. Nie ma już konstruktora domyślnego, bezargumentowego, pojawił się natomiast konstruktor przyjmujący jeden argument typu `string`. Jak już wiemy, taka konstrukcja jest niepoprawna, gdyż konstruktor klasy `Potomna` będzie próbował wywołać domyślny konstruktor klasy `Glowna`, którego po prostu nie ma. Już przy próbie komplikacji zostanie zatem zasygnalizowany błąd, który został zaprezentowany na rysunku 6.16.

Rysunek 6.16.
Brak konstruktora domyślnego w klasie bazowej powoduje błąd kompilacji



The screenshot shows a Windows Command Prompt window titled 'C:\Windows\system32\cmd.exe'. The command entered is 'csc Program.cs'. The output shows the Microsoft Visual C# Compiler version 1.0.0.50411 and a copyright notice from Microsoft Corporation. A red error message is displayed: 'Program.cs(13,10): error CS7036: There is no argument given that corresponds to the required formal parameter 'argument' of 'Glowna.Glowna(string)''. The prompt 'C:\>csc>-' is visible at the bottom.

Kod z listingu 6.16 możemy poprawić na dwa sposoby: dopisując konstruktor domyślny do klasy `Glowna` lub też wywołując za pomocą listy inicjalizacyjnej istniejący w niej konstruktor. W tym drugim przypadku należałoby zastosować następującą konstrukcję:

```
public Potomna():base("")  
{  
    Console.WriteLine("Konstruktor klasy Potomna");  
}
```

Oczywiście wartość przekazana konstruktorowi klasy bazowej (w przykładzie jest to pusty ciąg znaków "") jest przykładowa, może to być dowolna wartość typu `string`.

Wykonajmy jeszcze przykład, w którym obiekty przedstawionych wcześniej klas `Glowna` i `Potomna` będą polami dodatkowej klasy `Program`, a oba obiekty utworzymy w konstruktorze klasy `Program`. Będziemy zatem mieć do czynienia z sytuacją przedstawioną na listingu 6.17.

Listing 6.17. Ilustracja kolejności wykonywania konstruktorów

```
using System;  
  
public class Glowna  
{  
    public Glowna()  
    {  
        Console.WriteLine("Konstruktor klasy Glowna");  
    }  
}  
  
public class Potomna : Glowna  
{  
    public Potomna()  
    {  
        Console.WriteLine("Konstruktor klasy Potomna");  
    }  
}  
  
public class Program  
{  
    Glowna obiekt1;  
    Potomna obiekt2;  
    public Program()  
    {  
        Console.WriteLine("Konstruktor klasy Program");  
        obiekt1 = new Glowna();  
        obiekt2 = new Potomna();  
    }  
}
```

```

    }
    public static void Main()
    {
        Program obiektP = new Program();
    }
}

```

Klasy Główna i Potomna mają tu postać taką jak na listingu 6.15, z tą różnicą, że z Potomna została usunięta metoda Main. W klasie Program zostały zadeklarowane dwa pola o nazwach obiekt1 i obiekt2. Pierwsze z nich jest typu Główna, drugie typu Potomna. Oba są inicjowane w konstruktorze klasy Program, wcześniej jednak konstruktor ten przedstawia się, czyli wyświetla na ekranie informację, z jakiej pochodzi klasy. W metodzie Main z kolei tworzymy nowy obiekt klasy Program. Jaka zatem będzie kolejność wykonania konstruktorów w tym przykładzie i ile napisów pojawi się na ekranie?

Skoro tworzymy obiekt klasy Program, w pierwszej kolejności zostanie wywołany konstruktor tej klasy, a tym samym najpierw pojawi się na ekranie napis Konstruktor klasy Program. W konstruktorze tym jest tworzony obiekt klasy Główna, zatem wywołany zostanie konstruktor tej klasy i będzie wyświetlony napis Konstruktor klasy Główna. W kolejnym kroku tworzymy obiekt klasy Potomna. Klasa Potomna dziedziczy jednak po Głównej, zatem w tym kroku zostaną wywołane dwa konstruktory, najpierw klasy bazowej Głównej, a następnie potomnej Potomnej. W związku z tym na ekranie pojawi się ciąg napisów widoczny na rysunku 6.17.

Rysunek 6.17.
Złożone wywołania
konstruktorów

```

C:\cs>Program.exe
Konstruktor klasy Program
Konstruktor klasy Główna
Konstruktor klasy Główna
Konstruktor klasy Potomna
C:\cs>_

```

Wywoływanie metod w konstruktorach

W konstruktorach można wywoływać inne metody danej klasy. Wydaje się to oczywiste. Musimy jednak pamiętać, gdyż w połączeniu z polimorfizmem może nas to wprowadzić w pułapkę. Przyjrzyjmy się bowiem klasom przedstawionym na listingu 6.18.

Listing 6.18. Niebezpieczeństwa związane z wywoływaniem metod

```

using System;

public class A
{
    public A()
    {
        //Console.WriteLine("Konstruktor klasy A");
        f();
    }
    public virtual void f()

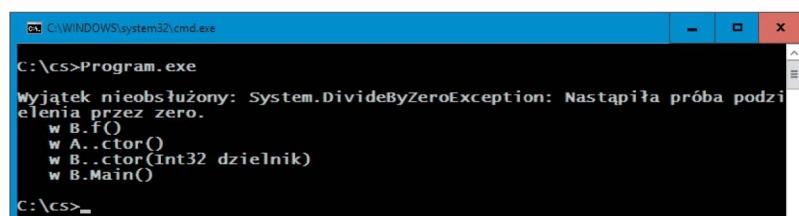
```

```
{  
    //Console.WriteLine("Klasa A metoda f()");  
}  
}  
  
public class B : A  
{  
    int dzielnik;  
    public B(int dzielnik)  
    {  
        //Console.WriteLine("Konstruktor klasy B");  
        this.dzielnik = dzielnik;  
    }  
    public override void f()  
    {  
        //Console.WriteLine("Klasa B metoda f()");  
        double wynik = 1 / dzielnik;  
        Console.WriteLine(  
            "Dzielenie całkowite 100 / {0} daje wynik: {1}", dzielnik, wynik);  
    }  
    public static void Main()  
    {  
        B b = new B(1);  
        b.f();  
    }  
}
```

Przy pierwszym spojrzeniu kod ten wygląda całkiem poprawnie. Klasa A zawiera publiczną metodę f, która nie robi nic, oraz wywołujący ją konstruktor domyślny. W komentarzach zostały umieszczone instrukcje Console.WriteLine, które później pozwolą nam dokładniej prześledzić sposób działania programu.

Klasa B dziedziczy po klasie A. Ma ona jeden konstruktor przyjmujący argument typu int. Wartość tego argumentu jest przypisywana polu typu int o nazwie dzielnik. Została również zdefiniowana metoda f, która wykonuje dzielenie wartości 1 przez wartość zapisaną w polu o nazwie dzielnik i wyświetla wynik tego dzielenia na ekranie. W metodzie Main tworzymy nowy obiekt klasy B, przekazując konstruktorowi wartość 1, zatem pole dzielnik przyjmuje wartość 1. Następnie wywołujemy metodę f. Spodziewamy się, że metoda ta zgodnie z naszymi intencjami wykona dzielenie 1/1, zatem na ekranie pojawi się napis 1 / dzielnik to: 1. Uruchommy więc taki program i zobaczymy, co się stanie. Przedstawiono to na rysunku 6.18.

Rysunek 6.18.
Pulpapka związana
z polimorficznym
wywoływaniem
metod



To zapewne bardzo niemiła niespodzianka, ewidentnie został wygenerowany wyjątek klasy DivideByZeroException; w metodzie f zostało wykonane dzielenie przez zero.

Jak to się jednak mogło stać, skoro — to nie ulega wątpliwości — konstruktorowi klasy B przekazaliśmy argument o wartości 1? Zatem polu `dzielnik` musiała zostać przypisana wartość 1. Skąd więc wyjątek? Aby dokładniej zobaczyć, co się tak naprawdę stało, usuńmy teraz znaki komentarza z instrukcji `Console.WriteLine` w obu klasach oraz ponownie skompilujmy i uruchommy program. Zobaczmy widok zaprezentowany na rysunku 6.19. Widać teraz wyraźnie, że konstruktor klasy B wcale nie zdążył się wykonać, wyjątek nastąpił wcześniej.

Rysunek 6.19.

Konstruktor klasy B
nie został do końca
wykonany

```
C:\cs>Program.exe
Konstruktor klasy A
Klasa B metoda f()
wyjątek nieobsłużony: System.DivideByZeroException: Nastąpiła próba podzielenia przez zero.
  w B.f()
  w A..ctor()
  w B..ctor(Int32 dzielnik)
  w B.Main()

C:\cs>_
```

Prześledźmy więc dokładniej kolejne etapy działania programu. W metodzie `Main` z klasy B tworzymy obiekt tej klasy. Powoduje to oczywiście wywołanie konstruktora klasy B, ale uwaga: zgodnie z tym, co zostało przedstawione już w tej lekcji, ponieważ klasa B dziedziczy po A, przed wykonaniem jej konstruktora jest wywoływany konstruktor domyślny klasy A. Spójrzmy więc, co się dzieje w konstruktorze klasy A. Otóż jest tam wywoływana metoda f. Skoro jednak sam obiekt jest klasy B (a metoda f jest wirtualna), wywołanie to będzie **polimorficzne**, a zatem zostanie wywołana metoda f z klasy B! Oznacza to, że fragment kodu:

```
double wynik = 1 / dzielnik;
Console.WriteLine("1 / dzielnik to: " + wynik);
```

zostanie wykonany, **zanim** jeszcze polu `dzielnik` zostanie przypisana jakakolwiek wartość. Pole `dzielnik` jest w tym momencie po prostu niezainicjowane, a jak wiadomo z lekcji 13., niezainicjowane jawnie pole typu `int` ma wartość... 0. Dlatego też w tym przykładzie został wygenerowany wyjątek `DivideByZeroException` — przecież przez zero dzielić nie wolno.

Koniecznie należy więc pamiętać, że również w konstruktorach wywołania metod są polimorficzne, czyli skojarzenie treści metody odbywa się w trakcie działania programu i brany jest pod uwagę rzeczywisty typ obiektu. To niestety może prowadzić do trudnych do wykrycia błędów, których mechanizm powstawania został przedstawiony w powyższym przykładzie.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 30.1

Napisz klasę `Glowna` zawierającą abstrakcyjną metodę `Wyswietl`. Z klasy tej wyprowadź klasę potomną `Potomna`.

Ćwiczenie 30.2

Napisz klasę First zawierającą abstrakcyjną metodę f, z tej klasy wyprowadź klasę potomną Second zawierającą abstrakcyjną metodę g. Z klasy Second wyprowadź klasę Third.

Ćwiczenie 30.3

Zmodyfikuj kod z listingu 6.17 tak, aby obiekty obiekt1 i obiekt2 były inicjowane nie w konstruktorze klasy Program, ale w momencie ich deklaracji. Zaobserwuj kolejność wykonywania konstruktorów.

Ćwiczenie 30.4

Zmodyfikuj kod z listingu 6.17 tak, aby klasa Program dziedziczyła po Potomna. Jaka będzie kolejność wykonania konstruktorów?

Ćwiczenie 30.5

Popraw program z listingu 6.18 tak, aby nie występował błąd dzielenia przez 0.

Interfejsy

Lekcja 31. Tworzenie interfejsów

W lekcji 30. przedstawione były klasy abstrakcyjne, czyli takie, w których część metod (lub wszystkie) nie miała implementacji. Implementacja tych metod musiała być zrealizowana w klasach potomnych (o ile klasy potomne nie były również zadeklarowane jako abstrakcyjne). Podczas tej lekcji poznamy interfejsy, które można postraktować jako klasy czysto abstrakcyjne, nieposiadające żadnej implementacji.

Czym są interfejsy?

Interfejs to klasa czysto abstrakcyjna, czyli taka, w której wszystkie metody są traktowane jako abstrakcyjne. Deklarujemy go za pomocą słowa kluczowego interface. Interfejs może być publiczny, o ile użyte zostanie słowo public, lub wewnętrzny, o ile zostanie użyte słowo internal lub nie zostanie użyte żadne z tych słów. W tym drugim przypadku jest dostępny jedynie dla klas wchodzących w skład danego zestawu. Schematyczna konstrukcja interfejsu wygląda więc następująco:

```
[public | internal] interface nazwa_interfejsu
{
    składowe interfejsu
}
```

Składowymi interfejsu mogą być metody, właściwości oraz zdarzenia (omówione w lekcji 36. rozdziału 7.)⁶. Interfejs przypomina pod względem budowy klasę abstrakcyjną. Wszystkie składowe interfejsu są jednak zawsze domyślnie publiczne i tego zachowania nie można zmienić. Nie należy nawet używać słowa `public` — składowa po prostu zawsze jest publiczna.

Przykładowy interfejs o nazwie `IDrawable` (zwyczajowo nazwy interfejsów poprzedzane są dużą literą I) zawierający deklarację jednej tylko metody o nazwie `Draw` został przedstawiony na listingu 6.19.

Listing 6.19. Przykład prostego interfejsu

```
public interface IDrawable
{
    void Draw();
}
```

Tak zdefiniowany interfejs może być implementowany przez dowolną klasę. Mówimy, że dana klasa implementuje interfejs lub dziedziczy po interfejsie. Oznacza to, że zawiera ona definicje wszystkich zadeklarowanych w nim składowych. Jeśli pominiemy choć jedną składową, kompilator zgłosi błąd. Implementacja interfejsu wygląda tak samo jak dziedziczenie po zwykłej klasie, schematycznie:

```
[specyfikator dostępu] [abstract] class nazwa_klasy : nazwa_interfejsu
{
    /*
    składowe klasy
    */
}
```

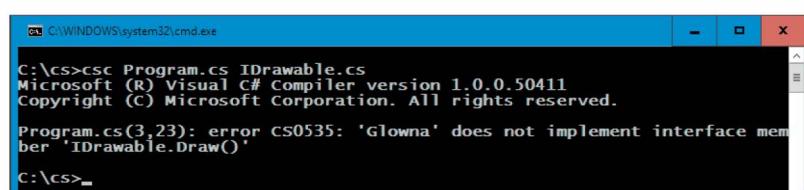
Jeśli zatem chcemy, aby przykładowa klasa `Glowna` implementowała interfejs `IDrawable` z listingu 6.19, musimy napisać:

```
public class Glowna : IDrawable
{
}
```

Próba kompilacji takiej klasy skończy się... błędem kompilacji. Jest to widoczne na rysunku 6.20.

Rysunek 6.20.

Brak definicji
metody
zadeklarowanej
w interfejsie



Zapomnieliśmy bowiem, że zgodnie z tym, co zostało napisane wcześniej, klasa, która implementuje interfejs, musi zawierać implementację wszystkich jego składowych⁷.

⁶ A także indeksery, które nie będą omawiane w tej publikacji.

⁷ Lub też musi być klasą abstrakcyjną.

W tym przypadku jedyną składową jest metoda Draw. Poprawna klasa Główna będzie zatem wyglądała tak, jak to zostało przedstawione na listingu 6.20. Oczywiście treść metody Draw może być dowolna.

Listing 6.20. Klasa implementująca interfejs IDrawable

```
public class Główna : IDrawable
{
    public void Draw()
    {
        System.Console.WriteLine("Draw");
    }
}
```

Widać już więc, że interfejs określa po prostu, jakie metody muszą znaleźć się w klasie, która go implementuje. Przypomnijmy sobie teraz przykłady z figurami z lekcji 29. i 30. (klasy Triangle, Rectangle, Circle). Wymuszaliśmy w tych klasach deklarację metody Draw. Odbywało się to poprzez umieszczenie w klasie nadrzędnej Shape abstrakcyjnej metody Draw. Możemy to jednak zrobić w inny sposób, wykorzystując interfejs IDrawable. Wystarczy, jeśli każda z klas będzie ten interfejs implementowała, tak jak jest to przedstawione na listingu 6.21. W takiej sytuacji również występuje konieczność deklaracji metody Draw w każdej z przedstawionych klas.

Listing 6.21. Zestaw klas dziedziczących po IDrawable

```
public class Circle : IDrawable
{
    public void Draw()
    {
        System.Console.WriteLine("Jestem okręgiem.");
    }
}

public class Rectangle : IDrawable
{
    public void Draw()
    {
        System.Console.WriteLine("Jestem prostokątem.");
    }
}

public class Triangle : IDrawable
{
    public void Draw()
    {
        System.Console.WriteLine("Jestem trójkątem.");
    }
}
```

Interfejsy a hierarchia klas

Przykład z listingu 6.21 umożliwił pokazanie, jak implementować jeden interfejs w wielu klasach, jednak w stosunku do przykładów z poprzednich lekcji została w nim zmieniona hierarchia klas. Wszak wcześniej wszystkie trzy klasy, Triangle, Rectangle oraz Circle, dziedziczyły po klasie Shape, co pozwalało na stosowanie wywołań polimorficznych. Zastosowanie interfejsów na szczęście nic tu nie zmienia, czyli wszystkie trzy klasy nadal mogą dziedziczyć po Shape i jednocześnie implementować interfejs IDrawable. Sytuacja taka jest przedstawiona na listingu 6.22.

Listing 6.22. Jednoczesne dziedziczenie po klasie bazowej i po interfejsie

```
using System;

public interface IDrawable
{
    void Draw();
}

public class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Jestem jakimś kształtem.");
    }
}

public class Circle : Shape, IDrawable
{
    public override void Draw()
    {
        Console.WriteLine("Jestem okręgiem.");
    }
}

public class Rectangle : Shape, IDrawable
{
    public override void Draw()
    {
        Console.WriteLine("Jestem prostokątem.");
    }
}

public class Triangle : Shape, IDrawable
{
    public override void Draw()
    {
        Console.WriteLine("Jestem trójkątem.");
    }
}
```

Jak widać, istnieje możliwość jednoczesnego dziedziczenia po klasie i po interfejsie. Nazwy klasy i interfejsu należy wtedy oddzielić znakiem przecinka, czyli schematycznie taka konstrukcja ma postać:

```
[modyfikator_dostępu] class potomna : bazowa, interfejs
{
    /* treść klasy */
}
```

Oczywiście klasa potomna dziedzicząca również po interfejsie musi zawierać składowe tego interfejsu. Tak jest w powyższym przykładzie; klasy Circle, Rectangle i Triangle zawierają definicje metody Draw. W tym wypadku każda z tych metod dodatkowo przesyłania metodę Draw z klasy bazowej (Shape). Zwróćmy uwagę, że ten sposób jest dobry, jeśli chcemy, aby niektóre z klas potomnych miały przesyłać metodę Draw z klasy bazowej. Wymusza to właśnie implementacja interfejsu IDrawable. Każda zatem klasa, która koniecznie ma przesyłać metodę Draw, powinna dziedziczyć i po bazowej (Shape), i po interfejsie (IDrawable), natomiast klasy, w których przesłonięcie nie jest konieczne, mogłyby dziedziczyć jedynie po Shape.

Co jednak zrobić w sytuacji, kiedy chcemy wymusić implementację metod interfejsu we wszystkich klasach potomnych? Można by wprowadzić dziedziczenie po interfejsie w każdej z tych klas, można to jednak też zrobić nieco inaczej. Otóż klasa bazowa mogłaby być abstrakcyjna i dziedziczyć po interfejsie, przy czym składowe wynikające z dziedziczenia interfejsu też powinny być wtedy abstrakcyjne. Taka sytuacja została zobrazowana na listingu 6.23.

Listing 6.23. Dziedziczenie interfejsu po klasie bazowej

```
using System;

public interface IDrawable
{
    void Draw();
}

public abstract class Shape : IDrawable
{
    public abstract void Draw();
}

public class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Jestem okręgiem.");
    }
}

public class Rectangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Jestem prostokątem.");
    }
}

public class Triangle : Shape
{
    public override void Draw()
```

```
{  
    Console.WriteLine("Jestem trójkątem.");  
}  
}
```

Interfejs `IDrawable` pozostał bez zmian. Klasa `Shape` dziedziczy po `IDrawable`, a więc musi zawierać deklarację metody `Draw`. Ta metoda została jednak oznaczona jako abstrakcyjna (przez co klasa również musi być abstrakcyjna). Dzięki takiej konstrukcji wszystkie klasy potomne od `Shape` muszą również zawierać metodę `Draw`, a nie muszą bezpośrednio dziedziczyć po `IDrawable`.

Interfejsy i właściwości

W C# interfejsy nie mogą zawierać pól, ale nic nie stoi na przeszkodzie, aby umieszczać w nich właściwości. Definicja właściwości interfejsu wygląda podobnie jak w przypadku klasy, z tą różnicą, że metody `get` i `set` mają jedynie deklaracje, a pozbawione są wnętrza (definicji). Schematyczny wygląd interfejsu zawierającego jedną właściwość z możliwością jej zapisu i odczytu (czyli z deklaracjami zarówno `get`, jak i `set`) jest następujący:

```
[public] interface INazwaInterfejsu  
{  
    typ_właściwości nazwa_właściwości  
    {  
        get;  
        set;  
    }  
}
```

Przykładowy interfejs o nazwie `IPunkt` zawierający właściwości `x` i `y` będzie miał zatem postać przedstawioną na listingu 6.24.

Listing 6.24. Interfejs `IPunkt`

```
public interface IPunkt  
{  
    int x  
    {  
        get;  
        set;  
    }  
    int y  
    {  
        get;  
        set;  
    }  
}
```

Warto teraz napisać treść klasy implementującej taki interfejs. Z powodzeniem mogłyby to być klasa o nazwie `Punkt`. Powinna ona wtedy zawierać prywatne pola pozwalające na przechowywanie wartości `x` i `y` oraz definicje właściwości `x` i `y` wraz z akcesorami

get i set umożliwiającymi wykonywanie operacji odczytu i zapisu. Treść takiej klasy znajduje się na listingu 6.25, a przykład jej użycia na listingu 6.26.

Listing 6.25. Implementacja interfejsu IPunkt

```
public class Punkt : IPunkt
{
    private int wspX, wspY;
    public int x
    {
        get
        {
            return wspX;
        }
        set
        {
            wspX = value;
        }
    }
    public int y
    {
        get
        {
            return wspY;
        }
        set
        {
            wspY = value;
        }
    }
}
```

Listing 6.26. Użycie klasy Punkt implementującej interfejs IPunkt

```
public class Program
{
    public static void Main()
    {
        Punkt punkt1 = new Punkt();
        punkt1.x = 100;
        punkt1.y = 200;
        Console.WriteLine("x = {0}", punkt1.x);
        Console.WriteLine("y = {0}", punkt1.y);
    }
}
```

Klasa Punkt zawiera dwa prywatne pola wspX i wspY przechowujące wartości współrzędnych x i y, a także właściwości x i y, których obecność jest niezbędna ze względu na dziedziczenie po interfejsie IPunkt. Akcesor get właściwości x zwraca po prostu wartość pola wspX, natomiast akcesor set właściwości x ustawia wartość pola wspX. Konstrukcja akcesorów dla właściwości y jest analogiczna.

W metodzie Main klasy Program jest wykonywany test operacji na obiekcie klasy Punkt. Najpierw jest on tworzony, następnie ustawiane są wartości właściwości x i y,

a na koniec wartości tych właściwości są wyświetlane za pomocą instrukcji `Console.WriteLine`. Dzięki temu można się przekonać, że implementacja interfejsu `IPunkt` w klasie `Punkt` zakończyła się sukcesem.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 31.1

Napisz kod interfejsu o nazwie `IRysowanie`, w którym zostaną zadeklarowane metody `Rysuj2D` i `Rysuj3D`. Następnie napisz klasę `Figura`, która będzie implementowała ten interfejs.

Ćwiczenie 31.2

Zmodyfikuj kod z listingu 6.22 tak, aby klasa `Shape` implementowała interfejs `IDrawable` oraz zawierała jeden konstruktor przyjmujący argument typu `string`. Przekazany w ten sposób ciąg znaków powinien być używany w komunikacie wyświetlonym przez metodę `Draw`.

Ćwiczenie 31.3

Napisz przykładowy interfejs zawierający dwie właściwości — jedną typu `int`, drugą typu `double` — oraz implementującą go klasę.

Ćwiczenie 31.4

Napisz przykładowy program korzystający z klasy powstałej w ćwiczeniu 31.3.

Ćwiczenie 31.5

Napisz interfejs `IPunkt3D` pochodny od interfejsu `IPunkt` z listingu 6.24. Przygotuj następnie klasę `Punkt3D` implementującą ten interfejs, pochodną od klasy `Punkt` z listingu 6.25. Powstały kod przetestuj np. z programem z listingu 6.4.

Lekcja 32. Implementacja kilku interfejsów

Wiadomo już, w jaki sposób tworzyć i implementować interfejsy w klasach. Zagadnienia te zostały omówione w lekcji 31. Kolejnym tematem, którym się zajmiemy, jest technika pozwalająca na implementację przez jedną klasę wielu interfejsów. Sprawdzimy przy tym, jak unikać niebezpieczeństw związanych z konfliktami nazw metod w interfejsach. W tej lekcji okaże się również, że interfejsy, tak jak inne klasy, podlegają regułom dziedziczenia. Zobaczmy, że jeden interfejs może przejąć metody nawet z kilku innych.

Implementowanie wielu interfejsów

W C# klasa potomna może dziedziczyć tylko po jednej klasie bazowej, nie ma więc znanego m.in. z C++ wielodziedziczenia. Istnieje natomiast możliwość implementowania wielu interfejsów. Interfejsy, które mają być implementowane przez klasę, należy oddzielić znakami przecinka. Schematycznie taka konstrukcja wygląda następująco:

```
[public] [abstract] class nazwa_klasy : interfejs1, interfejs2, ... , interfejsN
{
    /*
    pola i metody klasy
    */
}
```

Jeśli mamy na przykład dwa interfejsy, pierwszy o nazwie `PierwszyInterfejs`, definiujący metodę `f`, i drugi o nazwie `DrugiInterfejs`, definiujący metodę `g`, oraz klasę `MojaKlasa`, implementującą oba, całość przyjmie postać przedstawioną na listingu 6.27.

Listing 6.27. Jednoczesna implementacja dwóch interfejsów

```
public interface PierwszyInterfejs
{
    void f();
}

public interface DrugiInterfejs
{
    void g();
}

public class MojaKlasa : PierwszyInterfejs, DrugiInterfejs
{
    public void f()
    {
    }
    public void g()
    {
    }
}
```

Klasa `MojaKlasa` musi oczywiście zawierać definicje wszystkich metod zadeklarowanych w interfejsach `PierwszyInterfejs` i `DrugiInterfejs`. W tym przypadku są to metody `f` i `g`. Pominięcie którejkolwiek z nich spowoduje, rzecz jasna, błąd komplikacji.

Po co jednak klasa miałaby implementować wiele interfejsów? Czy nie lepiej byłoby po prostu napisać jeden? Okazuje się, że nie, tracimy bowiem wtedy uniwersalność kodu. Założmy, że mamy klasy `Telewizor` i `Radio` oraz dwa interfejsy `IWydajeDzwiek` oraz `IWyswietlaObraz`. Sytuacja taka jest zobrazowana na listingu 6.28.

Listing 6.28. Uniwersalność interfejsów

```
public interface IWydajeDzwiek
{
    void Graj();
}
```

```
public interface IWyswietlaObraz
{
    void Wyswietl();
}

public class Radio : IWydajeDzwiek
{
    public void Graj()
    {
        //instrukcje metody Graj
    }
}

public class Telewizor : IWydajeDzwiek, IWyswietlaObraz
{
    public void Graj()
    {
        //instrukcje metody Graj
    }
    public void Wyswietl()
    {
        //instrukcje metody Wyswietl
    }
}
```

Jak wiadomo, telewizor zarówno wyświetla obrazy, jak i wydaje dźwięki, implementuje zatem oba interfejsy: `IWyswietlaObraz` i `IWydajeDzwiek`, a więc zawiera również metody `Graj` i `Wyswietl`. Radio tymczasem obrazów nie wyświetla, a zatem implementuje jedynie interfejs `IWydajeDzwiek` i klasa ta zawiera tylko metodę `Graj`. Jak widać, dzięki temu, że jedna klasa może implementować wiele interfejsów, mogą być one bardziej uniwersalne. Gdyby taka możliwość nie istniała, dla telewizora musiałby powstać interfejs, np. o nazwie `IWyswietlaObrazIWydajeDzwiek`, w którym zostałyby zadeklarowane metody `Graj` i `Wyswietl`, co ograniczałoby jego zastosowanie. Może być to jednak ćwiczeniem do samodzielnego wykonania.

