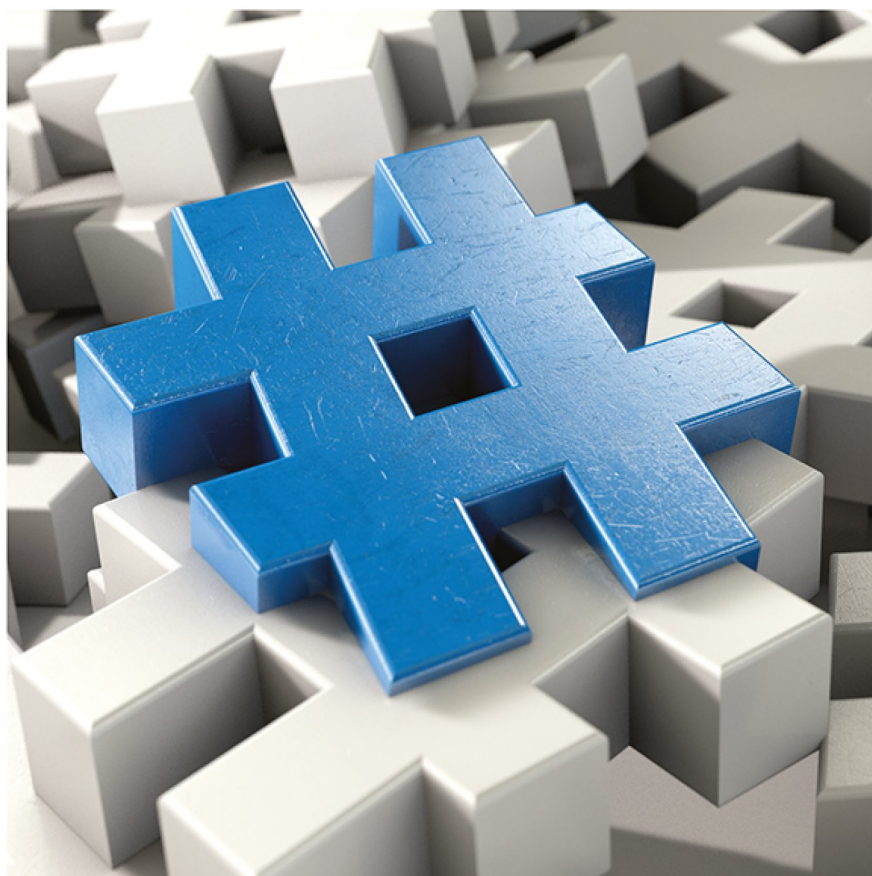


C#

 ĆWICZENIA

Wydanie IV



Marcin Lis

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

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

Wydawnictwo HELION

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

http://helion.pl/user/opinie/cwcs4_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Listingi do książki można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/cwcs4.zip>

ISBN: 978-83-283-2894-5

Copyright © Helion 2016

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

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

Spis treści

Wstęp	7
--------------	----------

Część I **Język programowania**

Rozdział 1.	Pierwsza aplikacja	11
	Język C#	11
	Jak właściwie nazywa się ten język?	12
	Środowisko uruchomieniowe	12
	Narzędzia	14
	Najprostszy program	15
	Kompilacja i uruchamianie	16
	Visual Studio	20
	Dyrektywa using	24
Rozdział 2.	Zmienne i typy danych	25
	Typy danych	25
	Operatory	37
	Komentarze	50

Rozdział 3.	Instrukcje	53
	Instrukcje warunkowe	53
	Pętle	61
	Instrukcja goto	72
	Wprowadzanie danych	76

Część II

Programowanie obiektowe

Rozdział 4.	Klasy i obiekty	93
	Klasy	93
	Metody	96
	Konstruktory	109
	Specyfikatory dostępu	113
	Dziedziczenie	120
	Słowo kluczowe this	124
Rozdział 5.	Tablice	127
	Deklarowanie tablic	128
	Inicjalizacja tablic	130
	Rozmiar tablicy	132
	Pętla foreach	135
	Tablice wielowymiarowe	138
Rozdział 6.	Wyjątki i obsługa błędów	145
	Obsługa błędów	145
	Blok try...catch	151
	Hierarchia wyjątków	157
	Własne wyjątki	160
	Sekcja finally	163
	Filtrowanie wyjątków	165
Rozdział 7.	Interfejsy	169
	Prosty interfejs	169
	Interfejsy w klasach potomnych	173
	Czy to interfejs?	180

Część III

Programowanie w Windows

Rozdział 8.	Pierwsze okno	193
	Utworzenie okna	193
	Wyświetlanie komunikatu	197
	Zdarzenie ApplicationExit	199
Rozdział 9.	Delegacje i zdarzenia	201
	Czym są delegacje?	201
	Jak obsługiwać zdarzenia?	206
Rozdział 10.	Komponenty	213
	Etykiety (Label)	213
	Przyciski (Button)	219
	Pola tekstowe (TextBox)	222
	Pola wyboru (CheckBox, RadioButton)	226
	Listy rozwijane (ComboBox)	232
	Listy zwykłe (ListBox)	235
	Menu	238

Wstęp

Czy warto poznawać C#? Jeszcze kilka lat temu to pytanie zadawało sobie bardzo wielu programistów na całym świecie. Język ten był (i jest) mocno promowany przez firmę Microsoft, gdzie powstały jego pierwsze specyfikacje i implementacje, ale nawet osoby, które nie przepadają za tą firmą, przyznawały, że C# jest udanym produktem. Co więcej, jest na tyle podobny do C++ i Javy, że programiści znający oba te języki od razu go polubią. Z kolei wszyscy ci, którzy dopiero rozpoczynają swoją przygodę z programowaniem, dostaną do dyspozycji język prostszy do nauki niż wspomniane C++, przy czym w tym przypadku „prostszy” wcale nie oznacza „o mniejszych możliwościach”. Co więcej, jeśli ktoś nauczy się C# jako pierwszego języka, nie będzie miał trudności z późniejszym poznaniem C++ czy Javy.

Czy zatem warto? Dziś już nie trzeba stawiać takiego pytania. Odpowiedź brzmi — na pewno tak. Minęły czasy, kiedy programista znał tylko jeden język programowania, więc niezależnie od tego, czy jest to dla nas pierwszy, czy kolejny język, warto się z nim zapoznać. Oczywiście trudno w tej chwili prorokować, czy osiągnie on tak dużą popularność jak wspomniani wyżej aktualni liderzy (choć wszystko na to

wskazuje¹), ale już teraz wiadomo, że w wielu zastosowaniach na pewno będzie wykorzystywany, choćby ze względu na wygodę i naprawdę duże możliwości. Dlatego też powstała niniejsza książka, która stanowi doskonały pierwszy krok do wyprawy w krainę C#, przydatny zarówno dla osób początkujących, jak i znających już inny język (bądź języki) programowania.

W książce opisano standard C# 6.0 (dostępny wraz z platformą .NET Framework 4.6), choć większość ćwiczeń będzie poprawnie działać nawet w pierwotnych, powstałych wiele lat temu, a obecnie rzadko spotykanych wersjach 1.0 i 1.2. Przedstawiony materiał jest również w pełni zgodny ze standardami C# 4.0 (dostępny wraz z platformą .NET Framework 4.0) oraz 5.0 (dostępny wraz z platformą .NET Framework 4.5).

¹ Wedle niektórych badań już na początku 2012 roku popularność C# zbliżyła się do C++, który systematycznie traci udziały w rynku.

Część I

Język programowania

1

Pierwsza aplikacja

Język C#

Język C# został opracowany w firmie Microsoft i wywodzi się z rodziny C/C++, choć zawiera również wiele elementów znanych programistom Javy, jak na przykład 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 dużo łatwiejszy niż tak popularny C++.

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

C# jest w pełni obiektowy (zorientowany obiektowo), zawiera wspomniane już mechanizmy odzyskiwania pamięci i obsługę wyjątków.

Jest też ściśle powiązany ze środowiskiem uruchomieniowym .NET, co oczywiście nie znaczy, że nie mogą powstawać jego implementacje przeznaczone dla innych platform. Oznacza to jednak, że C# doskonale sprawdza się w najnowszym środowisku Windows oraz w sposób bezpośredni może korzystać z udogodnień platformy .NET, co pozwala na szybkie i efektywne pisanie aplikacji.

Jak właściwie nazywa się ten język?

Pewne zamieszanie panuje w kwestii nazwy języka. Trzeba więc od razu podkreślić, że C# wymawiamy jako C-Sharp (czyli fonetycznie si-szarp lub ce-szarp) i jest to jedyna poprawna nazwa. To nie jest C-Hash, C-Number, C-Pound, C-Gate itp. Aby jednak zrozumieć, skąd wynikają te nieporozumienia, należy wrócić do genezy nazewnictwa języków z tej rodziny. Dlaczego np. tak popularny język C (1971) został oznaczony właśnie tą literą alfabetu? Otóż dlatego, że wywodzi się wprost z języka B (1969). Z kolei C++ to nic innego jak złożenie nazwy C i operatora ++, co oznacza, że jest to ulepszona wersja C.

Podobnie jest z C#, w którym pierwotnie za literą C stał znak **#** oznaczający w notacji muzycznej podwyższenie dźwięku o pół tonu¹ (znak o kodzie U+266f), co można interpretować jako kolejne, ulepszone C. Ponieważ jednak znak ten nie występuje na standardowej klawiaturze, podjęto decyzję, że będzie reprezentowany przez znak # (hash, znak o kodzie U+0023), niemniej wymawiany będzie jako *sharp* (fonetycznie: *szarp*). Czyli, podkreślmy raz jeszcze, nazwa tego języka powinna być wymawiana jako C-Sharp.

Środowisko uruchomieniowe

Programy pisane w technologii .NET niezależnie od zastosowanego języka wymagają specjalnego środowiska uruchomieniowego, tak zwanego CLR — *Common Language Runtime*. Próba uruchomienia takiego

¹ Stąd zapewne to nieszczęsne C-Cis, pojawiające się w niektórych polskich tłumaczeniach.

programu w czystym systemie Windows skończy się niepowodzeniem i komunikatami o braku odpowiednich bibliotek. Na szczęście począwszy od wersji XP, a więc obecnie praktycznie w każdym przypadku, platforma .NET jest instalowana standardowo wraz z systemem i nie trzeba podejmować dodatkowych czynności. Użytkownicy starszych wersji, o ile w XXI w. są tacy wśród czytelników tej książki, muszą wcześniej zainstalować pakiet .NET Framework (najlepiej w możliwie najnowszej wersji) dostępny na stronach Microsoftu (<http://www.microsoft.com/net>).

Skąd takie wymogi? Otóż program pisany w technologii .NET, czy to w C#, Visual Basicu czy innym języku, nie jest kompilowany do kodu natywnego danego procesora („rozumianego” bezpośrednio przez procesor), ale do kodu pośredniego² (przypomina to w pewnym stopniu *byte-code* znany z Javy). Ten kod pośredni 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 językach. Fragment prostego programu w tym wspólnym języku pośrednim widoczny jest na poniższym listingu.

```
.method public hidebysig static void Main(string[] args) cil managed
{
    .entrypoint
    // Code size 309 (0x135)
    .maxstack 3
    .locals init (int32 V_0, int32 V_1, int32 V_2, float64 V_3, float64 V_4,
        ↳ object[] V_5)
    IL_0000: ldc.i4.1
    IL_0001: stloc.0
    IL_0002: ldc.i4.s -5
    IL_0004: stloc.1
    IL_0005: ldc.i4.4
    IL_0006: stloc.2
    IL_0007: ldstr bytearray (50 00 61 00 72 00 61 00 6D 00 65 00 74 00 72
        ↳ 00 79 00 20 00 72 00 F3 00 77 00 6E 00 61 00 6E 00 69 00 61 00 3A 00
        ↳ 0A 00)
    IL_000c: call void [mscorlib]System.Console::WriteLine(string)
}
```

² Kompilacja kodu pośredniego do kodu natywnego procesora jest wykonywana przez kompilator *just-in-time* w momencie uruchomienia aplikacji. Środowisko uruchomieniowe może też przechowywać gotowe do uruchomienia prekompilowane i zoptymalizowane fragmenty kodu aplikacji.

Nie jest on może przy pierwszym spojrzeniu zbyt przejrzysty, ale osoby znające na przykład asembler z pewnością dostrzegą spore podobieństwa. Na szczęście, przynajmniej na początku przygody z C#, nie trzeba schodzić na tak niski poziom programowania. Możliwość obejrzania kodu pośredniego może być jednak przydatna np. przy wyszukiwaniu błędów we współpracujących ze sobą komponentach.

W tej chwili najważniejsze dla nas jest to, że aby uruchomić program napisany w C#, musimy mieć zainstalowany pakiet .NET Framework, który potrafi poradzić sobie z kodem pośrednim oraz zawiera biblioteki z definicjami przydatnych klas.

Narzędzia

Najpopularniejszym środowiskiem programistycznym służącym do tworzenia aplikacji C# jest oczywiście produkowany przez firmę Microsoft pakiet Visual C# — niegdyś dostępny jako oddzielny produkt, a obecnie jako część pakietu Visual Studio. Oczywiście wszystkie prezentowane w niniejszej książce przykłady mogą być tworzone przy użyciu tego właśnie produktu (a także niezależnych alternatyw takich jak Mono). Korzystać można z darmowej edycji Visual Studio Community dostępnej pod adresem <https://www.visualstudio.com/> (lub podobnym, o ile w przyszłości zostanie zmieniony).

Oprócz tego istnieje również darmowy kompilator C# (*csc.exe*) będący częścią pakietu .NET Framework (pakietu tego należy szukać pod wspomnianym już adresem <http://www.microsoft.com/net/> lub <http://msdn.microsoft.com/netframework/>), a w przypadku wersji 6.0 i wyższych występujący jako składowa zestawu Microsoft Build Tools (<https://www.visualstudio.com/pl-pl/downloads>). Jest to kompilator uruchamiany w wierszu poleceń, nie oferuje więc dodatkowego wsparcia przy budowaniu aplikacji tak jak Visual Studio, jednak do naszych celów jest całkowicie wystarczający.

Co więcej, właśnie ten kompilator będzie prezentowany podczas wykonywania ćwiczeń. Będzie tak przede wszystkim dlatego, że nie korzystając z wizualnych pomocy (szczególnie przy tworzeniu aplikacji z interfejsem graficznym), łatwiej jest zrozumieć zależności występujące w kodzie oraz zobaczyć, w jaki sposób realizowanych jest wiele mechanizmów języka, takich jak np. zdarzenia. Oczywiście Czytelnicy, którzy wolą korzystać z pełnego Visual Studio, mogą posługiwać się tym narzędziem.

Najprostszy program

Na początku napiszmy najprostszy chyba program, którego zadaniem będzie wyświetlenie na ekranie dowolnego tekstu. Nie jest to skomplikowane zadanie, pewne rzeczy trzeba będzie jednak przyjąć „na słowo”, dopóki nie zostaną omówione definicje klas, każdy bowiem program napisany w C# składa się ze zbioru klas. Ogólna struktura programu powinna wyglądać następująco:

```
using System;

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

Takiej właśnie struktury będziemy używać w najbliższych ćwiczeniach, przyjmując, że tak powinien wyglądać program. Kod wykonywalny, np. instrukcje wyprowadzające dane na ekran, umieszczać będziemy w oznaczonym miejscu tzw. funkcji Main(). Taką instrukcją jest:

```
Console.WriteLine("Tekst do wyświetlenia");
```

ĆWICZENIE

1.1 Pierwszy program

Napisz program wyświetlający na ekranie dowolny napis, np. Mój pierwszy program!.

```
using System;

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

Plik ten można zapisać na dysku pod nazwą *program.cs*. Oczywiście możliwe jest również nadanie innej, praktycznie dowolnej nazwy (np. *aplikacja.cs*).

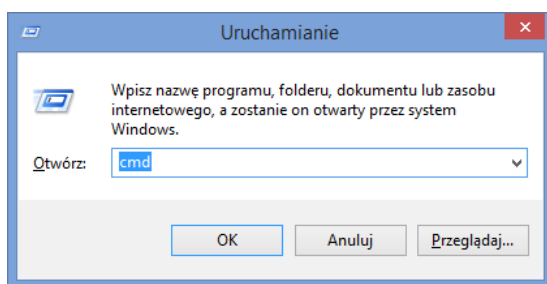


Wielkość liter w kodzie programu ma znaczenie. Jeśli się pomylimy, nie uda się utworzyć działającej aplikacji!

Kompilacja i uruchamianie

Kod z ćwiczenia 1.1 należy teraz skompilować i uruchomić. Kompilacja to proces przetworzenia kodu symbolicznego widocznego na listingu na kod zrozumiały dla danego środowiska uruchomieniowego (systemu operacyjnego wraz z zainstalowanymi dodatkami). Korzystać będziemy z kompilatora uruchamianego w wierszu poleceń — `csc.exe` (dostępnego w systemach Windows z zainstalowanym pakietem .NET Framework, a także Visual Studio i Microsoft Build Tools). W systemie Windows wciskamy na klawiaturze kombinację klawiszy *Windows+R*³, w polu *Uruchom* wpisujemy `cmd` lub `cmd.exe`⁴ i klikamy przycisk *OK* lub wciskamy klawisz *Enter* (rysunek 1.1). (W systemach XP i starszych pole *Uruchom* jest też dostępne bezpośrednio w menu *Start*⁵). Można też w dowolny inny sposób uruchomić aplikację `cmd.exe`. Pojawi się wtedy okno wiersza poleceń (wiersza polecenia), w którym będzie można wydawać komendy. Jego wygląd dla systemów z rodziny Windows 10 został przedstawiony na rysunku 1.2 (w innych wersjach wygląda bardzo podobnie).

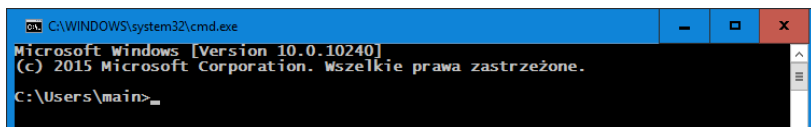
Rysunek 1.1.
Uruchamianie polecenia cmd w Windows 7



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



Rysunek 1.2. Okno konsoli (wiersza poleceń) w systemie Windows 10

Aby wykonać kompilację, w najprostszym przypadku jako parametr wywołania kompilatora *csc.exe* należy podać nazwę pliku z kodem źródłowym — wywołanie będzie więc miało postać:

ścieżka dostępu do kompilatora\csc.exe *ścieżka dostępu do pliku*\program.cs

Na przykład:

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

lub:

```
c:\Program Files\MSBuild\14.0\Bin\csc.exe c:\cs\program.cs
```

przy założeniu, że kod programu znajduje się w pliku *program.cs* w katalogu *c:\cs*.