Konflikty nazw

Kiedy jedna klasa implementuje wiele interfejsów, mogą powstać budzące wątpliwości sytuacje konfliktu nazw, które niestety mogą również prowadzić do powstawania błędów (na szczęście zwykle wykrywanych w trakcie kompilacji). Założmy, że mamy dwa interfejsy przedstawione na listingu 6.29.

Listing 6.29. Interfejsy zawierające taką samą metodę

```
public interface IPierwszyInterfejs
{
    void f();
}

public interface IDrugiInterfejs
{
    void f();
}
```

Są one w pełni poprawne, kod ten nie budzi żadnych wątpliwości. Co się natomiast stanie, kiedy przykładowa klasa `MojaKlasa` będzie miała implementować oba te interfejsy? Z kodu pierwszego interfejsu wynika, że powinna ona mieć zdefiniowaną metodę o nazwie `f`, z kodu drugiego wynika dokładnie to samo. Czy zatem klasa ta ma zawierać dwie metody `f`? Oczywiście nie, nie byłoby to przecież możliwe. Skoro jednak deklaracje metod `f` w obu interfejsach są takie same, oznacza to, że klasa implementująca oba interfejsy musi zawierać **jedną** metodę `f` o deklaracji `public void f()`. Zatem przykładowa klasa `MojaKlasa` będzie miała postać przedstawioną na listingu 6.30.

Listing 6.30. Klasa implementująca interfejsy zawierające taką samą metodę

```
public class MojaKlasa : IPierwszyInterfejs, IDrugiInterfejs
{
    public void f()
    {
        System.Console.WriteLine("Metoda f");
    }
}
```

Nieco inna, ale również jednoznaczna sytuacja jest wtedy, kiedy w dwóch interfejsach będziemy mieli metody o takiej samej nazwie, ale innych argumentach. Wtedy również będzie możliwa zaimplementowanie je w jednej klasie. Takie dwa przykładowe interfejsy zostały przedstawione na listingu 6.31.

Listing 6.31. Interfejsy zawierające metody o jednej nazwie, ale różnych argumentach

```
public interface IPierwszyInterfejs
{
    void f(int argument);
}

public interface IDrugiInterfejs
{
    int f(double argument);
}
```

W interfejsie pierwszym została zadeklarowana niezwracająca wyniku metoda `f` przyjmująca jeden argument typu `int`, natomiast w drugim metoda o takiej samej nazwie, przyjmująca jednak argument typu `double` i zwracającą wartość typu `int`. Skoro tak, przykładowa klasa `MojaKlasa` implementująca oba interfejsy będzie musiała mieć implementacje obu metod. Będzie zatem wyposażona w przeciążone metody `f` (lekcja 15.). Może ona wyglądać na przykład tak, jak to zostało przedstawione na listingu 6.32.

Listing 6.32. Interfejsy wymuszają przeciążanie metod

```
using System;

public class MojaKlasa : IPierwszyInterfejs, IDrugiInterfejs
{
    public void f()
    {
        Console.WriteLine("Metoda f");
    }
}
```

```
        }
        public void f(int argument)
        {
            Console.WriteLine("f:argument = " + argument);
        }
        public int f(double argument)
        {
            Console.WriteLine("f:argument = " + argument);
            return (int) argument;
        }
    }
```

Klasa ta implementuje dwa interfejsy o nazwach `IPierwszyInterfejs` i `IDrugiInterfejs`. Zawiera również trzy przeciążone metody `f`. Pierwsza z nich nie zwraca żadnych wartości oraz nie przyjmuje żadnych argumentów — nie jest też bezpośrednio związana z interfejsami. Metoda druga została wymuszona przez interfejs `IPierwszyInterfejs` i również nie zwraca żadnej wartości, przyjmuje natomiast argument typu `int` (tak jak zostało to zdefiniowane w interfejsie). Metoda trzecia została wymuszona przez interfejs `IDrugiInterfejs`. Przyjmuje ona jeden argument typu `double` oraz zwraca wartość typu `int`. Tak więc w tym przypadku kod jest również w pełni poprawny i jednoznaczny.

Zastanówmy się jednak, co się stanie, kiedy interfejsy przyjmą postać widoczną na listingu 6.33. Czy jakakolwiek klasa może je jednocześnie implementować?

Listing 6.33. Interfejsy zawierające bezargumentową metodę o tej samej nazwie

```
public interface IPierwszyInterfejs
{
    void f();
}

public interface IDrugiInterfejs
{
    int f();
}
```

Odpowiedź na zadane powyżej pytanie brzmi: nie. Cóż by się bowiem stało, gdyby nasza przykładowa klasa `MojaKlasa` miała implementować oba interfejsy przedstawione na listingu 6.33? Musiałaby mieć postać z listingu 6.34.

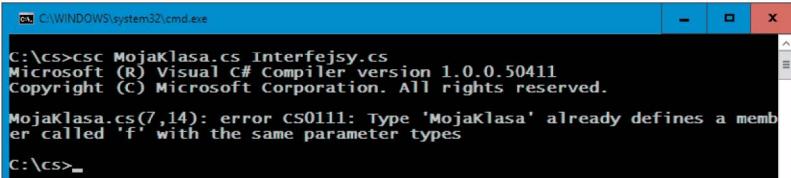
Listing 6.34. Błędna implementacja interfejsów

```
public class MojaKlasa : IPierwszyInterfejs, IDrugiInterfejs
{
    public void f()
    {
        Console.WriteLine("Metoda void f()");
    }
    public int f()
    {
        Console.WriteLine("Metoda int f()");
        return 0;
    }
}
```

Jest ona w sposób oczywisty nieprawidłowa. W jednej klasie nie mogą istnieć dwie metody o takiej samej nazwie, różniące się jedynie typem zwracanego wyniku. Próba komplikacji takiego kodu spowoduje powstanie błędu widocznego na rysunku 6.21.

Rysunek 6.21.

Nieprawidłowa implementacja interfejsów w klasie MojaKlasa



```
C:\>csc MojaKlasa.cs Interfejsy.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

MojaKlasa.cs(7,14): error CS0111: Type 'MojaKlasa' already defines a member called 'f' with the same parameter types

C:\>
```

Dziedziczenie interfejsów

Interfejsy można budować, stosując mechanizm dziedziczenia. Odbywa się to tak samo jak w przypadku klas. Interfejs potomny będzie zawierał wszystkie swoje składowe oraz wszystkie składowe z interfejsu bazowego. Schemat dziedziczenia w przypadku interfejsów wygląda następująco:

```
[public] interface interfejs_potomny : interfejs_bazowy
{
    //deklaracje składowych interfejsu
}
```

Jeśli mamy zatem przykładowy interfejs I1 zawierający metodę f, możemy wprowadzić z niego interfejs I2 zawierający metodę g. Taka sytuacja została przedstawiona na listingu 6.35.

Listing 6.35. Przykład dziedziczenia interfejsów

```
public interface I1
{
    void f();
}

public interface I2 : I1
{
    void g();
}
```

Jak widać, interfejs I2 dziedziczy po interfejsie I1, zatem zawiera własną metodę g oraz odziedziconą f. Przykładowa klasa, która będzie implementowała interfejs I2, będzie zatem musiała zawierać zarówno metodę f, jak i g, inaczej nie uda nam się skompilować jej kodu. Taka przykładowa klasa jest widoczna na listingu 6.36.

Listing 6.36. Klasa implementująca interfejs potomny

```
public class MojaKlasa : I2
{
    public void f()
    {
        //treść metody f
    }
}
```

```
public void g()
{
    //treść metody g
}
```

Interfejs, inaczej niż klasa, może dziedziczyć nie tylko po jednym, ale po wielu interfejsach. Interfejsy bazowe należy w takiej sytuacji umieścić po dwukropku, oddzielając ich nazwy przecinkami. Schematycznie konstrukcja taka wygląda następująco:

```
[public] interface nazwa_interfejsu : interfejs1, interfejs2, ... , interfejsN
{
    //składowe interfejsu
}
```

Jeśli mamy zatem dwa przykładowe interfejsy bazowe I1 i I2, zawierające metody f i g, a chcemy utworzyć interfejs potomny I3, który odziedziczy wszystkie ich właściwości, powinniśmy zastosować konstrukcję widoczną na listingu 6.37.

Listing 6.37. Dziedziczenie po wielu interfejsach

```
public interface I1
{
    void f();
}

public interface I2
{
    void g();
}

public interface I3 : I1, I2
{
    void h();
}
```

Oczywiście przykładowa klasa implementująca interfejs I3 będzie musiała mieć zdefiniowane wszystkie metody z interfejsów I1, I2 i I3, czyli metody f, g i h. Pominięcie którejkolwiek z nich spowoduje błąd komplikacji. Przykładowa klasa MojaKlasa implementująca interfejs I3 została przedstawiona na listingu 6.38.

Listing 6.38. Klasa implementująca interfejs I3

```
public class MojaKlasa : I3
{
    public void f() //wymuszona przez I1
    {
        //treść metody f
    }
    public void g() //wymuszona przez I2
    {
        //treść metody g
    }
    public void h() //wymuszona przez I3
```

```
{  
    //treść metody h  
}
```

Przy wykorzystywaniu mechanizmu dziedziczenia w interfejsach musimy pamiętać o możliwych konfliktach nazw; konflikty te były omawiane w poprzedniej części lekcji. Takie niebezpieczeństwo istnieje zarówno przy dziedziczeniu pojedynczym, jak i wielokrotnym. Na listingu 6.39 jest przedstawiony przykład nieprawidłowego dziedziczenia po jednym interfejsie bazowym. Otóż w interfejsie I1 została zadeklarowana metoda f niezwracająca wyniku, natomiast w interfejsie I2 również metoda o nazwie f, ale zwracająca wynik typu int. W takiej sytuacji interfejs I2 nie może dziedziczyć po I1, gdyż występuje konflikt nazw.

Listing 6.39. Nieprawidłowe dziedziczenie interfejsów

```
public interface I1  
{  
    void f();  
}  
  
public interface I2 : I1  
{  
    int f(); //Nieprawidłowo  
}
```

Ten błąd zostanie jednak odkryty dopiero przy próbie napisania kodu klasy dziedziczącej po I2 — sam kod interfejsów da się skompilować bez problemu⁸.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 32.1

Dopisz do interfejsu PierwszyInterfejs z listingu 6.27 bezargumentową metodę g o typie zwracanym void, a do interfejsu DrugiInterfejs bezargumentową metodę f o takim samym typie zwracanym. Jak powinna w takiej sytuacji wyglądać treść klasy MojaKlasa (czy wymaga dokonywania zmian)?

Ćwiczenie 32.2

Zmień kod interfejsów z listingu 6.29 tak, aby metoda f interfejsu PierwszyInterfejs przyjmowała argument typu int, a metoda f interfejsu DrugiInterfejs argument typu double. Jakich modyfikacji wymagać będzie klasa MojaKlasa z listingu 6.30, aby mogła współpracować z tak zmienionymi interfejsami?

⁸ Pojawi się jedynie ostrzeżenie o nieuzyciu słowa new.

Ćwiczenie 32.3

Zmodyfikuj kod z listingu 6.28 w taki sposób, aby klasa `Telewizor` implementowała tylko jeden interfejs o nazwie `IWyswietlaObrazIWydajeDzwiek`.

Ćwiczenie 32.4

Wykorzystując mechanizm dziedziczenia, połącz interfejsy `IWyswietlaObraz` i `IWydajeDzwiek` z listingu 6.28 tak, aby powstał jeden interfejs o nazwie `IWyswietlaObrazIWydajeDzwiek`.

Ćwiczenie 32.5

Zmień kod z listingów 6.31 i 6.32 tak, aby istnienie bezargumentowej metody `f` było wymuszane przez oba interfejsy.

Klasy zagnieżdżone

Lekcja 33. Klasa wewnętrz klasy

Przy omawianiu rodzajów klas w rozdziale 3. pojawiło się pojęcie klas wewnętrznych. Inaczej można je nazwać zagnieżdżonymi. Nie zajmowaliśmy się wtedy tym tematem, teraz nadszedł czas na przybliżenie tego rodzaju klas. Lekcja 32. jest poświęcona właśnie wprowadzeniu w tę nową tematykę. Zobaczmy, w jaki sposób tworzy się klasy zagnieżdżone oraz jakie mają one właściwości. Sprawdzimy, jak dostać się do ich składowych oraz jakie relacje zachodzą między klasą wewnętrzną a zewnętrzną.

Tworzenie klas zagnieżdżonych

Klasa zagnieżdżona (wewnętrzna), jak sama nazwa wskazuje, to klasa, która została zdefiniowana we wnętrzu innej. Konstrukcja taka początkowo może wydawać się nieco dziwna, w praktyce pozwala jednak na wygodne tworzenie różnych konstrukcji programistycznych. Schematyczna deklaracja klasy zagnieżdżonej wygląda następująco:

```
[specyfikator dostępu] class klasa_zewnętrzna
{
    [specyfikator dostępu] class klasa_zagnieżdżona
    {
        /*
         składowe klasy zagnieżdżonej
        */
    }
}
```

```
    składowe klasy zewnętrznej  
*/  
}
```

Jeśli więc chcemy utworzyć klasę o nazwie `Outside`, która będzie zawierała w sobie klasę `Inside`, możemy zastosować kod zaprezentowany na listingu 6.40.

Listing 6.40. Proste zagnieżdżanie klas

```
public class Outside  
{  
    class Inside  
    {  
    }  
}
```

W klasie zewnętrznej możemy bez problemów oraz bez żadnych dodatkowych zabiegów programistycznych korzystać z obiektów klasy zagnieżdżonej. Można je tworzyć, a także bezpośrednio odwoływać się do zdefiniowanych w nich pól i metod. Przykład odwołań do obiektu klasy zagnieżdżonej jest widoczny na listingu 6.41.

Listing 6.41. Odwołania do obiektów klasy zagnieżdżonej

```
using System;  
  
public class Outside  
{  
    class Inside  
    {  
        public int liczba = 100;  
        public void f()  
        {  
            Console.WriteLine("Inside:f liczba = " + liczba);  
        }  
    }  
    public void g()  
    {  
        Inside ins = new Inside();  
        ins.f();  
        ins.liczba = 200;  
        ins.f();  
    }  
    public static void Main()  
    {  
        new Outside().g();  
    }  
}
```

Klasa zagnieżdżona o nazwie `Inside` zawiera jedno publiczne pole typu `int` o nazwie `liczba`, któremu już podczas deklaracji jest przypisywana wartość 100, oraz jedną publiczną metodę o nazwie `f`, której zadaniem jest wyświetlenie wartości zapisanej w polu `liczba`. Klasa zewnętrzna `Outside` zawiera publiczną metodę o nazwie `g`, w której jest tworzony nowy obiekt klasy `Inside`. Następnie wywoływana jest metoda `f` tego

obiektu, a dalej polu `liczba` jest przypisywana wartość 200 i ponownie jest wywoływana metoda `f`. Oprócz metody `g` w klasie `Outside` znajduje się metoda `Main`, od której zaczyna się wykonywanie kodu programu. W tej metodzie jest tworzony nowy obiekt klasy `Outside` i wywoływana jest jego metoda `g`. Odbywa się to w jednej linii programu. Niestosowana do tej pory konstrukcja:

```
new Outside().g();
```

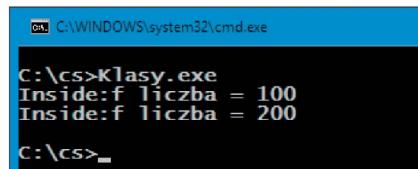
oznacza: „Utwórz obiekt klasy `Outside` (referencja do obiektu nie jest zapamiętywana w programie), a następnie wywołaj jego metodę o nazwie `g`”. Efekt działania tego fragmentu kodu jest taki sam jak rezultat wykonania dwóch instrukcji:

```
Outside out = new Outside();
out.g();
```

Ostatecznie w wyniku działania aplikacji zobaczymy na ekranie widok przedstawiony na rysunku 6.22. Widać więc wyraźnie, że w klasie zewnętrznej można bez problemów odwoływać się do składowych klasy zagnieżdzonej (wewnętrznej).

Rysunek 6.22.

*Odwolania do pól
i metod klasy
zagnieżdzonej*



Kilka klas zagnieżdżonych

W jednej klasie zewnętrznej może istnieć dowolna liczba klas zagnieżdżonych, nie ma pod tym względem ograniczeń. Dostęp do nich odbywa się w taki sam sposób, jak przedstawiono w poprzedniej części tej lekcji. Przykładowa klasa `Outside`, w której zostały zdefiniowane dwie klasy wewnętrzne, została przedstawiona na listingu 6.42.

Listing 6.42. Użycie dwóch klas zagnieżdżonych

```
using System;

public class Outside
{
    SecondInside secondIns = new SecondInside();
    class FirstInside
    {
        public int liczba = 100;
        public void f()
        {
            Console.WriteLine("FirstInside:f liczba = " + liczba);
        }
    }
    class SecondInside
    {
        public double liczba = 1.0;
        public void f()
        {
```

```
        Console.WriteLine("SecondInside:f liczba = " + liczba);
    }
}
public void g()
{
    FirstInside firstIns = new FirstInside();
    firstIns.f();
    firstIns.liczba = 200;
    firstIns.f();
    secondIns.f();
    secondIns.liczba = 2.5;
    secondIns.f();
}
public static void Main()
{
    new Outside().g();
}
```

Mamy tu do czynienia z trzema klasami — zewnętrzną `Outside` oraz dwoma wewnętrznymi: `FirstInside` i `SecondInside`. W klasie `FirstInside` znajduje się jedno pole typu `int` o nazwie `liczba` oraz jedna bezargumentowa metoda o nazwie `f`. Zadaniem tej metody jest wyświetlanie wartości pola `liczba`. Klasa `SecondInside` jest zbudowana w sposób analogiczny do `FirstInside`; zawiera jedno pole o nazwie `liczba` typu `double` oraz jedną bezargumentową metodę o nazwie `f`, której zadaniem jest wyświetlanie wartości tego pola.

Klasa zewnętrzna, `Outside`, zawiera pole typu `SecondInside` o nazwie `secondIns`, do którego już w trakcie deklaracji jest przypisywana referencja do nowego obiektu klasy `SecondInside`. Do dyspozycji mamy również dwie metody: `g` oraz `Main`. W `g` tworzymy zmienną typu `FirstInside` o nazwie `firstIns` i przypisujemy jej referencję do nowego obiektu klasy `FirstInside`, następnie wywołujemy metodę `f` tego obiektu, przypisujemy polu `liczba` wartość 200 oraz ponownie wywołujemy metodę `f`. W kolejnym kroku wywołujemy metodę `f` obiektu wskazywanego przez `secondIns`, przypisujemy polu `liczba` tego obiektu wartość 2.5 oraz ponownie wywołujemy jego metodę `f`. Nie musimy tworzyć obiektu `secondIns`, gdyż czynność ta została wykonana w trakcie deklaracji pola `secondIns`. Ostatecznie po skompilowaniu i uruchomieniu kodu na ekranie zobaczymy widok zaprezentowany na rysunku 6.23.

Rysunek 6.23.

Wykorzystanie
dwóch klas
zagnieżdżonych

```
C:\> C:\WINDOWS\system32\cmd.exe
C:\> Klasy.exe
FirstInside:f liczba = 100
FirstInside:f liczba = 200
SecondInside:f liczba = 1
SecondInside:f liczba = 2,5
C:\>
```

Składowe klas zagnieżdżonych

W dotychczasowych przykładach składowe klas zagnieżdżonych były oznaczone specyfikatorem dostępu `public`. Nie jest to jednak obligatoryjne; do klas zagnieżdżonych mają zastosowanie takie same zasady ustalania dostępu do składowych jak w przypadku zwykłych klas. Dostęp może być więc publiczny (`public`), wewnętrzny (`internal`), prywatny (`private`), chroniony (`protected`) bądź wewnętrzny chroniony (`protected internal`). Założymy, że mamy klasę wewnętrzną `Inside` podobną do przedstawionej na listingu 6.41, w której jednak pole `liczba` jest polem prywatnym, tak jak jest to widoczne na listingu 6.43.

Listing 6.43. Dostęp do składowych klasy zagnieżdżonej

```
using System;

public class Outside
{
    class Inside
    {
        private int liczba = 100;
        public void f()
        {
            Console.WriteLine("Inside:f liczba = " + liczba);
        }
    }
    public void g()
    {
        Inside ins = new Inside();
        ins.f();
        ins.liczba = 200; //Uwaga!
        ins.f();
    }
    public static void Main()
    {
        new Outside().g();
    }
}
```

Taki program nie będzie mógł być poprawnie skompilowany. Klasa zagnieżdżona `Inside` zawiera metodę `f` i pole `liczba`, ale tym razem pole to jest prywatne, a więc nie ma do niego dostępu spoza tej klasy. Nic więc nie stoi na przeszkodzie, aby w metodzie `g` klasy `Outside` wykonać instrukcję:

```
ins.f();
```

ale zaznaczona komentarzem instrukcja:

```
ins.liczba = 200;
```

jest już nieprawidłowa. Dlatego podczas próby komplikacji zostanie zgłoszony błąd widoczny na rysunku 6.24. Widać więc, że reguły dostępu do składowych klas zagnieżdżonych są respektowane. Na ten fakt powinny zwrócić uwagę osoby programujące np. w Javie, gdyż w tym języku ta kwestia została rozwiązana odmiennie.

Rysunek 6.24.

Nie można odwołać się do pola prywatnego klasy zagnieżdzonej

```
C:\WINDOWS\system32\cmd.exe
C:\>csc Klasy.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

Klasy.cs(17,9): error CS0122: 'Outside.Inside.liczba' is invalid
due to its protection level

C:\>_
```

Obiekty klas zagnieżdzonych

Wiemy, że obiekty klas zagnieżdzonych można tworzyć w klasach zewnętrznych. Co jednak zrobić, kiedy chcemy operować na obiekcie klasy zagnieżdzionej poza klasą zewnętrzną? Czyli co zrobić na przykład w sytuacji, kiedy klasą zewnętrzną jest `Outside`, wewnętrzną `Inside`, a my chcemy mieć dostęp do obiektów klasy `Inside` z zupełnie niezależnej klasy `Program`?

Możliwe są dwa rozwiązania tego problemu. Otóż w klasie zewnętrznej można umieścić metodę tworzącą i zwracającą obiekty klasy zagnieżdzionej lub też za pomocą odpowiedniej składni bezpośrednio powołać do życia obiekt klasy zagnieżdzonej. Zaczniemy od sposobu pierwszego. Na listingu 6.44 są widoczne przykładowe klasy `Inside` i `Outside`.

Listing 6.44. Klasa zagnieżdzona

```
using System;

public class Outside
{
    public class Inside
    {
        public void g()
        {
            Console.WriteLine("Inside:g()");
        }
        public Inside getInside()
        {
            return new Inside();
        }
    }
}
```

Klasa zewnętrzna `Outside` zawiera metodę `getInside`, która zwraca obiekt klasy `Inside`. Obiekt ten jest tworzony wewnątrz tej metody i zwracany za pomocą standardowej instrukcji `return`. Klasa zagnieżdzona `Inside` zawiera jedną metodę o nazwie `g`, wyświetlającą napis informujący o klasie, z której pochodzi. Chcemy utworzyć teraz klasę `Program`, która skorzysta z obiektu klasy `Inside`. Można to zrobić w sposób przedstawiony na listingu 6.45.

Listing 6.45. Użycie obiektu klasy zagnieżdzonej

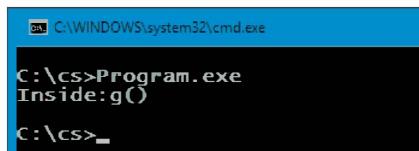
```
using System;

public class Program
{
    public static void Main()
    {
        Outside out1 = new Outside();
        out1.getInside().g();
    }
}
```

Tworzymy nowy obiekt klasy `Outside` i przypisujemy go zmiennej o nazwie `out1`. Następnie wywołujemy metodę `getInside` tego obiektu, która zwraca obiekt klasy `Inside`, oraz wywołujemy metodę `g` zwróconego obiektu. Po skompilowaniu i uruchomieniu klasy `Program` zobaczymy więc widok zaprezentowany na rysunku 6.25.

Rysunek 6.25.

Wywołanie metody
klasy zagnieżdzionej
z klasy Program



Przedstawiony sposób dostępu do obiektu klasy zagnieżdzonej jest jak najbardziej poprawny, ale ma jedną wadę. Otóż obiekt klasy `Inside` można wykorzystać tylko raz, do jednokrotnego wywołania metody `g`, gdyż nie przechowujemy referencji do niego. Tego typu wywołania są spotykane w praktyce, jeśli jednak chcemy zachować dostęp do obiektu zwróconego przez metodę `getInside`, musimy postąpić inaczej. Należy oczywiście zadeklarować zmienną, w której zostanie zapisana referencja.

Jakiego typu będzie ta zmienna? Odpowiedź, która nasuwa się w pierwszej chwili, brzmi: zmienna powinna być typu `Inside` (skoro tworzony jest obiekt klasy `Inside`). Jeśli jednak spróbujemy w klasie `Program` dokonać przykładowej deklaracji w postaci:

```
Inside ins;
```

nie osiągniemy zamierzonego celu. Kompilator nie zna osobnej klasy o nazwie `Inside` i zgłosi błąd. Klasa zagnieżdzona nie może bowiem istnieć samodzielnie, bez klasy zewnętrznej. Jest to również odzwierciedlone w deklaracji zmiennych klas zagnieżdzonych. Deklaracja taka powinna schematycznie wyglądać następująco:

```
klasa_zewnętrzna.klasa_zagnieżdzona nazwa_zmiennej;
```

A zatem prawidłowa deklaracja zmiennej klasy `Inside` w klasie niezależnej powinna mieć postać:

```
Outside.Inside ins;
```

Przykładowa klasa `Program` wykorzystująca taką deklarację jest widoczna na listingu 6.46.

Listing 6.46. Użycie obiektu klasy zagnieżdzonej w klasie niezależnej

```
public class Program
{
    public static void Main()
    {
        Outside out1 = new Outside();
        Outside.Inside ins1 = out1.getInside();
        ins1.g();
        ins1.g();
    }
}
```

Tworzymy nowy obiekt klasy `Outside` i przypisujemy go zmiennej `out1`. Następnie deklarujemy zmienną klasy `Inside`, wykorzystując poznaną przed chwilą konstrukcję. Zmienną tę inicjujemy, wywołując metodę `getInside` klasy `Outside`, zwracającą obiekt klasy `Inside`. W ten sposób zachowujemy referencję do obiektu, którą możemy już dowolnie wykorzystywać. W tym przypadku dwukrotnie wywoływana jest metoda `g`.

Pozostał jeszcze do omówienia drugi sposób tworzenia obiektów klas zagnieżdzonych, to znaczy bezpośrednie powoływanie ich do życia w klasach niezależnych. Co zatem zrobić w sytuacji, kiedy klasa zewnętrzna nie udostępnia żadnej metody zwracającej nowy obiekt klasy zagnieżdzionej, tak jak zostało to przedstawione na listingu 6.47?

Listing 6.47. Brak metody zwracającej nowy obiekt klasy zagnieżdzionej

```
using System;

public class Outside
{
    public class Inside
    {
        public void g()
        {
            Console.WriteLine("Inside:g()");
        }
    }
}
```

Otoż okazuje się, że w klasie niezależnej bez problemów można utworzyć obiekt klasy zagnieżdzionej. Zwrócić uwagę na sposób deklaracji zmiennej typu zagnieżdzionego, widocznego na listingu 6.46. Wynika z tego, że pełna nazwa typu zagnieżdzionego może być schematycznie przedstawiona jako:

klasa_zewnętrzna.klasa_zagnieżdzona

W analogiczny sposób wywołamy więc konstruktor takiej klasy, ogólnie:

new klasa_zewnętrzna.klasa_zagnieżdzona(argumenty_konstruktora)

Aby więc w omawianym przypadku utworzyć obiekt klasy zagnieżdzonej `Inside`, należy użyć konstrukcji o postaci:

new Outside.Inside();

tak jak zostało to pokazane na listingu 6.48. Warunkiem jest jednak, aby klasa wewnętrzna miała odpowiedni poziom dostępu. W tym przypadku został użyty dodatkowo modyfikator public (na listingu 6.47).

Listing 6.48. Tworzenie obiektu klasy zagnieżdzonej w klasie niezależnej

```
public class Program
{
    public static void Main()
    {
        Outside.Inside ins1 = new Outside.Inside();
        ins1.g();
    }
}
```

Rodzaje klas wewnętrznych

Dotychczas pojawiło się już kilka przykładów z klasami wewnętrznymi. We wszystkich przypadkach z wyjątkiem ostatniego przed definicją klasy nie znajdował się żaden modyfikator dostępu. To oznacza, że te klasy wewnętrzne (bez modyfikatora dostępu) były klasami prywatnymi. Dlatego właśnie w przykładzie z listingu 6.47 przy klasie Inside pojawiło się słowo public. Gdyby go nie było, program z listingu 6.48 nie dałby się skompilować ze względu na brak dostępu do klasy Inside.

Ogólnie rzecz ujmując, klasy zagnieżdżone można w pewnym sensie traktować jako składowe klas zewnętrznych, a w związku z tym można w stosunku do nich stosować również pozostałe modyfikatory dostępu. Klasy te mogą zatem być:

- ◆ publiczne,
- ◆ prywatne,
- ◆ chronione,
- ◆ wewnętrzne,
- ◆ wewnętrzne chronione.

Jeśli przed nazwą klasy zagnieżdżonej nie wystąpi żaden modyfikator dostępu, domyślnie będzie ona prywatna. Zobrazowano to w przykładzie widocznym na listingu 6.49.

Listing 6.49. Prywatne klasy zagnieżdżone

```
using System;

public class Outside
{
    private class Inside1
    {
        public void g()
        {
            Console.WriteLine("Inside1:g()");
        }
    }
}
```

```
    }
    class Inside2
    {
        public void g()
        {
            Console.WriteLine("Inside2:g()");
        }
    }
    public void f()
    {
        new Inside1().g();
        new Inside2().g();
    }
}

public class Program
{
    public static void Main()
    {
        //Outside.Inside1 ins1 = new Outside.Inside1();
        //Outside.Inside2 ins2 = new Outside.Inside2();
        Outside out1 = new Outside();
        out1.f();
    }
}
```

Tym razem w klasie `Outside` znalazły się dwie klasy zagnieżdżone: `Inside1` i `Inside2`. Pierwsza z nich jest jawnie prywatna, znajduje się bowiem przed nią modyfikator `private`. Druga klasa zagnieżdżona jest również prywatna. Wynika to z tego, że jeżeli przed definicją nie znajduje się żaden modyfikator dostępu, to kod jest traktowany tak, jakby obecny był modyfikator `private`. Zarówno `Inside1`, jak i `Inside2` zawierają publiczną metodę `g`, której zadaniem jest wyświetlenie nazwy tej metody i nazwy klasy, z której ona pochodzi.

W klasie `Program` w metodzie `Main` znajdują się trzy instrukcje tworzące obiekty. Pierwsza z nich próbuje utworzyć obiekt typu `Inside1`. To nie może się udać, jest to bowiem klasa jawnie prywatna. Podobnie druga instrukcja również jest nieprawidłowa, gdyż `Inside2` jest domyślnie prywatna. Trzecia instrukcja tworzy nowy obiekt klasy `Outside` i przypisuje go zmiennej `out1`. Następnie za pomocą tej zmiennej jest wywoływana metoda `f` tego obiektu. Te instrukcje są prawidłowe, ponieważ zarówno klasa `Outside`, jak i metoda `f` są publiczne.

Przyjrzymy się więc metodzie `f`. Najpierw jest w niej tworzony obiekt typu `Inside1` i jest wywoływana jego metoda `g`, a następnie obiekt typu `Inside2` i również jest wywoływana jego metoda `g`. Czy te instrukcje są prawidłowe? Tak. Klasa zewnętrzna (`Outside`) zawsze ma dostęp do klas zagnieżdżonych w sobie, są one niedostępne jedynie dla klas niezależnych, dokładnie tak samo jak inne składowe, takie jak pola i metody. Czy prawidłowe było wywołanie metod `g`? Również tak, ponieważ w obu klasach zagnieżdżonych (`Inside1` i `Inside2`) te metody są publiczne.

Dostęp do składowych klasy zewnętrznej

Ciekawą i wartą poruszenia kwestią jest to, że klasa zagnieżdzona ma pełny dostęp do składowych klasy zewnętrznej. I to niezależnie od ich poziomu dostępu. Oznacza to, że może operować nawet na składowych prywatnych! Zobaczmy to w praktyce. Odpowiedni przykład został zaprezentowany na listingu 6.50.

Listing 6.50. Modyfikacja wartości prywatnego pola klasy zewnętrznej

```
using System;

public class Outside
{
    private int wartosc;
    public class Inside
    {
        private Outside parent;
        public Inside(Outside obj)
        {
            parent = obj;
        }
        public void Ustaw(int val)
        {
            parent.wartosc = val;
        }
        public void Wyswietl()
        {
            Console.WriteLine("Pole wartość = {0}", wartosc);
        }
    }
}

public class Program
{
    public static void Main()
    {
        Outside out1 = new Outside();
        out1.Wyswietl();
        Outside.Inside ins1 = new Outside.Inside(out1);
        ins1.Ustaw(100);
        out1.Wyswietl();
    }
}
```

Przeanalizujmy ten program dokładniej. Mamy tu do czynienia z klasą zewnętrzną `Outside`, zawierającą prywatne pole typu `int` o nazwie `wartosc` oraz publiczną metodę `Wyswietl`, której zadaniem jest wyświetlenie wartości pola. W klasie nie ma jednak żadnej metody pozwalającej na zapis pola `wartosc`, a więc może się wydawać, że jego wartość nie może być w żaden sposób zmieniana. Byłoby tak faktycznie, gdyby nie istnienie zagnieżdzonej klasy `Inside`. Ma ona nieco inną budowę niż w poprzednich przykładach. Otóż zawiera prywatne pole typu `Outside` o nazwie `parent` oraz konstruktor przyjmujący jeden argument pozwalający na ustawienie wartości tego pola. Tak więc przy tworzeniu obiektów klasy `Inside` niezbędne będzie zawsze użycie obiektu klasy `Outside`.

Oprócz wymienionych składowych klasa `Inside` zawiera również metodę `Ustaw` przyjmującą jeden argument typu `int`. W treści tej metody wartość przekazanego jej argumentu jest przypisywana polu `wartosc` obiektu klasy `Outside` wskazywanego przez pole `parent`:

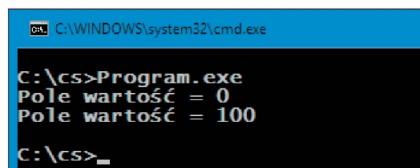
```
parent.wartosc = val;
```

W tym miejscu należy zwrócić uwagę, iż jest to możliwe tylko dlatego, że `Inside` jest zagnieżdżona w `Outside` i ma dostęp do jej składowych, nawet tych prywatnych. W innej sytuacji tego typu odwołanie skutkowałoby błędem komilacji.

W klasie `Program` w metodzie `Main` są używane obiekty typu `Outside` i `Inside`. Najpierw zmiennej `out1` jest przypisywany nowy obiekt klasy `Outside` oraz jest wywoływana jego metoda `Wyswietl`. Ponieważ pole `wartosc` nie zostało w żaden sposób zainicjalizowane, będzie ono miało wartość 0 i taka też wartość pojawi się na ekranie. Kolejna instrukcja to utworzenie nowego obiektu klasy `Inside` i przypisanie go zmiennej `ins1` — konstruktorowi został przekazany obiekt wskazywany przez `out1`. Następnie została wywołana metoda `Ustaw` obiektu `ins1`, tym samym prywatne pole `wartosc` obiektu `out1` zostało ustawione na 100, o czym przekonujemy się, wywołując ponownie metodę `Wyswietl` tego obiektu. Ostatecznie po uruchomieniu programu zobaczymy widok zaprezentowany na rysunku 6.26.

Rysunek 6.26.

Efekt modyfikacji prywatnego pola klasy zewnętrznej



Ćwiczenia do samodzielnego wykonania

Ćwiczenie 33.1

Napisz klasę o nazwie `Zewnętrzna` zawierającą wewnętrzną klasę o nazwie `Wewnętrzna`, o dostępie publicznym. W klasie `Zewnętrzna` umieść metodę zwracającą nowy obiekt klasy `Wewnętrzna`.

Ćwiczenie 33.2

Napisz klasę o nazwie `Zewnętrzna` zawierającą wewnętrzną klasę o nazwie `Wewnętrzna`, o dostępie prywatnym. Sprawdź, czy można napisać metody pozwalające na posługiwaniu się obiektami klasy `Wewnętrzna` w klasie niezależnej.

Ćwiczenie 33.3

Zmień kod z listingu 6.45 tak, aby zachować sposób działania programu (wywołanie metody `g` klasy `Inside`), ale by nie było konieczności używania zmiennej pomocniczej `out1`.

Ćwiczenie 33.4

Do klasy `Inside` z listingu 6.50 dopisz pusty konstruktor bezargumentowy. Spróbuj użyć go zamiast jednoargumentowego przy tworzeniu obiektu klasy `Inside`. Jakie będą tego konsekwencje przy próbie uruchomienia programu?

Typy uogólnione

Lekcja 34. Kontrola typów i typy uogólnione

Typy uogólnione pojawiły się w C# już w wersji 2.0. Mówimy o nich także jako o typach ogólnych, generycznych lub (potocznie) o generykach (ang. *generics*, *generic types*). Wszystkie te określenia funkcjonują zarówno w literaturze, jak i mowie potocznej i wszystkie mają swoich zwolenników i przeciwników. W książce będzie stosowany termin **typy uogólnione**. W skrócie można powiedzieć, że pozwalają one na konstruowanie kodu, który operuje nie na konkretnych typach danych (konkretnych klasach czy interfejsach), ale na typach nieokreślonych, ogólnych. Pełne omówienie tego tematu wykracza niestety poza ramy niniejszej publikacji. Ponieważ jednak każdy programista musi znać przynajmniej podstawy tego zagadnienia, lekcja 34. poświęcona została niezbędnym podstawom.

Jak zbudować kontener?

W lekcji 12. zostały przedstawione tablice, czyli struktury przechowujące dane różnych typów. Jednym z ich głównych ograniczeń była konieczność jawniej deklaracji wielkości. Zwykła tablica może przechowywać tylko tyle elementów, ile zostało określonych podczas jej tworzenia. Jednak w realnym programowaniu często nie da się określić z góry, ile liczb czy obiektów będzie faktycznie potrzebnych, stąd konieczne są struktury danych, które pozwalają na dynamiczne zwiększenie swojej wielkości, a tym samym możliwości przechowywania danych. Jak poradzić sobie z takim problemem? Można skorzystać z gotowego rozwiązania (w C# dostępne są różne tzw. klasy kontenerowe) lub napisać własną klasę symulującą zachowanie dynamicznej tablicy. Spróbujmy wykonać to zadanie. Klasę nazwiemy `TablicaInt` — będzie ona umożliwiała przechowywanie dowolnej wartości typu `int`. Dostęp do danych będzie realizowany za pomocą metod `get` oraz `set`. Trzeba też ustalić, w jaki sposób wewnątrz klasy przechowywać dane. Użyjemy do tego zwykłą tablicy typu `int`. Przykładowy kod został zaprezentowany na listingu 6.51.

Listing 6.51. Klasa realizująca funkcję dynamicznej tablicy

```
using System;

public class TablicaInt
{
    private int[] tab;
    public TablicaInt(int size)
    {
        if(size < 0)
        {
            throw new ArgumentOutOfRangeException("size < 0");
        }
        tab = new int[size];
    }
    public int Get(int index)
    {
        if(index >= tab.Length || index < 0)
        {
            throw new IndexOutOfRangeException("index = " + index);
        }
        else
        {
            return tab[index];
        }
    }
    public void Set(int index, int value){
        if(index < 0)
        {
            throw new IndexOutOfRangeException("index = " + index);
        }
        if(index >= tab.Length)
        {
            Resize(index + 1);
        }
        tab[index] = value;
    }
    protected void Resize(int size)
    {
        int[] newTab = new int[size];
        for(int i = 0; i < tab.Length; i++)
        {
            newTab[i] = tab[i];
        }
        tab = newTab;
    }
    public int Length
    {
        get
        {
            return tab.Length;
        }
    }
}
```

Klasa zawiera jedno prywatne pole `tab`, któremu w konstruktorze jest przypisywana nowo utworzona tablica liczb typu `int`. Rozmiar tej tablicy określa argument konstruktora. W przypadku stwierdzenia, że przekazana wartość jest mniejsza od zera, generowany jest systemowy wyjątek `ArgumentOutOfRangeException` (nie można bowiem tworzyć tablic o ujemnej liczbie elementów). Wymieniony wyjątek jest zdefiniowany w przestrzeni nazw `System`, można się więc do niego bezpośrednio odwoływać bez dodatkowych dyrektyw `using`.

Do pobierania danych służy metoda `Get`, która przyjmuje jeden argument — `index` — określający indeks wartości, jaka ma zostać zwrócona. Indeks pobieranego elementu nie może być w tym przypadku większy niż całkowity rozmiar wewnętrznej tablicy pomniejszony o 1 (jak pamiętamy, elementy tablicy są indeksowane od 0) ani też mniejszy od 0. Jeśli zatem indeks znajduje się poza zakresem, generowany jest znany nam dobrze z wcześniejszych lekcji wyjątek `IndexOutOfRangeException`:

```
throw new IndexOutOfRangeException("index = " + index);
```

Jeśli natomiast argument przekazany metodzie jest poprawny, zwrócona zostaje wartość odczytana spod wskazanego indeksu tablicy:

```
return tab[index];
```

Metoda `Set` przyjmuje dwa argumenty. Pierwszy — `index` — określa indeks elementu, który ma zostać zapisany, drugi — `value` — wartość, która ma się znaleźć pod tym indeksem. W tym przypadku na początku sprawdzamy, czy argument `index` jest mniejszy od 0, a jeśli tak, zgłaszamy wyjątek `IndexOutOfRangeException`. Jest to jasne działanie, nie można bowiem zapisywać ujemnych indeksów (choćż ciekawym rozwiązaniem byłoby wprowadzenie takiej możliwości; to dobre ćwiczenie do samodzielnego wykonania). Inaczej będzie jednak w przypadku, gdy zostanie ustalone, że indeks przekracza aktualny rozmiar tablicy. Skoro tablica ma dynamicznie zwiększać swoją wielkość w zależności od potrzeb, taka sytuacja jest w pełni poprawna. Zwiększamy więc wtedy tablicę, wywołując metodę `Resize`. Na końcu metody `Set` przypisujemy wartość wskazaną przez `value` komórce określonej przez `index`:

```
tab[index] = value;
```

Pozostało więc przyjrzeć się metodzie `Resize`. Nie wykonuje ona żadnych skomplikowanych czynności. Skoro nie można zwiększyć rozmiaru już raz utworzonej tablicy, trzeba utworzyć nową o rozmiarze wskazanym przez argument `size`:

```
int[] newTab = new int[size];
```

Po wykonaniu tej czynności niezbędne jest oczywiście przeniesienie zawartości starej tablicy do nowej⁹, co odbywa się w pętli `for`, a następnie przypisanie nowej tablicy polu `tab`:

```
tab = newTab;
```

⁹ Zwróćmy uwagę, że w przedstawionej realizacji tablica powiększana jest tylko do rozmiaru wynikającego ze zwiększenia największego indeksu o 1. W przypadku klasycznego wstawiania dużej liczby danych przy małym początkowym rozmiarze tablicy odbije się to niekorzystnie na wydajności. Warto się zastanowić, jak zapobiec temu niekorzystnemu zjawisku (patrz też sekcja „Ćwiczenia do samodzielnego wykonania”).

W kodzie klasy znajduje się również właściwość Length, która zawiera aktualny rozmiar tablicy, a dokładniej rzecz ujmując — zwraca aktualną wartość właściwości Length tablicy tab.

O tym, że nowa klasa działa prawidłowo, można się przekonać, używając obiektu typu TablicaInt w przykładowym programie. Przykładowy kod takiej aplikacji został przedstawiony na listingu 6.52, a efekt jego działania — na rysunku 6.27.

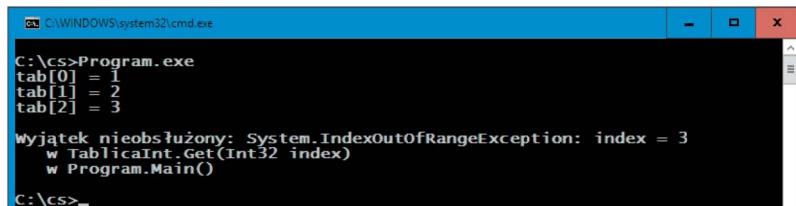
Listing 6.52. Testowanie działania klasy TablicaInt

```
using System;

public class Program
{
    public static void Main()
    {
        TablicaInt tab = new TablicaInt(2);
        tab.Set(0, 1);
        tab.Set(1, 2);
        tab.Set(2, 3);
        for(int i = 0; i < tab.Length; i++)
        {
            Console.WriteLine("tab[" + i + "] = " + tab.Get(i) + " ");
        }
        tab.Get(3);
    }
}
```

Rysunek 6.27.

Efekt działania kodu testującego nowy typ tablicy



Na początku tworzony jest nowy obiekt klasy TablicaInt o rozmiarze dwóch elementów, a następnie trzykrotnie wywoływana jest metoda Set. Pierwsze dwa wywołania ustaważą indeksy 0 i 1, zatem operują na istniejących od początku komórkach. Trzecie wywołanie powoduje jednak ustawnienie elementu o indeksie 2. Pierwotnie takiej komórki nie było, więc przy klasycznej tablicy należałoby się spodziewać wyjątku IndexOutOfRangeException. Ponieważ jednak metoda Set może dynamicznie zwiększać rozmiar tablicy (wywołując metodę Resize), również i trzecia instrukcja Set jest wykonywana poprawnie.

Zawartość obiektu tab jest następnie wyświetlana na ekranie za pomocą pętli for. Ostatnia instrukcja programu to próba pobrania za pomocą metody Get elementu o indeksie 3. Ponieważ jednak taki element nie istnieje (metoda Get nigdy nie zmienia rozmiaru tablicy), instrukcja ta powoduje wygenerowanie wyjątku. Całość działa zatem zgodnie z założeniami. Klasa TablicaInt ma jednak jedną wadę — może przechowywać tylko dane typu int. W kolejnym punkcie lekcji zostanie więc pokazane, jak można ją usprawnić.

Przechowywanie dowolnych danych

Klasa `TablicaInt` z listingu 6.51 mogła przechowywać tylko wartości typu `int`, czyli liczby całkowite. Co zatem zrobić, gdy trzeba zapisywać wartości innych typów? Można np. napisać kolejną klasę. Trudno jednak przygotowywać osobne klasy realizujące funkcje dynamicznej tablicy dla każdego możliwego typu danych. Byłoby to bardzo uciążliwe. Jest jednak proste rozwiążanie tego problemu. Jak wiadomo, każdy typ danych w rzeczywistości jest obiektowy i dziedziczy bezpośrednio lub pośrednio po klasie `Object`. W pierwszych lekcjach tego rozdziału znalazło się też wiele informacji o dziedziczeniu, rzutowaniu typów i polimorfizmie. Można się zatem domyślić, że wystarczy, aby ogólna klasa `Tablica` przechowywała referencje do typu `Object` — wtedy będą mogły być w niej zapisywane obiekty dowolnych typów. Spójrzmy na listing 6.53.

Listing 6.53. Dynamiczna tablica dla dowolnego typu danych

```
using System;

public class Tablica
{
    private Object[] tab;
    public Tablica(int size)
    {
        if(size < 0)
        {
            throw new ArgumentOutOfRangeException("size < 0");
        }
        tab = new Object[size];
    }
    public Object Get(int index)
    {
        if(index >= tab.Length || index < 0)
        {
            throw new IndexOutOfRangeException("index = " + index);
        }
        else
        {
            return tab[index];
        }
    }
    public void Set(int index, Object value){
        if(index < 0)
        {
            throw new IndexOutOfRangeException("index = " + index);
        }
        if(index >= tab.Length)
        {
            Resize(index + 1);
        }
        tab[index] = value;
    }
    protected void Resize(int size)
    {
        Object[] newTab = new Object[size];
        for(int i = 0; i < tab.Length; i++)
        {
```

```
        newTab[i] = tab[i];
    }
    tab = newTab;
}
public int Length
{
    get
    {
        return tab.Length;
    }
}
```

Struktura tego kodu jest bardzo podobna do przedstawionej na listingu 6.51, bo też bardzo podobna jest zasada działania. Metody Get, Set i Resize oraz właściwość Length wykonują analogiczne czynności, zmienił się natomiast typ przechowywanych danych, którym teraz jest Object. Tak więc metoda Get, pobierająca dane, przyjmuje wartość typu int, określającą indeks żądanego elementu, i zwraca wartość typu Object, a metoda Set przyjmuje dwa argumenty — pierwszy typu int, określający indeks komórki do zmiany, i drugi typu Object, określający nową wartość komórki. Taka klasa będzie pracowała ze wszystkimi typami danych, nawet typami prostymi. Aby się o tym przekonać, wystarczy uruchomić program z listingu 6.54.

Listing 6.54. Użycie nowej wersji klasy Tablica

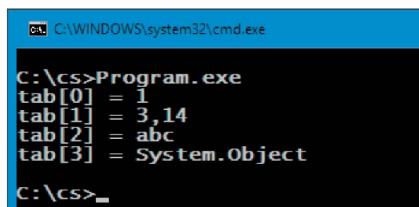
```
using System;

public class Program
{
    public static void Main()
    {
        Tablica tab = new Tablica(2);
        tab.Set(0, 1);
        tab.Set(1, 3.14);
        tab.Set(2, "abc");
        tab.Set(3, new Object());
        for(int i = 0; i < tab.Length; i++)
        {
            Console.WriteLine("tab[" + i + "] = " + tab.Get(i) + " ");
        }
    }
}
```

Pierwsze wywołanie tab.Set powoduje zapisanie wartości całkowitej 1 w pierwszej komórce (o indeksie 0). Wywołanie drugie zapisuje w kolejnej komórce (o indeksie 1) wartości rzeczywistej 3.14. Kolejna instrukcja to zapisanie pod indeksem 2 ciągu znaków abc. Jest to możliwe, mimo że metoda Set oczekuje wartości typu Object (nie jest to jednak problemem, gdyż jak wiadomo, wszystkie użyte typy danych dziedziczą po klasie Object). Ostatnie wywołanie Set powoduje zapis nowo utworzonego obiektu typu Object. Dalsze instrukcje działają tak samo jak w poprzednim przykładzie — w pętli typu for odczytywana jest zawartość tablicy. Po komplikacji i uruchomieniu

programu zobaczymy więc widok podobny do przedstawionego na rysunku 6.28. To dowód na to, że obiekt klasy `Tablica` może nie tylko dynamicznie zwiększać swoją pojemność, ale też przechowywać dane różnych typów.

Rysunek 6.28.
Obiekt klasy `tablica`
przechowuje dane
różnych typów



```
C:\cs>Program.exe
tab[0] = 1
tab[1] = 3,14
tab[2] = abc
tab[3] = System.Object
```

Problem kontroli typów

W poprzednim punkcie lekcji powstała uniwersalna klasa `Tablica` pozwalająca na przechowywanie danych dowolnych typów. Wydaje się, że to bardzo dobre rozwiązanie. Można powiedzieć: pełna wygoda, i faktycznie, w taki właśnie sposób często rozwiązywano problem przechowywania danych różnych typów przed nastaniem ery typów uogólnionych. Niestety, ta uniwersalność i wygoda niosą też ze sobą pewne zagrożenia. Aby to sprawdzić, spróbujmy przeanalizować program przedstawiony na listingu 6.55. Korzysta on z klasy `Tablica` z listingu 6.53 do przechowywania obiektów typów `Triangle` i `Rectangle`.

Listing 6.55. Problem niedostatecznej kontroli typów

```
using System;

class Triangle {}
class Rectangle
{
    public void Diagonal(){}
}

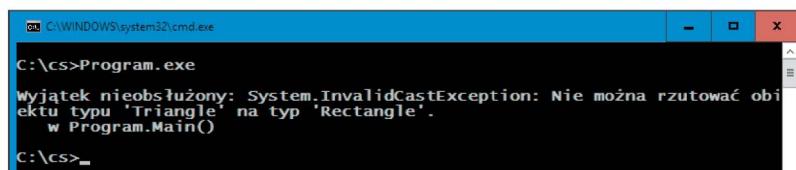
public class Program
{
    public static void Main()
    {
        Tablica rectangles = new Tablica(3);
        rectangles.Set(0, new Rectangle());
        rectangles.Set(1, new Rectangle());
        rectangles.Set(2, new Triangle());
        for(int i = 0; i < rectangles.Length; i++)
        {
            ((Rectangle) rectangles.Get(i)).Diagonal();
        }
    }
}
```

W metodzie `Main` utworzony został obiekt `rectangles` typu `Tablica`, którego zadaniem, jak można się domyślić na podstawie samej nazwy, jest przechowywanie obiektów klasy `Rectangle` (prostokąt). Za pomocą metody `Set` dodano do niego trzy elementy.

Następnie w pętli `for` została wywołana metoda `Diagonal` (przekątna) każdego z pobranych obiektów. Ponieważ metoda `Get` zwraca obiekt typu `Object`, przed wywołaniem metody `Get` niezbędne było dokonanie rzutowania na typ `Rectangle`.

Wszystko działałoby oczywiście prawidłowo, gdyby nie to, że trzecia instrukcja `Set` spowodowała umieszczenie w obiekcie `rectangles` obiekt typu `Triangle`, który nie ma metody `Diagonal` (trójkąt, ang. *triangle*, nie ma bowiem przekątnej, ang. *diagonal*). Kompilator nie ma jednak żadnej możliwości wychwycenia takiego błędu — skoro klasa `Tablica` może przechowywać dowolne typy obiektowe, to typem drugiego argumentu metody `Set` jest `Object`. Zawsze więc będzie wykonywane rzutowanie na typ `Object` (czyli formalnie trzecia instrukcja `Set` jest traktowana jako `rectangles.Set(2, (Object) new Triangle())`). Błąd objawi się zatem dopiero w trakcie wykonywania programu — przy próbie rzutowania trzeciego elementu pobranego z kontenera `rectangles` na typ `Rectangle` zostanie zgłoszony wyjątek `InvalidOperationException`, tak jak jest to widoczne na rysunku 6.29.

Rysunek 6.29.
Umieszczenie
nieprawidłowego
obiektu w tablicy
spowodowało
wyjątek `InvalidOperationException`



Oczywiście winą leży tu po stronie programisty — skoro nieuważnie umieszcza w kontenerze obiekt niewłaściwego typu, to nie może mieć pretensji, że aplikacja nie działa. Rozwiązaniem mogłaby być np. kontrola typu przy dodawaniu i pobieraniu obiektów z dynamicznej tablicy. To wymagałoby jednak rozbudowania kodu o dodatkowe instrukcje weryfikujące umieszczane w każdym miejscu, w którym następuje odwołanie do elementów przechowywanych w obiektach typu `Tablica`. Najwygodniejsze byłoby jednak pozostawienie elastyczności rozwiązania polegającego na możliwości przechowywania każdego typu danych, a przy tym przerzucenie części zadań związanych z kontrolą typów na kompilator. Jest to możliwe dzięki typom uogólnionym.