Ścieżka dostępu do kompilatora to katalog, w którym został zainstalowany pakiet .NET Framework. Znajduje się on w lokalizacji:

windows\Microsoft.NET\Framework*wersja*

gdzie *windows* oznacza katalog systemowy, a *wersja* określa wersję platformy .NET, np. v4.0.30319. W przypadku korzystania z narzędzi Microsoft Build Tools ścieżka dostępu to:

dysk:\Program Files\MSBuild*wersja*\Bin

Na przykład:

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

Nie można przy tym zapomnieć o podawaniu nazwy pliku zawsze z rozszerzeniem. Typowym rozszerzeniem plików z kodem źródłowym C# jest *cs*, chociaż nie jest to obligatoryjne i kompilator przyjmie bez problemów również dowolny inny plik zawierający poprawny tekst programu. W celu ułatwienia pracy warto dodać do zmiennej systemowej *path* ścieżkę dostępu do pliku wykonywalnego kompilatora, np. wydając (w wierszu poleceń; rysunek 1.3) polecenie⁶:

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

⁶ Oczywiście należy samodzielnie dostosować numer wersji platformy .NET. Użyty w poleceniu ciąg v4.0.30319 jest przykładowy.

lub:

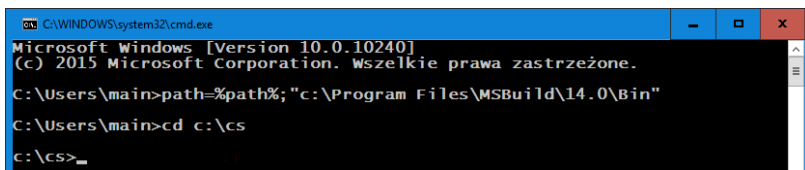
```
path=%path%; "c:\Program Files\MSBuild\14.0\Bin"
```

Wtedy kompilacja będzie mogła być wykonywana za pomocą komendy:

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

a jeżeli jesteśmy w katalogu `c:\cs`, to za pomocą polecenia:

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



Rysunek 1.3. Modyfikacja zmiennej systemowej `path` w wierszu poleceń

Zmianę katalogu bieżącego można osiągnąć dzięki komendzie:

```
cd ścieżka dostępu
```

Na przykład:

```
cd c:\cs
```

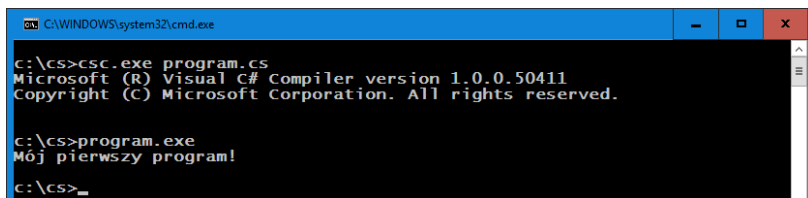
Po kompilacji powstanie plik wynikowy *program.exe*, który można uruchomić w wierszu poleceń (w konsoli systemowej), czyli tak jak każdą inną aplikację, o ile oczywiście został zainstalowany wcześniej pakiet .NET Framework.

Ć W I C Z E N I E

1.2 Skompilowanie i uruchomienie programu

Skompiluj i uruchom program napisany w ćwiczeniu 1.1.

Etap kompilacji oraz wynik działania programu widoczny jest na rysunku 1.4. O ile w kodzie źródłowym nie występują błędy, to — jak widać — kompilator nie wyświetla żadnych informacji oprócz noty copyright, generuje natomiast plik wykonywalny typu `exe`. Kompilator `csc` umożliwia stosowanie różnych opcji pozwalających na ingerencję w proces kompilacji. Są one pokazane w tabeli 1.1.



```
C:\WINDOWS\system32\cmd.exe
c:\cs>csc.exe program.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

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

c:\cs>
```

Rysunek 1.4. Kompilacja i uruchomienie pierwszego programu w C#

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.
/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 nazwy są zgodne z maską.
/win32icon:	–	nazwa pliku	Dołącza do pliku wynikowego podaną ikonę.
/debug	–	+ lub –	Włącza (+) oraz wyłącza (–) generowanie informacji dla debugera.
/optimize	/o	+ lub –	Włącza (+) oraz wyłącza (–) optymalizację kodu.
/incremental	/incr	+ lub –	Włącza (+) oraz wyłącza (–) kompilację przyrostową.
/warnaserror	–	–	Włącza tryb traktowania ostrzeżeń jako błędów.

Tabela 1.1. Wybrane opcje kompilatora csc — ciąg dalszy

Nazwa opcji	Forma skrócona	Parametr	Znaczenie
/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

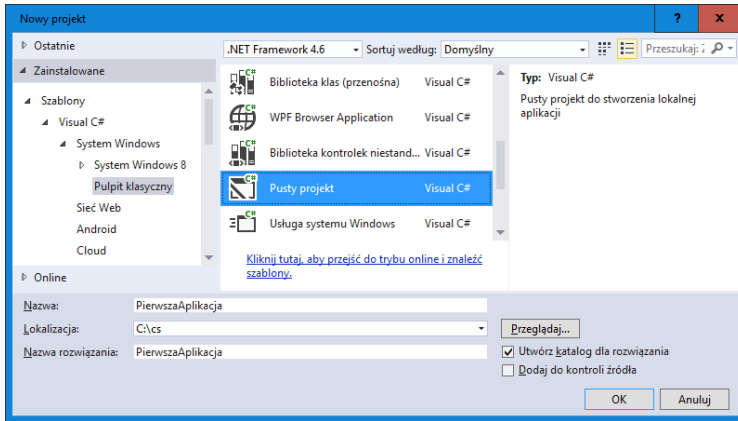
Do wykonywania ćwiczeń zawartych w książce z powodzeniem wystarczy opisany wyżej kompilator uruchamiany w wierszu poleceń (*csc.exe*), dostępny w pakietach .NET Framework i Microsoft Build Tools. Dla osób, które wolałyby jednak wykorzystać do nauki pakiet Visual Studio (może to być z powodzeniem darmowa wersja Community), zostało zamieszczonych kilka poniższych ćwiczeń pokazujących, w jaki sposób stworzyć projekt i dokonać kompilacji.

Ć W I C Z E N I E

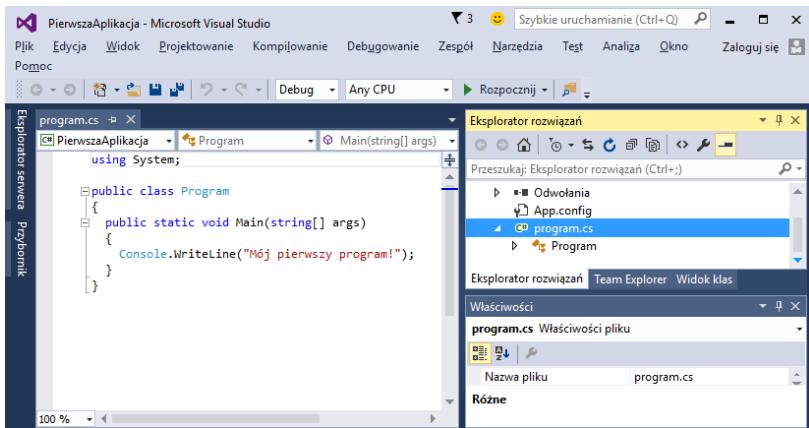
1.3 Pierwszy projekt w Visual Studio

Uruchom Visual Studio, utwórz pusty projekt, a następnie dodaj do niego przygotowany plik *program.cs*. Dokonaj kompilacji kodu.

Po uruchomieniu pakietu należy utworzyć nowy projekt, klikając odpowiednią ikonę, wybierając z menu *Plik (File)* pozycję *Nowy projekt (New Project)* lub wciskając kombinację klawiszy *Ctrl+Shift+N*. Następnie należy wybrać opcję *Pusty projekt (Empty Project)* i zatwierdzić wybór przyciskiem *OK* (rysunek 1.5). W polu *Nazwa (Name)* można podać nazwę projektu, a w polu *Lokalizacja (Location)* — katalog projektu. Następnie (po utworzeniu pustego projektu) korzystając z menu *Projektowanie (Project)* i opcji *Dodaj istniejący element (Add Existing Item)*, należy dodać do projektu plik *program.cs*. Jeżeli zawartość dodanego pliku nie pojawi się na ekranie, można ją wyświetlić, klikając dwukrotnie jego nazwę w oknie *Eksplorator rozwiązań (Solution Explorer)*; rysunek 1.6).



Rysunek 1.5. Wybór szablonu projektu Visual Studio



Rysunek 1.6. Plik program.cs dodany do projektu w pakiecie Visual Studio

Projekt najlepiej następnie zapisać, wybierając z menu *Plik* (*File*) opcję *Zapisz wszystko* (*Save all*). Plik wynikowy tworzy się, wybierając z menu *Kompilowanie* (*Build*) pozycję *Kompiluj rozwiązanie* (*Build Solution*) lub wciskając klawisze *Ctrl+Shift+B*. Plik wykonywalny zostanie zapisany w katalogu projektu (tam gdzie został zapisany projekt), w podkatalogu *nazwa projektu/Bin/Release* lub *nazwa projektu/Bin/Debug* (w zależności od tego, jaki został wybrany typ generowanego kodu; lista rozwijana z pozycjami *Debug* i *Release* widoczna na rysunku 1.6). Będzie to aplikacja konsolowa i należy ją uruchamiać w wierszu poleceń.

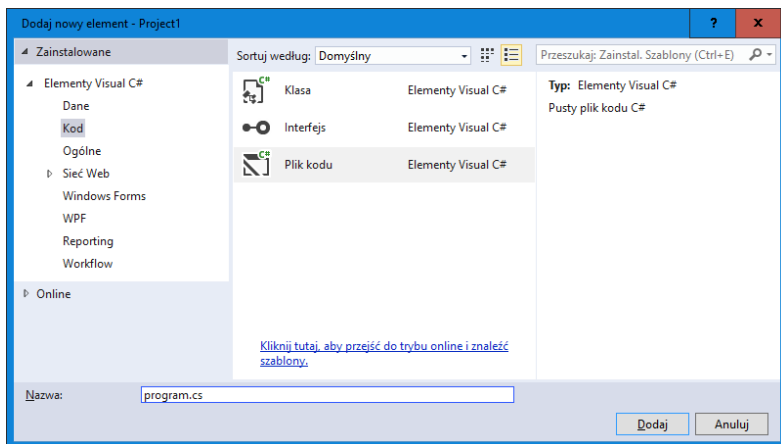
Jeżeli kod miałby być pisany bezpośrednio w edytorze Visual Studio, należałoby postąpić inaczej niż w ćwiczeniu 1.3. Do wyboru są dwa sposoby — można napisać cały kod od zera lub też kazać wygenerować pakietowi szkielet aplikacji i dopiero ten szkielet wypełnić instrukcjami. Do ćwiczeń z niniejszej książki lepszy jest sposób pierwszy, jako że kod generowany automatycznie będzie się nieco różnił od prezentowanych dalej przykładów. Niemniej w dwóch kolejnych ćwiczeniach zostanie pokazane, jak zastosować oba sposoby.

Ć W I C Z E N I E

1.4 Dodawanie nowego pliku do istniejącego projektu

W Visual Studio utwórz pusty projekt (*Empty Project*), dodaj do projektu nowy plik i zapisz w nim kod ćwiczenia 1.1.

Po utworzeniu projektu należy — analogicznie jak w ćwiczeniu 1.3 — z menu *Projektowanie (Project)* wybrać pozycję *Dodaj nowy element (Add New Item)*. W kolejnym oknie zaznaczamy *Kod (Code)* i *Plik kodu (Code file)*, w polu *Nazwa (Name)* wpisujemy nazwę pliku, w którym chcemy umieścić kod programu, i klikamy *Dodaj (Add)* (rysunek 1.7). Następnie w edytorze wprowadzamy instrukcje z ćwiczenia 1.1. Utworzony w ten sposób projekt kompilujemy w sposób opisany w ćwiczeniu 1.3.



Rysunek 1.7. Tworzenie nowego pliku z kodem źródłowym w Visual Studio

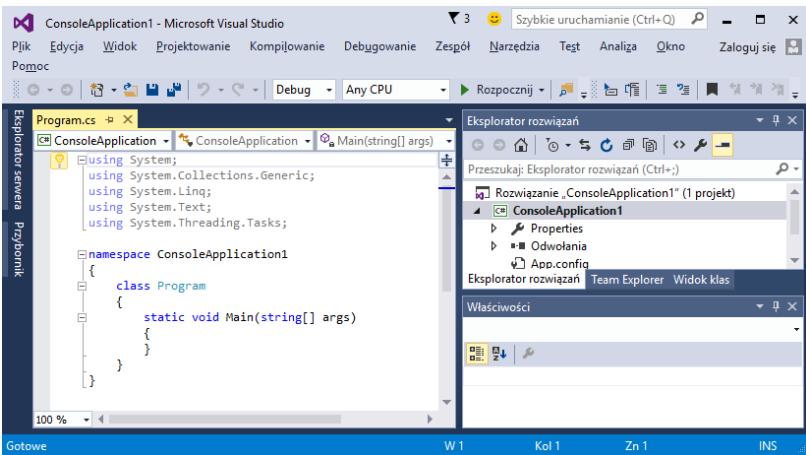
ĆWICZENIE

1.5 Tworzenie projektu konsolowego

W Visual Studio utwórz projekt konsolowy C# (*Console Application*) i zapisz w nim kod realizujący zadanie z ćwiczenia 1.1.

Tym razem po uruchomieniu Visual Studio z menu *Plik* (File) również wybieramy pozycje *Nowy* i *Projekt* (New i Project), jednak zamiast opcji *Pusty projekt* szukamy opcji *Aplikacja konsolowa* (*Console Application*) i po jej zaznaczeniu klikamy przycisk OK. Utworzony zostanie szkielet aplikacji (rysunek 1.8). Pomędzy znakami nawiasu klamrowego występującego za frazą `static void Main(string[] args)` wpisujemy instrukcję:

```
Console.WriteLine("Mój pierwszy program!");
```



Rysunek 1.8. Wygenerowany przez Visual Studio szkielet aplikacji konsolowej

Kompilacji i uruchomienia programu dokonujemy w sposób identyczny jak w poprzednich dwóch ćwiczeniach.

Dyrektywa using

We wszystkich dotychczasowych ćwiczeniach na początku kodu programu pojawiała się dyrektywa `using System`. Oznacza ona, że program będzie korzystał z klas zdefiniowanych w **przestrzeni nazw** `System`. Przypomina to nieco instrukcję `import` znaną z Javy. Taki zapis wskazuje kompilatorowi, gdzie ma szukać **klasy** `Console` i **metody** `WriteLine`, które wykorzystaliśmy do wyświetlenia napisu na ekranie. (Wyróżnione pojęcia zostaną bliżej wyjaśnione w dalszej części książki).

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 ćwiczenia 1.1 w C# 6.0 i wyższych wersjach może również wyglądać tak jak w ćwiczeniu 1.6. Po użyciu zapisu `using static Console` w treści metody `Main` zamiast pisać `Console.WriteLine` można użyć samego zapisu `WriteLine`. Sposób działania kodu wcale się przez to nie zmienia.

Ć W I C Z E N I E

1.6 Dyrektywa using w C# 6.0

Napisz taką wersję programu z ćwiczenia 1.1 (dla C# w wersji 6.0 i wyższych), aby wewnątrz kodu zamiast zapisu `Console.WriteLine` można było stosować zapis `WriteLine`.

```
using System;
using static Console;

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



2

Zmienne i typy danych

Typy danych

Zmienna to miejsce w programie, w którym można przechowywać jakieś dane, np. liczby czy ciągi znaków (napisy). Każda zmienna ma swoją nazwę, która ją jednoznacznie identyfikuje, oraz typ określający, jakiego rodzaju dane może ona przechowywać. Przykładowo zmienna typu `int` może przechowywać liczby całkowite, a zmienna typu `float` — liczby rzeczywiste.

Typy danych możemy podzielić na typy wartościowe (ang. *value types*), do których zaliczymy typy proste (inaczej podstawowe, ang. *primitive types*, *simple types*), wyliczeniowe (ang. *enum types*) i strukturalne (ang. *struct types*) oraz typy odnośnikowe (referencyjne, ang. *reference types*), do których zaliczamy typy klasowe, interfejsowe, delegacyjne oraz tablicowe. Ta mnogość nie powinna jednak przerażać. Na początku nauki wystarczy zapoznać się z podstawowymi typami: prostymi arytmetycznymi, typem `boolean` oraz `string`.

Typy arytmetyczne

Typy arytmetyczne, będące podzbiorem typów prostych, można podzielić na typy całkowitoliczbowe (ang. *integral types*) oraz typy zmiennoprzecinkowe (zmiennopozycyjne, ang. *floating-point types*). Pierwsze służą do reprezentacji liczb całkowitych, drugie — liczb rzeczywistych (z częścią ułamkową). Typy całkowitoliczbowe w C# to:

- ❑ sbyte,
- ❑ byte,
- ❑ char,
- ❑ short,
- ❑ ushort,
- ❑ int,
- ❑ uint,
- ❑ long,
- ❑ ulong.

Zakresy możliwych do przedstawiania za ich pomocą liczb oraz ilość bitów, na których są one zapisywane, przedstawione zostały 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 liczb	Znaczenie
sbyte	od -128 do 127	8-bitowa liczba ze znakiem
byte	od 0 do 255	8-bitowa liczba bez znaku
char	U+0000 do U+FFFF	16-bitowy znak Unicode
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

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, został zaliczony do typów arytmetycznych całkowitoliczbowych.

Typy zmiennoprzecinkowe występują tylko w trzech odmianach:

- ❑ `float`,
- ❑ `double`,
- ❑ `decimal`.

Zakres oraz precyzja liczb, jakie można za ich pomocą przedstawić, zestawione są w tabeli 2.2. Typ `decimal` służy do reprezentowania wartości, dla których ważniejsza jest precyzja, a nie maksymalny zakres reprezentowanych wartości (np. danych finansowych).

Tabela 2.2. Typy zmiennoprzecinkowe w C#

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

Typ `bool` (Boolean)

Zmienne tego typu mogą przyjmować tylko dwie wartości: `true` i `false` (prawda i fałsz). 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, tak jak w wymienionych językach. Oznacza to, że poniższy fragment kodu jest niepoprawny.

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

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

Deklarowanie zmiennych

Aby w programie użyć jakiejś zmiennej, wpierw trzeba ją zadeklarować, tzn. podać jej typ oraz nazwę. Ogólna deklaracja wygląda następująco:

```
typ_zmiennej nazwa_zmiennej;
```

Po takim zadeklarowaniu zmienna jest już gotowa do użycia, tzn. można jej przypisywać różne wartości bądź też wykonywać na niej różne operacje, np. dodawanie. Pierwsze przypisanie wartości zmiennej nazywa się **inicjalizacją zmiennej** lub **inicjacją zmiennej**.

Jeśli chcemy wyświetlić zawartość zmiennej na ekranie, wystarczy użyć — tak jak w ćwiczeniu 1.1 — instrukcji `Console.WriteLine`, podając nazwę zmiennej jako parametr:

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

Jeżeli chcemy równocześnie wyświetlić jakiś łańcuch znaków (napis), możemy wykonać to w sposób następujący:

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

Jeżeli zmiennych jest kilka i mają występować w różnych miejscach łańcucha znakowego, najlepiej zastosować następującą konstrukcję:

```
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`.

Ć W I C Z E N I E

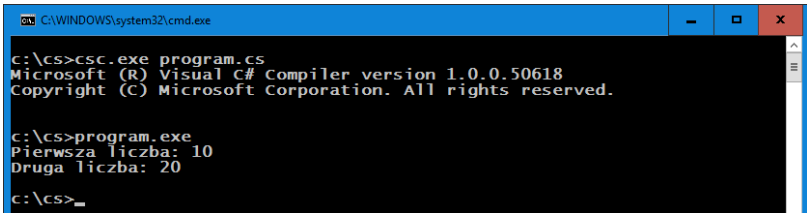
2.1 Wyświetlanie wartości zmiennych arytmetycznych

Zadeklaruj dwie zmienne całkowite i przypisz im dowolne wartości. Wyświetl wartości zmiennych na ekranie, tak jak pokazano na rysunku 2.1.

```
using System;

public class main
{
    public static void Main()
    {
        int pierwszaLiczba;
        int drugaLiczba;
```

```
pierwszaLiczba = 10;
drugaLiczba = 20;
Console.WriteLine("Pierwsza liczba: " + pierwszaLiczba);
Console.WriteLine("Druga liczba: " + drugaLiczba);
}
}
```



```
C:\WINDOWS\system32\cmd.exe
c:\cs>csc.exe program.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50618
Copyright (C) Microsoft Corporation. All rights reserved.

c:\cs>program.exe
Pierwsza liczba: 10
Druga liczba: 20
c:\cs>
```

Rysunek 2.1. Wynik działania programu z ćwiczenia 2.1

W programie powstały dwie zmienne typu `int`, czyli mogące przechowywać wartości całkowite: `pierwszaLiczba` i `drugaLiczba`. Pierwszej z nich przypisana została wartość 10, a drugiej — wartość 20. W ostatnich dwóch instrukcjach wartości zmiennych zostały wyświetlone na ekranie. Użyty został opisany wyżej sposób, w którym napis ujęty w znaki cudzysłowu jest łączony ze zmienną (wartością zmiennej) za pomocą znaku `+`.

Wartość zmiennej można przypisać już w trakcie deklaracji (mamy wtedy do czynienia z jednoczesną deklaracją i inicjalizacją), pisząc:

```
typ_zmiennej nazwa_zmiennej = wartość;
```

Można również zadeklarować wiele zmiennych danego typu, oddzielając ich nazwy przecinkami. Część z nich może też być od razu zainicjalizowana:

```
typ_zmiennej nazwa1, nazwa2, nazwa3;
typ_zmiennej nazwa1 = wartość1, nazwa2, nazwa3 = wartość2;
```

Zmienne w C#, podobnie jak w Javie, C czy C++, ale inaczej niż w Pascalu, można deklarować praktycznie w dowolnym miejscu programu, czyli wtedy, gdy są faktycznie potrzebne.

Ć W I C Z E N I E

2.2 Jednoczesna deklaracja i inicjalizacja zmiennych

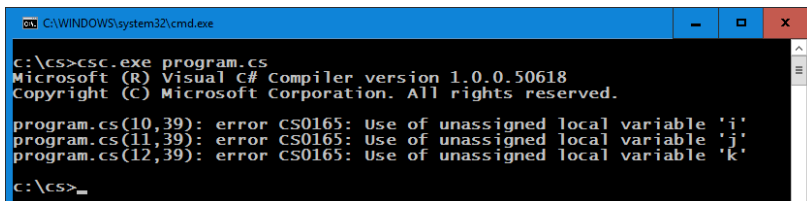
Zadeklaruj i jednocześnie zainicjalizuj dwie zmienne typu całkowitego. Wyświetl ich wartości na ekranie.

```
using System;

public class main
{
    public static void Main()
    {
        int pierwszaLiczba = 10;
        int drugaLiczba = 20;
        Console.WriteLine("Pierwsza liczba: " + pierwszaLiczba);
        Console.WriteLine("Druga liczba: " + drugaLiczba);
    }
}
```

Rozwiązanie w tym ćwiczeniu jest bardzo podobne do poprzedniego przykładu. Tym razem jednak deklaracje zmiennych zostały połączone z ich inicjalizacją (przypisaniem pierwotnych wartości).

Należy zwrócić uwagę, że wprowadzić zmienna nie musi być zainicjalizowana w momencie deklaracji, ale musi być zainicjalizowana przed pierwszym jej użyciem. Jeśli tego nie zrobimy, zostanie zgłoszony błąd kompilacji (rysunek 2.2). Sprawdźmy to, wykonując kolejne ćwiczenie.

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The prompt shows the command "c:\cs>csc.exe program.cs" and the output from the Microsoft (R) Visual C# Compiler version 1.0.0.50618. The output includes the copyright notice and three error messages: "program.cs(10,39): error CS0165: Use of unassigned local variable 'i'", "program.cs(11,39): error CS0165: Use of unassigned local variable 'j'", and "program.cs(12,39): error CS0165: Use of unassigned local variable 'k'". The prompt ends with "c:\cs>_".

```
c:\cs>csc.exe program.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50618
Copyright (C) Microsoft Corporation. All rights reserved.

program.cs(10,39): error CS0165: Use of unassigned local variable 'i'
program.cs(11,39): error CS0165: Use of unassigned local variable 'j'
program.cs(12,39): error CS0165: Use of unassigned local variable 'k'

c:\cs>_
```

Rysunek 2.2. Próba wykorzystania niezainicjalizowanej zmiennej powoduje błąd kompilacji

Ć W I C Z E N I E

2.3 Konieczność inicjalizacji zmiennych przed użyciem

Zadeklaruj kilka zmiennych typu całkowitego w jednym wierszu. Część z nich zainicjalizuj. Spróbuj wyświetlić wartości wszystkich zmiennych na ekranie.

```
using System;

public class main
{
    public static void Main()
    {
        int pierwszaLiczba = 10, drugaLiczba = 20, i, j, k;
        Console.WriteLine("pierwszaLiczba: " + pierwszaLiczba);
        Console.WriteLine("drugaLiczba: " + drugaLiczba);
        Console.WriteLine("zmienna i: " + i);
        Console.WriteLine("zmienna j: " + j);
        Console.WriteLine("zmienna k: " + k);
    }
}
```

Zgodnie z przewidywaniami przedstawionego programu nie udało się skompilować, co jest widoczne na rysunku 2.2. Powodem jest brak przypisania wartości zmiennym *i*, *j* oraz *k* połączony z próbą odczytu ich zawartości (w programie można pozostawić niezainicjalizowane zmienne, jeśli nie nastąpi próba ich użycia, choć zwykle nie ma to sensu).

Ć W I C Z E N I E

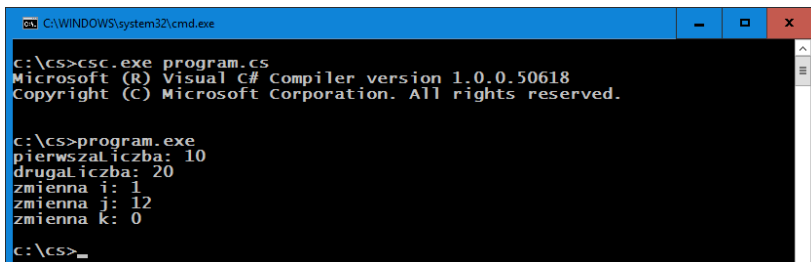
2.4 Inicjalizacja zmiennych

Popraw kod z ćwiczenia 2.3 tak, aby nie występowały błędy kompilacji. Skompiluj i uruchom otrzymany kod (rysunek 2.3).

```
using System;

public class main
{
    public static void Main()
    {
        int pierwszaLiczba = 10, drugaLiczba = 20, i, j, k;
        i = 1;
        j = 12;
        k = 0;
        Console.WriteLine("pierwszaLiczba: " + pierwszaLiczba);
        Console.WriteLine("drugaLiczba: " + drugaLiczba);
    }
}
```

```
        Console.WriteLine("zmienna i: " + i);  
        Console.WriteLine("zmienna j: " + j);  
        Console.WriteLine("zmienna k: " + k);  
    }  
}
```



```
C:\WINDOWS\system32\cmd.exe  
c:\cs>csc.exe program.cs  
Microsoft (R) Visual C# Compiler version 1.0.0.50618  
Copyright (C) Microsoft Corporation. All rights reserved.  
  
c:\cs>program.exe  
pierwsza liczba: 10  
druga liczba: 20  
zmienna i: 1  
zmienna j: 12  
zmienna k: 0  
c:\cs>_
```

Rysunek 2.3. Prawidłowo zainicjalizowane zmienne mogą zostać użyte w programie

Nazewnictwo zmiennych

Przy nazywaniu zmiennych obowiązują pewne zasady. Nazwa taka może składać się z dużych i małych liter oraz cyfr i znaków podkreślenia, ale nie może się zaczynać od cyfry. Stosowanie polskich znaków (ogólniej: wszelkich znaków narodowych) jest dozwolone, choć z reguły nie korzysta się z tej możliwości, ograniczając się wyłącznie do znaków alfabetu łacińskiego (zależy to jednak od danego projektu programistycznego; często też stosuje się nazwy wywodzące się z języka angielskiego). Nazwa zmiennej powinna także odzwierciedlać funkcję pełnioną w programie. Jeżeli zmienna określa np. liczbę punktów w jakimś zbiorze, najlepiej ją nazwać `liczbaPunktow` lub nawet `liczbaPunktow ↪WZbiorze`. Mimo że tak długa nazwa może wydawać się dziwna, poprawia jednak bardzo czytelność programu oraz ułatwia jego analizę. Naprawdę warto stosować ten sposób. Często przyjmuje się też, co również jest bardzo wygodne, że nazwę zmiennej rozpoczynamy małą literą, a poszczególne człony tej nazwy (wyrazy, które się na nią składają) piszemy wielką literą (tzw. zapis *lower camel case*), dokładnie tak jak w powyższych przykładach.

Typy odnośnikowe

Typy odnośnikowe, nazywane również referencyjnymi, służą do deklarowania zmiennych, które są odwołaniami do obiektów. Samymi obiektami zajmiemy się dopiero w rozdziale 4., w tej chwili przyjrzymy się tylko, w jaki sposób deklarujemy tego rodzaju zmienne.

Czynność ta jest wykonywana, podobnie jak w przypadku zmiennych typów podstawowych, za pomocą instrukcji w postaci:

```
typ_zmiennej nazwa_zmiennej;
```

lub:

```
typ_zmiennej nazwa_zmiennej_1, nazwa_zmiennej_2, nazwa_zmiennej_3;
```

W ten sposób zadeklarowane zostało jednak tylko tzw. odniesienie (ang. *reference*) do zmiennej obiektowej, a nie sam obiekt. Takiemu odniesieniu przypisana jest domyślnie wartość pusta (`null`), czyli praktycznie nie można wykonywać na nim żadnej operacji. Dopiero po utworzeniu odpowiedniego obiektu w pamięci można powiązać go z tak zadeklarowaną zmienną. Jeśli zatem napiszemy:

```
int a;
```

uzyskamy gotową do użycia zmienną typu całkowitego. Możemy jej przypisać np. wartość 10. Żeby jednak można było skorzystać z tablicy, musimy zadeklarować zmienną odnośnikową typu tablicowego, utworzyć obiekt tablicy i powiązać go ze zmienną. Dopiero wtedy będziemy mogli swobodnie odwoływać się do kolejnych elementów. Pisząc zatem:

```
int[] tablica;
```

zadeklarujemy odniesienie do tablicy, która będzie zawierała elementy typu `int`, czyli 32-bitowe liczby całkowite. Sama tablica powstanie dopiero po przypisaniu:

```
int[] tablica = new int[wielkość_tablicy];
```

Tym tematem zajmiemy się bliżej w rozdziale 5.

Typ string

Typ `string` służy do reprezentacji łańcuchów znakowych, inaczej napisów. Jeśli chcemy umieścić w programie łańcuch znaków, napis, należy go ująć w cudzysłów prosty, czyli na jego początku i na końcu umieścić znaki cudzysłowu¹, np.:

```
"To jest napis"
```

Nie jest to jednak typ bezpośrednio wbudowany w język, jak ma to miejsce w Pascalu, ale typ referencyjny (odnośnikowy) — podobnie jak w Javie. Nie ma konieczności jawnego wywoływania konstruktora klasy `String`, czyli zmienne można deklarować tak jak zmienne typów prostych, np.:

```
string zmienna = "napis";
```

Po takiej deklaracji utworzony zostanie jednak obiekt klasy (typu) `string`, zatem na zmiennej `zmienna` możemy wykonywać dowolne operacje możliwe do wykonania na klasie `string`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
<code>\a</code>	Sygnał dźwiękowy (ang. <i>alert</i>)	0x0007
<code>\b</code>	Cofnięcie o jeden znak (ang. <i>backspace</i>)	0x0008
<code>\f</code>	Nowa strona (ang. <i>form feed</i>)	0x000C
<code>\n</code>	Nowa linia (ang. <i>new line</i>)	0x000A
<code>\r</code>	Powrót karetki (przesunięcie na początek linii, ang. <i>carriage return</i>)	0x000D

¹ W rzeczywistości jest to znak zastępczy cudzysłowu, pseudocudzysłów. W książce będzie jednak używane popularne określenie „cudzysłów” (choć formalnie nie jest ono w pełni prawidłowe).

² Dokładniej rzecz ujmując, słowo `string` jest używanym w C# aliasem dla klasy `String` zdefiniowanej w przestrzeni nazw `System` i będącej bezpośrednim odwzorowaniem typu wspólnego `String` występującego w platformie .NET. Również pozostałe typy proste są aliasami dla struktur z przestrzeni nazw `System`.

Tabela 2.3. Sekwencje znaków specjalnych — ciąg dalszy

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

Typ object

Typ `object` jest typem nadrzędnym, z którego wyprowadzone są wszystkie inne typy danych. Nazwa `object` jest aliasem dla typu `Object` zdefiniowanego w przestrzeni nazw `System` (`System.Object`). Nie będziemy zagłębiać się w niuanse wewnętrznej realizacji typów w C#, warto jedynie wspomnieć, że w rzeczywistości nawet typy proste są typami referencyjnymi, a zmienne tych typów zachowują się (choć z pewnymi ograniczeniami) jak zmienne typów obiektowych. Oznacza to np., że można dla takiej zmiennej wywołać metody danej klasy (bądź struktury). Oto przykład:

```
int a = 10;  
string b = a.ToString();
```

Co więcej, możliwe jest również zastosowanie konstrukcji:

```
string b = 10.ToString();
```

Te wiadomości nie są jednak niezbędne na początku nauki C#. Podane tu wyjaśnienia staną się też bardziej zrozumiałe po przeczytaniu rozdziału 4.

Typy wyliczeniowe

Do utworzenia typu wyliczeniowego jest używane słowo `enum` pozwalające na tworzenie wyliczeń. Wyliczenie to po prostu zbiór wartości, które będzie można przypisywać danym tego typu. Wyliczenie powstaje w następujący sposób:

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

Na przykład:

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

W takim przypadku poszczególne elementy wyliczenia otrzymają wartości domyślnego typu bazowego, którym jest `int`. Kompilator sam nada więc wartości poszczególnym elementom, zaczynając od 1 (czyli wartość `czerwony` wewnętrznie będzie reprezentowana przez 1, `zielony` — przez 2, a `niebieski` — przez 3).

Można też samodzielnie określić typ bazowy oraz wartości przypisane danym elementom wyliczenia. Wtedy stosuje się definicję rozszerzoną:

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

Na przykład:

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

Przy tym typem bazowym może być tylko taki, który reprezentuje wartości całkowite, czyli: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` i `ulong`.

Wartość null

Jest to pewien specjalny typ danych, który oznacza po prostu nic (`null`). Wartości te używane są, gdy trzeba wskazać, że dana zmienna referencyjna jest pusta, czyli nie został do niej przypisany żaden obiekt. Typ `null` występuje w większości obiektowych języków programowania, w Object Pascalu (języku, na którym oparte jest środowisko Delphi) zamiast `null` stosuje się słowo `nil`.

Operatory

Poznaliśmy już zmienne, musimy jednak wiedzieć, jakie możemy wykonywać na nich operacje. Operacje są wykonywane za pomocą różnych operatorów, np. odejmowania, dodawania, przypisania itd. Operatory te można podzielić na następujące grupy:

- ❑ arytmetyczne,
- ❑ bitowe,
- ❑ logiczne,
- ❑ przypisania,
- ❑ porównania,
- ❑ pozostałe.

Operatory arytmetyczne

Występujące w C# operatory arytmetyczne zostały przedstawione w tabeli 2.4. W praktyce korzystanie z większości 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.

Tabela 2.4. Operatory arytmetyczne w C#

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

Ć W I C Z E N I E

2.5 Proste operacje arytmetyczne

Zadeklaruj dwie zmienne typu całkowitego. Wykonaj na nich kilka operacji arytmetycznych. Wyniki wyświetl na ekranie.

```
using System;

public class main
{
    public static void Main()
    {
        int a, b, c;
        a = 10;
        b = 25;
        c = b - a;
        Console.WriteLine("a = " + a);
        Console.WriteLine("b = " + b);
        Console.WriteLine("b - a = " + c);
        c = a * b;
        Console.WriteLine("a * b = " + c);
    }
}
```

Deklarujemy tu trzy zmienne typu całkowitoliczbowego o nazwach a, b i c. Zmiennym a i b przypisujemy wartości liczbowe, odpowiednio 10 i 25, zmiennej c zaś wynik odejmowania b – a, czyli zawierać ona będzie liczbę 15. Kolejne kroki to wyświetlenie wyników dotychczasowych działań i przypisań na ekranie. Następnie przypisujemy zmiennej c wynik mnożenia a * b (czyli liczbę 250) i wartość tego działania również wyświetlamy na ekranie.

Do operatorów arytmetycznych należy również znak %, przy czym nie oznacza on obliczania procentów, ale dzielenie modulo (resztę z dzielenia). Na przykład wynik działania 12 % 5 wynosi 2 (bo 5 mieści się w 10 dwa razy i pozostawia resztę 2).