Korzystanie z typów uogólnionych

Koncepcja użycia typów uogólnionych w klasie realizującej funkcję dynamicznej tablicy będzie następująca: tablica wciąż ma mieć możliwość przechowywania dowolnego typu danych, ale podczas korzystania z niej chcemy mieć możliwość zadedykowania, jaki typ zostanie użyty w konkretnym przypadku. Jak to zrobić? Po pierwsze, trzeba zaznaczyć, że klasa będzie korzystała z typów uogólnionych; po drugie, trzeba zastąpić konkretny typ danych (w tym przypadku typ `Object`) typem ogólnym. Tak więc za nazwą klasy w nawiasie kątowym (ostrym, ostrokatnym; budujemy go ze znaków mniejszości i większości) umieszczaamy określenie typu ogólnego, a następnie używamy tego określenia wewnątrz klasy, zastępując nim typ konkretny. Zapewne brzmi to nieco zawile, spójrzmy zatem od razu na listing 6.56.

Listing 6.56. Uogólniona klasa Tablica

```
using System;

public class Tablica<T>
{
    private T[] tab;
    public Tablica(int size)
    {
        if(size < 0)
        {
            throw new ArgumentOutOfRangeException("size < 0");
        }
        tab = new T[size];
    }
    public T Get(int index)
    {
        if(index >= tab.Length || index < 0)
        {
            throw new IndexOutOfRangeException("index = " + index);
        }
        else
        {
            return tab[index];
        }
    }
    public void Set(int index, T value){
        if(index < 0)
        {
            throw new IndexOutOfRangeException("index = " + index);
        }
        if(index >= tab.Length)
        {
            Resize(index + 1);
        }
        tab[index] = value;
    }
    protected void Resize(int size)
    {
        T[] newTab = new T[size];
        for(int i = 0; i < tab.Length; i++)
        {
            newTab[i] = tab[i];
        }
        tab = newTab;
    }
    public int Length
    {
        get
        {
            return tab.Length;
        }
    }
}
```

Za nazwą klasy w nawiasie kątowym pojawił się symbol T. To informacja, że klasa będzie korzystała z typów uogólnionych oraz że symbolem typu ogólnego jest T (ten symbol można zmienić na dowolny inny, niekoniecznie jednoliterowy, zwyczajowo jednak używa się pojedynczych liter, zaczynając od wielkiej litery T). Ten symbol został umieszczony w każdym miejscu kodu klasy, gdzie wcześniej występował konkretny typ przechowywanych danych (był to typ `Object`). Tak więc instrukcja:

```
private T[] tab;
```

oznacza, że w klasie znajduje się prywatne pole zawierające tablicę, której typ zostanie określony dopiero w kodzie aplikacji przy tworzeniu obiektu klasy `Tablica`. Deklaracja:

```
public T Get(int index)
```

oznacza metodę `Get` zwracającą typ danych, który zostanie dokładniej określony przy tworzeniu obiektu klasy `Tablica` itd. Każde wystąpienie T to deklaracja, że właściwy typ danych będzie określony później. Dzięki temu powstała uniwersalna klasa, która może przechowywać dowolne dane, ale która pozwala na kontrolę typów już w momencie komilacji. Sposób działania nowej klasy można sprawdzić za pomocą programu widocznego na listingu 6.57.

Listing 6.57. Testowanie klasy korzystającej z uogólniania typów

```
//tutaj klasy Triangle i Rectangle z listingu 6.55

public class Program
{
    public static void Main()
    {
        Tablica<Rectangle> rectangles = new Tablica<Rectangle>(0);
        rectangles.Set(0, new Rectangle());
        rectangles.Set(1, new Rectangle());
        //rectangles.Set(2, new Triangle());
        for(int i = 0; i < rectangles.Length; i++)
        {
            (rectangles.Get(i)).Diagonal();
        }
    }
}
```

Klasy `Triangle` i `Rectangle` mogą pozostać w takiej samej postaci jak na listingu 6.55. Ewentualnie można dodać do metody `Diagonal` instrukcję wyświetlaną dowolny komunikat na ekranie, tak aby widać było efekty jej działania. Nie to jest jednak istotą przykładu. W metodzie `Main` deklarowany jest obiekt `rectangles` typu `Tablica`. Ponieważ korzystamy z najnowszej wersji klasy używającej typów uogólnionych, już przy deklaracji konieczne było podanie, jakiego rodzaju obiekty będą przechowywane w tablicy. Nazwa typu docelowego jest umieszczana w nawiasie kątowym. Tej konstrukcji należy użyć zarówno przy deklaracji zmiennej, jak i przy tworzeniu nowego obiektu. Zatem zapis:

```
Tablica<Rectangle> rectangles
```

oznacza powstanie zmiennej `rectangles` typu `Tablica`, której będzie można przypisać obiekt typu `Tablica` zawierający obiekty typu `Rectangle`, a zapis:

```
new Tablica<Rectangle>(0)
```

oznacza powstanie obiektu typu `Tablica` korzystającego z danych typu `Rectangle`. W efekcie przy w metodach `Get` i `Set` będzie można korzystać wyłącznie z obiektów typów `Rectangle` (lub obiektów klas pochodnych od `Rectangle`, por. materiał z lekcji 28. i 29.). Warto też zauważyc, że w pętli `for` odczytującej dane z tablicy brak jest rzutowania — nie było to konieczne, gdyż wiadomo, że metoda `Get` zwróci obiekty typu `Rectangle`.

Nie da się też popełnić pomyłki z listingu 6.55, związanej z zapisaniem w tablicy obiektu typu `Triangle`. Instrukcja:

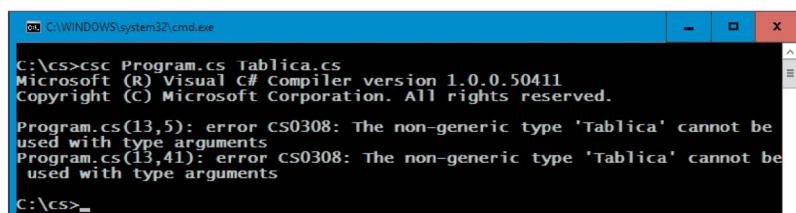
```
rectangles.Set(2, new Triangle());
```

została ujęta w komentarz, gdyż jest nieprawidłowa. Usunięcie komentarza spowoduje błąd komplikacji przedstawiony na rysunku 6.30. Widać więc wyraźnie, że teraz sam kompilator dba o kontrolę typów i nie dopuści do użycia niewłaściwego typu danych. Do przechowywania obiektów typu `Triangle` trzeba utworzyć osobną tablicę przystosowaną do pracy z tym typem danych, np.:

```
Tablica<Triangle> triangles = new Tablica<Triangle>(1);
```

Rysunek 6.30.

Próba
zastosowania
niewłaściwego typu
kończy się błędem
kompilacji



Ćwiczenia do samodzielnego wykonania

Ćwiczenie 34.1

Napisz program wypełniający obiekt klasy `TablicaInt` (w postaci z listingu 6.51) liczbami od 1 do 100, wykorzystujący do tego celu pętlę `for`. Początkowym rozmiarem tablicy ma być 1. Pętla ma mieć natomiast taką postać, aby nastąpiła co najwyżej jedna realokacja danych (jedno wywołanie metody `Resize`).

Ćwiczenie 34.2

Zmodyfikuj kod z listingu 6.51 tak, aby podczas wstawiania dużej liczby elementów przy małym początkowym rozmiarze tablicy nie występowało niekorzystne zjawisko bardzo częstej realokacji danych (częstego wywoływanie metody `Resize`).

Ćwiczenie 34.3

Popraw kod pętli `for` z listingu 6.55 tak, aby program uruchamiał się bez błędów. Możesz użyć operatora `is`, badającego, czy obiekt jest danego typu (np. `obj1 is string` da wartość `true`, jeśli obiekt `obj1` jest typu `string`), albo skorzystać z innego sposobu.

Ćwiczenie 34.4

Napisz program przechowujący przykładowe obiekty klas `Triangle`, `Rectangle` i `Circle`. Do przechowywania danych użij obiektów klasy `Tablica` w wersji z listingu 6.56.

Ćwiczenie 34.5

Napisz kod klasy, która będzie mogła przechowywać pojedynczą wartość dowolnego typu. Typ przechowywanej danej ma być ustalany przy tworzeniu obiektów tej klasy. Dostęp do przechowywanej wartości powinien być możliwy wyłącznie przez właściwość o dowolnej nazwie. Zawrzyj w kodzie metodę zwracającą tekstową reprezentację przechowywanego obiektu.

Ćwiczenie 34.6

Napisz przykładowy program ilustrujący użycie klasy z ćwiczenia 34.5 do przechowywania różnych typów danych.



Rozdział 7.

Aplikacje z interfejsem graficznym

Wszystkie prezentowane dotychczas programy pracowały w trybie tekstowym, a wyniki ich działania można było obserwować w oknie konsoli. To pozwalało na zapoznawanie się z wieloma podstawowymi konstrukcjami języka bez zaprzatania uwagi sposobem działania aplikacji pracujących w trybie graficznym. Jednak większość współczesnych programów oferuje graficzny interfejs użytkownika. Skoro więc przedstawiono już tak wiele cech języka C#, przyjrzyjmy się również sposobom tworzenia aplikacji okienkowych. Temu właśnie zagadniemu poświęcone są trzy kolejne lekcje. W lekcji 35. zajmiemy się podstawami tworzenia okien i menu, w 36. — ważnym tematem delegacji i zdarzeń, a w 37. — przykładami zastosowania takich komponentów, jak przyciski, etykiety, pola tekstowe i listy rozwijane.

Lekcja 35. Tworzenie okien

Lekcja 33. jest poświęcona podstawowym informacjom, których znajomość jest niezbędna do tworzenia aplikacji z interfejsem graficznym. Zostanie w niej pokazane, jak utworzyć okno aplikacji, nadać mu tytuł i ustalić jego rozmiary. Przedstawione będą właściwości i metody klasy `Form`, a także taki sposób kompilacji kodu źródłowego, aby w tle nie pojawiało się okno konsoli. Nie zabraknie również tematu dodawania do okna wielopoziomowego menu.

Pierwsze okno

Dotąd przedstawiano w książce programy konsolowe; najwyższy czas zobaczyć, jak tworzyć aplikacje z interfejsem graficznym. Podobnie jak w części pierwszej, kod będziemy pisać „ręcznie”, nie korzystając z pomocy narzędzi wizualnych, takich jak edytor form pakietu Visual Studio. Dzięki temu dobrze przeanalizujemy mechanizmy rządzące aplikacjami okienkowymi.

Podstawowy szablon kodu pozostanie taki sam jak w przypadku przykładów tworzonych w części pierwszej. Dodatkowo będzie trzeba poinformować kompilator o tym, że chcemy korzystać z klas zawartych w przestrzeni `System.Windows.Forms`. W związku z tym na początku kodu programów pojawi się dyrektywa `using` w postaci `using System.Windows.Forms`.

Do utworzenia podstawowego okna będzie potrzebna klasa `Form` z platformy .NET. Należy utworzyć jej instancję oraz przekazać ją jako argument w wywołaniu instrukcji `Application.Run()`. A zatem w metodzie `Main` powinna znaleźć się linia: `Application.Run(new Form());`. Tak więc kod tworzący najprostszą aplikację okienkową będzie miał postać taką jak na listingu 7.1.

Listing 7.1. *Utworzenie okna aplikacji*

```
using System.Windows.Forms;

public class Program
{
    public static void Main()
    {
        Application.Run(new Form());
    }
}
```

Jak widać, struktura programu jest taka sama jak w przypadku aplikacji konsolowej. Powstała klasa `Program` zawierająca publiczną i statyczną metodę `Main`, od której rozpocznie się wykonywanie kodu. W metodzie `Main` znalazła się instrukcja:

```
Application.Run(new Form());
```

czyli wywołanie metody `Run` z klasy `Application` (widać więc, że jest to również metoda statyczna; lekcja 19.), i został jej przekazany jako argument nowo utworzony obiekt typu `Form`. To właśnie sygnał do uruchomienia aplikacji okienkowej, a ponieważ argumentem jest obiekt typu `Form` — również do wyświetlenia okna na ekranie.

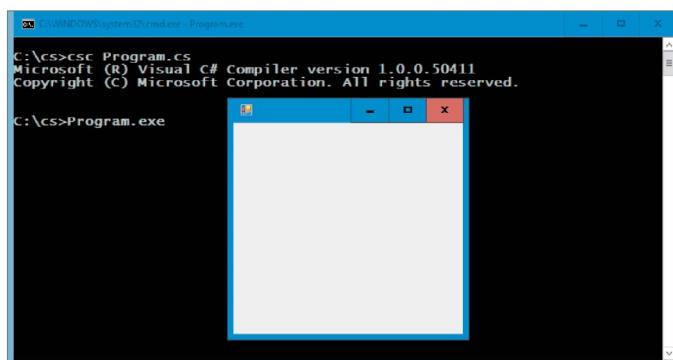
Jeśli teraz zapiszemy przedstawiony kod w pliku `Program.cs`, skompilujemy go za pomocą polecenia:

```
csc Program.cs
```

i uruchomimy, zobaczymy na ekranie widok taki jak na rysunku 7.1. Faktycznie mamy na ekranie typowe okno, które co prawda „nie robi” niczego pozytecznego, ale zauważmy, że mamy do dyspozycji działające przyciski służące do minimalizacji, maksymalizacji oraz zamknięcia, a także typowe menu systemowe. Osoby programujące w Javie powinny zwrócić też uwagę, że faktycznie taką aplikację można zamknąć, klikając odpowiedni przycisk.

Rysunek 7.1.

Prosta aplikacja
okienkowa



Nie będziemy jednak zapewne zadowoleni z jednej rzeczy. Otóż niezależnie od tego, czy tak skompilowany program uruchomimy z poziomu wiersza poleceń czy też klikając jego ikonę, zawsze w tle pojawić się będzie okno konsoli — widać to na rysunku 7.1. Powód takiego zachowania jest prosty. Domyslnie kompilator zakłada, że tworzymy aplikację konsolową. Dopiero ustawienie odpowiedniej opcji komplikacji zmieni ten stan rzeczy (tabela 1.1 z rozdziału 1.). Zamiast zatem pisać:

```
csc program.cs
```

należy skorzystać z polecenia:

```
csc /target:winexe program.cs
```

lub z formy skróconej:

```
csc /t:winexe program.cs
```

Po jego zastosowaniu powstanie plik typu *exe*, który będzie się uruchamiał tak jak zwykły program działający w środowisku graficznym.

Klasa Form

Jak można było przekonać się w poprzednim punkcie lekcji, okno aplikacji jest opisywane przez klasę *Form*. Poznajmy więc właściwości i metody przez nią udostępniane. Ponieważ jest ich bardzo dużo, skupimy się jedynie na wybranych z nich, które mogą być przydatne w początkowej fazie nauki C# i .NET. Zostały one zebrane w tabelach 7.1 i 7.2. Pozwalają m.in. na zmianę typu, wyglądu i zachowania okna.

Tabela 7.1. Wybrane właściwości klasy *Form*

Typ	Nazwa właściwości	Znaczenie
bool	AutoScaleMode	Ustala tryb automatycznego skalowania okna.
bool	AutoScroll	Określa, czy w oknie mają się automatycznie pojawiać paski przewijania.
bool	AutoSize	Określa, czy forma (okno) może automatycznie zmieniać rozmiary zgodnie z trybem określonym przez AutoSizeMode.

Tabela 7.1. Wybrane właściwości klasy Form — ciąg dalszy

Typ	Nazwa właściwości	Znaczenie
AutoSizeMode	AutoSizeMode	Określa tryb automatycznej zmiany rozmiarów formy.
Color	BackColor	Określa aktualny kolor tła.
Image	BackgroundImage	Określa obraz tła okna.
Bounds	Bounds	Określa rozmiar oraz położenie okna.
Size	ClientSize	Określa rozmiar obszaru roboczego okna.
ContextMenu	ContextMenu	Określa powiązane z oknem menu kontekstowe.
Cursor	Cursor	Określa rodzaj kurSORA wyświetlanego, kiedy wskaźnik myszy znajdzie się nad oknem.
Font	Font	Określa rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się w oknie.
Color	ForeColor	Określa kolor używany do rysowania obiektów w oknie (kolor pierwszoplanowy).
FormBorderStyle	FormBorderStyle	Ustala typ ramki okalającej okno.
int	Height	Określa wysokość okna.
Icon	Icon	Ustala ikonę przypisaną do okna.
int	Left	Określa w pikselach położenie lewego górnego rogu w poziomie.
Point	Location	Określa współrzędne lewego górnego rogu okna.
MainMenu	Menu	Menu główne przypisane do okna.
bool	Modal	Decyduje, czy okno ma być modalne.
string	Name	Określa nazwę okna.
Control	Parent	Referencja do obiektu nadrzędnego okna.
bool	ShowInTaskbar	Decyduje, czy okno ma być wyświetlane na pasku narzędziowym.
Size	Size	Określa wysokość i szerokość okna.
String	Text	Określa tytuł okna (tekst na pasku tytułu).
int	Top	Określa w pikselach położenie lewego górnego rogu w pionie.
bool	Visible	Określa, czy okno ma być widoczne.
int	Width	Określa w pikselach szerokość okna.
FormWindowState	WindowState	Reprezentuje bieżący stan okna.

Tabela 7.2. Wybrane metody klasy Form

Typ zwracany	Metoda	Opis
void	Activate	Aktywuje okno.
void	Close	Zamyka okno.
Graphics	CreateGraphics	Tworzy obiekt pozwalający na wykonywanie operacji graficznych.
void	Dispose	Zwalnia zasoby związane z oknem.

Tabela 7.2. Wybrane metody klasy Form — ciąg dalszy

Typ zwracany	Metoda	Opis
void	Hide	Ukrywa okno przed użytkownikiem.
void	Refresh	Wymusza odświeżenie okna.
void	ResetBackColor	Ustawia domyślny kolor tła.
void	ResetCursor	Ustawia domyślny kurSOR.
void	ResetFont	Ustawia domyślną czcionkę.
void	ResetForeColor	Ustawia domyślny kolor pierwszoplanowy.
void	ResetText	Ustawia domyślny tytuł okna.
void	Scale	Wykonuje przeskalowanie okna.
void	SetBounds	Ustala położenie i rozmiary okna.
void	Show	Wyświetla okno.
void	Update	Odrysowuje (aktualnia) unieważnione obszary okna.

W tabeli 7.1 znajdziemy między innymi właściwość Text. Pozwala ona na zmianę napisu na pasku tytułu okna. Sprawdźmy, jak to zrobić w praktyce. Trzeba będzie w nieco inny sposób utworzyć obiekt klasy Form, tak aby możliwe było zapamiętanie referencji do niego, a tym samym modyfikowanie właściwości. Odpowiedni przykład jest widoczny na listingu 7.2.

Listing 7.2. Okno zawierające określony Tytuł

```
using System.Windows.Forms;

public class Program
{
    public static void Main()
    {
        Form mojeOkno = new Form();
        mojeOkno.Text = "Tytuł okna";
        Application.Run(mojeOkno);
    }
}
```

Tym razem tworzymy zmienną typu mojeOkno i przypisujemy jej referencję do nowego obiektu typu Form. Dzięki temu możemy zmieniać jego właściwości. Modyfikujemy więc właściwość Text, przypisując jej ciąg znaków Tytuł okna — oczywiście można go zmienić na dowolny inny. Po dokonaniu tego przypisania przekazujemy zmienną mojeOkno jako argument metody Run, dzięki czemu aplikacja okienkowa rozpoczyna swoje działanie, a okno (forma, formatka) pojawia się na ekranie. Przyjmie ono postać widoczną na rysunku 7.2.

Rysunek 7.2.

Okno ze zdefiniowanym
w kodzie tytułem



Zauważmy jednak, że taki sposób modyfikacji zachowania okna sprawdzi się tylko w przypadku wyświetlania prostych okien dialogowych. Typowa aplikacja z reguły jest znacznie bardziej skomplikowana oraz zawiera wiele różnych zdefiniowanych przez nas właściwości. Najlepiej byłoby więc wprowadzić swoją własną klasę pochodną od `Form`. Tak też właśnie najczęściej się postępuje. Jak by to wyglądało w praktyce, zobrazowano w przykładzie widocznym na listingu 7.3, w którym powstanie okna o zadanych tytuł i rozmiarze.

Listing 7.3. Okno o zadanym tytule i rozmiarze

```
using System.Windows.Forms;

public class MainForm : Form
{
    public MainForm()
    {
        Text = "Tytuł okna";
        Width = 320;
        Height = 200;
    }
}

public class Program
{
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Powstała tu klasa `MainForm` dziedzicząca po `Form`, a więc przejmująca, jak już doskonale wiadomo (lekce z rozdziału 3.), jej cechy i właściwości. Jest to bardzo prosta konstrukcja, zawierająca jedynie konstruktor, w którym ustalany jest tytuł (modyfikacja właściwości `Text`), szerokość (modyfikacja właściwości `Width`) oraz wysokość (modyfikacja właściwości `Height`) okna. Druga klasa — `Program` — ma postać bardzo podobną do tej przedstawionej na listingu 7.1, z tą różnicą, że jako argument metody `Run` jest przekazywany nowo utworzony obiekt naszej klasy — `MainForm` — a nie `Form`. Tak więc tym razem okno aplikacji jest reprezentowane przez klasę `MainForm`. Zwrócmy też uwagę, że można by to rozwiązać nieco inaczej. Czy bowiem na pewno potrzebna jest klasa `Program`? To oczywiście zależy od struktury całej aplikacji, ale w tym przypadku na pewno można by się jej pozbyć. Pozostanie to jednak jako ćwiczenie do samodzielnego wykonania.

Przeanalizujmy jeszcze jeden przykład z wykorzystaniem wiadomości z lekcji 15. Otóż napiszmy aplikację okienkową, w której tytuł i rozmiary okna będą wprowadzane z wiersza poleceń. Przykład tak działającego programu jest widoczny na listingu 7.4.

Listing 7.4. Okno o tytule i rozmiarze wprowadzanych z wiersza poleceń

```
using System;
using System.Windows.Forms;

public class MainForm : Form
```

```
{  
    public MainForm(string tytul, int szerokosc, int wysokosc)  
    {  
        Text = tytul;  
        Width = szerokosc;  
        Height = wysokosc;  
    }  
}  
  
public class Program  
{  
    public static void Main(string[] args)  
    {  
        string tytul;  
        int szerokosc, wysokosc;  
        if(args.Length < 3)  
        {  
            tytul = "Tytuł domyślny";  
            szerokosc = 320;  
            wysokosc = 200;  
        }  
        else  
        {  
            tytul = args[0];  
            try  
            {  
                szerokosc = Int32.Parse(args[1]);  
                wysokosc = Int32.Parse(args[2]);  
            }  
            catch(Exception)  
            {  
                szerokosc = 320;  
                wysokosc = 200;  
            }  
        }  
        Application.Run(new MainForm(tytul, szerokosc, wysokosc));  
    }  
}
```

Klasa `MainForm` ma teraz nieco inną postać niż we wcześniejszych przykładach. Konstruktor przyjmuje trzy argumenty określające parametry okna. Są to: `tytul` — określający tytuł, `szerokosc` — określający szerokość oraz `wysokosc` — określający wysokość okna. Argumenty konstruktora przypisywane są właściwościom `Text`, `Width` i `Height`.

Dużo więcej pracy wymagała natomiast modyfikacja metody `Main` z klasy `Program`. Zaczyna się ona od zadeklarowania zmennych pomocniczych `tytul`, `szerokosc` i `wysokosc`, którym zostaną przypisane dane wymagane przez konstruktor `MainForm`. Następnie sprawdzane jest, czy przy wywołaniu programu zostały podane co najmniej trzy argumenty. Jeśli nie, zmienne inicjalizowane są wartościami domyślnymi, którymi są:

- ◆ Tytuł domyślny — dla zmiennej `tytul`;
- ◆ 320 — dla zmiennej `szerokosc`;
- ◆ 200 — dla zmiennej `wysokosc`.

Jeśli jednak dane zostały przekazane, trzeba je odpowiednio przetworzyć. Z tytułem nie ma problemu — przyjmujemy, że jest to po prostu pierwszy otrzymany ciąg, do konujemy więc bezpośredniego przypisania:

```
tytuł = args[0];
```

Inaczej jest z wysokością i szerokością. Muszą być one przetworzone na wartości typu `int`. Nie można przy tym zakładać, że na pewno będą one poprawne — użytkownik może przecież wprowadzić w wierszu polecień dowolne dane. Dlatego też dwa wywołania metody `Parse` przetwarzające drugi (`args[1]`) i trzeci (`args[2]`) argument zostały ujęte w blok `try...catch`. Dzięki temu, jeśli otrzymane dane nie będą reprezentowały poprawnych wartości całkowitych, ~~z~~nienny szerokość i wysokość zostaną przypisane wartości domyślne 320 i 200.

Ostatecznie wszystkie zmienne pomocnicze są używane jako argumenty konstruktora obiektu klasy `MainForm`, który stanowić będzie główne okno aplikacji:

```
Application.Run(new MainForm(tytuł, szerokosc, wysokosc));
```

W prosty sposób można więc będzie sterować parametrami okna z poziomu wiersza polecen.

Tworzenie menu

Większość aplikacji okienkowych posiada menu — to jeden z podstawowych elementów interfejsu graficznego. Warto więc zobaczyć, jak wyposażyc okno programu w takie udoskonalenie. Należy w tym celu skorzystać z klas `MainMenu` oraz `MenuItem`. Pierwsza z nich opisuje pasek menu, natomiast druga — poszczególne pozycje menu. Najpierw należy utworzyć obiekt klasy `MainMenu` oraz obiekty typu `MenuItem` odpowiadające poszczególnym pozycjom, a następnie połączyć je ze sobą za pomocą właściwości `MenuItem`s i metody `Add`. Tekst znajdujący się w pozycjach menu modyfikuje się za pomocą właściwości `Text` klasy `MenuItem`. Jak to wygląda w praktyce, zobrazowano w przykładzie widocznym na listingu 7.5.

Listing 7.5. Budowa menu

```
using System.Windows.Forms;

public class MainForm : Form
{
    public MainForm()
    {
        Text = "Moja aplikacja";
        Width = 320;
        Height = 200;

        MainMenu mm = new MainMenu();

        MenuItem mi1 = new MenuItem();
        MenuItem mi2 = new MenuItem();
```

```
    mi1.Text = "Menu 1";
    mi2.Text = "Menu 2";

    mm.MenuItems.Add(mi1);
    mm.MenuItems.Add(mi2);

    Menu = mm;
}

public class Program
{
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Główna część kodu jest zawarta w konstruktorze klasy `MainForm`. Na początku jest ustalany tytuł oraz rozmiary okna aplikacji, a następnie jest tworzony obiekt typu `MainMenu`, który będzie głównym menu powiązany z oknem programu. Obiekt ten jest przypisywany zmiennej `mm`:

```
MainMenu mm = new MainMenu();
```

Po utworzeniu menu głównego tworzone są jego dwie pozycje. Odbywa się to przez wywołanie konstruktorów klasy `MenuItem`:

```
MenuItem mi1 = new MenuItem();
MenuItem mi2 = new MenuItem();
```

Nowe obiekty przypisywane są zmiennej `mi1` i `mi2`, tak aby można się było do nich w prosty sposób odwoływać w dalszej części kodu. Każda pozycja menu powinna mieć przypisany tekst i swoją nazwę, dlatego też modyfikowana jest właściwość `Text` obiektów `mi1` i `mi2`:

```
mi1.Text = "Menu 1";
mi2.Text = "Menu 2";
```

Pozycje menu trzeba w jakiś sposób powiązać z menu głównym, czyli po prostu dodać je do menu głównego. Odbywa się to przez wywołanie metody `Add` właściwości `MenuItems` obiektu klasy `MainMenu`, czyli obiektu `mm`:

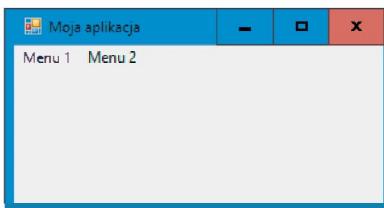
```
mm.MenuItems.Add(mi1);
mm.MenuItems.Add(mi2);
```

Na zakończenie trzeba dołączyć menu główne do aplikacji, co odbywa się przez przypisanie go właściwości `Menu` okna:

```
Menu = mm;
```

Samo uruchomienie aplikacji przebiega w taki sam sposób jak w poprzednich przykładach, jej wygląd zobrazowano natomiast na rysunku 7.3.

Rysunek 7.3.
Aplikacja
zawierająca menu



Z takiego menu nie będziemy jednak zadowoleni, nie zawiera ono przecież żadnych pozycji. A w realnej aplikacji rozwijane menu może być przecież nawet wielopoziomowe. Trzeba więc nauczyć się, jak dodawać do menu kolejne pozycje. Na szczęście jest to bardzo proste. Otóż każdy obiekt typu `MenuItem` zawiera odziedziczoną po klasie `Menu` właściwość `MenuItem`s, która określa wszystkie jego podmenu, czyli pozycje, które ma zawierać. Każda pozycja może więc zawierać inne pozycje menu. W ten sposób można zbudować wielopoziomową strukturę o dowolnej wielkości. Utworzymy więc teraz aplikację mającą menu główne zawierające jedną pozycję, ta pozycja będzie zawierała trzy kolejne, a ostatnia z tych trzech — kolejne trzy. Brzmi to nieco zawile, ale chodzi o strukturę widoczną na rysunku 7.4. Została ona utworzona przez kod z listingu 7.6.

Listing 7.6. Budowa rozwijanego menu

```
using System.Windows.Forms;

public class MainForm : Form
{
    public MainForm()
    {
        Text = "Moja aplikacja";
        Width = 320;
        Height = 200;

        MainMenu mm = new MainMenu();
        MenuItem mi1 = new MenuItem("Menu 1");

        MenuItem m1p1 = new MenuItem("Pozycja 1");
        MenuItem m1p2 = new MenuItem("Pozycja 2");
        MenuItem m1p3 = new MenuItem("Pozycja 3");

        MenuItem m1p3p1 = new MenuItem("Pozycja 1");
        MenuItem m1p3p2 = new MenuItem("Pozycja 2");
        MenuItem m1p3p3 = new MenuItem("Pozycja 3");

        m1p3.MenuItems.Add(m1p3p1);
        m1p3.MenuItems.Add(m1p3p2);
        m1p3.MenuItems.Add(m1p3p3);

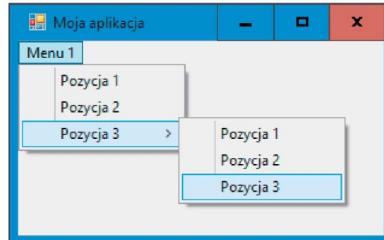
        mi1.MenuItems.Add(m1p1);
        mi1.MenuItems.Add(m1p2);
        mi1.MenuItems.Add(m1p3);

        mm.MenuItems.Add(mi1);
        Menu = mm;
    }
}
```

```
public class Program
{
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Rysunek 7.4.

Aplikacja zawierająca wielopoziomowe menu



Na początku jest tworzone menu główne oraz jego jedyna pozycja, powstają więc obiekty typu `MainMenuItem` i `MenuItem`:

```
MainMenu mm = new MainMenu();
MenuItem mi1 = new MenuItem("Menu 1");
```

Został tu użyty drugi z konstruktorów klasy `MenuItem`, przyjmujący jeden argument typu `string`, określający, jaki tekst ma być wyświetlany na danej pozycji. Dzięki temu unikamy konieczności późniejszego przypisywania danych właściwości `Text`.

Następnie powstają pozycje menu `mi1`; kolejne obiekty typu `MenuItem` są przypisywane zmiennym `m1p1` (czyli: menu 1, pozycja 1), `m1p2` i `m1p3`:

```
MenuItem m1p1 = new MenuItem("Pozycja 1");
MenuItem m1p2 = new MenuItem("Pozycja 2");
MenuItem m1p3 = new MenuItem("Pozycja 3");
```

Dalsze trzy instrukcje to utworzenie pozycji, które będą przypisane do menu `m1p3`. Nowo powstałe obiekty są przypisywane zmiennym `m1p3p1` (czyli: menu 1, pozycja 3, pozycja 1), `m1p3p2` i `m1p3p3`:

```
MenuItem m1p3p1 = new MenuItem("Pozycja 1");
MenuItem m1p3p2 = new MenuItem("Pozycja 2");
MenuItem m1p3p3 = new MenuItem("Pozycja 3");
```

Kiedy wszystkie obiekty są gotowe, trzeba je połączyć w spójną całość. Najpierw dodawane są pozycje do menu `m1p3`:

```
m1p3.MenuItems.Add(m1p3p1);
m1p3.MenuItems.Add(m1p3p2);
m1p3.MenuItems.Add(m1p3p3);
```

a następnie do menu `mi1`:

```
mi1.MenuItems.Add(m1p1);
mi1.MenuItems.Add(m1p2);
mi1.MenuItems.Add(m1p3);
```

Na zakończenie menu `mi1` jest dodawane do menu głównego `mm`, a menu główne do okna aplikacji:

```
mm.MenuItems.Add(mi1);
Menu = mm;
```

Po skompilowaniu i uruchomieniu programu przekonamy się, że faktycznie powstałe menu jest wielopoziomowe, tak jak zostało to zaprezentowane na rysunku 7.4. Z jego wyglądu powinniśmy już być zadowoleni; brakuje jednak jeszcze jednego elementu. Otóż takie menu jest nieaktywne, tzn. po wybraniu dowolnej pozycji nic się nie dzieje. Oczywiście nic dziać się nie może, skoro nie przypisaliśmy im żadnego kodu wykonywalnego. Aby to zrobić, trzeba znać zagadnienia delegacji i zdarzeń, o czym traktuje kolejna lekcja.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 35.1

Zmodyfikuj program z listingu 7.3 w taki sposób, aby nie było konieczności użycia klasy `Program`, a aplikacja zawierała jedynie klasę `MainForm`.

Ćwiczenie 35.2

Napisz aplikację okienkową, w której tytuł i rozmiary okna będą wczytywane z pliku tekstowego o nazwie przekazanej jako argument wywołania. W przypadku niepodania nazwy pliku bądź wykrycia nieprawidłowego formatu danych powinny zostać zastosowane wartości domyślne.

Ćwiczenie 35.3

Napisz aplikację okienkową, której pierwotne położenie na ekranie będzie określane za pomocą wartości przekazanych z wiersza polecen (właściwości umożliwiające zmianę położenia okna znajdziesz w tabeli 7.1; aby mieć możliwość samodzielnego ustalania początkowej pozycji okna trzeba też zmienić wartość właściwości `StartPosition` obiektu typu `Form` na `FormStartPosition.Manual`).

Ćwiczenie 35.4

Napisz aplikację zawierającą menu główne z jedną pozycją, która z kolei zawiera menu z trzema pozycjami. Każda z tych trzech pozycji powinna zawierać kolejne trzy pozycje.

Ćwiczenie 35.5

Napisz aplikację okienkową zawierającą menu. Struktura menu powinna być wczytywana z pliku tekstowego o nazwie przekazanej w postaci argumentu z wiersza polecen.

Lekcja 36. Delegacje i zdarzenia

Lekcja 36. została poświęcona delegacjom i zdarzeniom. To dosyć ważny temat. Oba te mechanizmy są ze sobą powiązane i pozwalają na asynchroniczną komunikację między obiektami. Asynchroniczną, czyli taką, w której obiekt informuje o zmianie swojego stanu wtedy, gdy taka zmiana nastąpi, a odbiorca informacji nie czeka na nią aktywnie — może być mu ona przekazana w dowolnej chwili. Lekcja ta jest umieszczona w rozdziale omawiającym aplikacje okienkowe dlatego, że delegacje i zdarzenia są niezbędne do obsługi graficznego interfejsu użytkownika (co zostanie pokazane w lekcji 37.). Nie należy jednak wyciągać z tego wniosku, że omawiane mechanizmy służą wyłącznie do tego celu. Są one uniwersalne i można ich również używać w wielu innych sytuacjach.

Koncepcja zdarzeń i delegacji

Znaczenie terminu **zdarzenie** (ang. *event*) jest zgodne z jego intuicyjnym rozumieniem. Może to być jednokrotne lub dwukrotne kliknięcie myszą, rozwinięcie menu, przesunięcie kurSORA, otworzenie i zamknięcie okna, uruchomienie lub zamknięcie aplikacji itp. Zdarzenia są więc niezbędne do obsługi graficznego interfejsu użytkownika. Nie jest to jednak jedyne ich zastosowanie, zdarzenie to przecież również odebranie danych np. interfejsu sieciowego czy informacja o zakończeniu długich obliczeń. Takiemu realnemu zdarzeniu odpowiada opisujący je *byt programistyczny*.

Samo wystąpienie zdarzenia to jednak nie wszystko, musi być ono przecież w jakiś sposób powiązane z kodem, który zostanie wykonany po jego wystąpieniu. W C# do takiego powiązania służy mechanizm tzw. **delegacji** (ang. *delegation*)¹. Dzięki temu wystąpienie danego zdarzenia może spowodować wywołanie konkretnej metody bądź nawet kilku metod. Historycznie delegacje wywodzą się ze znanych z języka C wskaźników do funkcji i tzw. funkcji zwrotnych (ang. *callback functions*), jest to jednak mechanizm dużo nowocześniejszy i bezpieczniejszy.

Tak więc przykładową klasę opisującą okno aplikacji można wyposażyć w zestaw zdarzeń. Dzięki temu będzie wiadomo, na jakie zdarzenia obiekt tej klasy zareaguje. Aby taka reakcja była możliwa, do zdarzenia musi być przypisany obiekt delegacji przechowujący listę metod wywoływanych w odpowiedzi na to zdarzenie. Żeby jednak nie przedłużać tych nieco teoretycznych dywagacji na temat mechanizmów obsługi zdarzeń, od razu przejdźmy do korzystania z delegacji na bardzo prostym przykładzie.

Tworzenie delegacji

Jak utworzyć delegacje? Należy użyć słowa **delegate**, po którym następuje określenie delegacji. Schematycznie wyglądałoby to następująco:

modyfikator_dostępu delegate określenie_delegacji;

¹ Spotyka się również określenia „ten delegat” lub „ta delegata”, oba jednak wydają się niezbyt naturalne. W książce będzie stosowany wyłącznie termin „ta delegacja”.

przy czym *określenie delegacji* można potraktować jak deklarację funkcji (metody) pasującej do tej delegacji. Gdyby to miała być na przykład funkcja o typie zwracanym `void` i nieprzyjmująca argumentów, deklaracja mogłaby wyglądać tak:

```
public delegate void Delegacja();
```

Jak jej użyć? W najprostszym przypadku tak, jakby była to referencja (odniesienie) do funkcji. Delegacja pozwoli więc na wywołanie dowolnej metody o deklaracji zgodnej z deklaracją delegacji. Brzmi to nieco zawile. Wykonajmy zatem od razu przykład takiego zastosowania. Jest on widoczny na listingu 7.7.

Listing 7.7. Pierwsza delegacja

```
using System;

public class Program
{
    public delegate void Delegacja();
    public static void Metoda1()
    {
        Console.WriteLine("Została wywołana metoda Metoda1.");
    }
    public static void Metoda2(string napis)
    {
        Console.WriteLine("Została wywołana metoda Metoda2.");
    }
    public static void Main()
    {
        Delegacja del1 = Metoda1;
        //Delegacja del2 = Metoda2;
        del1();
        //del2();
    }
}
```

Powstała delegacja o nazwie `Delegacja`, pasująca do każdej bezargumentowej metody (funkcji) o typie zwracanym `void`:

```
public delegate void Delegacja();
```

Jest ona składową klasy `Program`, gdyż tylko w tej klasie będzie używana. To nie jest jednak obligatoryjne. W rzeczywistości deklaracja delegacji jest deklaracją nowego typu danych. Mogłaby być więc umieszczona poza zasięgiem klasy `Program`, np. w sposób następujący:

```
using System;
public delegate void Delegacja();

public class Program
{
    //treść klasy Program
}
```

Z tego typu deklaracji skorzystamy jednak w dalszej części lekcji.

W klasie `Program` zostały umieszczone dwie metody: `Metoda1` i `Metoda2`. Pierwsza z nich ma typ zwracany `void` i nie przyjmuje argumentów, natomiast druga również ma typ zwracany `void`, ale przyjmuje jeden argument typu `string`. Jak można się domyślić, `Metoda1` pasuje do delegacji `Delegacja`, a `Metoda2` — nie.

W metodzie `Main` następuje utworzenie obiektu delegacji i przypisanie mu metody `Metoda1`:

```
Delegacja dell = Metoda1;
```

Jest to więc bardzo podobna konstrukcja jak przypisywanie wartości innym obiektem, tyle że wartością przypisywaną jest metoda². Można też oddzielić utworzenie delegacji od przypisania metody, zatem poprawnym zapisem byłoby też:

```
Delegacja dell;
dell = Metoda1;
```

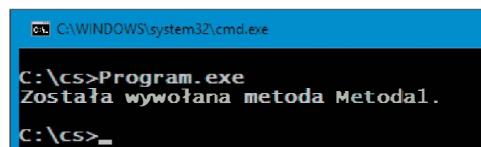
Co jednak uzyskaliśmy w taki sposób? Otóż to, że delegacja `dell` została powiązana z metodą `Metoda1`, a więc możemy ją potraktować jak referencję do tej metody. Możliwe jest zatem wywołanie metody `Metoda1` przez obiekt delegacji, co robimy za pomocą instrukcji:

```
dell();
```

O tym, że instrukcja ta faktycznie spowodowała wywołanie metody `Metoda1`, przekonamy się, kompilując i uruchamiając program. Ukaże się nam wtedy widok przedstawiony na rysunku 7.5.

Rysunek 7.5.

Wywołanie metody poprzez delegację



W metodzie `Main` znajdują się również dwie instrukcje w komentarzu:

```
//Delegacja del2 = Metoda2;
//del2();
```

Są one nieprawidłowe. Nie można przypisać metody `Metoda2` obiektowi delegacji `del2`, jako że ich deklaracje są różne. `Metoda2` przyjmuje bowiem jeden argument typu `string`, podczas gdy delegacja jest bezargumentowa. Skoro nie powstał obiekt `del2`, nie można też potraktować go jako referencji do metody i dokonać wywołania.

Oczywiście można przypisać delegacji metodę przyjmującą argumenty. Wystarczy ją odpowiednio zadeklarować. Tego typu działanie jest wykonywane w programie wiadocznym na listingu 7.8.

² Ta konstrukcja jest dostępna, począwszy od C# 2.0. W rzeczywistości zostanie wywołany konstruktor delegacji i powstanie odpowiedni obiekt. We wcześniejszych wersjach języka należy użyć konstrukcji o postaci `Delegacja dell = new Delegacja(Metoda1);`. Ten sposób zostanie wykorzystany w dalszej części książki.

Listing 7.8. Delegacje i argumenty

```
using System;

public class Program
{
    public delegate void Delegacja(string str);
    public static void Metoda1(string napis)
    {
        Console.WriteLine(napis);
    }
    public static void Main()
    {
        Delegacja dell = Metoda1;
        dell("To jest test.");
    }
}
```

Tym razem deklaracja delegacji jest zgodna z każdą metodą niezwracającą wyniku i przyjmującą jeden argument typu `string`:

```
public delegate void Delegacja(string str);
```

W klasie `Program` została więc umieszczona taka metoda, jest to `Metoda1`. Przyjmuje ona argument typu `string` i wyświetla jego zawartość na ekranie.

W metodzie `Main` powstał obiekt delegacji `Delegacja` o nazwie `dell` i została mu przypisana metoda `Metoda1`. Można go więc traktować jako odniesienie do metody `Metoda1` i wywołać ją przez zastosowanie instrukcji:

```
dell("To jest test.");
```

Efektem jej działania będzie pojawienie się na ekranie napisu `To jest test..`

Podobnie jest z metodami zwracającymi jakieś wartości. Aby móc je wywoływać poprzez delegacje, należy te delegacje odpowiednio zadeklarować. Założymy więc, że interesuje nas powiązanie z delegacją metody przyjmującej dwa argumenty typu `int` i zwracającej wynik typu `int` (np. wartość wynikającą z dodania argumentów). Działający w ten sposób program został przedstawiony na listingu 7.9.

Listing 7.9. Zwracanie wartości przy wywoaniu metody przez delegację

```
using System;

public class Program
{
    public delegate int Delegacja(int arg1, int arg2);
    public static int Dodaj(int argument1, int argument2)
    {
        int wynik = argument1 + argument2;
        return wynik;
    }
    public static void Main()
    {
```

```
Delegacja dell = Dodaj;
int wartosc = dell(4, 8);
Console.WriteLine("Wynikiem jest {0}.", wartosc);
}
```

W klasie Program została umieszczona deklaracja delegacji Delegacja:

```
int Delegacja(int arg1, int arg2);
```

mamy więc tu do czynienia z typem zwracanym int i dwoma argumentami typu int. Taka deklaracja odpowiada statycznej metodzie Dodaj, która również zwraca wartość typu int i przyjmuje dwa argumenty tego typu. We wnętrzu metody następuje dodanie wartości argumentów i zwrócenie wyniku za pomocą instrukcji return.

Skoro deklaracje delegacji i metody są zgodne, możliwe jest ich powiązanie, czyli obiekt delegacji może stać się referencją do metody, tak też dzieje się w funkcji Main. Instrukcja:

```
Delegacja dell = Dodaj;
```

powoduje utworzenie obiektu dell typu Delegacja i przypisanie mu metody Dodaj. Obiekt ten jest następnie używany do wywołania metody Dodaj i przypisania wyniku zmiennej pomocniczej wartosc:

```
int wartosc = dell(4, 8);
```

Wartość tej zmiennej jest z kolei wyświetlana na ekranie za pomocą instrukcji Console.
→WriteLine.

Delegacja jako funkcja zwrotna

Gdyby zastosowania delegacji ograniczały się do przedstawionych w poprzednim podpunkcie, ich użyteczność byłaby dosyć ograniczona. Wyobraźmy sobie jednak nieco bardziej złożoną sytuację. Otóż czasami przydatne jest, aby klasa mogła użyć kodu zewnętrznego. Z taką sytuacją często mamy do czynienia np. w przypadku klas lub metod realizujących algorytm sortowania operujące na elementach dowolnego typu. Niezbędne jest wtedy dostarczenie metody porównującej dwa elementy. Do tego doskonale nadają się właśnie delegacje.

Jak w praktyce realizuje się takie zastosowanie, sprawdzimy jednak na prostszym przykładzie, który nie będzie wymagał implementacji żadnego skomplikowanego algorytmu. Otóż założymy, że utworzyliśmy klasę o nazwie Kontener, która posiada dwie właściwości (mogłyby to być pola): w1 i w2. Pierwsza będzie typu int, a druga typu double. Klasa będzie też zawierała metodę Wyswietl, której zadaniem będzie wyświetlanie wartości elementów na ekranie. Kod działający w opisany sposób jest widoczny na listingu 7.10.

Listing 7.10. Klasa kontenerowa

```
public class Kontener
{
    public int w1
    {
        get
        {
            return 100;
        }
    }
    public double w2
    {
        get
        {
            return 2.14;
        }
    }
    public void Wyswietl()
    {
        System.Console.WriteLine("w1 = {0}", w1);
        System.Console.WriteLine("w2 = {0}", w2);
    }
}
```

Obie właściwości są tylko do odczytu. Pierwsza z nich po prostu zwraca wartość 100, a druga — 2.14. Metoda `Wyswietl` wyświetla wartość właściwości w dwóch wierszach na ekranie. Aby przetestować działanie tej klasy, wystarczy użyć instrukcji:

```
Kontener k = new Kontener();
k.Wyswietl();
```

Przy takiej konstrukcji postać metody `Wyswietl` jest standardowa i z góry ustalona. Zmiana sposobu wyświetlania wiązałaby się ze zmianą kodu klasy, a to nie zawsze jest możliwe. Możemy np. nie mieć dostępu do kodu źródłowego, lecz jedynie biblioteki *dll*, zawierającej już skompilowany kod. Trzeba więc spowodować, aby treść metody wyświetlającej mogła być dostarczana z zewnątrz klasy. Pomoże nam w tym właśnie mechanizm delegacji.

Wyrzućmy zatem z kodu klasy `Kontener` metodę `Wyswietl`, a na jej miejsce wprowadźmy inną, o następującej postaci:

```
public void WyswietlCallBack(DelegateWyswietl del)
{
    del(this);
}
```

Jest to metoda przyjmująca jako argument obiekt delegacji typu `DelegateWyswietl` o nazwie `del`. Wewnątrz tej metody następuje wywołanie delegacji z przekazaniem jej argumentu `this`, a więc bieżącego obiektu typu `Kontener`. Wynika z tego, że deklaracja delegacji powinna odpowiadać metodom przyjmującym argument typu `Kontener`. Utwórzmy więc taką deklarację:

```
public delegate void DelegateWyswietl(Kontener obj);
```

Odpowiada ona wszystkim metodom o typie zwracanym void przyjmującym argument typu Kontener. Połączony teraz wszystko oraz dopisany kod korzystający z nowych mechanizmów. Całość przyjmie wtedy postać widoczną na listingach 7.11 i 7.12.

Listing 7.11. Klasa Kontener korzystająca z metody zwrotnej

```
public delegate void DelegateWyswietl(Kontener obj);

public class Kontener
{
    public int w1
    {
        get
        {
            return 100;
        }
    }
    public double w2
    {
        get
        {
            return 2.14;
        }
    }
    public void WyswietlCallBack(DelegateWyswietl del)
    {
        del(this);
    }
}
```

Na początku została umieszczona deklaracja delegacji `DelegateWyswietl`, a w kodzie klasy `Kontener` metoda `Wyswietl` została wymieniona na `WyswietlCallBack`. Te zmiany zostały już opisane wcześniej. Co się natomiast dzieje w klasie `Program` (listing 7.12)? Otóż została ona wyposażona w statyczną metodę `Wyswietl` przyjmującą jeden argument typu `Kontener`. Wewnątrz tej metody za pomocą instrukcji `Console.WriteLine` wyświetlane są właściwości `w1` i `w2` obiektu otrzymanego w postaci argumentu. Jak można się domyślić, to właśnie ta metoda zostanie połączona z delegacją.

Listing 7.12. Użycie metody zwrotnej do wyświetlania wyniku

```
public class Program
{
    public static void Wyswietl(Kontener obj)
    {
        System.Console.WriteLine(obj.w1);
        System.Console.WriteLine(obj.w2);
    }
    public static void Main()
    {
        Kontener k = new Kontener();
        DelegateWyswietl del = Wyswietl;
        k.WyswietlCallBack(del);
    }
}
```

Przyjrzyjmy się teraz metodzie `Main` z klasy `Program`. Najpierw jest tworzony nowy obiekt k typu `Kontener`:

```
Kontener k = new Kontener();
```

Następnie jest tworzony obiekt delegacji `DelegateWyswietl` i wiązany z metodą `Wyswietl` klasy `Main`:

```
DelegateWyswietl del = Wyswietl;
```

Trzecia instrukcja to natomiast wywołanie metody `WyswietlCallBack` obiektu `k` i przekazanie jej obiektu delegacji:

```
k.WyswietlCallBack(del);
```

To tak, jakbyśmy metodzie `WyswietlCallBack` przekazali referencję do metody `Wyswietl` klasy `Main`. Innymi słowy, informujemy obiekt k klasy `Kontener`, że do wyświetlanego swoich danych ma użyć zewnętrznej metody `Wyswietl` dostarczonej przez wywołanie `WyswietlCallBack`. W ten sposób spowodowaliśmy, że obiekt klasy `Kontener` korzysta z zewnętrznego kodu.

Co więcej, możemy napisać kilka różnych metod wyświetlających dane i korzystać z nich zamiennie, tworząc odpowiednie delegacje i wywołując metodę `WyswietlCallBack`. Nie będzie to wymagało żadnych modyfikacji klasy `Kontener` (to właśnie było przecież naszym celem). Zobrazowano to w przykładzie widocznym na listingu 7.13.

Listing 7.13. Wywoływanie za pomocą delegacji różnych metod

```
public class Program
{
    public static void Wyswietl1(Kontener obj)
    {
        System.Console.WriteLine(obj.w1);
        System.Console.WriteLine(obj.w2);
    }
    public static void Wyswietl2(Kontener obj)
    {
        System.Console.WriteLine("Wartość w1 to {0}.", obj.w1);
        System.Console.WriteLine("Wartość w2 to {0}.", obj.w2);
    }
    public static void Main()
    {
        Kontener k = new Kontener();
        DelegateWyswietl del1 = Wyswietl1;
        DelegateWyswietl del2 = Wyswietl2;
        k.WyswietlCallBack(del1);
        k.WyswietlCallBack(del2);
    }
}
```

W klasie `Program` mamy tym razem dwie metody zajmujące się wyświetlaniem danych z obiektów typu `Kontener`. Pierwsza — `Wyswietl1` — wyświetla same wartości, natomiast druga — `Wyswietl2` — wartości uzupełnione o opis słowny. W metodzie

Main najpierw został utworzony obiekt k typu Kontener, a następnie dwie delegacje: dell i del2. Pierwsza z nich została powiązana z metodą Wyswietl1:

```
DelegateWyswietl dell = Wyswietl1;
```

a druga z metodą Wyswietl2:

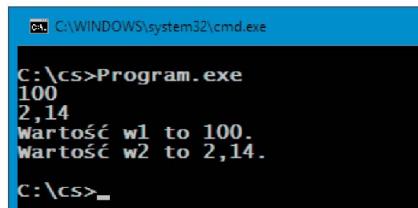
```
DelegateWyswietl del2 = Wyswietl2;
```

Obie delegacje zostały użyte w wywołaniach metody WyswietlCallBack:

```
k.WyswietlCallBack(dell);
k.WyswietlCallBack(del2);
```

Dzięki temu spowodowaliśmy, że obiekt k użył do wyświetlenia swoich danych metod Wyswietl1 i Wyswietl2 pochodzących z klasy Main. Po komplikacji i uruchomieniu programu zobaczymy zatem widok przedstawiony na rysunku 7.6.

Rysunek 7.6.
Efekt działania kodu z listingu 7.13



Delegacja powiązana z wieloma metodami

Na samym początku lekcji zostało wspomniane, że delegacja pozwala na powiązanie zdarzenia nawet z kilkoma metodami. A zatem musi istnieć możliwość przypisania jej większej liczby metod niż tylko jednej, tak jak miało to miejsce w dotychczasowych przykładach. Tak jest w istocie, każdy obiekt delegacji może być powiązany z dowolną liczbą metod; można je do tego obiektu dodawać, a także — uwaga — odejmować od niego. Odejmowanie oznacza w tym przypadku usunięcie powiązania delegacji z daną metodą. Oczywiście należy pamiętać, że każda powiązana metoda musi mieć deklarację zgodną z deklaracją delegacji. Przekonajmy się, jak to będzie wyglądało w praktyce. Zobrazowano to w przykładzie widocznym na listingu 7.14 (korzysta on z klasy Kontener i delegacji z listingu 7.11).

Listing 7.14. Użycie delegacji do wywołania kilku metod

```
using System;

public class Program
{
    public static void WyswietlW1(Kontener obj)
    {
        Console.WriteLine("Wartość w1 to {0}.", obj.w1);
    }
    public static void WyswietlW2(Kontener obj)
    {
        Console.WriteLine("Wartość w2 to {0}.", obj.w2);
    }
}
```

```
public static void Main()
{
    Kontener k = new Kontener();
    DelegateWyswietl del1 = WyswietlW1;
    DelegateWyswietl del2 = WyswietlW2;

    DelegateWyswietl del3 = del1 + del2;
    k.WyswietlCallBack(del3);