Ć W I C Z E N I E

2.6 Dzielenie modulo

Zadeklaruj kilka zmiennych. Wykonaj na nich operacje dzielenia modulo. Wyniki wyświetl na ekranie.

```
using System;

public class main
{
    public static void Main()
    {
        int a, b, c;
        a = 10;
        b = 25;
        c = b % a;
        Console.WriteLine("b % a = " + c);
        Console.WriteLine("11 % 3 = " + 11 % 3);
        c = a * b;
        Console.WriteLine("(a * b) % 120 = " + c % 120);
    }
}
```

W kodzie wykonywane są w sumie trzy różne operacje dzielenia modulo (uzyskiwania reszty z dzielenia). Pierwsza obejmuje dwie zmienne ($b \% a$), druga — bezpośrednio dwie wartości liczbowe ($11 \% 3$), a trzecia — zmienną i liczbę ($c \% 120$). Uzyskujemy więc trzy różne wyniki (5, 2 i 10), które są wyświetlane na ekranie. Warto zwrócić uwagę, że operacje uzyskiwania reszty w instrukcjach `Console.WriteLine` były wykonywane jako pierwsze, przed operacjami wykonywanymi za pomocą operatora `+` (oznaczającymi w tym przypadku tworzenie napisu), co wynika z priorytetów operatorów (ta kwestia jest opisana w jednym z kolejnych punktów). Formalnie w celu poprawienia czytelności można jednak zastosować zwykły nawias i odpowiednie linie kodu zapisać tak:

```
Console.WriteLine("11 % 3 = " + (11 % 3));
Console.WriteLine("(a * b) % 120 = " + (c % 120));
```

Kolejne operatory typu arytmetycznego to operator inkrementacji i dekrementacji. Operator inkrementacji, czyli zwiększania, powoduje przyrost wartości zmiennej o jeden. Ma on postać `++` i może występować w formie przyrostkowej (postinkrementacyjnej) bądź przedrostkowej (preinkrementacyjnej). Oznacza to, że jeśli istnieje zmienna o nazwie `x`, formą przedrostkową będzie `++x`, natomiast przyrostkową — `x++`.

Oba te wyrażenia zwiększą wartość zmiennej *x* o jeden, jednak wcale nie są one równoważne. Otóż *x++* zwiększa wartość zmiennej po jej wykorzystaniu, natomiast *++x* przed wykorzystaniem. Takie rozróżnienie może być niekiedy bardzo pomocne przy pisaniu programu.

ĆWICZENIE

2.7 Operator inkrementacji

Przeanalizuj poniższy kod. Nie uruchamiaj programu, ale zastanów się, jaki będzie wyświetlony ciąg liczb. Następnie po uruchomieniu kodu sprawdź swoje przypuszczenia.

```
using System;

public class main
{
    public static void Main()
    {
        /*1*/ int x = 1, 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);
    }
}
```

Dla ułatwienia poszczególne kroki w programie zostały oznaczone kolejnymi liczbami. Wynikiem działania tego kodu będzie ciąg liczb 2, 2, 3, 3, 6. Dlaczego? Na początku zmienna *x* przyjmuje wartość 1. W kroku 2. występuje operator *++x*, zatem najpierw jest ona zwiększana o jeden (*x* = 2), a dopiero potem wyświetlana na ekranie. W wierszu o numerze 3 jest odwrotnie. Najpierw jest wyświetlana wartość zmiennej *x* (*x* = 2), a dopiero potem zwiększana o 1 (*x* = 3). W wierszu 4. po prostu wyświetlamy wartość *x* (*x* = 3). W wierszu 5. najpierw zmiennej *y* przypisywana jest dotychczasowa wartość *x* (*x* = 3, *y* = 3), a następnie wartość *x* jest zwiększana o jeden (*x* = 4). W wierszu 6. wyświetlamy wartość *y* (*y* = 3). W wierszu 7. najpierw zwiększamy wartość *x* o jeden (*x* = 5), a następnie przypisujemy tę wartość zmiennej *y* (*y* = 5). W wierszu 8., ostatnim, zwiększamy *y* o jeden (*y* = 6) i wyświetlamy na ekranie.



Operator dekrementacji, czyli `--`, działa analogicznie do `++`, ale zamiast zwiększać wartości zmiennych — zmniejsza je, oczywiście zawsze o jeden.

Ć W I C Z E N I E

2.8 Operator dekrementacji

Zmień kod z ćwiczenia 2.7 tak, aby operator `++` został zastąpiony operatorem `--`. Następnie przeanalizuj działanie powstałego programu i sprawdź, czy otrzymany wynik jest taki sam jak na ekranie po uruchomieniu kodu.

```
using System;

public class main
{
    public static void Main()
    {
        /*1*/ int x = 1, 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);
    }
}
```

Tym razem wynikiem działania programu będzie ciąg liczb 0, 0, -1, -1, -4. Na początku zmienna `x` przyjmuje wartość 1. W kroku 2. występuje operator `--x`, zatem najpierw jest ona zmniejszana o jeden (`x = 0`), a dopiero potem wyświetlana na ekranie. W wierszu o numerze 3 jest odwrotnie. Najpierw wartość zmiennej `x` jest wyświetlana (`x = 0`), a dopiero potem zmniejszana o 1 (`x = -1`). W wierszu 4. po prostu wyświetlamy wartość `x` (`x = -1`). W wierszu 5. najpierw zmiennej `y` przypisywana jest dotychczasowa wartość `x` (`x = -1, y = -1`), a następnie wartość `x` jest zmniejszana o jeden (`x = -2`). W wierszu 6. wyświetlamy wartość `y` (`y = -1`). W wierszu 7. najpierw zmniejszamy wartość `x` o jeden (`x = -3`), a następnie przypisujemy tę wartość zmiennej `y` (`y = -3`). W wierszu 8., ostatnim, zmniejszamy `y` o jeden (`y = -4`) i wyświetlamy na ekranie.



Do operatorów arytmetycznych można także zaliczyć jednoargumentowe operatory zmiany znaku, które zapisujemy jako $+$ i $-$. Ich działanie jest zgodne z zasadami matematyki, a więc zapis $+$ wartość nie powoduje żadnej zmiany [np. $+1$ to plus jeden, natomiast $+(-1)$ to minus jeden], natomiast $-$ wartość powoduje zmianę wartości na przeciwną [np. $-(-1)$ to plus jeden, a $-(+1)$ i $-(1)$ to minus jeden].

Działania operatorów arytmetycznych na liczbach całkowitych nie trzeba chyba dokładniej wyjaśniać, z dwoma może wyjątkami. Otóż co się stanie, jeżeli wynik dzielenia dwóch liczb całkowitych nie będzie liczbą całkowitą? Na szczęście odpowiedź jest prosta: odrzucona będzie część ułamkowa. Zatem wynikiem działania $7/2$ w arytmetyce liczb całkowitych będzie 3 („prawdziwym” wynikiem jest 3,5, która to wartość jest zaokrąglana w dół do najbliższej liczby całkowitej, czyli 3).

Ć W I C Z E N I E

2.9 Dzielenie całkowitoliczbowe

Wykonaj dzielenie zmiennych typu całkowitego. Sprawdź rezultaty w sytuacji, gdy rzeczywisty wynik jest ułamkiem.

```
using System;

public class main
{
    public static void Main()
    {
        int a, b, c;
        a = 8;
        b = 3;
        c = 2;
        Console.WriteLine("a = " + a);
        Console.WriteLine("b = " + b);
        Console.WriteLine("c = " + c);
        Console.WriteLine("a / b = " + a / b);
        Console.WriteLine("a / c = " + a / c);
        Console.WriteLine("b / c = " + b / c);
    }
}
```

Wykonywane są tu trzy różne dzielenia. Pierwsze to a / b , czyli $8/3$. Jego rzeczywistym wynikiem jest 2,6, jednak ze względu na użycie arytmetyki całkowitoliczbowej na ekranie pojawi się 2, ponieważ część ułamkowa będzie utracona. Drugie dzielenie to a / c , czyli $8/2$.

Ono nie budzi żadnych wątpliwości. Wynikiem jest po prostu 4. W trzecim przypadku wykonywana operacja to b / c , czyli $3/2$. Właściwym wynikiem jest oczywiście 1,5, jednak skutek zaokrąglenia powstanie wartość 1.

Drugim problemem jest to, co się stanie, jeżeli przekroczymy zakres jakiejś zmiennej. Pamiętamy np., że zmienna typu `sbyte` jest zapisywana na 8 bitach i przyjmuje wartości od -128 do 127 (tabela 2.1). Spróbujmy zatem przypisać zmiennej tego typu wartość 128.

ĆWICZENIE

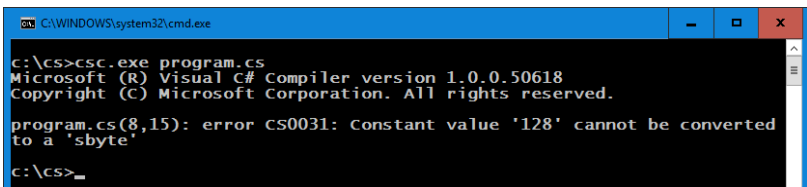
2.10 Przekroczenie dopuszczalnego zakresu zmiennej na etapie kompilacji kodu

Zadeklaruj zmienną typu `sbyte`. Przypisz jej wartość 128. Spróbuj wykonać kompilację otrzymanego kodu.

```
using System;

public class main
{
    public static void Main()
    {
        sbyte zmienna;
        zmienna = 128;
        Console.WriteLine("zmienna = " + zmienna);
    }
}
```

Kompilacja tego przykładu nie powiedzie się (rysunek 2.4). Kompilator wykryje, że próbujemy przekroczyć dopuszczalny zakres dla wartości typu `sbyte`, i nie pozwoli na to. Można powiedzieć, że w tego typu sytuacji nie ma problemu.



Rysunek 2.4. Próba przekroczenia dopuszczalnego zakresu zmiennej powoduje błąd kompilacji

Niestety, kompilator nie zawsze wykryje tego typu błąd. Może się zdarzyć, że zakres przekroczyliśmy nie w trakcie kompilacji, ale podczas wykonywania programu. Co się wtedy stanie? Można się o tym przekonać, wykonując kolejne ćwiczenie.

Ć W I C Z E N I E

2.11 Przekroczenie dopuszczalnego zakresu zmiennej na etapie wykonania programu

Zadeklaruj zmienną typu `sbyte` i przypisz jej wartość 127. Następnie wykonaj operację arytmetyczną zwiększającą wartość zmiennej, a zatem powodującą przekroczenie dopuszczalnej wartości. Wyświetl wynik na ekranie.

```
using System;

public class main
{
    public static void Main()
    {
        sbyte zmienna;
        zmienna = 127;
        zmienna++;
        Console.WriteLine("zmienna = " + zmienna);
    }
}
```

Po uruchomieniu kodu okaże się, że zmienna ma wartość... -128. Skąd ten wynik, przecież $127 + 1$ to z pewnością nie jest -128? Ponieważ maksymalną wartością dla typu `sbyte` jest 127, zmienna takiego typu nie może przyjąć prawidłowej wartości, czyli 128. Stąd następuje, można powiedzieć, „przekręcenie licznika”, co zobrazowano na rysunku 2.5.

Rysunek 2.5.
Przekroczenie
dopuszczalnego
zakresu
dla typu `sbyte`



Operatory bitowe

Operatory bitowe, jak sama nazwa wskazuje, służą do wykonywania operacji na bitach. Przypomnijmy zatem podstawowe wiadomości o systemach liczbowych. W systemie dziesiętnym wykorzystywanych jest dziesięć cyfr, od 0 do 9, w systemie szesnastkowym dodatkowo litery od A do F, a w systemie ósemkowym cyfry od 0 do 7. W systemie dwójkowym będą zatem wykorzystywane jedynie dwie cyfry — 0 i 1. Kolejne liczby budowane są z tych dwóch cyfr dokładnie tak samo jak w systemie dziesiętnym, przedstawiono to w tabeli 2.5. Widać wyraźnie, że np. 4 dziesiętnie to 100 dwójkowo, a 10 dziesiętnie to 1010 dwójkowo.

Tabela 2.5. Reprezentacja liczb w różnych systemach

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

Na tak zdefiniowanych liczbach można wykonywać znane ze szkoły operacje bitowe, takie jak AND (iloczyn bitowy), OR (suma bitowa), XOR (bitowa alternatywa wykluczająca) oraz negacja bitowa. Symbolem

operatora AND jest znak ampersand (&), operatora OR — pionowa kreśka (|), operatora XOR — daszek, strzałka w górę (^), negacji — tylda (~). Dodatkowo dostępne są także operacje przesunięć bitów w prawo i w lewo. Zestawienie tych operacji przedstawione jest w tabeli 2.6.

Tabela 2.6. Operatory bitowe

Rodzaj działania	Symbol w C#
bitowe AND	&
bitowe OR	
bitowe XOR	^
bitowa negacja	~
przesunięcie bitowe w lewo	<<
przesunięcie bitowe w prawo	>>

Operatory logiczne

Argumentami operacji takiego typu muszą być wyrażenia posiadające wartość logiczną, czyli true lub false (prawda lub fałsz). Przykładowo wyrażenie `10 < 20` jest niewątpliwie prawdziwe (10 jest mniejsze od 20), zatem jego wartość logiczna jest równa true. W grupie tej wyróżniamy trzy operatory: logiczne AND (&&, &), logiczne OR (||, |) i logiczną negację (!).

Warto zauważyć, że w części przypadków stosowania operacji logicznych w celu otrzymania wyniku wystarczy obliczyć tylko pierwszy argument. Wynika to oczywiście z właściwości operatorów, bo jeśli wynikiem obliczenia pierwszego argumentu jest wartość true, a wykonujemy operację OR, to niezależnie od stanu drugiego argumentu wartością całego wyrażenia będzie true. Podobnie jest przy stosowaniu operatora AND — jeżeli wartością pierwszego argumentu będzie false, to i wartością całego wyrażenia również będzie false.

Dlatego też w C# mamy po dwa operatory sumy oraz iloczynu. Jeśli użyjemy symboli & i |, przetwarzane będzie całe wyrażenie niezależnie od stanu argumentów, natomiast dla symboli && i || używane będzie opisane wyżej skrócone przetwarzanie wyrażeń (jeżeli obliczenie jednego argumentu umożliwia ustalenie wyniku, pozostałe argumenty nie są przetwarzane).

Operatory przypisania

Operacje przypisania są dwuargumentowe i powodują przypisanie wartości argumentu znajdującego się z prawej strony do umieszczonego z lewej. Najprostszym operatorem tego typu jest klasyczny znak równości, np.: `zmienna = 3`. Oprócz niego mamy do dyspozycji operatory łączące klasyczne przypisanie z innym operatorem arytmetycznym bądź bitowym. Operatory przypisania występujące w C# zostały przedstawione w tabeli 2.7.

Tabela 2.7. Operatory przypisania i ich znaczenie w C#

Argument1	Operator	Argument2	Znaczenie
x	=	y	x = y
x	+=	y	x = x + y
x	-=	y	x = x - y
x	*=	y	x = x * y
x	/=	y	x = x / y
x	%=	y	x = x % y
x	&=	y	x = x & y
x	=	y	x = x y
x	^=	y	x = x ^ y
x	<<=	y	x = x << y
x	>>=	y	x = x >> y

Operatory porównania (relacyjne)

Operatory porównania służą do porównywania argumentów. Wynikiem takiego porównania jest wartość logiczna `true` (jeśli jest ono prawdziwe) lub `false` (jeśli jest fałszywe), np. wyrażenie `2 < 3` ma wartość `true`, bo 2 jest mniejsze od 3, a wyrażenie `4 < 1` ma wartość `false`, bo 4 jest większe, a nie mniejsze od 1. Do dyspozycji mamy operatory porównania zawarte w tabeli 2.8.

Tabela 2.8. Operatory porównania w C#

Operator	Opis
==	Zwraca true, jeśli argumenty są sobie równe.
!=	Zwraca true, jeśli argumenty są różne.
>	Zwraca true, jeśli argument lewostronny jest większy od prawostronnego.
<	Zwraca true, jeśli argument lewostronny jest mniejszy od prawostronnego.
>=	Zwraca true, jeśli argument lewostronny jest większy lub równy prawostronnemu.
<=	Zwraca true, jeśli argument lewostronny jest mniejszy lub równy prawostronnemu.

Operator warunkowy (?:)

Operator warunkowy ma składnię następującą:

warunek ? *wartość1* : *wartość2*;

Zapis ten należy rozumieć tak: jeżeli *warunek* jest prawdziwy, wyrażenie przybiera *wartość1*, w przeciwnym razie — *wartość2*. Aby lepiej sobie to uzmysłować, wykonajmy proste ćwiczenie.

Ć W I C Z E N I E

2.12 Użycie operatora warunkowego

Wykorzystaj operator warunkowy do zmodyfikowania wartości dowolnej zmiennej typu całkowitego (int).

```
using System;

public class main
{
    public static void Main()
    {
        int x = 1, y;
        y = (x == 1) ? 10 : 20;
        Console.WriteLine("y = " + y);
    }
}
```

W powyższym ćwiczeniu najważniejszy jest wiersz:

```
y = (x == 1) ? 10 : 20;
```


który oznacza: jeżeli x jest równe 1, przypisz zmiennej y wartość 10, w przeciwnym razie przypisz zmiennej y wartość 20. Ponieważ zmienną x zainicjalizowaliśmy wartością 1, na ekranie zostanie wyświetlony ciąg znaków $y = 10$. Nawias okrągły użyty w wyrażeniu nie jest formalnie konieczny, ale zwiększa czytelność kodu programu i z reguły jest stosowany. Formalnie prawidłowa byłaby również instrukcja:

```
y = x == 1 ? 10 : 20;
```

Pozostałe operatory

Oprócz operatorów wymienionych wcześniej w języku C# występuje jeszcze kilkanaście innych, których dokładne omówienie wykracza poza ramy tematyczne tej książki. Zostały one jednak uwzględnione w tabeli w punkcie „Priorytety operatorów”.

Priorytety operatorów

Skoro znamy już operatory, musimy jeszcze wiedzieć, w jakiej kolejności są wykonywane. Wiadomo np., że mnożenie jest „silniejsze” od dodawania, zatem najpierw mnożymy, potem dodajemy. W C# jest podobnie, siła każdego operatora jest ściśle określona. Przedstawiono to w tabeli 2.9³. Im wyższa pozycja w tabeli, tym wyższy priorytet operatora. Operatory znajdujące się na jednym poziomie (w jednym wierszu) mają ten sam priorytet.

Tabela 2.9. Priorytety operatorów w C#

Operatory	Symbole
wybór składowej, wywołanie funkcji, indeksowanie tablicy, inkrementacja i dekrementacja przyrostkowa, tworzenie obiektów, kontrola typów, delegacje, ustalenie rozmiaru, dostęp do składowej z jednoczesną dereferencją	., f(), [], ++, --, new, typeof, checked, unchecked, delegate, sizeof ⁴ , ->

³ W tabeli uwzględniono również operatory występujące w C#, ale nieomawiane w książce.

⁴ Priorytet tego operatora został podwyższony w C# 5.0.

Tabela 2.9. Priorytety operatorów w C# — ciąg dalszy

Operatory	Symbole
ustalenie znaku wartości, negacje, inkrementacja i dekrementacja przedrostkowa, rzutowanie typów, wstrzymanie wykonania, uzyskanie adresu, dereferencja	+, -, !, ~, ++, --, ()x, await ⁴ , &x, *x ⁴
mnożenie, dzielenie, dzielenie modulo	*, /, %
dodawanie, odejmowanie	+, -
przesunięcia bitowe	<<, >>
relacyjne, testowanie typów	<, >, <=, >=, is, as
równość, różność	==, !=
bitowe AND	&
bitowe XOR	^
bitowe OR	
logiczne AND	&&
logiczne OR	
obsługa przypisania null (<i>null-coalescing</i>)	?? ⁵
warunkowy	?: ⁶
przypisania, lambda	=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, =, =>

Komentarze

Komentowanie tekstu źródłowego programu jest ważne ze względu na zachowanie przejrzystości. Zaniechanie tej czynności powoduje zwykle, że sam programista po pewnym czasie musi poświęcić dużo czasu na analizowanie własnego kodu. Oczywiście w krótkich programach, takich jak prezentowane w tej książce, zazwyczaj nie trzeba używać komentarzy, należy jednak wiedzieć, w jaki sposób można je stosować.

⁵ Operator dostępny, począwszy od C# 6.0.

⁶ Priorytet tego operatora został podwyższony w C# 3.0.

W C# istnieją dwa sposoby zastosowania komentarza. Oba zapożyczone z języków programowania C, C++ czy Java. Pierwszy typ składa się z dwóch ukośników (//), komentarz zaczyna się wtedy od miejsca wystąpienia tych dwóch znaków i obowiązuje do końca danej linii.

Drugi rodzaj komentarza zaczyna się od sekwencji znaków /*, a kończy sekwencją */. Jest to tzw. komentarz blokowy, jako że cały blok tekstu znajdujący się pomiędzy wymienionymi sekwencjami znaków jest ignorowany przez kompilator.

Należy pamiętać, że komentarzy blokowych nie wolno zagnieżdżać. Użycie sekwencji w postaci:

```
/*Komentarz blokowy  
/*Komentarz zagnieżdżony*/  
*/
```

spowoduje błąd kompilacji. Nic nie stoi natomiast na przeszkodzie, aby wewnątrz komentarza blokowego użyć komentarza składającego się z dwóch ukośników:

```
/*Komentarz blokowy  
// Komentarz wewnętrzny  
*/
```

Ć W I C Z E N I E

2.13 Komentarz wierszowy

Użyj komentarza składającego się z dwóch ukośników do opisanego fragmentu kodu programu.

```
using System;  
  
public class main  
{  
    public static void Main()  
    {  
        // inicjalizacja zmiennych  
        int x = 1, y;  
  
        // stwierdzenie, czy x jest równe 1  
        y = (x == 1)? 10 : 20;  
  
        // wypisanie wyników na ekranie  
        Console.WriteLine("y = " + y);  
    }  
}
```


Ć W I C Z E N I E

2.14 Komentarz blokowy

Użyj komentarza blokowego w kodzie programu z ćwiczenia 2.12.

```
using System;

public class main
{
    public static void Main()
    {
        int x = 1, y;
        /*
        Przykład użycia komentarza blokowego.
        W tym miejscu korzystamy z wyrażenia warunkowego.
        */
        y = (x == 1)? 10 : 20;
        Console.WriteLine("y = " + y);
    }
}
```



3

Instrukcje

Instrukcje warunkowe

Instrukcja if...else

Bardzo często w programie trzeba sprawdzić jakiś warunek i od tego, czy jest on prawdziwy, czy fałszywy, zależy dalsze wykonywanie różnych instrukcji. Do takiego sprawdzania służy właśnie instrukcja warunkowa if...else. Ma ogólną postać:

```
if (wyrażenie warunkowe)
{
    // instrukcje do wykonania, jeżeli warunek jest prawdziwy
}
else
{
    // instrukcje do wykonania, jeżeli warunek jest fałszywy
}
```

Może występować w wersji skróconej bez bloku else:

```
if (wyrażenie warunkowe)
{
    // instrukcje do wykonania, jeżeli warunek jest prawdziwy
}
```

Jeżeli w bloku if bądź else występuje tylko jedno polecenie, dopuszcza się pominięcie nawiasu klamrowego:

```
if (wyrażenie warunkowe)
    instrukcja1;
else
    instrukcja2;
```

Ta forma nie będzie jednak stosowana w książce (powoduje ona nie-spójność kodu i zwiększa podatność na pomyłki¹).

Spróbujmy zatem wykorzystać instrukcję if...else do sprawdzenia, czy zmienna całkowita jest mniejsza od 0.

Ć W I C Z E N I E

3.1 Wykorzystanie instrukcji if

Wykorzystaj instrukcję warunkową if...else do zweryfikowania, czy wartość zmiennej arytmetycznej jest mniejsza od 0. Wyświetl odpowiedni komunikat na ekranie.

```
using System;

public class Program
{
    public static void Main(string[] args)
    {
        int liczba = -5;
        if (liczba < 0)
        {
            Console.WriteLine("Zmienna jest mniejsza od zera.");
        }
        else
        {
            Console.WriteLine("Zmienna nie jest mniejsza od zera.");
        }
    }
}
```

¹ Choć nie jest to pogląd wyznawany przez wszystkich programistów.

Zawarta w kodzie instrukcja `if` bada warunek `liczba < 0`. Jeżeli warunek jest prawdziwy (czyli wartość zapisana w zmiennej `liczba` jest mniejsza od 0), wykonywana jest instrukcja z bloku `if`. Gdy warunek jest fałszywy (czyli wartość zapisana w zmiennej `liczba` nie jest mniejsza od 0), wykonywana jest instrukcja z bloku `else`. Ponieważ na początku zmiennej `liczba` została przypisana wartość `-5`, zostanie wykonany blok `if` i na ekranie pojawi się odpowiedni napis.

A teraz coś nieco bardziej skomplikowanego. Zajmijmy się klasycznym przykładem obliczania pierwiastków równania kwadratowego. Mamy równanie w postaci:

$$A \times x^2 + B \times x + C = 0$$

Aby obliczyć jego rozwiązanie, liczymy tzw. deltę (Δ), która równa jest:

$$B^2 - 4 \times A \times C$$

Jeżeli delta jest większa od 0, mamy dwa pierwiastki:

$$x_1 = \frac{-B + \sqrt{\Delta}}{2 \times A}$$

$$x_2 = \frac{-B - \sqrt{\Delta}}{2 \times A}$$

Jeżeli delta jest równa 0, istnieje tylko jedno rozwiązanie:

$$x = \frac{-B}{2 \times A}$$

W trzecim przypadku, jeżeli delta jest mniejsza od 0, równanie nie ma rozwiązań w zbiorze liczb rzeczywistych. Skoro jest tutaj tyle warunków do sprawdzenia, stanowi to doskonały przykład do potrenowania zastosowania instrukcji `if...else`. Aby nie komplikować zagadnienia, nie będziemy się w tej chwili zajmować wczytywaniem parametrów równania z klawiatury, ale podamy je bezpośrednio w kodzie.

Przed przystąpieniem do realizacji tego zadania trzeba się jeszcze dowiedzieć, w jaki sposób można uzyskać pierwiastek z danej liczby. Na szczęście nie jest to wcale skomplikowane, wystarczy skorzystać z konstrukcji `Math.Sqrt(wartość)`. Aby zatem dowiedzieć się, jaki jest pierwiastek kwadratowy z liczby 4, należy napisać:

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

Oczywiście zamiast liczby można też podać w takim wywołaniu zmienną, a wynik działania wypisać na ekranie, np.:

```
int pierwszaLiczba = 4;  
double drugaLiczba = Math.Sqrt(pierwszaLiczba);  
Console.WriteLine(drugaLiczba);
```

ĆWICZENIE

3.2 Obliczanie pierwiastków równania kwadratowego

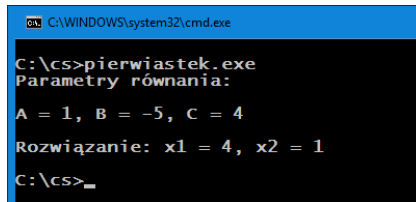
Wykorzystaj operacje arytmetyczne oraz instrukcję `if...else` do obliczenia pierwiastków równania kwadratowego o parametrach podanych bezpośrednio w kodzie programu. Wyniki wyświetl na ekranie (rysunek 3.1).

```
using System;  
  
public class Pierwiastek  
{  
    public static void Main(string[] args)  
    {  
        double parametrA = 1, parametrB = -5, parametrC = 4;  
  
        Console.WriteLine("Parametry równania:\n");  
        Console.WriteLine("A = " + parametrA + ", B = " + parametrB +  
            ", C = " + parametrC + "\n");  
  
        if (parametrA == 0)  
        {  
            Console.WriteLine("To nie jest równanie kwadratowe: A = 0!");  
        }  
        else  
        {  
            double delta = parametrB * parametrB - 4 * parametrA * parametrC;  
            if (delta < 0)  
            {  
                Console.WriteLine("Delta < 0.");  
                Console.WriteLine(  
                    "Brak rozwiązań w zbiorze liczb rzeczywistych");  
            }  
            else  
            {  
                double wynik;  
                if (delta == 0)  
                {  
                    wynik = - parametrB / (2 * parametrA);  
                    Console.WriteLine("Rozwiązanie: x = " + wynik);  
                }  
                else
```



```
{
    wynik = (- parametrB + Math.Sqrt(delta)) / (2 * parametrA);
    Console.Write("Rozwiązanie: x1 = " + wynik);
    wynik = (- parametrB - Math.Sqrt(delta)) / (2 * parametrA);
    Console.WriteLine(", x2 = " + wynik);
}
}
```

Rysunek 3.1.
*Wynik działania
programu
obliczającego
pierwiastki
równania
kwadratowego*



Na początku kodu są definiowane zmienne określające parametry równania, a ich wartości wyświetlane są na ekranie. Następnie za pomocą pierwszej instrukcji warunkowej `if` badany jest stan parametru `A`, czyli zmiennej `parametrA`. Po stwierdzeniu, że zmienna `parametrA` równa jest 0, wyświetlana jest informacja, że nie mamy do czynienia z równaniem kwadratowym. W przeciwnym wypadku wykonywane są instrukcje bloku `else`. Jest w nim obliczana delta, a następnie wykonywana kolejna instrukcja `if` badająca, czy delta jest mniejsza od 0. Jeśli jest mniejsza, wyświetlana jest informacja o braku rozwiązań. W przeciwnym razie (delta większa od 0 bądź równa 0) w bloku `else` jest wykonywana kolejna instrukcja warunkowa badająca tym razem, czy delta jest równa 0. Gdy jest równa 0, jedyny wynik jest obliczany i wyświetlany na ekranie. W przeciwnym przypadku (delta większa od 0) obliczane są dwa pierwiastki i oba rozwiązania pojawiają się na ekranie.

Instrukcja `if...else if`

Jak pokazano w ćwiczeniu 3.2, instrukcję warunkową można zagnieżdżać, tzn. wewnątrz jednego bloku `if` może występować kolejna taka instrukcja, a w niej następna itd. Taka budowa kodu powoduje jednak, że przy wielu zagnieżdżeniach staje się bardzo nieczytelny.

Aby tego uniknąć, można wykorzystać instrukcję `if...else if`. Zamiast stworzyć mniej wygodną konstrukcję, taką jak przedstawiona poniżej:

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

całość można zapisać dużo prościej i czytelniej w następującej postaci:

```
if (warunek1){  
    // instrukcje 1  
}  
else if (warunek2){  
    // instrukcje 2  
}  
else if (warunek3){  
    // instrukcje 3  
}  
else{  
    // instrukcje 4  
}
```

Ć W I C Z E N I E

3.3 Wykorzystanie instrukcji `if...else if`

Zmodyfikuj program napisany w ćwiczeniu 3.2 tak, aby używana była instrukcja `if...else if`.

```
using System;  
  
public class Pierwiastek  
{  
    public static void Main(string[] args)  
    {  
        double parametrA = 1, parametrB = -5, parametrC = 4;  
  
        Console.WriteLine("Parametry równania:\n");  
    }  
}
```

```
Console.WriteLine("A = " + parametrA + ", B = " + parametrB +
    ", C = " + parametrC + "\n");

if (parametrA == 0)
{
    Console.WriteLine("To nie jest równanie kwadratowe: A = 0!");
}
else
{
    double delta = parametrB * parametrB - 4 * parametrA * parametrC;
    double wynik;

    if (delta < 0)
    {
        Console.WriteLine("Delta < 0.");
        Console.WriteLine(
            "Brak rozwiązań w zbiorze liczb rzeczywistych");
    }
    else if (delta == 0)
    {
        wynik = -parametrB / (2 * parametrA);
        Console.WriteLine("Rozwiązanie: x = " + wynik);
    }
    else
    {
        wynik = (-parametrB + Math.Sqrt(delta)) / (2 * parametrA);
        Console.WriteLine("Rozwiązanie: x1 = " + wynik);
        wynik = (-parametrB - Math.Sqrt(delta)) / (2 * parametrA);
        Console.WriteLine("x2 = " + wynik);
    }
}
}
```

Instrukcja switch

Kiedy do sprawdzenia jest wiele warunków, instrukcja switch pozwala wygodnie zastąpić ciągi instrukcji if...else if. Jeśli mamy na przykład fragment kodu:

```
if (liczba == 1){
    // instrukcje1
}
else if (liczba == 2){
    // instrukcje2
}
```

```
else if (liczba == 3){  
    // instrukcje3  
}  
else{  
    // instrukcje4  
}
```

to możemy zastąpić go poniższym:

```
switch (liczba){  
    case 1:  
        // instrukcje1;  
        break;  
    case 2:  
        // instrukcje2;  
        break;  
    case 3:  
        // instrukcje3;  
        break;  
    default:  
        // instrukcje4;  
        break;  
}
```

Sprawdzamy tu po kolei, czy zmienna `liczba` jest równa 1, potem 2 i w końcu 3. Jeżeli zgodność zostanie stwierdzona, wykonywane są instrukcje po odpowiedniej klauzuli `case`. Jeżeli wartość zapisana w `liczba` nie jest równa żadnej z wymienionych liczb, wykonywane są instrukcje po słowie `default`. Instrukcja `break` powoduje wyjście z bloku `switch`. Jeśli zatem `liczba` będzie równa 1, zostaną wykonane instrukcje `instrukcje1`, jeśli `liczba` będzie równa 2, zostaną wykonane instrukcje `instrukcje2`, jeśli `liczba` będzie równa 3, zostaną wykonane instrukcje `instrukcje3`. Gdyby `liczba` nie była równa ani 1, ani 2, ani 3, zostaną wykonane instrukcje `instrukcje4`.

Ć W I C Z E N I E

3.4 Użycie instrukcji switch

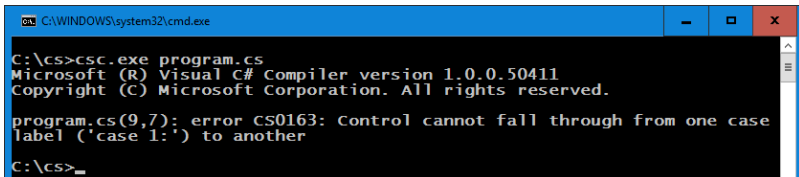
Zadeklaruj zmienną typu całkowitego i przypisz jej dowolną wartość. Korzystając z instrukcji `switch`, sprawdź, czy wartość ta równa jest 1 lub 2. Wyświetl odpowiedni komunikat na ekranie.

```
using System;  
  
public class Program  
{  
    public static void Main(string[] args)  
    {  

```

```
int liczba = 1;
switch (liczba){
    case 1:
        Console.WriteLine("liczba = 1");
        break;
    case 2:
        Console.WriteLine("liczba = 2");
        break;
    default:
        Console.WriteLine(
            "Zmienna liczba nie jest równa ani 1, ani 2.");
        break;
}
}
```

Warto w tym miejscu zauważyć, że w C#, inaczej niż w językach takich jak C, C++ czy Java, nie można pominąć instrukcji `break` powodującej wyjście z instrukcji `switch`. Próba taka skończy się błędem kompilacji (rysunek 3.2). Ściślej rzecz ujmując, nie musi być to dokładnie instrukcja `break`, ale dowolna instrukcja, w wyniku której zostanie opuszczony blok `switch`. Dokładniejsze wyjaśnienie i przykład takiej konstrukcji znajduje się przy opisie instrukcji `goto`.



Rysunek 3.2. Próba pominięcia instrukcji `break` kończy się błędem kompilacji

Pętle

Pętle w językach programowania pozwalają na wykonywanie powtarzających się czynności. Dokładnie w ten sam sposób działają one również w C#. Jeśli chcemy np. wypisać na ekranie 10 razy napis C#, możemy zrobić to, pisząc 10 razy `Console.WriteLine("C#");`. Lepiej jednak wykorzystać w tym celu pętle.

Pętla for

Pętla typu for ma następującą składnię:

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

wyrażenie początkowe jest stosowane do zainicjalizowania zmiennej używanej jako licznik liczby wykonań pętli; *wyrażenie warunkowe* określa warunek, jaki musi być spełniony, aby wykonać kolejne przejście w pętli; *wyrażenie modyfikujące* używane jest zwykle do modyfikacji zmiennej będącej licznikiem. To jednak podział zupełnie umowny, ponieważ ta konstrukcja jest bardzo elastyczna i możliwe są rozmaite modyfikacje jej podstawowej formy.

Ć W I C Z E N I E

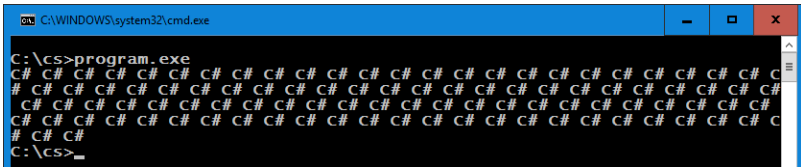
3.5 Prosta pętla typu for

Wykorzystując pętlę typu for, napisz program wyświetlający na ekranie 100 razy napis C#.

```
using System;

public class Program
{
    public static void Main(string[] args)
    {
        for (int i = 1; i <= 100; i++)
        {
            Console.Write("C# ");
        }
    }
}
```

Wynik działania tego prostego programu widoczny jest na rysunku 3.3. Zmienna *i* to tzw. **zmienna iteracyjna**, której na początku przypisujemy wartość 1 (`int i = 1`). Następnie w każdym przebiegu pętli jest ona zwiększana o 1 (`i++`) i wykonywana jest instrukcja `Console.WriteLine("C# ");`. Wszystko trwa tak długo, aż *i* osiągnie wartość 100 (`i <= 100`). Warto zwrócić uwagę, że zamiast `Console.WriteLine` użyty został zapis `Console.Write`. Dzięki temu kolejne napisy wyświetlane są obok siebie, bez przechodzenia za każdym razem do nowego wiersza (przejście jest wykonywane tylko wtedy, gdy napisy nie mieszczą się w danym wierszu).