    Console.WriteLine("--");

    del3 -= del2;
    k.WyswietlCallBack(del3);

    Console.WriteLine("--");

    del3 += del2;
    k.WyswietlCallBack(del3);
}
```

W klasie Program znalazły się dwie metody wyświetlające dane z obiektów typu Kontener: WyswietlW1 i WyswietlW2. Pierwsza wyświetla wartość właściwości w1, a druga — właściwości w2. W metodzie Main powstał nowy obiekt typu Kontener oraz dwie delegacje typu DelegateWyswietl: del1 i del2. Pierwszej została przypisana metoda WyswietlW1, a drugiej — WyswietlW2. Bardzo ciekawa jest natomiast instrukcja tworząca trzecią delegację, del3:

```
DelegateWyswietl del3 = del1 + del2;
```

Wygląda to jak dodawanie delegacji, a oznacza utworzenie obiektu del3 i przypisanie mu wszystkich metod powiązanych z delegacjami del1 i del2. Skoro więc obiekt del1 był powiązany z metodą WyswietlW1, a del2 z metodą WyswietlW2, to del3 będzie powiązany zarówno z WyswietlW1, jak i WyswietlW2. Przekonujemy się o tym dzięki wywołaniu:

```
k.WyswietlCallBack(del3);
```

Faktycznie spowoduje ono uruchomienie obu metod.

Skoro delegacje można dodawać, to można je też odejmować:

```
del3 -= del2;
```

Oczywiście oznacza to usunięcie z delegacji del3 wszystkich odwołań do metod występujących w delegacji del2. Dlatego też kolejna instrukcja:

```
k.WyswietlCallBack(del3);
```

spowoduje wywołanie tylko metody WyswietlW1.

Usuniętą delegację można bez problemów ponownie dodać za pomocą operatora `+=`:

```
de13 += de12;
```

Nic też nie stoi na przeszkodzie, aby jedna metoda została dodana do delegacji kilka razy. Zatem kolejna instrukcja:

```
de13 += de12;
```

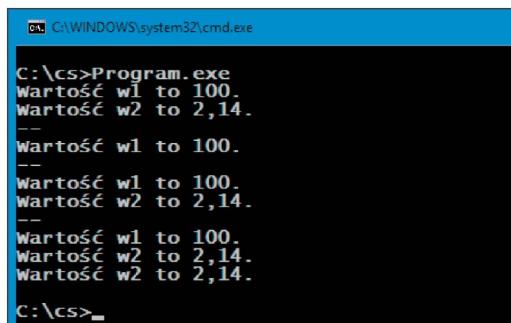
powoduje, że w delegacji `de13` znajdują się trzy odwołania: jedno do metody `WyswietlW1` i dwa do metody `WyswietlW2`. Dlatego też ostatnia instrukcja:

```
k.WyswietlCallBack(de13);
```

powoduje wyświetlenie w sumie trzech napisów. Ostatecznie po skompilowaniu i uruchomieniu programu zobaczymy więc widok zaprezentowany na rysunku 7.7.

Rysunek 7.7.

Efekt dodawania i odejmowania delegacji



C:\>Program.exe
Wartość w1 to 100.
Wartość w2 to 2,14.
--
Wartość w1 to 100.
Wartość w2 to 2,14.
--
Wartość w1 to 100.
Wartość w2 to 2,14.
Wartość w2 to 2,14.
C:\>_

Metody do delegacji mogą być też dodawane bezpośrednio, a nie — jak w powyższym przykładzie — za pomocą innych delegacji. Oznacza to, że można utworzyć delegację `de13` i przypisać jej metodę `WyswietlW1`:

```
DelegateWyswietl de13 = WyswietlW1;
```

a następnie dodać do niej metodę `WyswietlW2`:

```
de13 += WyswietlW2;
```

Po dodaniu metody można ją oczywiście usunąć za pomocą operatora `-=`:

```
de13 -= WyswietlW2;
```

Można więc łatwo poprawić kod z listingu 7.14 w taki sposób, aby występował w nim tylko jeden obiekt delegacji (co będzie ćwiczeniem do samodzielnego wykonania).

Zdarzenia

Podstawy zdarzeń

Zdarzenie to coś takiego jak kliknięcie myszą czy wybranie pozycji z menu — rozumiemy to intuicyjnie. Taki też opis pojawił się na początku niniejszej lekcji. Ale zdarzenie to również składowa klasy, dzięki której obiekt może informować o swoim

stanie. Założmy, że napisaliśmy aplikację okienkową, taką jak w lekcji 35., wykorzystując do tego klasę `Form`. Użytkownik tej aplikacji kliknął w dowolnym miejscu jej okna. W dużym uproszczeniu możemy powiedzieć, że obiekt okna otrzymał wtedy z systemu informację o tym, że nastąpiło kliknięcie. Dzięki temu, że w klasie `Form` zostało zdefiniowane zdarzenie o nazwie `Click`, istnieje możliwość wywołania w odpowiedzi na nie dowolnego kodu. Można się zapewne domyślić, że w tym celu wykorzystuje się delegacje.

Zanim jednak przejdziemy do obsługi zdarzeń w aplikacjach okienkowych (lekcja 37.), postaramy się najpierw przeanalizować sam ich mechanizm. Na początku skonstrujmy bardzo prostą klasę, zawierającą jedno prywatne pole `x` typu `int` oraz metody `getX` i `setX`, pozwalające na pobieranie jego wartości. Klasę tę nazwiemy `Kontener`, jako że przechowuje pewną daną. Jej postać jest widoczna na listingu 7.15.

Listing 7.15. Prosta klasa przechowująca wartość typu `int`

```
public class Kontener
{
    private int x;
    public int getX()
    {
        return x;
    }
    public void setX(int arg)
    {
        x = arg;
    }
}
```

Tego typu konstrukcje wykorzystywaliśmy już wielokrotnie, nie trzeba więc wyjaśniać, jak działa ten kod. Założymy teraz, że klasa `Kontener` powinna informować swoje otoczenie o tym, że przypisywana polu `x` wartość jest ujemna. Jak to zrobić? W pierwszej chwili mogą się nasunąć takie pomysły, jak użycie wyjątków czy zwarcanie przez metodę `setX` jakiejś wartości. O pierwszym pomyśle należy od razu zapomnieć — wyjątki stosujemy tylko do obsługi sytuacji wyjątkowych, najczęściej związanych z wystąpieniem błędu. Tymczasem przypisanie wartości ujemnej polu `x` żadnym błędem nie jest. Pomysł drugi można rozważyć, ale nie jest on zbyt wygodny. Wymaga on, aby w miejscu wywołania metody `setX` badać zwróconą przez nią wartość. My tymczasem chcemy mieć niezależny mechanizm informujący o fakcie przypisania wartości ujemnej. Trzeba więc użyć zdarzeń.

Do ich definiowania służy słowo kluczowe `event`. Jednak wcześniej należy utworzyć odpowiadającą zdarzeniu delegację, a zatem postępowanie jest dwuetapowe:

1. Definiujemy delegację w postaci:

[modyfikator_dostępu] `typ_zwracany delegate nazwa_delegacji(argumenty);`

2. Definiujemy co najmniej jedno zdarzenie w postaci:

[modyfikator_dostępu] `event nazwa_delegacji nazwa_zdarzenia;`

Przyjmiemy, że w odpowiedzi na powstanie zdarzenia polegającego na przypisaniu wartości ujemnej polu `x` będzie mogła być wywoływana dowolna bezargumentowa metoda o typie zwracanym `void`. W związku z tym delegacja powinna mieć postać:

```
public delegate void OnUjemneEventDelegate();
```

Skoro tak, deklaracja zdarzenia będzie wyglądała następująco:

```
public event OnUjemneEventDelegate OnUjemne;
```

Typem zdarzenia jest więc `OnUjemneEventDelegate` (nazwa delegacji), a nazwą `OnUjemne` (analogicznie do `Click` lub `onClick`, `onKeyDown` itp., z takimi nazwami spotkamy się w lekcji 37.). Ponieważ zdarzenie powinno być składową klasy `Kontener`, jego deklarację należy umieścić wewnątrz klasy. Kod będzie miał wtedy postać taką jak na listingu 7.16.

Listing 7.16. Klasa z obsługą zdarzeń

```
public delegate void OnUjemneEventDelegate();  
  
public class Kontener  
{  
    private int x;  
    public event OnUjemneEventDelegate OnUjemne;  
    public int getX()  
    {  
        return x;  
    }  
    public void setX(int arg)  
    {  
        x = arg;  
        if(arg < 0)  
        {  
            if(OnUjemne != null)  
            {  
                OnUjemne();  
            }  
        }  
    }  
}
```

Tak więc na początku kodu pojawiła się deklaracja delegacji, a wewnątrz klasy `Kontener`, za polem `x`, deklaracja zdarzenia `OnUjemne`. Modyfikacji wymagała również metoda `setX`, która musi teraz badać stan przekazanego jej argumentu. Najpierw więc następuje przypisanie:

```
x = arg;
```

a następnie sprawdzenie, czy `arg` jest mniejsze od 0. Jeśli tak, należy wywołać zdarzenie. Odpowiada za to instrukcja:

```
OnUjemne();
```

Co to oznacza? Skoro typem zdarzenia jest delegacja `OnUjemneEventDelegate`, w rzeczywistości będzie to wywołanie tej delegacji, a więc wszystkich metod z nią powiązanych.

Bardzo ważne jest natomiast sprawdzenie, czy obiekt delegacji faktycznie istnieje. Będzie istniał, jeśli został utworzony za pomocą operatora new (ten temat nie był poruszany w książce) lub będzie utworzony pośrednio przez dodanie do niego metody za pomocą operatora +=, tak jak zostało to opisane w podrozdziale poświęconym delegacjom. Dlatego też zanim nastąpi wywołanie, sprawdzany jest warunek `OnUjemne != null`. Jeśli jest on prawdziwy, czyli `OnUjemne` jest różne od `null`, obiekt delegacji istnieje, może więc nastąpić wywołanie `OnUjemne()`. Jeśli warunek jest fałszywy, czyli `OnUjemne` jest równe `null`, obiekt delegacji nie istnieje i zdarzenie nie jest wywoływane.

Sprawdźmy, jak użyć takiej konstrukcji w praktyce. Wykorzystamy klasę Kontener z listingu 7.16 w programie widocznym na listingu 7.17.

Listing 7.17. Obsługa zdarzeń generowanych przez klasę Kontener

```
using System;

public class Program
{
    public static void OnUjemneKomunikat()
    {
        Console.WriteLine("Przypisano ujemną wartość.");
    }
    public static void Main()
    {
        Kontener k = new Kontener();
        k.OnUjemne += OnUjemneKomunikat;
        k.setX(10);
        Console.WriteLine("k.x = {0}", k.getX());
        k.setX(-10);
        Console.WriteLine("k.x = {0}", k.getX());
    }
}
```

W klasie Program została umieszczona bezargumentowa metoda `OnUjemneKomunikat` o typie zwracanym `void`. Jej deklaracja odpowiada więc delegacji `OnUjemneEventDelegate`, a tym samym będzie ona mogła być wywoływana w odpowiedzi na zdarzenie `OnUjemne`. Zadaniem tej metody jest po prostu wyświetlenie komunikatu o tym, że przypisana została ujemna wartość.

W metodzie `Main` najpierw został utworzony obiekt `k` klasy `Kontener`, a następnie zdarzeniu `OnUjemne` została przypisana metoda `OnUjemneKomunikat`:

```
k.OnUjemne += OnUjemneKomunikat;
```

To nic innego jak informacja, że w odpowiedzi na zdarzenie `OnUjemne` ma zostać wywołana metoda `OnUjemneKomunikat`. Taką metodę (przypisaną zdarzeniu) często nazywa się **procedurą obsługi zdarzenia**.

Aby się przekonać, że taka konstrukcja faktycznie działa zgodnie z założeniami, wywołujemy dwukrotnie metodę `setX`, a po każdym wywołaniu za pomocą metod `Console.WriteLine` i `getX()` wyświetlimy również stan pola `x`. Pierwsze wywołanie `setX`:

```
k.setX(10);
```

powoduje przypisanie polu x wartości dodatniej — nie jest więc generowane zdarzenie OnUjemne. Na ekranie pojawi się więc tylko napis:

```
k.x = 10
```

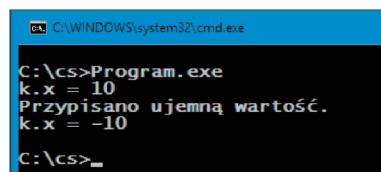
Drugie wywołanie setX:

```
k.setX(-10);
```

powoduje przypisanie polu x wartości ujemnej. A zatem w tym przypadku jest generowane zdarzenie OnUjemne i są wykonywane powiązane z nim metody. W omawianym przykładzie jest to metoda OnUjemneKomunikat. Tak więc na ekranie pojawi się dodatkowa informacja o przypisaniu wartości ujemnej, tak jak jest to widoczne na rysunku 7.8.

Rysunek 7.8.

Wywołanie metody powiązanej ze zdarzeniem



```
C:\cs>Program.exe
Przypisano ujemną wartość.
k.x = -10
```

Skoro można dodać metodę do zdarzenia, można ją również odjąć. W praktyce spotkamy się z sytuacjami, kiedy konieczna będzie rezygnacja z obsługi zdarzenia. Pośłużymy się wtedy operatorem -=. Jeśli bowiem wiązanie metody ze zdarzeniem miało postać:

```
k.OnUjemne += OnUjemneKomunikat;
```

to „odwiązywanie” będzie wyglądało następująco:

```
k.OnUjemne -= OnUjemneKomunikat;
```

Sprawdzmy to. Odpowiedni przykład jest widoczny na listingu 7.18.

Listing 7.18. Dodawanie i odejmowanie procedur obsługi

```
using System;

public class Program
{
    public static void OnUjemneKomunikat()
    {
        Console.WriteLine("Przypisano ujemną wartość.");
    }
    public static void Main()
    {
        Kontener k = new Kontener();
        k.OnUjemne += OnUjemneKomunikat;
        k.setX(-10);
        Console.WriteLine("k.x = {0}", k.getX());

        k.OnUjemne -= OnUjemneKomunikat;
        k.setX(-10);
        Console.WriteLine("k.x = {0}", k.getX());
    }
}
```

Metoda `OnUjemneKomunikat` pozostała bez zmian w stosunku do poprzedniego przykładu. W taki sam sposób tworzony jest też obiekt typu `Kontener` oraz następuje przypisanie metody `OnUjemneKomunikat` do zdarzenia:

```
k.OnUjemne += OnUjemneKomunikat;
```

Następnie jest wywoływana metoda `setX` z argumentem równym `-10`. Wiadomo więc, że na pewno powstanie zdarzenie `OnUjemne`, a na ekranie pojawi się napis wygenerowany w metodzie `OnUjemneKomunikat` oraz informacja o stanie pola wygenerowana przez pierwszą instrukcję:

```
Console.WriteLine("k.x = {0}", k.getX());
```

W dalszej części kodu następuje jednak usunięcie metody `OnUjemneKomunikat` z obsługi zdarzenia:

```
k.OnUjemne -= OnUjemneKomunikat;
```

W związku z tym kolejne wywołanie `setX` z ujemnym argumentem nie spowoduje żadnej reakcji. Informacja o stanie pola zostanie wyświetlona dopiero w kolejnej instrukcji `Console.WriteLine`. Ostatecznie po komplikacji i uruchomieniu programu zobaczymy więc widok zaprezentowany na rysunku 7.9.

Rysunek 7.9.
Efekt działania kodu z listingu 7.18

```
C:\cs>Program.exe
Przypisano ujemną wartość.
k.x = -10
k.x = -10
C:\cs>-
```

Obiekt generujący zdarzenie

Kiedy nabiera się większej praktyki w pisaniu aplikacji z obsługą zdarzeń, dochodzi się do przekonania, że bardzo często przydatne jest, aby metoda obsługująca zdarzenie (np. `OnUjemneKomunikat` z poprzedniego przykładu) miała dostęp do obiektu, który zdarzenie wygenerował, a często także do innych parametrów. Jest ona wtedy zdolna określić jego stan i podjąć bardziej zaawansowane działania. Postarajmy się więc tak zmodyfikować poprzednie przykłady, aby metodzie `OnUjemneKomunikat` był przekazywany właściwy obiekt. Modyfikacje nie będą bardzo duże, ale będą dotyczyć większości fragmentów kodu. Dodatkowo przypiszemy do zdarzenia dwie różne metody (procedury obsługi). Przykład realizujący tak postawione zadania znajduje się na listingach 7.19 i 7.20. Na listingu 7.19 została umieszczona klasa `Kontener` wraz z delegacją `OnUjemneEventDelegate`, a na listingu 7.20 — klasa `Program`.

Listing 7.19. Dostęp do obiektu generującego zdarzenie

```
public delegate void OnUjemneEventDelegate(Kontener obj);

public class Kontener
{
    private int x;
```

```
public event OnUjemneEventDelegate OnUjemne;
public int getX()
{
    return x;
}
public void setX(int arg)
{
    x = arg;
    if(arg < 0)
    {
        if(OnUjemne != null)
        {
            OnUjemne(this);
        }
    }
}
```

Zmieniła się deklaracja delegacji. Ma ona teraz postać:

```
public delegate void OnUjemneEventDelegate(Kontener obj);
```

a zatem obiektowi delegacji będzie można przypisywać metody o typie zwracanym `void` i przyjmujące argument typu `Kontener`. W klasie `Kontener` nieznacznie się zmieniła metoda `setX`. Tym razem bowiem wygenerowanie zdarzenia, a więc wywołanie delegacji, wymaga podania argumentu typu `Kontener`. Ponieważ założyliśmy, że wszystkie metody obsługujące nasze zdarzenie mają otrzymywać dostęp do obiektu generującego to zdarzenie, w wywołaniu delegacji trzeba przekazać wskazanie na obiekt bieżący, czyli `this`:

```
OnUjemne(this);
```

Przejdźmy do klasy `Program` widocznej na listingu 7.20.

Listing 7.20. Użycie klasy `Kontener` i delegacji `OnUjemneEventDelegate`

```
using System;

public class Program
{
    public static void OnUjemneKomunikat1(Kontener obj)
    {
        Console.WriteLine(
            "Przypisana wartość jest ujemna i równa {0}.", obj.getX());
    }
    public static void OnUjemneKomunikat2(Kontener obj)
    {
        Console.WriteLine(
            "Przypisano ujemną wartość = {0}.", obj.getX());
    }
    public static void Main()
    {
        Kontener k = new Kontener();
        k.OnUjemne += OnUjemneKomunikat1;
        k.setX(-10);
```

```

        Console.WriteLine("--");

        k.OnUjemne += OnUjemneKomunikat2;
        k.setX(-20);
    }
}

```

Zostały w niej umieszczone dwie metody, które będą procedurami obsługi zdarzenia `OnUjemne`. Są to `OnUjemneKomunikat1` i `OnUjemneKomunikat2`. Obie otrzymują jeden argument typu `Kontener` o nazwie `obj`. Dzięki odpowiedniemu wywołaniu delegacji w metodzie `setX` argumentem tym będzie obiekt generujący zdarzenie. W związku z tym obie metody mogą wyświetlić różne komunikaty zawierające aktualny stan pola `x`.

W metodzie `Main` został utworzony obiekt klasy `Kontener`, a następnie zdarzeniu `OnUjemne` została przypisana metoda `OnUjemneKomunikat1`:

```

Kontener k = new Kontener();
k.OnUjemne += OnUjemneKomunikat1;

```

Dlatego też po wywołaniu:

```
k.setX(-10);
```

na ekranie pojawia się zdefiniowany w tej metodzie komunikat zawierający informacje o wartości pola `x`. Dalej jednak zdarzeniu została przypisana również druga metoda, `OnUjemneKomunikat2`:

```
k.OnUjemne += OnUjemneKomunikat2;
```

A zatem od tej chwili wywołanie metody `setX` z wartością ujemną jako argumentem będzie powodowało wywołanie zarówno `OnUjemneKomunikat1`, jak i `OnUjemneKomunikat2`. Tak więc instrukcja:

```
k.setX(-20);
```

spowoduje ukazanie się dwóch komunikatów, tak jak jest to widoczne na rysunku 7.10.

Rysunek 7.10.
Wywołanie kilku metod w odpowiedzi na zdarzenie

```

C:\WINDOWS\system32\cmd.exe
C:\cs>Program.exe
Przypisana wartość jest ujemna i równa -10.
--
Przypisana wartość jest ujemna i równa -20.
Przypisano ujemną wartość = -20.
C:\cs>_

```

Wiele zdarzeń w jednej klasie

Zdarzenia należy traktować tak jak inne składowe klasy, nic nie stoi więc na przeszkodzie, aby zawierała ich ona kilka, kilkadesiąt czy nawet kilkaset — nie ma pod tym względem ograniczeń. W praktyce najczęściej mamy do czynienia z liczbą od kilku do kilkunastu. Wykonajmy więc przykład, w którym klasa `Kontener` będzie zawierała trzy zdarzenia. Pierwsze informujące o tym, że argument przypisywany polu `x` jest mniejszy od zera, drugie — że jest większy, i trzecie — że jest równy zero. Nawiązemy je, jakżeby inaczej, `OnUjemne`, `OnDodatnie` i `OnZero`.

Zastanówmy się teraz, ile typów delegacji będzie nam w związku z tym potrzebnych? Oczywiście wystarczy tylko jeden. Ponieważ każde zdarzenie można obsługiwać przez metody o takich samych typach, będzie potrzebny też jeden typ delegacji. Deklaracja przyjmie postać:

```
public delegate void OnPrzypisanieEventDelegate(Kontener obj);
```

Jak widać, w stosunku do poprzedniego przykładu została zmieniona tylko nazwa.

Kod najnowszej wersji klasy Kontener został przedstawiony na listingu 7.21.

Listing 7.21. Klasa z kilkoma zdarzeniami

```
public delegate void OnPrzypisanieEventDelegate(Kontener obj);

public class Kontener
{
    private int x;
    public event OnPrzypisanieEventDelegate OnUjemne;
    public event OnPrzypisanieEventDelegate OnDodatnie;
    public event OnPrzypisanieEventDelegate OnZero;
    public int getX()
    {
        return x;
    }
    public void setX(int arg)
    {
        x = arg;
        if(arg < 0)
        {
            if(OnUjemne != null)
            {
                OnUjemne(this);
            }
        }
        else if(arg > 0)
        {
            if(OnDodatnie != null)
            {
                OnDodatnie(this);
            }
        }
        else
        {
            if(OnZero != null)
            {
                OnZero(this);
            }
        }
    }
}
```

We wnętrzu klasy zostały zdefiniowane trzy zdarzenia: OnUjemne, OnDodatnie i OnZero, wszystkie typu delegacji OnPrzypisanieEventDelegate. Tak więc każdemu z nich będzie mogła być przypisana dowolna metoda o typie zwracanym `void`, przyjmująca jeden

argument typu Kontener. Ponieważ każde ze zdarzeń odpowiada innemu stanowi argumentu arg metody setX, w jej wnętrzu została umieszczona złożona instrukcja warunkowa if else. Gdy arg jest mniejsze od zera, wywoływanie jest zdarzenie (a tym samym delegacja) OnUjemne:

```
OnUjemne(this);  
gdy jest większe od zera — zdarzenie OnDodatnie:
```

```
OnDodatnie(this);  
a w pozostałym przypadku (czyli gdy arg jest równe 0) — zdarzenie OnZero:  
OnZero(this);
```

Przed każdym takim wywołaniem jest też sprawdzane, czy dany obiekt zdarzenia istnieje, czyli czy jest ono różne od null.

Użyjmy więc obiektu klasy Kontener i wykorzystajmy wszystkie trzy zdarzenia. Odpowiedni kod został zaprezentowany na listingu 7.22.

Listing 7.22. Obsługa kilku zdarzeń

```
using System;  
  
public class Program  
{  
    public static void OnUjemneKomunikat(Kontener obj)  
    {  
        Console.WriteLine(  
            "Przypisano ujemną wartość = {0}.", obj.getX());  
    }  
    public static void OnDodatnieKomunikat(Kontener obj)  
    {  
        Console.WriteLine(  
            "Przypisano dodatnią wartość = {0}.", obj.getX());  
    }  
    public static void OnZeroKomunikat(Kontener obj)  
    {  
        Console.WriteLine("Przypisano wartość = 0.");  
    }  
    public static void Main()  
    {  
        Kontener k = new Kontener();  
        k.OnUjemne += OnUjemneKomunikat;  
        k.OnDodatnie += OnDodatnieKomunikat;  
        k.OnZero += OnZeroKomunikat;  
  
        k.setX(10);  
        k.setX(0);  
        k.setX(-10);  
    }  
}
```

W programie zostały zdefiniowane trzy metody, które będą reagowały na zdarzenia:

- ◆ `OnUjemneKomunikat` — dla zdarzenia `OnUjemne`;
- ◆ `OnDodatnieKomunikat` — dla zdarzenia `OnDodatnie`;
- ◆ `OnZeroKomunikat` — dla zdarzenia `OnZero`.

Zadaniem każdej z nich jest wyświetlenie odpowiedniego komunikatu wraz ze stanem pola `x`. Odczytanie wartości tego pola odbywa się przez wywołanie metody `getX` obiektu przekazanego w postaci argumentu. Taką samą technikę zastosowaliśmy w przypadku programu z listingu 7.20. Oczywiście w przypadku metody `OnZeroKomunikat` nie ma potrzeby odczytywania stanu pola `x`, gdyż wiadomo, że jest to 0.

W metodzie `Main` najpierw tworzony jest obiekt typu `Kontener`, a następnie każdemu z jego zdarzeń przypisywana jest odpowiadająca temu zdarzeniu metoda:

```
k.OnUjemne += OnUjemneKomunikat;  
k.OnDodatnie += OnDodatnieKomunikat;  
k.OnZero += OnZeroKomunikat;
```

Po wykonaniu tych czynności trzykrotnie wywoływana jest metoda `setX`, ustawiająca wartość pola `x` na 10, 0 i -10, tak aby można było obserwować powstawanie kolejnych zdarzeń. Efekt działania programu jest widoczny na rysunku 7.11.

Rysunek 7.11.

Efekt obsługi kilku zdarzeń

```
C:\>Program.exe  
Przypisano dodatnią wartość = 10.  
Przypisano wartość = 0.  
Przypisano ujemną wartość = -10.  
C:\>-
```

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 36.1

Do programu z listingu 7.7 dopisz taką delegację, aby za jej pośrednictwem mogła zostać wywołana metoda `Metoda2`.

Ćwiczenie 36.2

Napisz metodę przyjmującą dwa argumenty typu `double`, zwracającą wynik ich odejmowania w postaci ciągu znaków, oraz odpowiadającą jej delegację. Wywołaj tę metodę za pośrednictwem delegacji.

Ćwiczenie 36.3

Zmień kod z listingu 7.14 w taki sposób, aby był w nim używany tylko jeden obiekt delegacji.

Ćwiczenie 36.4

Napisz klasę zawierającą metodę przyjmującą argument typu `string`. Gdyby otrzymany argument mógł być przekonwertowany na wartość całkowitą, powinno zostać wygenerowane zdarzenie `OnCalkowita`, a w przypadku możliwości konwersji na wartość rzeczywistą — zdarzenie `OnRzeczywista`.

Ćwiczenie 36.5

Napisz program testujący kod klasy z ćwiczenia 36.4.

Lekcja 37. Komponenty graficzne

Każda aplikacja okienkowa oprócz menu, które poznaliśmy już w lekcji 33., jest także wyposażona w wiele innych elementów graficznych, takich jak przyciski, etykiety, pola tekstowe czy listy rozwijane³. W .NET znajduje się oczywiście odpowiedni zestaw klas, które pozwalają na zastosowanie tego rodzaju komponentów, nazywanych również kontrolkami (ang. *controls*). Większość z nich jest zdefiniowana w przestrzeni nazw `System.Windows.Forms`. Jest ich bardzo wiele, część z nich będzie przedstawiona w ostatniej, 37. lekcji. Zostanie także wyjaśnione, w jaki sposób wyświetlać okna dialogowe z komunikatami tekstowymi.

Wyświetlanie komunikatów

W trakcie działania aplikacji często zachodzi potrzeba wyświetlenia informacji dla użytkownika. W przypadku programów pracujących w trybie tekstowym treść komunikatu mogła być prezentowana po prostu w konsoli. Aplikacje okienkowe oczywiście nie mogą działać w taki sposób, ważne informacje prezentuje się więc za pomocą okien dialogowych. Na szczęście nie trzeba tworzyć ich samodzielnie (choć nic nie stoi na przeszkodzie) — proste komunikaty wyświetlimy za pomocą predefiniowanych klas i metod.

Jedną z takich klas jest `MessageBox`, udostępniająca metodę `Show`. Jako argument tej metody należy podać tekst, który ma się pojawić na ekranie. Prosty program wyświetlający okno dialogowe widoczne na rysunku 7.12 został zaprezentowany na listingu 7.23.

Listing 7.23. Wyświetlenie okna dialogowego

```
using System.Windows.Forms;

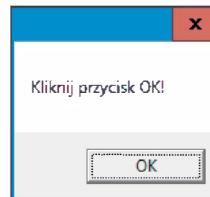
public class Program
{
    public static void Main()
    {
```

³ Często spotyka się też termin „lista rozwijalna”.