Rysunek 3.3. Wynik działania pętli *for* z ćwiczenia 3.5

Wyrażenie modyfikujące jest zwykle używane do modyfikacji zmiennej iteracyjnej. Taką modyfikację można jednak wykonać również wewnątrz pętli. Struktura tego typu wygląda następująco:

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

Ć W I C Z E N I E

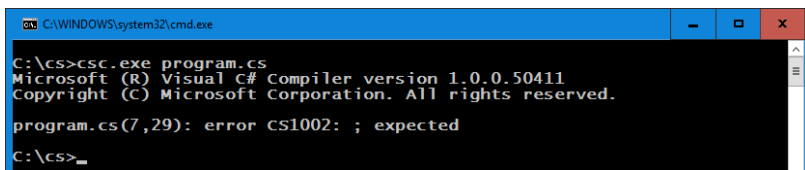
3.6 Wyrażenie modyfikujące w bloku instrukcji

Zmodyfikuj kod z ćwiczenia 3.5 tak, aby wyrażenie modyfikujące znalazło się w bloku instrukcji znajdujących się wewnątrz pętli.

```
using System;

public class Program
{
    public static void Main(string[] args)
    {
        for (int i = 1; i <= 100;)
        {
            Console.Write("C# ");
            i++;
        }
    }
}
```

Należy zwrócić uwagę, że mimo iż wyrażenie modyfikujące w ćwiczeniu 3.6 jest wewnątrz pętli, średnik znajdujący się po `i <= 100` jest niezbędny! Jeśli o nim zapomnimy, kompilator zgłosi błąd (rysunek 3.4).



```
C:\WINDOWS\system32\cmd.exe
C:\cs>csc.exe program.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

program.cs(7,29): error CS1002: ; expected

C:\cs>
```

Rysunek 3.4. Pominięcie średnika w pętli *for* powoduje błąd kompilacji

Kolejną ciekawą możliwością jest połączenie wyrażeń warunkowego i modyfikującego. Konkretnie należy spowodować, aby wyrażenie warunkowe było jednocześnie wyrażeniem modyfikującym. Tworzenie takich konstrukcji umożliwiają np. operatory inkrementacji i dekrementacji.

Ć W I C Z E N I E

3.7 Połączenie wyrażenia warunkowego i modyfikującego

Napisz taką pętlę typu *for*, aby wyrażenie warunkowe było jednocześnie wyrażeniem modyfikującym.

```
using System;

public class Program
{
    public static void Main(string[] args)
    {
        for (int i = 1; i++ <= 100;)
        {
            Console.WriteLine("C# ");
        }
    }
}
```

Tak jak w poprzednich przykładach można się tu pozbyć wyrażenia początkowego. Należy je przenieść przed pętlę. Schematyczna konstrukcja wygląda następująco:

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


Ć W I C Z E N I E

3.8 Wyrażenie początkowe przed pętlą

Zmodyfikuj pętlę typu `for` w taki sposób, aby wyrażenie początkowe znalazło się przed pętlą, a wyrażenie modyfikujące — wewnątrz pętli.

```
using System;

public class Program
{
    public static void Main(string[] args)
    {
        int i = 1;
        for (; i <= 100;)
        {
            Console.Write("C# ");
            i++;
        }
    }
}
```

Skoro zaszliśmy już tak daleko w pozbywaniu się wyrażeń sterujących, usuńmy również wyrażenie warunkowe, bo i to jest możliwe!

Ć W I C Z E N I E

3.9 Usunięcie wszystkich wyrażeń

Zmodyfikuj pętlę typu `for` w taki sposób, aby wyrażenie początkowe znalazło się przed pętlą, natomiast wyrażenia modyfikujące i warunkowe — wewnątrz pętli.

```
using System;

public class Program
{
    public static void Main(string[] args)
    {
        int i = 1;
        for (; ; )
        {
            Console.Write("C# ");
            if (i++ >= 100) break;
        }
    }
}
```

Jak widać, i taką pętlę można utworzyć. Ponownie trzeba tu przypomnieć, że średniki (tym razem już dwa) w nawiasie występującym po `for` są niezbędne! Warto też zwrócić uwagę na zmianę kierunku nierówności. Wcześniej sprawdzaliśmy, czy `i` jest mniejsze bądź równe 100, a teraz sprawdzamy, czy jest większe bądź równe. Dzieje się tak dlatego, że we wcześniejszych ćwiczeniach kontrolowane było, czy pętla ma się dalej wykonywać, natomiast w obecnym sprawdzamy, czy ma się zakończyć. Przy okazji w przykładzie zostało pokazane, w jaki sposób wykorzystuje się instrukcję `break` w pętli (wcześniej była użyta w instrukcji `switch`; ćwiczenie 3.4). Służy ona do natychmiastowego przerywania wykonywania pętli. Gdy zatem wystąpi `break`, pętla kończy działanie.

Kolejna przydatna instrukcja, `continue`, powoduje rozpoczęcie następnej iteracji (przebiegu) pętli — w miejscu jej wystąpienia wykonywanie bieżącej iteracji jest przerywane i rozpoczyna się kolejny przebieg. Najlepiej zobaczyć to na konkretnym przykładzie.

Ć W I C Z E N I E

3.10 Instrukcja `continue` w pętli `for`

Napisz program wyświetlający na ekranie liczby od 1 do 20, które nie są podzielne przez 2. Skorzystaj z pętli `for` i instrukcji `continue`.

```
using System;

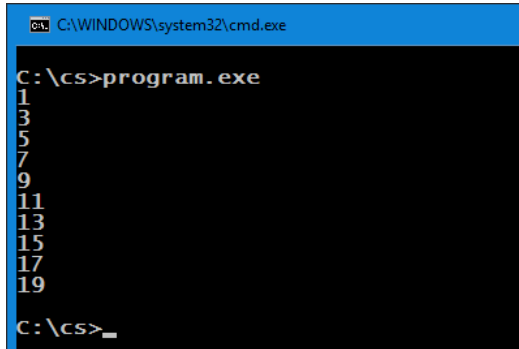
public class Program
{
    public static void Main(string[] args)
    {
        for (int i = 1; i <= 20; i++)
        {
            if (i % 2 == 0) continue;
            Console.WriteLine(i);
        }
    }
}
```

Wynik działania takiej aplikacji widoczny jest na rysunku 3.5. Jak pamiętamy, `%` to operator dzielenia modulo — dostarcza resztę z dzielenia. W każdym przebiegu sprawdzamy, czy `i` modulo 2 jest równe 0 (czy reszta z dzielenia `i` przez 2 jest równa 0). Jeśli jest (zatem `i` jest podzielne przez 2), przerywamy bieżącą iterację instrukcją `continue` (tym

samym rozpoczynamy kolejną iterację). W przeciwnym przypadku (i modulo 2 jest różne od 0) wykonujemy instrukcję `Console.WriteLine(i)`. Ostatecznie na ekranie otrzymamy wszystkie liczby od 1 do 20, które nie są podzielne przez 2.

Rysunek 3.5.

*Program
wyświetlający
liczby od 1 do 20
niepodzielne
przez 2*



```
C:\WINDOWS\system32\cmd.exe

C:\cs>program.exe
1
3
5
7
9
11
13
15
17
19
C:\cs>
```

Oczywiście ten sam program można napisać bez użycia instrukcji `continue`, co zostało pokazane w ćwiczeniu 3.11.

Ć W I C Z E N I E**3.11 Liczby niepodzielne
przez 2 bez instrukcji continue**

Zmodyfikuj kod z ćwiczenia 3.12 tak, aby nie było konieczności użycia instrukcji `continue`.

```
using System;

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

Tym razem instrukcja wyświetlająca wartość `i` jest wykonywana tylko wtedy, gdy prawdziwy jest warunek `i % 2 != 0`, a więc wtedy, gdy reszta z dzielenia `i` przez 2 jest różna od 0 (co oznacza, że wartość `i` jest nieparzysta). Dzięki takiej konstrukcji został osiągnięty taki sam efekt jak w ćwiczeniu 2.10, ale bez użycia instrukcji `continue`.

Pętla while

Pętla typu `for` służy raczej do wykonywania z góry znanej liczby operacji (wprowadzonej bezpośrednio lub zależnej np. od stanu zmiennej), w pętli `while` zwykle nie jest ona znana (może być np. wynikiem działania pewnej funkcji). Nie jest to jednak podział obligatoryjny. Pętlę `while` można napisać tak, aby była dokładnym funkcjonalnym odpowiednikiem pętli `for`, a pętlę `for` tak, aby była odpowiednikiem pętli `while`. Ogólna konstrukcja pętli typu `while` jest następująca:

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

Instrukcje są wykonywane dopóty, dopóki wyrażenie warunkowe jest prawdziwe. Oznacza to, że gdzieś w pętli musi nastąpić modyfikacja warunku bądź też musi być wywołana instrukcja `break` (ogólniej mówiąc, dowolna instrukcja powodująca opuszczenie pętli). Inaczej pętla będzie się wykonywała w nieskończoność!

Ć W I C Z E N I E

3.12 Wykorzystanie pętli while

Napisz przy użyciu pętli typu `while` program wyświetlający na ekranie 100 razy napis C#.

```
using System;

public class Program
{
    public static void Main(string[] args)
    {
        int i = 1;
        while (i <= 100)
        {
```

```
        Console.Write("C# ");  
        i++;  
    }  
}
```

Taka pętla działa, dopóki prawdziwy jest warunek `i <= 100`. Ponieważ początkową wartością `i` jest 1, a we wnętrzu pętli wartość tej zmiennej w każdym przebiegu jest zwiększana o 1 (`i++`), pętla (a tym samym instrukcja `Console.Write("C# ");`) zostanie wykonana dokładnie 100 razy.

Ć W I C Z E N I E

3.13 Połączenie wyrażenia modyfikującego i warunkowego

Zmodyfikuj kod z ćwiczenia 3.12 tak, aby wyrażenie warunkowe zmieniło jednocześnie wartość zmiennej `i`.

```
using System;  
  
public class Program  
{  
    public static void Main(string[] args)  
    {  
        int i = 1;  
        while (i++ <= 100)  
        {  
            Console.Write("C# ");  
        }  
    }  
}
```

Instrukcja `i++` występująca w ćwiczeniu 3.12 wewnątrz pętli została przeniesiona do wyrażenia warunkowego. Tym samym wartość zmiennej zmienia się już w tym wyrażeniu. Efekt działania będzie jednak ten sam — 100 napisów C#. Należy jednak zwrócić uwagę, że powstały program nie jest dokładnym odpowiednikiem poprzedniego kodu. Otóż wewnątrz pętli (między znakami `{ i }`) wartość `i` tym razem zmienia się nie od 1 do 100, ale od 2 do 101. Można się o tym przekonać, zmieniając instrukcję wyświetlającą napis w taki sposób, aby podawała również stan zmiennej `i`, np.:

```
Console.Write("C# (" + i + ") ");
```

Zmiana kodu na taki, który będzie dokładnym funkcjonalnym odpowiednikiem tego z ćwiczenia 3.12, pozostanie jednak zadaniem do samodzielnego wykonania.

ĆWICZENIE

3.14 Pętla while i liczby nieparzyste

Korzystając z pętli `while`, napisz program wyświetlający na ekranie liczby od 1 do 20 niepodzielne przez 2.

```
using System;

public class Program
{
    public static void Main(string[] args)
    {
        int i = 1;
        while (i <= 20)
        {
            if (i % 2 != 0)
            {
                Console.WriteLine(i);
            }
            i++;
        }
    }
}
```

To zadanie analogiczne do przedstawionego w ćwiczeniu 3.10. Bardzo podobna jest też struktura rozwiązania — wewnątrz pętli badany jest warunek `i % 2 != 0` i tylko wtedy, gdy jest on prawdziwy (i nie jest podzielne przez 2), wykonywana jest instrukcja wyświetlająca bieżącą wartość na ekranie. Użyty został wariant pętli `while` z ćwiczenia 3.12, w którym modyfikacja zmiennej `i` odbywa się wewnątrz pętli. Oczywiście możliwe byłoby również skorzystanie z wariantu, w którym modyfikacja jest wykonywana w wyrażeniu warunkowym (jak w ćwiczeniu 3.13), co jednak wymagałoby odpowiedniego dostosowania warunku i początkowego stanu zmiennej.

Pętla do...while

Oprócz przedstawionej już pętli `while` istnieje inna jej odmiana — jest to `do...while`. Jej konstrukcja jest następująca:

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

Zapis ten oznacza: wykonuj *instrukcje* dopóty, dopóki *warunek* jest prawdziwy. Spróbujemy zatem wykonać zadanie przedstawione w ćwiczeniu 3.12, ale skorzystamy z pętli typu `do...while`.

Ć W I C Z E N I E

3.15 Wykorzystanie pętli `do...while`

Napisz przy użyciu pętli `do...while` program wyświetlający na ekranie 100 razy napis `C#`.

```
using System;

public class Program
{
    public static void Main(string[] args)
    {
        int i = 1;
        do
        {
            Console.Write("C# ");
        }
        while (i++ < 100);
    }
}
```

Mogłoby się wydawać, że to przecież to samo co zwykła pętla `while`. Wydaje się wręcz, że to po prostu odwrócona pętla `while`. Jest jednak pewna różnica powodująca, że `while` i `do...while` nie są dokładnymi odpowiednikami. W pętli `do...while` instrukcje wykonane będą co najmniej raz, nawet wtedy, kiedy warunek jest na pewno fałszywy. Dzieje się tak dlatego, że sprawdzenie warunku zakończenia pętli odbywa się dopiero po jej pierwszym przebiegu.

Ć W I C Z E N I E

3.16 Pętla `do...while` z fałszywym warunkiem

Zmodyfikuj kod z ćwiczenia 3.15 w taki sposób, aby wyrażenie warunkowe na pewno było fałszywe. Zaobserwuj wyniki działania programu.

```
using System;

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

```
{
    int i = 101;
    do
    {
        Console.Write("C# ");
    }
    while (i++ < 100);
}
```

Tym razem mimo że warunek był fałszywy (początkowa wartość zmiennej *i* to 101, które na pewno jest większe niż użyte w wyrażeniu warunkowym 100), na ekranie pojawi się jeden napis C#.

Jeszcze dobitniej fakt wykonania jednego przebiegu pętli niezależnie od prawdziwości warunku można pokazać, stosując konstrukcję:

```
do
{
    Console.Write("C# ");
}
while (false);
```

W tym przypadku już na pierwszy rzut oka widać, że warunek jest fałszywy. Mimo to instrukcja z wnętrza pętli zostanie wykonana jeden raz.

Pętla foreach

Pętla typu `foreach` służy do przeglądania zawartości tablicy lub kolekcji obiektów. Jej opis zostanie podany w rozdziale 5.

Instrukcja goto

Instrukcja `goto`, czyli instrukcja umożliwiająca skok do określonego miejsca w programie, od dawna jest uznawana za niebezpieczną i powodującą wiele problemów. Niemniej w C# jest obecna, choć zaleca się jej stosowanie jedynie w przypadku bloków `switch...case` oraz wychodzenia z zagnieżdżonych pętli, w których to sytuacjach jest faktycznie użyteczna. Schemat jej użycia jest następujący:

```
goto etykieta;
```


gdzie *etykieta* jest zdefiniowanym miejscem w programie. Nie można jednak w ten sposób „wskoczyć” do bloku instrukcji, tzn. nie jest możliwa konstrukcja:

```
for(int j = 0; j < 500; j++)
{
    if(j == 100) goto label1;
}
for(int i = 0; i < 1000; i++)
{
    label1:;
    // instrukcje wnętrza pętli
}
```

Nic jednak nie stoi na przeszkodzie, aby zdefiniować etykietę przed lub za drugą pętlą, pisząc:

```
for(int j = 0; j < 500; j++)
{
    if(j == 100) goto label1;
}
label1:
for(int i = 0; i < 1000; i++)
{
    // instrukcje wnętrza pętli
}
```

Taki kod jest już jak najbardziej poprawny.

ĆWICZENIE

3.17 Przerwanie zagnieżdżonej pętli za pomocą goto

Napisz przykładowy program, w którym instrukcja goto jest wykorzystywana do wyjścia z zagnieżdżonej pętli for.

```
using System;

public class Program
{
    public static void Main()
    {
        for(int i = 0; i < 500; i++)
        {
            for(int j = 0; j < 1000; j++)
            {
                if((i == 100) && (j > 200))
                {
                    goto label1;
                }
            }
        }
    }
}
```

```
    }  
  }  
  return;  
  label1:  
    Console.WriteLine("Pętla została przerwana!");  
}  
}
```

Wykorzystujemy tutaj dwie pętle: zewnętrzną, gdzie zmienną iteracyjną jest *i*, oraz wewnętrzną, gdzie zmienną iteracyjną jest *j*. Wewnątrz drugiej pętli znajduje się warunek, który należy odczytywać następująco: jeżeli *i* równe jest 100 oraz *j* jest większe od 200, idź do miejsca w kodzie oznaczonego etykietą *label1*. Zatem po osiągnięciu warunku pętle zostaną przerwane, na ekranie zostanie wyświetlony napis *Pętla została przerwana*, a aplikacja zakończy działanie.

Należy zauważyć, że przed etykietą znajduje się instrukcja *return* powodująca zakończenie działania funkcji (metody) *Main* (bliższe wyjaśnienie tych kwestii zostanie podane w jednym z kolejnych rozdziałów), co jest równoznaczne z zakończeniem działania programu. Gdyby jej nie było, napis o przerwaniu pętli wyświetlany byłby zawsze, niezależnie od spełnienia warunku wewnątrz pętli. Oczywiście w tym przypadku warunek zawsze zostanie spełniony, zatem *return* nie jest niezbędne, ale już w prawdziwej aplikacji zapomnienie o tym drobnym z pozoru fakcie może przysporzyć wielu problemów.

Spróbujmy teraz wykorzystać instrukcję *goto* w drugim z zalecanych przypadków jej użycia, tzn. w bloku *switch...case*.

Ć W I C Z E N I E

3.18 Instrukcja *goto* w bloku *switch...case*

Napisz przykładowy kod, w którym instrukcja *goto* jest wykorzystywana w połączeniu z blokiem *switch...case*.

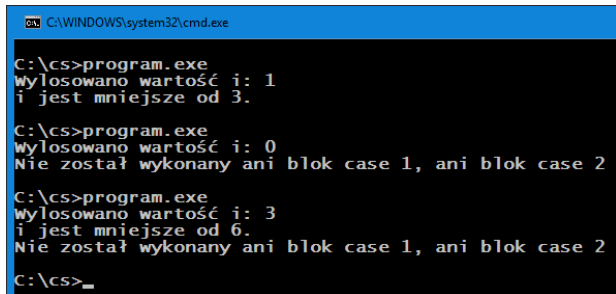
```
using System;  
  
public class Program  
{  
    public static void Main()  
    {  
        Random r = new Random();  
        int i = r.Next(5);  
        Console.WriteLine("Wylosowano wartość i: " + i);  
        switch(i)
```

```
{
    case 1 :
        Console.WriteLine("i jest mniejsze od 3.");
        break;
    case 2 :
        goto case 1;
    case 3 :
        Console.WriteLine("i jest mniejsze od 6.");
        goto default;
    case 4 :
        goto case 3;
    case 5 :
        goto case 3;
    default :
        Console.WriteLine(
            "Nie został wykonany ani blok case 1, ani blok case 2");
        break;
}
}
```

Przykładowe wyniki działania tej prostej aplikacji widoczne są na rysunku 3.6. Zmienną `i` typu `int` przypisujemy losową liczbę całkowitą z zakresu 0 – 5. Odpowiada za to linia `int i = r.Next(5);`, gdzie `r` jest zmienną wskazującą na obiekt typu `Random` utworzony za pomocą instrukcji `Random r = new Random();`. Taki obiekt służy właśnie do generowania losowych, a dokładniej pseudolosowych liczb.

Rysunek 3.6.

Przykładowe
wyniki działania
programu
z ćwiczenia 3.18



```
C:\WINDOWS\system32\cmd.exe
C:\>cs>program.exe
wylosowano wartość i: 1
i jest mniejsze od 3.
C:\>cs>program.exe
wylosowano wartość i: 0
Nie został wykonany ani blok case 1, ani blok case 2
C:\>cs>program.exe
wylosowano wartość i: 3
i jest mniejsze od 6.
Nie został wykonany ani blok case 1, ani blok case 2
C:\>cs>_
```

Następnie w bloku `switch...case` rozpatrywane są następujące sytuacje:

- ☐ `i` jest równe 0 — wyświetlamy napis `Nie został wykonany ani blok case 1, ani blok case 2;`
- ☐ `i` jest równe 1 lub 2 — wyświetlamy napis `i jest mniejsze od 3;`
- ☐ `i` jest równe 3, 4 lub 5 — wyświetlamy napis `i jest mniejsze od 6` oraz `nie został wykonany ani blok case 1, ani blok case 2.`



Wprowadzanie danych

Wiadomo już, jak wyprowadzać dane na konsolę, czyli po prostu jak wyświetlać je na ekranie. Bardzo przydałaby się jednak możliwość ich wprowadzania do programu. Nie jest to bardzo skomplikowane zadanie, choć napotkamy pewne trudności związane z koniecznością wykonywania konwersji typów danych.

Argumenty wiersza poleceń

Przekazywanie danych do aplikacji jako argumentów w wierszu poleceń przy wywoływaniu programu jest sposobem dobrze znanym większości programistów. Taka możliwość występuje prawie w każdym z popularnych języków programowania. Nie inaczej jest w C#, gdzie aby z tych danych skorzystać, należy odpowiednio zadeklarować funkcję Main. Konkretnie deklaracja powinna wyglądać następująco:

```
public static void main(string[] args)
{
    // instrukcje
}
```

Jak widać, argumenty są przekazywane do aplikacji w postaci tablicy obiektów typu string, czyli tablicy napisów. Ponieważ w C#, podobnie jak w Javie, tablice są obiektami, nie trzeba dodatkowo przekazywać liczby argumentów, jak ma to miejsce w C i C++. Rozmiar tablicy określany jest przez właściwość Length (więcej informacji o tablicach można znaleźć w rozdziale 5.).

Ć W I C Z E N I E

3.19 Odczytanie argumentów z wiersza poleceń

Wyświetl na ekranie wszystkie argumenty wywołania programu podane przez użytkownika w wierszu poleceń.

```
using System;

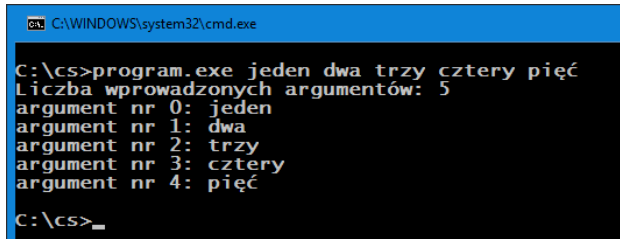
public class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine(
            "Liczba wprowadzonych argumentów: {0}", args.Length);
    }
}
```

```
for(int i = 0; i < args.Length; i++)  
{  
    Console.WriteLine("argument nr {0}: {1}", i, args[i]);  
}  
}
```

Dane są wyprowadzane na ekran (wyświetlane na ekranie) przy użyciu typowej pętli for. Przykładowy efekt działania programu widoczny jest na rysunku 3.7.

Rysunek 3.7.

*Program
wyświetlający
argumenty
podane
w wierszu
polecień*



```
C:\WINDOWS\system32\cmd.exe  
  
C:\cs>program.exe jeden dwa trzy cztery pięć  
Liczba wprowadzonych argumentów: 5  
argument nr 0: jeden  
argument nr 1: dwa  
argument nr 2: trzy  
argument nr 3: cztery  
argument nr 4: pięć  
  
C:\cs>_
```

Przy pracy z argumentami przekazywanymi z wiersza poleceń napotkamy pewien problem. Założmy, że chcemy napisać program, który dodaje do siebie dwie liczby całkowite i wyświetla wynik tego działania na ekranie. Wydawać by się mogło, że najprostsze rozwiązanie wygląda tak jak na poniższym listingu:

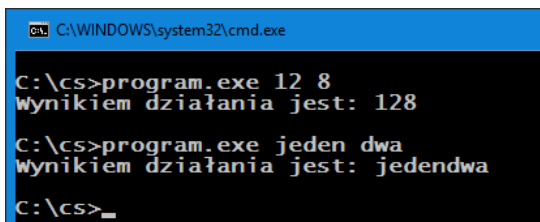
```
using System;  
  
public class main  
{  
    public static void Main(string[] args)  
    {  
        if(args.Length < 2)  
        {  
            Console.WriteLine(  
                "Należy podać dwa argumenty w wierszu poleceń!");  
            return;  
        }  
        Console.WriteLine(  
            "Wynikiem działania jest: {0}", args[0] + args[1]);  
    }  
}
```

Najpierw sprawdzamy, czy zostały przekazane co najmniej dwa argumenty. Jeśli tak nie jest (właściwość `Length` tablicy `args` ma wartość mniejszą niż 2), wyświetlamy stosowny komunikat na ekranie i opusz-

czamy metodę Main za pomocą instrukcji return (co jest równoznaczne z zakończeniem działania programu). Jeżeli jednak są co najmniej dwa argumenty, dodajemy je do siebie za pomocą operatora dodawania (`args[0] + args[1]`), a wynik wyświetlamy na ekranie (wynik zostanie wstawiony w miejscu oznaczonym `{0}`). Gdy uruchomimy taki program, może nas spotkać niespodzianka. Przykładowe wyniki zostały przedstawione na rysunku 3.8.

Rysunek 3.8.

*W wyniku
dodawania
zostały złączone
argumenty
przekazane
do programu*



```
C:\WINDOWS\system32\cmd.exe

C:\cs>program.exe 12 8
wynikiem działania jest: 128

C:\cs>program.exe jeden dwa
wynikiem działania jest: jedendwa

C:\cs>_
```

Oczywiście nie są prawidłowe. Powodem jest to, że argumenty pobrane z wiersza poleceń zawsze są ciągami znaków (przecież tablica `args` jest typu `string`) i w rzeczywistości wykonujemy operacje na dwóch napisach, a nie dwóch liczbach! Zatem wynikiem dodawania 12 i 8 będzie w powyższym programie 128 (napis) zamiast spodziewanego 20 (wartość całkowita). Aby aplikacja działała poprawnie, należy najpierw wykonać konwersję argumentów z typu `string` do typu `int`, a dopiero potem wykonywać działanie. Taką konwersję wykonamy za pomocą następującej konstrukcji:

```
int zmienna = Int32.Parse("zapis_liczby");
```

Ć W I C Z E N I E

3.20

Konwersja danych wprowadzanych w wierszu poleceń

Napisz program wykonujący dodawanie dwóch liczb podanych jako parametry w wierszu poleceń.

```
using System;

public class Program
{
    public static void Main(string[] args)
    {
        int l1, l2;
        if(args.Length < 2)
        {
```

```
        Console.WriteLine(
            "Należy podać dwa argumenty w wierszu poleceń!");
        return;
    }
    try
    {
        i1 = Int32.Parse(args[0]);
        i2 = Int32.Parse(args[1]);
    }
    catch(Exception)
    {
        Console.WriteLine(
            "Jeden z argumentów nie jest poprawną liczbą!");
        return;
    }
    Console.WriteLine("Wynikiem działania jest: {0}", i1 + i2);
}
}
```

Za konwersję z typu string (ciąg znaków) na typ int (wartość całkowita) odpowiada wspomniana wyżej instrukcja `Int32.Parse`². Odczytane wartości są zapisywane w zmiennych pomocniczych `i1` i `i2`. Dodatkowo poprzez zastosowanie bloku `try...catch` sprawdzane jest również, czy operacja ta zakończyła się sukcesem. Jeżeli konwersja się nie uda, zostaną wykonane instrukcje z bloku `catch` (wyświetlenie komunikatu i zakończenie działania programu). Jeśli konwersja się uda, wartość dodawania `i1` do `i2` jest wyświetlana na ekranie. (Dokładniejsze wytłumaczenie znaczenia bloku `try...catch` znajduje się w rozdziale 6., w którym opisano stosowanie wyjątków).

Powróćmy teraz do aplikacji powstałej w ćwiczeniach 3.2 i 3.3. Obliczała ona pierwiastki równania kwadratowego, jednak argumenty tego równania były podawane bezpośrednio w kodzie. Za każdym razem, kiedy następowała konieczność ich zmiany, trzeba było ponownie kompilować program. Wygoda tamtego rozwiązania pozostawiała więc wiele do życzenia. Teraz, kiedy wiadomo już, w jaki sposób stosować argumenty, podając je w wierszu poleceń, i jak wykonać konwersję danych, można pokusić się o spore usprawnienie tamtej aplikacji.

² Dokładniej rzecz ujmując, jest to wywołanie statycznej metody `Parse` struktury `Int32` zdefiniowanej w przestrzeni nazw `System`.

Ć W I C Z E N I E

3.21 Rozwiązywanie równań kwadratowych

Napisz program obliczający pierwiastki równania kwadratowego, w którym parametry równania są wprowadzane w wierszu poleceń.

```
using System;

class Pierwiastek
{
    public static void Main(string[] args)
    {
        double parametrA, parametrB, parametrC;

        if(args.Length < 3)
        {
            Console.WriteLine(
                "Wywołanie programu: pierwiastek.exe parametrA parametrB  

                 ↳parametrC");
            return;
        }

        try
        {
            parametrA = Double.Parse(args[0]);
            parametrB = Double.Parse(args[1]);
            parametrC = Double.Parse(args[2]);
        }
        catch(Exception)
        {
            Console.WriteLine(
                "Jeden z parametrów równania nie jest poprawną liczbą!");
            return;
        }

        Console.WriteLine("Wprowadzone parametry równania:\n");
        Console.WriteLine("A: " + parametrA + " B: " + parametrB +  

                           " C: " + parametrC + "\n");

        if (parametrA == 0)
        {
            Console.WriteLine("To nie jest równanie kwadratowe: A = 0!");
        }
        else
        {
            double delta = parametrB * parametrB - 4 * parametrA * parametrC;
            double wynik;

            if (delta < 0)
            {
```

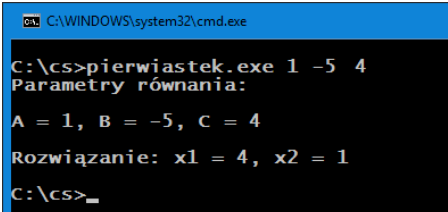


```
        Console.WriteLine("Delta < 0.");
        Console.WriteLine(
            "Brak rozwiązań w zbiorze liczb rzeczywistych.");
    }
    else if (delta == 0)
    {
        wynik = -parametrB / (2 * parametrA);
        Console.WriteLine("Rozwiązanie: x = " + wynik);
    }
    else
    {
        wynik = (-parametrB + Math.Sqrt(delta)) / (2 * parametrA);
        Console.WriteLine("Rozwiązanie: x1 = " + wynik);
        wynik = (-parametrB - Math.Sqrt(delta)) / (2 * parametrA);
        Console.WriteLine("x2 = " + wynik);
    }
}
}
```

W kodzie za pomocą pierwszej instrukcji `if` badane jest, czy z wiersza poleceń zostały przekazane co najmniej 3 argumenty. Jeżeli argumentów jest zbyt mało (wartość `args.Length` mniejsza niż 3), informujemy o tym użytkownika za pomocą komunikatu i kończymy działanie programu przez wywołanie instrukcji `return`. Jeżeli liczba argumentów jest właściwa (są co najmniej 3), następuje próba przetworzenia ich na wartości typu `Double`. Odbywa się to w sposób analogiczny do przedstawionego w ćwiczeniu 3.20, z tą różnicą, że została użyta instrukcja `Double.Parse(napis)` (struktura `Double` zamiast `Int32`). Dzięki temu ciągi znaków uzyskane z wiersza poleceń są przetwarzane na wartości rzeczywiste (typu `double`), a nie całkowite (typu `int`). Podobnie jak w ćwiczeniu 3.20 został też użyty blok `try...catch`, zatem zostanie obsługowana sytuacja, w jakiej program wywołano z ciągami znaków, które nie reprezentują prawidłowych liczb. Obliczenia pierwiastków równania są natomiast wykonywane na zasadach przedstawionych w ćwiczeniach 3.2 i 3.3. Efekt przykładowego wywołania programu został zaprezentowany na rysunku 3.9.

Rysunek 3.9.


*Równania
kwadratowe
o parametrach
podanych
w wierszu
poleceń*



```
C:\WINDOWS\system32\cmd.exe

C:\cs>pierwiastek.exe 1 -5 4
Parametry równania:
A = 1, B = -5, C = 4
Rozwiązanie: x1 = 4, x2 = 1
C:\cs>_
```

Należy zwrócić uwagę, że sposób wprowadzania liczb rzeczywistych z separatorem dziesiętnym zależy od ustawień regionalnych systemu. Konkretnie od tego, jaki symbol został ustalony jako separator dziesiętny. W ustawieniach polskich domyślnie jest to przecinek, przy ustawieniach angielskich — kropka.



Instrukcja ReadLine

Wprowadzanie danych do programu w wierszu poleceń nie zawsze jest wygodne. Często chcielibyśmy wykonywać tę czynność już w trakcie działania aplikacji. Jest to możliwe przy użyciu instrukcji³ `Console.ReadLine()`, która zwraca wprowadzoną przez użytkownika jedną linię tekstu (ciąg znaków zakończony znakiem końca linii).

Ć W I C Z E N I E

3.22 Wczytywanie wierszy tekstu

Napisz program, który w pętli wczytuje kolejne wiersze tekstu wprowadzane przez użytkownika i wyświetla je na ekranie. Program powinien zakończyć działanie po odczytaniu ciągu znaków `quit`.

```
using System;

public class Program
{
    public static void Main()
    {
        string str;
        Console.WriteLine(
            "Wprowadzaj ciągi znaków. Aby zakończyć, wpisz quit.");
        while((str = Console.ReadLine()) != "quit")
        {
            Console.WriteLine(str);
        }
    }
}
```

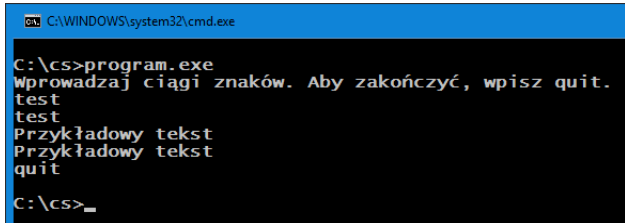
³ Dokładniej: dzięki statycznej metodzie `ReadLine` z klasy `Console`.

W kodzie została zadeklarowana pomocnicza zmienna `str` typu `string`, a za nią znajduje się pętla typu `while` odczytująca wiersze tekstu wprowadzane z klawiatury. Warunek pętli `while` ma tu skondensowaną formę. Wykonywanych jest kilka operacji. Najpierw następuje odczyt jednej linii tekstu (`Console.ReadLine()`), następnie uzyskany ciąg znaków przypisywany jest zmiennej `str` (`str = Console.ReadLine()`), w kolejnej fazie wartość tej zmiennej jest porównywana z ciągiem znaków `quit`.

Jeżeli zatem odczytany ciąg znaków (zawarty w zmiennej `str`) jest różny od ciągu `quit`, zostanie wykonana instrukcja z wnętrza pętli wyświetlająca odczytane dane na ekranie. Jeśli natomiast z klawiatury został wprowadzony ciąg `quit`, pętla, a tym samym cały program zostaną zakończone. Przykładowy efekt działania programu został przedstawiony na rysunku 3.10.

Rysunek 3.10.

Program
odczytujący
wiersze tekstu
wprowadzane
z klawiatury



```
C:\WINDOWS\system32\cmd.exe
C:\cs>program.exe
Wprowadzaj ciągi znaków. Aby zakończyć, wpisz quit.
test
test
Przykładowy tekst
Przykładowy tekst
quit
C:\cs>
```

Należy w tym miejscu zwrócić również uwagę, że zapis, taki jak w ćwiczeniu 3.22, może w pewnych sytuacjach spowodować nieprawidłową reakcję aplikacji. Dlaczego? Otóż wywołanie `ReadLine()` w sytuacji, gdy nie ma już nic więcej do odczytania, zamiast ciągu znaków zwróci wartość `null`. Przy odczycie z konsoli ma to miejsce po wciśnięciu kombinacji `Ctrl+Z` i zatwierdzeniu klawiszem `Enter`⁴. Jest to sygnał do zakończenia odczytu danych. Zaprezentowany program reaguje tylko na ciąg `quit`, nie respektuje sygnału `Ctrl+Z`. Można to jednak zmienić po przebudowie pętli.