```
        MessageBox.Show("Kliknij przycisk OK!");  
    }  
}
```

Rysunek 7.12.

Okno dialogowe
wyświetlone
za pomocą
metody Show



Klasa `MessageBox` jest zdefiniowana w przestrzeni nazw `System.Windows.Forms`, dlatego też na początku kodu znajduje się odpowiednia dyrektywa `using`. Metoda `Show` jest statyczna, tak więc do jej wywołania nie jest konieczne tworzenie instancji obiektu typu `MessageBox`. Przedstawiony kod może być skompilowany jako aplikacja konsolowa bądź okienkowa (z przełącznikiem kompilatora `/t:winexe`). W obu przypadkach postać okna będzie taka sama.

Gdybyśmy chcieli, aby okno (widoczne na rysunku 7.12) miało jakiś napis na pasku tytułu, należy użyć drugiego argumentu metody `Show`, schematycznie:

```
MessageBox.Show("tekst", "tytuł_okna");
```

Tej wersji użyjemy już w kolejnym przykładzie.

Obsługa zdarzeń

Lekcja 36. poświęcona była obsłudze zdarzeń i delegacji. Zostały w niej przedstawione informacje niezbędne do sprawnej obsługi graficznego interfejsu użytkownika. Jak bowiem reagować np. na kliknięcie przycisku? Oczywiście za pomocą procedury obsługi odpowiedniego zdarzenia. Otóż każda klasa opisująca dany komponent (kontrolkę), np. menu, przycisk, listę rozwijaną itp., ma zdefiniowany zestaw zdarzeń. Taki typowym zdarzeniem jest np. `Click`, powstające wtedy, kiedy użytkownik kliknie dany komponent. Jeśli powiążemy takie zdarzenie z odpowiednią metodą, kod tej metody zostanie wykonany.

Większość typowych zdarzeń jest obsługiwana w standardowy sposób, ich typem jest delegacja o nazwie `EventHandler`, zdefiniowana w przestrzeni nazw `System`. Ma ona postać:

```
public delegate void EventHandler(Object sender, EventArgs ea);
```

Odpowiadają więc jej wszystkie metody o typie zwracanym `void`, przyjmujące dwa argumenty: pierwszy typu `Object`, drugi typu `EventArgs`. Pierwszy argument zawiera referencję do obiektu, z którego pochodzi zdarzenie (np. kiedy kliknięty został przycisk, a zdarzeniem jest `Click`, będzie to referencja do tego przycisku), a drugi — obiekt typu `EventArgs` zawierający dodatkowe parametry zdarzenia. W tej lekcji nie będziemy jednak z nich (argumentów) korzystać.

Zanim jednak przystąpimy do obsługi zdarzeń związanych z komponentami, przyjrzymy się jednemu, związanemu z samą aplikacją. Jak wiadomo, uruchomienie aplikacji okienkowej wymaga użycia metody Run klasy Application. Ta klasa posiada również m.in. statyczne zdarzenie o nazwie ApplicationExit. Jest ono wywoływanie, gdy program kończy swoje działanie. Napiszmy więc taką aplikację okienkową, która używa tego zdarzenia i podczas zamknięcia wyświetli okno dialogowe. Tak działający kod został zaprezentowany na listingu 7.24.

Listing 7.24. Użycie zdarzenia ApplicationExit

```
using System;
using System.Windows.Forms;

public class MainForm : Form
{
    public MainForm()
    {
        Application.ApplicationExit += new EventHandler(OnExit);
    }
    private void OnExit(Object sender, EventArgs ea)
    {
        MessageBox.Show("No cóż, ta aplikacja kończy działanie!",
                       "Uwaga!");
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Ogólna struktura tej aplikacji jest taka sama jak w przypadku przykładów z lekcji 35. Jest to klasa `MainForm` pochodna od `Form`, zawierająca statyczną metodę `Main`, w której jest wywoływana instrukcja:

```
Application.Run(new MainForm());
```

powodująca wyświetlenie okna (formatki) i rozpoczęcie pracy w trybie graficznym.

Dodatkowo w klasie zdefiniowano konstruktor oraz metodę `OnExit`. W konstruktorze zdarzeniu `ApplicationExit` za pomocą operatora `+=` została przypisana nowa delegacja powiązana z metodą `OnExit`. Użyta instrukcja jest nieco inna niż w przykładach z lekcji 36., gdyż został użyty operator `new`. Cała instrukcja:

```
Application.ApplicationExit += new EventHandler(OnExit);
```

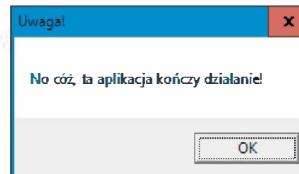
oznacza: „Utwórz nowy obiekt delegacji typu `EventHandler`, dodaj do niego metodę `OnExit` i przypisz go do zdarzenia `ApplicationExit`”. Taka postać jest kompatybilna z wersjami C# i .NET poniżej 2.0. Gdybyśmy chcieli zamiast takiej konstrukcji użyć sposobu z lekcji 36., należałoby zastosować instrukcję:

```
Application.ApplicationExit += OnExit;
```

Znaczenie jest takie samo, jest to jedynie skrócona i wygodniejsza forma zapisu.

Co się natomiast dzieje w metodzie `OnExit`? Jest w niej zawarta tylko jedna instrukcja — wywołanie metody `Show` klasy `MessageBox`, powodującej wyświetlenie na ekranie okna dialogowego z komunikatem. Korzystamy z dwuargumentowej wersji tej metody, dzięki czemu okno będzie miało również napis na pasku tytułu, tak jak jest to widoczne na rysunku 7.13. Oczywiście, okno to pojawi się podczas zamknięcia aplikacji.

Rysunek 7.13.
Okno pojawiające się podczas kończenia pracy aplikacji



Menu

W lekcji 33. wyjaśniono, w jaki sposób tworzy się i dodaje do okna aplikacji menu. Niestety, powstałe menu nie reagowało na wybór pozycji, nie był jeszcze wtedy omówiony temat zdarzeń i delegacji. Teraz, kiedy zaprezentowany został już materiał z lekcji 36., można nadrobić zaległości. Napiszemy zatem aplikację okienkową zawierającą menu główne z pozycją *Plik* i dwiema podpozycjami, tak jak jest to widoczne na rysunku 7.14. Wybór pierwszej z nich będzie powodował wyświetlenie jej nazwy w osobnym oknie dialogowym, a drugiej — zakończenie pracy programu. Działający w opisany sposób kod znajduje się na listingu 7.25.

Listing 7.25. Menu reagujące na wybór pozycji

```
using System;
using System.Windows.Forms;

public class MainForm : Form
{
    private void OnWyswietlKomunikat(object sender, EventArgs ea)
    {
        MessageBox.Show(
            "Została wybrana pozycja: Wyświetl komunikat", "Komunikat");
    }
    private void OnWyjdź(object sender, EventArgs ea)
    {
        Application.Exit();
    }

    public MainForm()
    {
        Text = "Moja aplikacja";
        Width = 320;
        Height = 200;

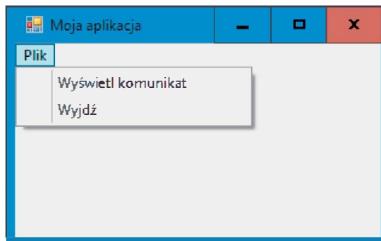
        MainMenu mm = new MainMenu();
        MenuItem miPlik = new MenuItem("Plik");

        MenuItem miWyswietlKomunikat =
            new MenuItem("Wyświetl komunikat");
        MenuItem miWyjdź = new MenuItem("Wyjdź");
    }
}
```

```
    miWyswietlKomunikat.Click +=  
        new EventHandler(OnWyswietlKomunikat);  
  
    miWyjdz.Click += new EventHandler(OnWyjdz);  
  
    miPlik.MenuItems.Add(miWyswietlKomunikat);  
    miPlik.MenuItems.Add(miWyjdz);  
  
    mm.MenuItems.Add(miPlik);  
    Menu = mm;  
}  
public static void Main()  
{  
    Application.Run(new MainForm());  
}
```

Rysunek 7.14.

Menu z dwoma podpozycjami



Sposób budowania menu jest taki sam jak w przypadku przykładu z listingu 7.6. Najpierw tworzone są obiekty: `mm` typu `MainMenu` oraz `miPlik` typu `MenuItem`. Odzwierciedlają one menu główne (`mm`) oraz jego jedyną pozycję o nazwie *Plik* (`miPlik`). Następnie tworzone są dwie podpozycje: `miWyswietlKomunikat` i `miWyjdz`. Jak widać, nazwy tworzone są z połączenia skrótu nazwy klasy (`mm` = `MainMenu`, `mi` = `MenuItem`) oraz nazwy danej pozycji menu.

W klasie `MenuItem` znajduje się zdarzenie `Click`, które jest wywoływanie, kiedy dana pozycja menu zostanie wybrana (kliknięta) przez użytkownika. Wynika z tego, że aby wykonać jakiś kod powiązany z takim kliknięciem, każdej pozycji należy przypisać (poprzez delegację) odpowiednią procedurę obsługi. Dla pozycji `miWyswietlKomunikat` będzie to metoda `OnWyswietlKomunikat`, a dla pozycji `miWyjdz` — metoda `OnWyjdz`. Dlatego też po utworzeniu obiektów menu następuje powiązanie zdarzeń i metod:

```
    miWyswietlKomunikat.Click +=  
        new EventHandler(onWyswietlKomunikat);  
    miWyjdz.Click += new EventHandler(onWyjdz);
```

Zadaniem metody `OnWyswietlKomunikat` jest wyświetlenie informacji o tym, która pozycja została wybrana. Jest to wykonywane przez wywołanie metody `Show` klasy `MessageBox`. Zadaniem metody `OnWyjdz` jest z kolei zakończenie pracy aplikacji, zatem jedyną jej instrukcją jest wywołanie metody `Exit` klasy `Application`, powodującej wyjście z programu. To wszystko. Żadne dodatkowe czynności nie są potrzebne. Tak przygotowane menu będzie już reagowało na polecenia użytkownika.

Etykiety

Etykiety tekstowe to jedne z najprostszych komponentów graficznych. Umożliwiają one wyświetlanie tekstu. Aby utworzyć etykietę, należy skorzystać z klasy `Label`. Konstruktor klasy `Label` jest bezargumentowy i tworzy pustą etykietę. Po utworzeniu etykiety można jej przypisać tekst, modyfikując właściwość o nazwie `Text`. Etykiętę umieszczamy w oknie, wywołując metodę `Add` właściwości `Controls`. Wybrane właściwości klasy `Label` zostały zebrane w tabeli 7.3. Prosty przykład obrazujący użycie etykiety jest widoczny na listingu 7.26, natomiast efekt jego działania na rysunku 7.15.

Tabela 7.3. Wybrane właściwości klasy `Label`

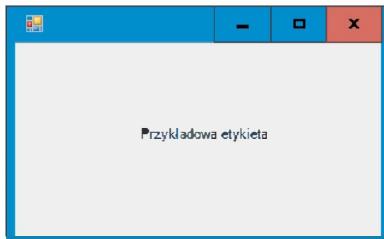
Typ	Nazwa właściwości	Znaczenie
<code>bool</code>	<code>AutoSize</code>	Określa, czy etykieta ma automatycznie dopasowywać swój rozmiar do zawartego na niej tekstu.
<code>Color</code>	<code>BackColor</code>	Określa kolor tła etykiety.
<code>BorderStyle</code>	<code>BorderStyle</code>	Określa styl ramki otaczającej etykietę.
<code>Bounds</code>	<code>Bounds</code>	Określa rozmiar oraz położenie etykiety.
<code>Cursor</code>	<code>Cursor</code>	Określa rodzaj kurSORA wyświetlanego, kiedy wskaźnik myszy znajdzie się nad etykietą.
<code>Font</code>	<code>Font</code>	Określa rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się na etykiecie.
<code>Color</code>	<code>ForeColor</code>	Określa kolor tekstu etykiety.
<code>int</code>	<code>Height</code>	Określa wysokość etykiety.
<code>Image</code>	<code>Image</code>	Obraz wyświetlany na etykiecie.
<code>int</code>	<code>Left</code>	Określa położenie lewego górnego rogu w poziomie, w pikselach.
<code>Point</code>	<code>Location</code>	Określa współrzędne lewego górnego rogu etykiety.
<code>string</code>	<code>Name</code>	Nazwa etykiety.
<code>Control</code>	<code>Parent</code>	Referencja do obiektu zawierającego etykietę (nadziednego).
<code>Size</code>	<code>Size</code>	Określa wysokość i szerokość etykiety.
<code>string</code>	<code>Text</code>	Określa tekst wyświetlany na etykiecie.
<code>ContentAlignment</code>	<code>TextAlign</code>	Określa położenie tekstu na etykiecie.
<code>int</code>	<code>Top</code>	Określa położenie lewego górnego rogu w pionie, w pikselach.
<code>bool</code>	<code>Visible</code>	Określa, czy etykieta ma być widoczna.
<code>int</code>	<code>Width</code>	Określa rozmiar etykiety w poziomie.

Listing 7.26. Umieszczenie etykiety w oknie aplikacji

```
using System.Windows.Forms;

public class MainForm : Form
{
    private Label label = new Label();
    public MainForm()
    {
        Width = 320;
        Height = 200;
        label.Text = "Przykładowa etykieta";
        label.AutoSize = true;
        label.Left = (ClientSize.Width - label.Width) / 2;
        label.Top = (ClientSize.Height - label.Height) / 2;
        Controls.Add(label);
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Rysunek 7.15.
Okno zawierające
etykię



Etykieta została utworzona jako prywatna składowa klasy `MainForm`. Jej właściwość `AutoSize` została ustawiona na `true`, dzięki czemu etykieta będzie zmieniała automatycznie swój rozmiar, dopasowując go do przypisywanego jej tekstu.

Jak widać, na rysunku tekst etykiety znajduje się w centrum formy (okna). Takie umiejscowienie obiektu uzyskujemy poprzez modyfikację właściwości `Top` oraz `Left`. Aby uzyskać odpowiednie wartości, wykonujemy tu proste działania matematyczne:

$$\begin{aligned} \text{Współrzędna X} &= (\text{długość okna} - \text{długość etykiety}) / 2 \\ \text{Współrzędna Y} &= (\text{wysokość okna} - \text{wysokość etykiety}) / 2 \end{aligned}$$

Oczywiście działania te są wykonywane po przypisaniu etykiecie tekstu, inaczej obliczenia nie uwzględniałyby jej prawidłowej wielkości. Do uzyskania szerokości, a szczególnie wysokości formy używamy właściwości `ClientWidth` (szerokość) oraz `ClientHeight` (wysokość). Podaję one bowiem rozmiar okna po odliczeniu okalającej ramki oraz paska tytułowego i ewentualnego menu, czyli po prostu wielkość okna, którą mamy do naszej dyspozycji i umieszczania w oknie innych obiektów. Po dokonaniu wszystkich obliczeń i przypisów etykieta jest dodawana do formy za pomocą instrukcji:

```
Controls.Add(label);
```

Przyciski

Obsługą i wyświetlaniem przycisków zajmuje się klasa `Button`. Podobnie jak w przypadku klasy `Label`, konstruktor jest bezargumentowy i w wyniku jego działania powstaje przycisk bez napisu na jego powierzchni, a zmiana tekstu na przycisku następuje poprzez modyfikację właściwości `Text`. W odróżnieniu od etykiet przyciski powinny jednak reagować na kliknięcia myszą, przy ich stosowaniu niezbędne będzie zatem użycie zdarzenia `Click`, które oczywiście w klasie `Button` jest zdefiniowane. Wybrane właściwości tej klasy zostały zebrane w tabeli 7.4. Przykładowa aplikacja zawierająca przycisk, którego kliknięcie spowoduje wyświetlenie komunikatu, jest widoczna na listingu 7.27, natomiast jej wygląd zobrazowano na rysunku 7.16.

Tabela 7.4. Wybrane właściwości klasy `Button`

Typ	Nazwa właściwości	Znaczenie
<code>Color</code>	<code>BackColor</code>	Określa kolor tła przycisku.
<code>Bounds</code>	<code>Bounds</code>	Określa rozmiar oraz położenie przycisku.
<code>Cursor</code>	<code>Cursor</code>	Określa rodzaj kurSORA wyświetlanego, kiedy wskaźnik myszy znajdzie się nad przyciskiem.
<code>FlatStyle</code>	<code>FlatStyle</code>	Modyfikuje styl przycisku.
<code>Font</code>	<code>Font</code>	Określa rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się na przycisku.
<code>Color</code>	<code>ForeColor</code>	Określa kolor tekstu przycisku.
<code>int</code>	<code>Height</code>	Określa wysokość przycisku.
<code>Image</code>	<code>Image</code>	Obraz wyświetlany na przycisku.
<code>int</code>	<code>Left</code>	Określa położenie lewego górnego rogu w poziomie, w pikselach.
<code>Point</code>	<code>Location</code>	Określa współrzędne lewego górnego rogu przycisku.
<code>string</code>	<code>Name</code>	Nazwa przycisku.
<code>Control</code>	<code>Parent</code>	Referencja do obiektu zawierającego przycisk (obiektu nadrzędnego).
<code>Size</code>	<code>Size</code>	Określa wysokość i szerokość przycisku.
<code>string</code>	<code>Text</code>	Tekst wyświetlany na przycisku.
<code>ContentAlignment</code>	<code>TextAlign</code>	Określa położenie tekstu na przycisku.
<code>int</code>	<code>Top</code>	Określa położenie lewego górnego rogu w pionie, w pikselach.
<code>bool</code>	<code>Visible</code>	Określa, czy przycisk ma być widoczny.
<code>int</code>	<code>Width</code>	Określa rozmiar przycisku w poziomie, w pikselach.

Listing 7.27. Aplikacja zawierająca przycisk

```
using System;
using System.Windows.Forms;

public class MainForm:Form
{
```

```
private Button button1 = new Button();

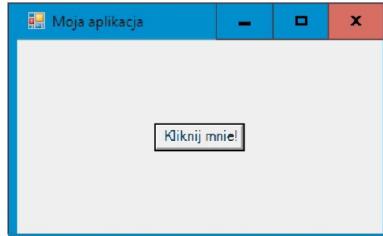
private void OnButton1Click(object sender, EventArgs ea)
{
    MessageBox.Show("Przycisk został kliknięty!",
                    "Komunikat");
}

public MainForm()
{
    Width = 320;
    Height = 200;
    Text = "Moja aplikacja";
    button1.Text = "Kliknij mnie!";
    button1.Left = (ClientSize.Width - button1.Width) / 2;
    button1.Top = (ClientSize.Height - button1.Height) / 2;

    button1.Click += new EventHandler(OnButton1Click);

    Controls.Add(button1);
}
public static void Main()
{
    Application.Run(new MainForm());
}
```

Rysunek 7.16.
Wygląd okna
aplikacji
z listingu 7.27



Schemat budowy aplikacji z listingu 7.27 jest taki sam jak w przypadku poprzedniego przykładu. Przycisk button1 został zdefiniowany jako prywatna składowa klasy MainForm, a jego właściwości zostały określone w konstruktorze. Najpierw został przypisany napis, który ma się znaleźć na przycisku, a następnie zostało zdefiniowane jego położenie. W tym celu użyto takich samych wzorów jak w przypadku etykiety z listingu 7.26. W taki sam sposób komponent ten został też dodany do okna aplikacji; odpowiada za to instrukcja:

```
Controls.Add(button1);
```

Procedurą obsługi zdarzenia Click jest metoda OnButton1Click. Powiązanie zdarzenia i metody (za pośrednictwem obiektu delegacji EventHandler) następuje dzięki instrukcji:

```
button1.Click += new EventHandler(OnButton1Click);
```

W metodzie OnButton1Click za pomocą instrukcji MessageBox.Show wyświetlany jest komunikat informacyjny.

Pola tekstowe

Pola tekstowe definiowane są przez klasę `TextBox`. Dysponuje ona bezargumentowym konstruktorem oraz sporym zestawem właściwości. Wybrane z nich zostały przedstawione w tabeli 7.5. Dostęp do tekstu zawartego w polu otrzymujemy przez odwołanie się do właściwości o nazwie `Text`. Przykład prostego sposobu użycia pola tekstowego jest widoczny na listingu 7.28.

Tabela 7.5. Wybrane właściwości klasy `TextBox`

Typ	Nazwa właściwości	Znaczenie
<code>bool</code>	<code>AutoSize</code>	Określa, czy pole tekstowe ma automatycznie dopasowywać swój rozmiar do zawartego w nim tekstu.
<code>Color</code>	<code>BackColor</code>	Określa kolor tła pola tekstowego.
<code>Image</code>	<code>BackgroundImage</code>	Obraz znajdujący się w tle okna tekstowego.
<code>BorderStyle</code>	<code>BorderStyle</code>	Określa styl ramki otaczającej pole tekstowe.
<code>Bounds</code>	<code>Bounds</code>	Określa rozmiar oraz położenie pola tekstowego.
<code>Cursor</code>	<code>Cursor</code>	Rodzaj kurSORA wyświetlonego, kiedy wskaźnik myszy znajdzie się nad polem tekstowym.
<code>Font</code>	<code>Font</code>	Określa rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się w polu.
<code>Color</code>	<code>ForeColor</code>	Określa kolor tekstu pola tekstowego.
<code>int</code>	<code>Height</code>	Określa wysokość pola tekstowego.
<code>int</code>	<code>Left</code>	Określa położenie lewego górnego rogu w poziomie, w pikselach.
<code>string[]</code>	<code>Lines</code>	Tablica zawierająca poszczególne linie tekstu zawarte w polu tekstowym.
<code>Point</code>	<code>Location</code>	Określa współrzędne lewego górnego rogu pola tekstowego.
<code>int</code>	<code>MaxLength</code>	Maksymalna liczba znaków, które można wprowadzić do pola tekstowego.
<code>bool</code>	<code>Modified</code>	Określa, czy zawartość pola tekstowego była modyfikowana.
<code>bool</code>	<code>Multiline</code>	Określa, czy pole tekstowe ma zawierać jedną, czy wiele linii tekstu.
<code>string</code>	<code>Name</code>	Nazwa pola tekstowego.
<code>Control</code>	<code>Parent</code>	Referencja do obiektu zawierającego pole tekstowe (obiektu nadrzędnego).
<code>char</code>	<code>PasswordChar</code>	Określa, jaki znak będzie wyświetlany w polu tekstowym w celu zamaskowania wprowadzanego tekstu; aby skorzystać z tej opcji, właściwość <code>Multiline</code> musi być ustawiona na <code>false</code> .
<code>bool</code>	<code>ReadOnly</code>	Określa, czy pole tekstowe ma być ustawione w trybie tylko do odczytu.

Tabela 7.5. Wybrane właściwości klasy *TextBox* — ciąg dalszy

Typ	Nazwa właściwości	Znaczenie
string	SelectedText	Zaznaczony fragment tekstu w polu tekstowym.
int	SelectionLength	Liczba znaków w zaznaczonym fragmencie tekstu.
int	SelectionStart	Indeks pierwszego znaku zaznaczonego fragmentu tekstu.
Size	Size	Okręsła rozmiar pola tekstowego.
string	Text	Tekst wyświetlany w polu tekstowym.
ContentAlignment	TextAlign	Okręsła położenie tekstu w polu tekstowym.
int	Top	Okręsła położenie lewego górnego rogu w pionie, w pikselach.
bool	Visible	Okręsła, czy pole tekstowe ma być widoczne.
int	Width	Okręsła rozmiar pola tekstowego w poziomie.
bool	WordWrap	Okręsła, czy słowa mają być automatycznie przenoszone do nowej linii, kiedy nie mieścią się w bieżącej; aby zastosować tę funkcję, właściwość <i>Multiline</i> musi być ustawiona na <i>true</i> .

Listing 7.28. Użycie pola tekstowego

```

using System;
using System.Drawing;
using System.Windows.Forms;

public class MainForm:Form
{
    private Button button1 = new Button();
    private Label label1 = new Label();
    private TextBox textBox1 = new TextBox();

    private void OnButton1Click(object sender, EventArgs ea)
    {
        label1.Text = textBox1.Text;
    }

    public MainForm()
    {
        Width = 320;
        Height = 200;
        Text = "Moja aplikacja";

        label1.Text = "Tekst etykiety";
        label1.Top = 30;
        label1.Left = (ClientSize.Width - label1.Width) / 2;
        label1.TextAlign = ContentAlignment.MiddleCenter;

        button1.Text = "Kliknij tu!";
        button1.Left = (ClientSize.Width - button1.Width) / 2;
        button1.Top = 120;

        button1.Click += new EventHandler(OnButton1Click);
    }
}

```

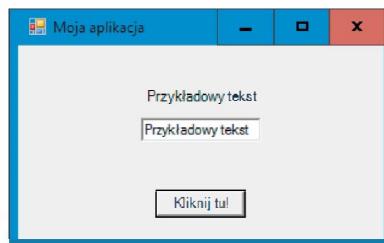
```
textBox1.Top = 60;
textBox1.Left = (ClientSize.Width - textBox1.Width) / 2;

Controls.Add(label1);
Controls.Add(button1);
Controls.Add(textBox1);
}
public static void Main()
{
    Application.Run(new MainForm());
}
}
```

Program ten umieszcza w oknie aplikacji etykietę, pole tekstowe oraz przycisk, tak jak jest to widoczne na rysunku 7.17. Po kliknięciu przycisku tekst wpisany do pola jest przypisywany etykiecie. Korzystamy tu oczywiście ze zdarzenia Click klasy Button. W konstruktorze klasy MainForm są tworzone wszystkie trzy kontrolki oraz są im przypisywane właściwości. Położenie w pionie (właściwość Top) jest ustalane arbitralnie, natomiast położenie w poziomie jest dopasowywane automatycznie do początkowych rozmiarów okna — korzystamy w tym celu z takich samych wzorów jak we wcześniejszych przykładach. Do przycisku jest również przypisywana procedura obsługi zdarzenia, którą jest metoda OnButton1Click:

```
button1.Click += new EventHandler(OnButton1Click);
```

Rysunek 7.17.
Pole tekstowe
w oknie aplikacji



Należy też zwrócić w tym miejscu uwagę na użycie właściwości TextAlign etykiety label1 ustalającej sposób wyrównywania tekstu. Właściwość ta jest typu wyliczeniowego ContentAlignment, którego składowe zostały przedstawione w tabeli 7.6. Instrukcja:

```
label1.TextAlign = ContentAlignment.MiddleCenter;
```

powoduje zatem, że tekst będzie wyśrodkowany w poziomie i w pionie. Typ ContentAlignment jest zdefiniowany w przestrzeni nazw System.Drawing (stąd też odpowiednia dyrektywa using znajdująca się na początku kodu).

Tabela 7.6. Składowe typu wyliczeniowego ContentAlignment

Składowa	Znaczenie
BottomCenter	Wyrównywanie w pionie w dół i w poziomie do środka.
BottomLeft	Wyrównywanie w pionie w dół i w poziomie do lewej.
BottomRight	Wyrównywanie w pionie w dół i w poziomie do prawej.
MiddleCenter	Wyrównywanie w pionie do środka i w poziomie do środka.

Tabela 7.6. Składowe typu wyliczeniowego *ContentAlignment* — ciąg dalszy

Składowa	Znaczenie
MiddleLeft	Wyrównywanie w pionie do środka i w poziomie do lewej.
MiddleRight	Wyrównywanie w pionie do środka i w poziomie do prawej.
TopCenter	Wyrównywanie w pionie do góry i w poziomie do środka.
TopLeft	Wyrównywanie w pionie do góry i w poziomie do lewej.
TopRight	Wyrównywanie w pionie do góry i w poziomie do prawej.

Metoda `OnButton1Click` wykonywana po kliknięciu przycisku `button1` jest bardzo prosta i zawiera tylko jedną instrukcję:

```
label1.Text = textBox1.Text;
```

Jest to przypisanie właściwości `Text` pola tekstowego `textBox1` właściwości `Text` etykiety `label1`. Tak więc po jej wykonaniu tekst znajdujący się w polu zostanie umieszczony na etykiecie.

Listy rozwijane

Listy rozwijane można tworzyć dzięki klasie `ComboBox`. Listę jej wybranych właściwości przedstawiono w tabeli 7.7. Dla nas najważniejsza będzie w tej chwili właściwość `Items`, jako że zawiera ona wszystkie elementy znajdujące się na liście. Właściwość ta jest w rzeczywistości kolekcją elementów typu `object`⁴. Dodawanie elementów można zatem zrealizować, stosując konstrukcję:

```
comboBox.Items.Add("element");
```

natomiast ich usuwanie, wykorzystując:

```
comboBox.Items.Remove("element");
```

gdzie `comboBox` jest referencją do obiektu typu `ComboBox`.

Jeżeli chcemy jednak dodać większą liczbę elementów naraz, najwygodniej jest zastosować metodę `AddRange` w postaci:

```
comboBox.Items.AddRange
(
    new[] object{
        "Element 1"
        "Element 2"
        ...
        "Element n"
    }
);
```

⁴ W rzeczywistości jest to właściwość typu `ObjectCollection`, implementującego interfejsy `IList`, `ICollection` i `INumerable`, jednak dokładne omówienie tego tematu wykracza poza ramy niniejszej publikacji.

Wybranie przez użytkownika elementu z listy wykrywa się dzięki zdarzeniu o nazwie SelectedIndexChanged. Odniesienie do wybranego elementu znajdziemy natomiast we właściwości SelectedItem. Te wiadomości w zupełności wystarczają do napisania prostego programu ilustrującego działanie tej kontrolki. Taki przykład został zaprezentowany na listingu 7.29.

Tabela 7.7. Wybrane właściwości klasy ComboBox

Typ	Nazwa właściwości	Znaczenie
Color	BackColor	Określa kolor tła listy.
Bounds	Bounds	Określa rozmiar oraz położenie listy.
Cursor	Cursor	Określa rodzaj kurSORA wyświetlanego, kiedy wskaźnik myszy znajdzie się nad listą.
int	DropDownWidth	Określa szerokość rozwijanej części kontrolki.
Font	Font	Określa rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się na liście.
Color	ForeColor	Określa kolor tekstu.
int	Height	Określa wysokość listy.
int	ItemHeight	Określa wysokość pojedynczego elementu listy.
ObjectCollection	Items	Lista elementów znajdujących się na liście.
int	Left	Określa położenie lewego górnego rogu w poziomie, w pikselach.
Point	Location	Określa współrzędne lewego górnego rogu listy.
int	MaxDropDownItems	Maksymalna liczba elementów, które będą wyświetlane po rozwinięciu listy.
int	MaxLength	Maksymalna liczba znaków wyświetlanych w polu edycyjnym listy.
string	Name	Nazwa listy.
Control	Parent	Referencja do obiektu zawierającego listę.
int	SelectedIndex	Indeks aktualnie zaznaczonego elementu.
object	SelectedItem	Aktualnie zaznaczony element.
Size	Size	Określa wysokość i szerokość listy.
bool	Sorted	Określa, czy elementy listy mają być posortowane.
string	Text	Tekst wyświetlany w polu edycyjnym listy.
int	Top	Określa położenie lewego górnego rogu w pionie, w pikselach.
bool	Visible	Określa, czy lista ma być widoczna.
int	Width	Określa rozmiar listy w poziomie.

Listing 7.29. Ilustracja działania listy rozwijanej

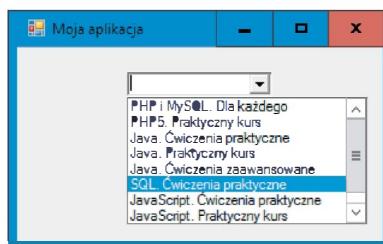
```
using System;
using System.Windows.Forms;

public class MainForm : Form
```

```
{  
    private ComboBox cb = new ComboBox();  
  
    private void OnCbSelect(object sender, EventArgs ea)  
    {  
        string s = ((ComboBox)sender).SelectedItem.ToString();  
        MessageBox.Show("Wybrano element: " + s, "Komunikat");  
    }  
  
    public MainForm()  
    {  
        Width = 320;  
        Height = 200;  
        Text = "Moja aplikacja";  
  
        cb.Items.AddRange  
        (  
            new object[]  
            {  
                "PHP i MySQL. Dla każdego",  
                "PHP5. Praktyczny kurs",  
                "Java. Ćwiczenia praktyczne",  
                "Java. Praktyczny kurs",  
                "Java. Ćwiczenia zaawansowane",  
                "SQL. Ćwiczenia praktyczne",  
                "JavaScript. Ćwiczenia praktyczne",  
                "JavaScript. Praktyczny kurs",  
                "Tablice informatyczne. AJAX"  
            }  
        );  
        cb.Left = (ClientSize.Width - cb.Width) / 2;  
        cb.Top = 20;  
        cb.DropDownWidth = 200;  
        cb.SelectedIndexChanged += OnCbSelect;  
        Controls.Add(cb);  
    }  
    public static void Main()  
    {  
        Application.Run(new MainForm());  
    }  
}
```

W oknie aplikacji została umieszczona lista rozwijana zawierająca 9 pozycji, tak jak jest to widoczne na rysunku 7.18. Wybranie dowolnej pozycji spowoduje wyświetlenie jej nazwy w osobnym oknie dialogowym. Lista została zdefiniowana jako prywatne pole cb typu ComboBox. W konstruktorze klasy MainForm zostało ustalone jej umiejscowienie na formacie (modyfikacja właściwości Left oraz Top), szerokość części rozwijanej (właściwość DropDownWidth), a także została określona jej zawartość. Poszczególne elementy listy dodano za pomocą metody AddRange właściwości Items, zgodnie z podanym wcześniej schematem.

Rysunek 7.18.
Lista rozwijana
umieszczona
w oknie aplikacji



Ponieważ lista ma reagować na wybranie dowolnej z pozycji, niezbędnego było również oprogramowanie zdarzenia SelectedIndexChanged. Procedurą obsługi tego zdarzenia jest metoda OnCbSelect, została ona więc powiązana ze zdarzeniem za pomocą instrukcji:

```
cb.SelectedIndexChanged += OnCbSelect;
```

Jak widać, tym razem zastosowaliśmy sposób odmienny niż we wcześniejszych przykładach z tej lekcji, a poznany w lekcji 36. Oczywiście obiekt delegacji można też utworzyć jawnie, stosując konstrukcję:

```
cb.SelectedIndexChanged += new EventHandler(OnCbSelect);
```

W metodzie OnCbSelect niezbędne jest uzyskanie nazwy wybranej pozycji, tak by mogła być wyświetlona w oknie dialogowym. Korzystamy zatem z pierwszego argumentu metody. Jak pamiętamy, tym argumentem jest obiekt, który zapoczątkował zdarzenie, a więc lista cb. Stosujemy zatem instrukcję:

```
string s = ((ComboBox)sender).SelectedItem.ToString();
```

Najpierw następuje rzutowanie argumentu na typ ComboBox, następnie odwołujemy się do właściwości SelectedItem (jest ona typu Object) i wywołujemy jej metodę ToString. Uzyskana w ten sposób wartość jest przypisywana zmiennej pomocniczej s, która jest używana w kolejnej instrukcji do skonstruowania wyświetlanego na ekranie ciągu znaków.

Ćwiczenia do samodzielnego wykonania

Ćwiczenie 37.1

Napisz aplikację okienkową zawierającą przycisk, którego kliknięcie będzie powodowało kończenie pracy programu.

Ćwiczenie 37.2

Napisz aplikację okienkową zawierającą przycisk i etykietę. Każde kliknięcie przycisku powinno zmieniać kolor tekstu etykiety z czarnego na biały i odwrotnie (kolor zmienisz, korzystając ze struktury Color z przestrzeni nazw System.Drawing oraz odpowiedniej właściwości).

Ćwiczenie 37.3

Napisz aplikację okienkową zawierającą przycisk i pole tekstowe. Po kliknięciu przycisku tekst znajdujący się w polu powinien stać się tytułem okna aplikacji.

Ćwiczenie 37.4

Zmodyfikuj przykład z listingu 7.29 w taki sposób, aby w metodzie `OnCbSelect` nie trzeba było używać argumentu `sender` i aby działanie aplikacji się nie zmieniło.

Ćwiczenie 37.5

Napisz aplikację okienkową zawierającą menu z pozycjami *Odczytaj* i *Zapisz* oraz pole tekstowe. Wybranie pozycji *Odczytaj* powinno powodować odczyt danych tekstowych z pliku *dane.txt* i wprowadzenie ich do pola tekstowego, a wybranie pozycji *Zapisz* — zapisanie tekstu znajdującego się w polu do pliku *dane.txt*. Jeżeli określona operacja nie będzie mogła być wykonana, należy wyświetlić odpowiedni komunikat.

Zakończenie

Lekcja 37. kończy książkę. Udało się w niej zaprezentować całkiem szeroki zakres materiału, pozwalający na swobodne poruszanie się w tematyce C#. Oczywiście nie zostały przedstawione wszystkie aspekty programowania w tej technologii — książka musiałaby mieć wtedy nie 400, lecz co najmniej 1400 stron — jednak zawarta tu wiedza to solidne podstawy pozwalające na samodzielne programowanie.

Czytelnik poznał więc wszystkie podstawowe konstrukcje języka, a także wiele aspektów programowania zorientowanego obiektowo. Wie, co to są wyjątki i jak za ich pomocą obsługiwać sytuacje wyjątkowe, przyswoił sobie techniki wejścia-wyjścia, wczytywanie i zapisywanie danych, operacje na plikach. Umie posłużyć się wieloma klasami .NET, tworzyć aplikacje z interfejsem graficznym, używać komponentów (kontrolek), takich jak przyciski, listy rozwijane czy menu.

Programowanie to jednak taka dziedzina informatyki, w której wciąż trzeba się uczyć, szukać nowych rozwiązań, wymyślać nowe zastosowania, śledzić na bieżąco pojawiające się nowe standardy i możliwości. Nie można spocząć na laurach. Tak więc jeśli Czytelnik przebrnął przez te 400 stron, to właśnie znalazł się na początku długiej i faszynującej drogi; aby pełnej samych sukcesów!

Autor

Skorowidz

.NET Framework, 13, 16

A

akcesor set, 191
aliasy, 297
aplikacje
 konsolowe, 19
 okienkowe, 361
argumenty, 374
 konstruktörów, 148
 metody, 131
 metody Main, 139
automatyczne
 konwersje wartości, 54
 wywołanie konstruktora, 313

B

badanie poprawności danych, 203
bitowa alternatywa wykluczająca, 60
blok
 default, 80
 finally, 229
 try...catch, 203, 208, 218
błąd
 aplikacji, 100
 kompilacji, 55, 175, 315
błędna
 hierarchia wyjątków, 217
 implementacja interfejsów, 330

C

chronione pola, 169
ciagi znaków, 35, 233, 35, 233
CIL, Common Intermediate Language, 12
CLR, Common Language Runtime, 12
CLS, Common Language Specification, 12

D

deklaracja, 39
 metody, 122
 zmiennej, 39
 wielu zmiennych, 41
dekrementacja, 51
delegacja, 371, 379
 OnUjemneEventDelegate, 387
destruktor, 145, 154
dodawanie
 delegacji, 381
 metody, 123
 procedur obsługi, 385
 znaków, 236
dokument XML, 29
dostęp
 chroniony, protected, 169
 do klasy, 165
 do obiektu generującego zdarzenie, 386
 do składowych klasy zagnieżdzonej, 338
 do składowych klasy zewnętrznej, 344
 prywatny, private, 168
 publiczny, public, 166

dynamiczna tablica, 347, 350
dyrektywa using, 129
dziedziczenie, 156, 174, 302, 307, 322
 interfejsu, 200, 323, 331
 struktury, 199
dzielenie przez zero, 212

E

etykiety, 397

F

FCL, Framework Class Library, 12
formatowanie danych, 240
funkcje zwrotne, 375

H

hierarchia klas, 322
hierarchia wyjątków, 214

I

IDE, Integrated Development Environment, 15
iloczyn

 bitowy, 58
 logiczny (`&&`), 61
 logiczny (`&`), 62
implementacja interfejsów, 325, 330
informacja o błędzie, 100
informacje o pliku, 269
inicjalizacja, 40

 pól, 198
 tablic, 101
 właściwości, 196
 zmiennej, 40
inicjalizator, 152
inkrementacja, 51
instalacja

 .NET Framework, 13
 Mono, 15
 MonoDevelop, 15
 Visual Studio, 14
instrukcja

instrukcje
 sterujące, 67
 warunkowe, 67
interfejs, 199, 319, 322, 324
 graficzny, 359
 IDrawable, 321
interpolacja łańcuchów znakowych, 46

J

jednostki komplikacji, 126
język C#, 9

K

katalog, 259
klasa, 118
 BinaryReader, 285
 BinaryWriter, 283
 Button, 399
 ComboBox, 405
 Console, 248
 Convert, 238
 DirectoryInfo, 259
 FileInfo, 266
 FileStream, 272
 FileSystemInfo, 258
 Form, 361
 Kontener, 377, 384
 Label, 397
 MainForm, 365
 StreamReader, 279
 StreamWriter, 281
 TablicaInt, 349, 350
 TextBox, 401
 Triangle, 307

klasy
 abstrakcyjne, 309, 319
 chronione, 165, 342
 kontenerowe, 376
 pochodne, 299
 potomne, 156
 prywatne, 165, 342
 publiczne, 165, 342
 statyczne, 129
 wewnętrzne, 165, 342
 wewnętrzne chronione, 165, 342
 zagniezione, 334
 zewnętrzne, 344
klawiatura, 255
klawisze specjalne, 253
kod
 pośredni, CIL, 12
 źródłowy, 11

kolejność wykonywania konstruktorów, 315
 kolory, 253
 komentarz, 27
 blokowy, 27
 liniowy, 28
 XML, 29
 komplikacja, 11
 just-in-time, 12
 kompilator, 11
 csc.exe, 12, 19
 mcs, 23
 komponenty graficzne, 392
 komunikat, 392
 o błędzie, 214
 konflikty nazw, 328
 konsola, 17
 konsolidacja, 12
 konstruktor, 145, 147
 bezargumentowy, 199
 domyślny, 314
 inicjalizujący właściwość, 195
 przyjmujący argumenty, 148
 struktury, 199
 konstruktor klasy
 bazowej i potomnej, 160
 BinaryReader, 285
 BinaryWriter, 283
 kontener, 346
 kontrola typów, 346, 352
 konwersja, 239, 293
 typów prostych, 290, 293
 konwersje wartości, 54

L

linia tekstu, 255
 linkowanie, 12
 lista plików, 261
 listy
 inicjalizacyjne, 152
 listy rozwijane, 404, 405
 literał null, 38
 literały, 36
 całkowitoliczbowe, 36
 logiczne, 38
 łańcuchowe, 38
 zmiennoprzecinkowe, 37
 znakowe, 38
 logiczna negacja, 62

Ł

łańcuchy znakowe, 35, 233
 łączenie napisów, 45

M
 manifest, 127
 menu, 366, 368, 395
 Kompilacja, 21
 rozwijane, 368
 Tools, 21
 z dwoma podpozycjami, 396
 metadane, 127
 metoda, 122
 Draw, 311
 DrawShape, 306, 312
 Main, 125, 127, 136
 ToString, 296, 297
 metody
 abstrakcyjne, 309
 klasy BinaryReader, 285, 286
 klasy BinaryWriter, 283
 klasy Console, 249
 klasy Convert, 238
 klasy DirectoryInfo, 260
 klasy FileInfo, 267
 klasy FileStream, 272
 klasy FileInfo, 259
 klasy Form, 362
 klasy StreamReader, 279
 klasy StreamWriter, 281
 klasy string, 243
 operujące na polu, 160
 prywatne, 307
 statyczne, 183
 ustawiające pola, 132
 wirtualne, 303, 305
 zwracające wyniki, 125
 zwrotne, 377
 modyfikator sealed, 174
 modyfikatory dostępu, 164
 Mono, 15, 22
 MonoDevelop, 15, 23

N

nawiasy klamrowe, 68
 nazwy zmiennych, 42
 negacja bitowa, 59
 nieprawidłowe dziedziczenie interfejsów, 333
 niszczenie obiektu, 154

O

obiekt, 118
 generujący zdarzenie, 386
 jako argument, 134

- obiekt
 - klasy zagnieżdzonej, 341
 - wyjątku, 191
- obiekty klas zagnieżdżonych, 339
- obsługa
 - błędów, 191, 203
 - kilkę zdarzeń, 390
 - zdarzeń, 383, 393
- odczyt
 - danych binarnych, 285
 - danych binarnych z pliku, 286
 - danych tekstowych, 278
 - danych z pliku, 276
 - danych z pliku tekstopowego, 279
 - plików, 271
 - znaków, 248
- odśmiecacz, 154
- odwołanie do składowej, 292
- okno, 359, 363, 364
 - dialogowe, 393
 - konsoli, 17
- opcja
 - Debug, 21
 - Release, 21
- opcje kompilatora csc, 19
- operacje
 - arytmetyczne, 50
 - bitowe, 57
 - logiczne, 61
 - na katalogach, 259
 - na plikach, 266
 - na tablicach, 98
 - odczytu i zapisu, 274
 - przypisania, 63
 - strumieniowe, 278
- operator
 - „, 121
 - dekrementacji, 53
 - inkrementacji, 52
 - new, 101, 106, 120
 - warunkowy, 76, 81
- operatorы
 - arytmetyczne, 50
 - bitowe, 57
 - logiczne, 61
 - porównywania, 64
 - przypisania, 63, 64
- ostrzeżenie kompilatora, 159
- P**
- pakiet
 - .NET Framework, 12
 - Microsoft Build Tools 2015, 18
- Visual C#, 12
- Visual Studio, 12
- Xamarin Studio, 15, 25
- parametr, 131
- pętla
 - do...while, 88
 - for, 83
 - foreach, 89
 - while, 86
- pierwsza aplikacja, 16
- platforma .NET, 12
- pliki
 - cs, 16
 - pośrednie, 11
 - wykonywalne, 11
- pobieranie
 - linii tekstu, 255
 - zawartości katalogu, 260
- pola
 - statyczne, 184
 - tekstowe, 401, 403
- pole
 - chronione, 169
 - prywatne, 168
 - publiczne, 166
 - sygnalizujące stan operacji, 205
 - tylko do odczytu, 175–177
- polecenie
 - cd, 18
 - cmd, 17
- polimorficzne wywoływanie metod, 317
- polimorfizm, 289, 302
- poprawność danych, 203
- późne wiązanie, 299, 302
- priorytety operatorów, 65
- problem kontroli typów, 352
- procedura obsługi zdarzenia, 384
- programowanie obiektowe, 117, 289
- propagacja wyjątku, 210
- prywatne
 - klasy zagnieżdzone, 342
 - pola, 168
- przechowywanie dowolnych danych, 350
- przechwytywanie
 - wyjątku, 209
 - wielu wyjątków, 215, 217
 - wyjątku ogólnego, 215
- przeciążanie
 - konstruktorów, 149
 - metod, 131, 138, 329
- przekazywanie argumentów
 - przez referencję, 141
 - przez wartość, 140
- przekroczenie zakresu, 55, 57

przekształcanie współrzędnych, 171

przerwanie

- wykonywania pętli, 94
- instrukcji switch, 79

przesłanianie

- metod, 179, 296

- pól, 182

przestrzeń nazw, 127

przesunięcie bitowe

- w lewo, 60

- bitowe w prawo, 61

przetwarzanie

- ciągów, 242

- znaków specjalnych, 48

przyciski, 399

przypisanie, 39

publiczne pola, 167

pusty ciąg znaków, 236

R

rodzaje

- klas wewnętrznych, 342

- wyjątków, 212

rozpoznawanie klawiszy specjalnych, 252

rzeczywisty typ obiektu, 300

rzutowanie typów obiektowych, 291–295

S

sekcja finally, 228

sekwenca ucieczki, 47

składowe

- klas zagnieżdżonych, 338

- klasy zewnętrznej, 344

- statyczne, 183

- typu wyliczeniowego, 403

słowo kluczowe

- namespace, 127

- sealed, 174

- this, 151

- void, 123

specyfikatory formatów, 241

sprawdzanie poprawności danych, 188

stałe napisowe, 36

standardowe wejście i wyjście, 247

standardy C#, 10

statyczne metody, 183

statyczne pola, 184

struktura, 196

- ConsoleKeyInfo, 250, 252

- struktura kodu, 26

- struktura tablicy, 97

- struktura właściwości, 187

strumienie, 278

wejściowe, 278

wyjściowe, 278

suma

- bitowa, 59

- logiczna (|), 62

- logiczna (||), 62

sygnalizacja błędów, 190

system

- plików, 258

- wejścia-wyjścia, 233

szkielet

- aplikacji, 20, 25

- klasy, 119

Ś

ścieżka dostępu, 18

środowisko uruchomieniowe, CLR, 12

T

tablica, 97

tablice

- dwuwymiarowe, 104, 109

- nieregularne, 111

- tablic, 107

- trójkątne, 114

technologia Silverlight, 23

tekst programu, 21

testowanie

- klasy, 158

- konstruktora, 149

tworzenie

- delegacji, 371

- interfejsów, 319

- katalogów, 263

- klas zagnieżdżonych, 334

- menu, 366

- obiektu, 120

- obiektu klasy zagnieżdżonej, 342

- okien aplikacji, 359

- okna aplikacji, 360

- pliku, 267

- struktur, 196, 197

- tablic, 98

- tablicy dwuwymiarowej, 106

- tablicy nieregularnej, 113

- własnych wyjątków, 225

typ

- bool, 34

- char, 34

- ContentAlignment, 403

- Object, 295

- string, 35, 243

typy

- arytmetyczne całkowitoliczbowe, 32
- arytmetyczne zmiennoprzecinkowe, 33
- danych, 31
- obiektów, 118
- odnośnikowe, 32
- proste, 32, 297
- strukturalne, 35
- uogólnione, 346, 353
- wartościowe, 32
- wyliczeniowe, 34, 403

U

- uniwersalność interfejsów, 327
- uogólnianie typów, 355
- uogólniona klasa, 354
- uruchomienie programu, 16, 22
- usuwanie
 - katalogów, 265
 - plików, 270
- użycie
 - bloku try...catch, 208
 - break, 91
 - delegacji, 379
 - dyrektywy using, 129
 - etykiety, 398
 - instrukcji continue, 95
 - instrukcji goto, 79
 - instrukcji if, 68
 - instrukcji if...else if, 74
 - instrukcji switch, 77
 - klas zagnieżdżonych, 336
 - klasy, 121
 - klasy Convert, 239
 - klasy Kontener, 387
 - klasy Tablica, 351
 - komentarza blokowego, 27
 - komentarza liniowego, 29
 - komentarza XML, 30
 - listy rozwijanej, 405
 - metody zwrotnej, 377
 - obiektu klasy zagnieżdzonej, 340, 341
 - operatora dekrementacji, 53
 - operatora inkrementacji, 52
 - operatora new, 106
 - operatora warunkowego, 81
 - pętli do...while, 88
 - pętli foreach, 90
 - pola tekstowego, 402
 - prostej właściwości, 187
 - przeciążonych konstruktorów, 150
 - sekcji try...finally, 228, 230
 - struktury, 197

- właściwości Length, 110
- właściwości Message, 213
- zdarzenia ApplicationExit, 394

V

- Visual Studio, 14, 19, 14, 19
- Visual Studio Community, 12

W

- wczesne wiązanie, 302
- wczytywanie pojedynczego znaku, 250
- wczytywanie tekstu, 255
- wiązanie
 - czasu wykonania, 302
 - dynamiczne, 302
 - statyczne, 302
- własne wyjątki, 220
- właściwości, 187, 324
 - implementowane automatycznie, 195
 - klasy Button, 399
 - klasy ComboBox, 405
 - klasy Console, 248
 - klasy DirectoryInfo, 259
 - klasy FileInfo, 266
 - klasy FileStream, 272
 - klasy FileSystemInfo, 259
 - klasy Form, 361
 - klasy Label, 397
 - klasy TextBox, 401, 402
 - niezwiązane z polami, 194
 - struktury ConsoleKeyInfo, 250
 - tylko do odczytu, 192
 - tylko do zapisu, 193
- właściwość
 - Length, 102, 109
 - Message, 213
 - typu ObjectCollection, 404
- wnętrze klas, 170
- wprowadzanie liczb, 256
- współrzędne
 - biegunowe, 171
 - kartezjańskie, 171
- wybór typu projektu, 20, 24
- wyjątek, 190, 207
 - DivideByZeroException, 213, 222, 318
 - IndexOutOfRangeException, 213
 - InvalidOperationException, 301, 353
 - ValueOutOfRangeException, 191
- wyjątki
 - hierarchia, 214
 - ponowne zgłoszenie, 223
 - propagacja, 210

przechwycenie, 209
przechwytywanie, 215
warunkowe, 226
własne, 220, 225
zgłaszać, 221
wypełnianie tablicy, 102, 114
wyprowadzanie danych na ekran, 43
wyrażenia lambda, 143
 definicja metody, 144
wyrażenie
 modyfikujące, 84
 początkowe, 84, 85
 warunkowe, 84
wyświetlanie
 katalogu bieżącego, 260
 komunikatów, 392
 liczb, 242
 listy podkatalogów, 261
 nazwy plików, 263
 okna dialogowego, 392
 napisu, 44
 pojedynczych znaków, 235
 wartości zmiennych, 43
 zawartości tablicy, 105
 znaków specjalnych, 46
wywołanie
 konstruktorów, 163, 313, 316
 metody, 123
 metody przez delegację, 374
 polimorficzne, 304, 306
 metod, 302
 metod w konstruktorach, 316

X

Xamarin Studio, 23

Z

zabronienie dziedziczenia, 174
zagnieżdżanie
 bloków try...catch, 218
 instrukcji if...else, 69
 komentarzy blokowych, 28
zagnieżdżone pętle for, 93
zakresy liczb, 33
zamiana ciągów na wartości, 238
zapis
 danych binarnych, 283
 danych binarnych do pliku, 284
 danych do pliku, 274
 danych tekstowych, 281
 danych w pliku tekstowym, 281
 plików, 271
 wartości, 36
zdarzenia, 371, 381, 389
zdarzenie ApplicationExit, 394
zestaw, 127
 bibliotek, FCL, 12
zgłaszać
 ponowne wyjątku, 223
 przechwyconego wyjątku, 223
 własnych wyjątków, 226
 wyjątek, 221
zintegrowane środowisko programistyczne, IDE, 15
zmiana kolorów na konsoli, 254
zmienna systemowa path, 18
zmienne, 39
 odnośnikowe, 121
 typów odnośnikowych, 42
znaczniki komentarza XML, 29
znaki, 233
 specjalne, 35, 46, 237