⁴ Odpowiada to pojawieniu się w standardowym wejściu znaku kontrolnego o kodzie 1A (26 dziesiętnie).

Ć W I C Z E N I E

3.23 Odczyt danych z rozpoznawaniem sekwencji specjalnej

Popraw kod z ćwiczenia 3.22 tak, aby reagował na kombinację *Ctrl+Z*.

```
using System;

public class Program
{
    public static void Main()
    {
        string str;
        Console.WriteLine(
            "Wprowadzaj ciągi znaków. Aby zakończyć, wpisz quit.");
        while(true)
        {
            str = Console.ReadLine();
            if(str == "quit" || str == null)
            {
                break;
            }
            Console.WriteLine(str);
        }
    }
}
```

W porównaniu z kodem z ćwiczenia 3.22 występuje tu sporo różnic. Zupełnie inna jest konstrukcja pętli. Ponieważ wyrażeniem warunkowym jest `true`, pętla nie zakończy się samoczynnie i konieczne będzie jej przerwanie. Wewnątrz pętli ciąg znaków odczytany z konsoli (z reguły wprowadzony przez użytkownika z klawiatury) jest odczytywany i zapisywany w zmiennej `str`. Następnie badane jest, czy w `str` znajduje się ciąg `quit` oraz czy `str` jest równe `null` (oznaczałoby to, że w ciągu wejściowym znalazł się znak specjalny *Ctrl+Z*). Po spełnieniu jednego z tych warunków należy zakończyć pętlę, jest więc wykonywana instrukcja `break`. Jeżeli jednak oba warunki są fałszywe, odczytane dane (zapisane w zmiennej `str`) pojawiają się na ekranie.

Skoro wiadomo już, jak wczytywać dane z klawiatury, można pokusić się o zmodyfikowanie programu do obliczania pierwiastków równania kwadratowego tak, aby parametry były wprowadzane w trakcie jego działania. W programie będziemy teraz prosić użytkownika o podanie kolejnych liczb i podstawimy je pod zmienne `parametrA`, `parametrB` i `parametrC`.

Oczywiście przed przypisaniem danych do wspomnianych zmiennych trzeba wykonać konwersję z typu `string` na typ `double` (lub też `int`, jeśli decydujemy się na obsługę wyłącznie wartości całkowitych). Co jednak należy zrobić, kiedy użytkownik nie poda prawidłowej liczby, ale np. wpisze losową kombinację liter? Najlepiej byłoby poprosić o ponowne wprowadzenie parametru. Jak to wykonać? Najwygodniej będzie skorzystać z pętli `while` lub `do...while` i prosić użytkownika o wprowadzanie liczby tak długo, aż poda poprawną.

Skoro jednak mamy trzy zmienne, a tym samym trzy parametry do wprowadzenia, to nie ma sensu pisać trzech pętli wyglądających praktycznie tak samo. Lepiej utworzyć dodatkową funkcję (metodę; ten temat został bliżej omówiony w rozdziale 4.), której zadaniem będzie dostarczenie prawidłowej liczby rzeczywistej wprowadzonej przez użytkownika.

Ć W I C Z E N I E

3.24 Równania kwadratowe o parametrach podawanych w trakcie działania aplikacji

Napisz program obliczający pierwiastki równania kwadratowego, w którym parametry są wprowadzane w trakcie jego działania.

```
using System;

class Pierwiastek
{
    public static double pobierzLiczbe(string param)
    {
        double liczba = 0;
        bool sukces;
        do
        {
            Console.Write("Proszę podać {0} parametr równania: ", param);
            try
            {
                liczba = Double.Parse(Console.ReadLine());
                sukces = true;
            }
            catch (Exception)
            {
                Console.WriteLine(
                    "To nie jest prawidłowa liczba rzeczywista!");
                sukces = false;
            }
        }
    }
}
```

```
        while(!sukces);
        return liczba;
    }
    public static void Main(string[] args)
    {
        double parametrA = pobierzLiczbe("pierwszy");
        double parametrB = pobierzLiczbe("drugi");
        double parametrC = pobierzLiczbe("trzeci");

        Console.WriteLine("Wprowadzone parametry równania:\n");
        Console.WriteLine("A = " + parametrA + ", B = " + parametrB +
            ", C = " + parametrC + "\n");

        if (parametrA == 0)
        {
            Console.WriteLine("To nie jest równanie kwadratowe: A = 0!");
        }
        else
        {
            double delta = parametrB * parametrB - 4 * parametrA * parametrC;
            double wynik;

            if (delta < 0)
            {
                Console.WriteLine("Delta < 0");
                Console.WriteLine("Brak rozwiązań w zbiorze liczb rzeczywistych.");
            }
            else if (delta == 0)
            {
                wynik = -parametrB / (2 * parametrA);
                Console.WriteLine("Rozwiązanie: x = " + wynik);
            }
            else
            {
                wynik = (-parametrB + Math.Sqrt(delta)) / (2 * parametrA);
                Console.WriteLine("Rozwiązanie: x1 = " + wynik);
                wynik = (-parametrB - Math.Sqrt(delta)) / (2 * parametrA);
                Console.WriteLine("x2 = " + wynik);
            }
        }
    }
}
```

Za pobieranie wartości liczbowych od użytkownika odpowiada osobna funkcja (metoda; dokładne wyjaśnienie tego terminu znajdzie się w rozdziale 4.) o nazwie `pobierzLiczbe`. Otrzymuje ona jeden argument typu `string` (ciąg znaków), który zostanie użyty w komunikacie wyświetlanym na ekranie. Wewnątrz funkcji zostały zadeklarowane dwie

zmienne: liczba oraz sukces. W pierwszej będzie zapisywana wartość odczytana z klawiatury (w postaci liczby typu `double`), druga będzie sygnalizowała, czy odczytany ciąg udało się przetworzyć na liczbę (`sukces = true`), czy też nie (`sukces = false`).

Odczyt wartości przeprowadzany jest w pętli `do...while`. Najpierw wyświetlany jest komunikat zawierający określenie (pobrane z argumentu `param`), który z parametrów jest aktualnie wczytywany. Następnie w bloku try wykonywana jest złożona instrukcja:

```
liczba = Double.Parse(Console.ReadLine());
```

Następuje odczytanie z konsoli wiersza tekstu, przekształcenie go na wartość typu `double` oraz przypisanie tej wartości zmiennej `liczba`. Zmiennej `sukces` jest też przypisywana wartość `true`. To oznacza, że jeśli konwersja ciągu znaków na liczbę powiedzie się, pętla zostanie zakończona (warunek kontynuacji pętli to `!sukces`, a więc gdy zmienna `sukces` ma wartość `true`, pętla jest kończona), a wartość zapisana w zmiennej `liczba` stanie się wynikiem działania funkcji (odpowiada za to instrukcja `return liczba;`).

Jeżeli jednak konwersja ciągu pobranego z klawiatury nie uda się, zostanie wykonany blok `catch`. To oznacza, że na ekranie pojawi się komunikat z informacją o błędnych danych, a zmienna `sukces` otrzyma wartość `false`. Skoro tak się stanie, pętla `do...while` będzie kontynuowana, a użytkownik zobaczy ponowną prośbę o podanie wartości liczbowej.

Zawartość metody `Main`, od której rozpoczyna się wykonywanie kodu programu, jest prawie taka sama jak we wcześniejszych przykładach dotyczących rozwiązywania równań kwadratowych — zasady obliczeń się przecież nie zmieniły. Różnice dotyczą samego początku kodu. Wartości zmiennych `parametrA`, `parametrB` i `parametrC` są po prostu pobierane z opisaney wyżej funkcji `pobierzLiczbe`. Zapis:

```
double parametrA = pobierzLiczbe("pierwszy");
```

oznacza, że ma powstać zmienna o nazwie `parametrA` oraz typie `double` i ma jej być przypisana wartość wynikająca z działania funkcji `pobierzLiczbe`, której został przekazany ciąg znaków `pierwszy`. Dokładne zasady budowy własnych funkcji (właściwie metod) oraz zwracania przez nie wartości zostały opisane w kolejnym rozdziale.



Na tym można by zakończyć ćwiczenia z wprowadzania danych i rozwiązywania równań kwadratowych, ale warto wykonać jeszcze jeden przykład. Wygodnie byłoby, gdyby program umożliwiał podawanie parametrów zarówno w wierszu poleceń, jak i w trakcie jego działania. Użytkownik mógłby wybrać wygodniejszy sposób. Co więcej, jeśli przy podawaniu danych w wierszu poleceń pomyliłby się, aplikacja pozwoliłaby na ponowne ich wprowadzenie.

Nie jest to przecież skomplikowane zadanie. Wystarczy połączyć kod z ćwiczenia 3.21 z kodem z ćwiczenia 3.24.

Ć W I C Z E N I E

3.25 Różne sposoby podawania parametrów dla aplikacji

Napisz program rozwiązujący równanie kwadratowe o zadanych parametrach będących dowolnymi liczbami rzeczywistymi. Parametry mogą być wprowadzane zarówno w trakcie działania programu, jak i w wierszu poleceń.

```
using System;

class Pierwiastek
{
    public static double pobierzLiczbe(string param)
    {
        // Tutaj treść metody pobierzLiczbę z ćwiczenia 3.24.
    }
    public static void Main(string[] args)
    {
        bool liniaKomend = true;
        double parametrA = 0, parametrB = 0, parametrC = 0;

        if(args.Length < 3)
        {
            liniaKomend = false;
        }
        if(liniaKomend)
        {
            try
            {
                parametrA = Double.Parse(args[0]);
                parametrB = Double.Parse(args[1]);
                parametrC = Double.Parse(args[2]);
            }
            catch(Exception)
            {
            }
        }
    }
}
```



```
        Console.WriteLine("Jeden z wprowadzonych parametrów nie jest
        ↳poprawną liczbą!");
        liniaKomend = false;;
    }
}

if(!liniaKomend)
{
    parametrA = pobierzLiczbe("pierwszy");
    parametrB = pobierzLiczbe("drugi");
    parametrC = pobierzLiczbe("trzeci");
}

Console.WriteLine("Wprowadzone parametry równania:\n");
Console.WriteLine("A = " + parametrA + ", B = " + parametrB +
    ", C = " + parametrC + "\n");

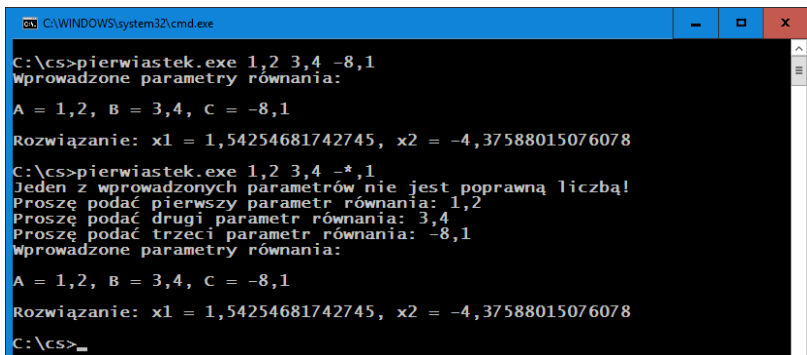
// Tutaj dalsza treść metody Main z ćwiczenia 3.24.
}
}
```

Zarówno treść metody `pobierzLiczbe`, jak i część metody `Main` wykonująca obliczenia pozostały bez zmian w porównaniu z poprzednimi przykładami, dlatego też nie zostały uwzględnione na listingu. Modyfikacjom uległ natomiast początek kodu metody `Main`. Pojawiła się nowa zmienna o nazwie `liniaKomend`, wskazująca, czy argumenty mają być pobierane z wiersza poleceń. Jej początkowa wartość wynosi `true`.

Za deklaracjami zmiennych znajduje się instrukcja warunkowa sprawdzająca liczbę argumentów otrzymanych z wiersza poleceń. Jeżeli jest ich mniej niż 3, stan zmiennej `liniaKomend` zmienia się na `false`. Tym samym wspomniane argumenty zostaną pominięte, a program poprosi o ich ręczne wprowadzenie. Jeżeli jednak są przynajmniej 3 argumenty, wartością zmiennej pozostaje `true`, a więc wykonywane są instrukcje przetwarzające dane z tablicy `args`. Gdyby przy przetwarzaniu danych zawartych w `args` wystąpił błąd, wykonany będzie blok `catch`, a tym samym stan zmiennej `liniaKomend` zmieni się na `false`.

Kolejna instrukcja warunkowa bada wartość zapisaną w `liniaKomend`. Wartość `true` oznacza, że w wierszu poleceń zostały podane co najmniej 3 argumenty i udało się je przetworzyć na wartości rzeczywiste. Skoro tak, blok `if` jest pomijany i program przystępuje do obliczeń. Z kolei wartość `false` oznacza, że argumenty nie zostały przekazane lub są błędne (nie są wartościami rzeczywistymi). Wtedy parametry równania są pobierane podobnie jak w ćwiczeniu 3.24, czyli za pomocą metody `pobierzLiczbe`.

Przykładowy efekt dwóch różnych wywołań programu został przedstawiony na rysunku 3.11. Jeżeli w wierszu poleceń podane zostały 3 parametry i każdy z nich jest prawidłową liczbą, zostaną one użyte do rozwiązania równania (pierwsze wywołanie na rysunku). Jeśli jednak parametrów jest mniej lub przy wprowadzaniu któregośkolwiek z nich został popełniony błąd (drugie wywołanie na rysunku), aplikacja poprosi o ponowne ich wprowadzenie.



```
C:\WINDOWS\system32\cmd.exe
C:\cs>pierwiastek.exe 1,2 3,4 -8,1
Wprowadzone parametry równania:
A = 1,2, B = 3,4, C = -8,1
Rozwiązanie: x1 = 1,54254681742745, x2 = -4,37588015076078
C:\cs>pierwiastek.exe 1,2 3,4 -*,1
Jeden z wprowadzonych parametrów nie jest poprawną liczbą!
Proszę podać pierwszy parametr równania: 1,2
Proszę podać drugi parametr równania: 3,4
Proszę podać trzeci parametr równania: -8,1
Wprowadzone parametry równania:
A = 1,2, B = 3,4, C = -8,1
Rozwiązanie: x1 = 1,54254681742745, x2 = -4,37588015076078
C:\cs>
```

Rysunek 3.11. Parametry równania mogą być podawane w postaci liczb rzeczywistych

Część II

Programowanie obektowe

4

Klasy i obiekty

Klasy

Każdy program w C# składa się z klas. Dotychczas używaliśmy tylko jednej klasy, która nazywała się Program lub Pierwiastek. Wróćmy zatem do pierwszego programu z ćwiczeń, wyświetlającego na ekranie napis. Kod wyglądał następująco:

```
using System;

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

Zostało wtedy przyjęte założenie, że szkielet kolejnych aplikacji służących do nauki struktur języka programowania ma właśnie tak wyglądać. Teraz nadszedł czas, aby wyjaśnić dlaczego. Wszystko stanie się jasne po przeczytaniu tego właśnie rozdziału.

W klasach zawarty jest kod wykonywalny, który realizuje przypisane mu zadania. Klasy są także opisami obiektów, a każdy obiekt jest instancją, czyli wystąpieniem danej klasy. Oznacza to, że klasa definiuje typ danego obiektu.

Dla osoby nieobeznanej z programowaniem obiektowym brzmi to zapewne całkowicie niezrozumiale, nie należy się jednak przejmować, w rzeczywistości nie ma w tym nic skomplikowanego. Skoro klasa definiuje typ obiektu, to przypomnijmy sobie, czym jest typ. Typ określa rodzaj wartości, jakie może przyjmować dany byt programistyczny, np. zmienna. Jeśli zatem typem zmiennej jest `int`, może ona przyjmować wartości typu `int`, czyli liczby całkowite z danego przedziału.

Czym są obiekty? Można je określić jako byty programistyczne, które potrafią przechowywać jakieś wartości oraz wykonywać pewne operacje. Klasy natomiast to właśnie opisy takich obiektów. Obrazowo można powiedzieć, że klasa to pewnego rodzaju plan, na podstawie którego w programie tworzone są obiekty. Aby jednak nie poprzestać na suchych definicjach, spróbujemy wykonać konkretny przykład. Załóżmy, że chcemy w programie zapisać dane dotyczące punktów na ekranie. Taka klasa, nazwiemy ją `Punkt`, powinna przechowywać dwie współrzędne: `x` i `y`.

Ć W I C Z E N I E

4.1 Prosta klasa

Napisz kod klasy, w której można będzie przechowywać dane dotyczące punktów na ekranie.

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

Składowymi klas są pola i metody. Pola to atrybuty, w których można przechowywać dane dotyczące klasy. Metody to kod wykonywalny, za pomocą którego można przeprowadzać różne operacje. W tym przypadku klasa zawiera tylko dwa pola, `x` i `y`, które opisują położenie punktu na ekranie. Słowo `public` oznacza, że zarówno do klasy, jak i do pól nie ma ograniczeń dostępu. Zostanie to dokładniej wyjaśnione w podrozdziale „Specyfikatory dostępu”.



Zadeklarujmy teraz zmienną typu Punkt. Jest to bardzo proste:

```
Punkt nowyPunkt;
```

Jak widać, najpierw podaje się nazwę klasy, a potem nazwę zmiennej, podobnie jak dla zmiennych poznanych już wcześniej typów podstawowych. Tak zadeklarowana zmienna jest jednak pusta — dokładniej mówiąc, powstała w ten sposób jedynie referencja, inaczej odniesienie (ang. *reference*), do obiektu klasy Punkt — ale można jej przypisać obiekt klasy Punkt. Aby to zrobić, należy go najpierw utworzyć. Wykorzystujemy w tym celu operator `new` w postaci:

```
new NazwaKlasy();
```

W omawianym przypadku cała konstrukcja będzie wyglądała następująco:

```
Punkt nowyPunkt = new Punkt();
```

Można też najpierw zadeklarować referencję, a dopiero potem utworzyć i przypisać jej obiekt danej klasy:

```
Punkt nowyPunkt;  
nowyPunkt = new Punkt();
```

Warto zwrócić uwagę, że np. w C++ jest inaczej. W tym języku instrukcja:

```
Punkt nowyPunkt;
```

spowodowałaby utworzenie obiektu typu Punkt, a nie tylko referencji do niego. Jeżeli ktoś jest przyzwyczajony do C++, powinien o tym pamiętać, gdyż jest to częsta przyczyna błędów.

Gdy powstanie obiekt typu Punkt, można odwoływać się do jego pól, stosując konstrukcję:

```
nazwa_obiektu.nazwa_pola
```

Jeżeli zatem zmiennej o nazwie `nowyPunkt` został przypisany obiekt typu Punkt¹ (np. za pomocą jednej z pokazanych wyżej instrukcji), to aby przypisać wartość 100 polu `x`, należy użyć instrukcji:

```
nowyPunkt.x = 100;
```

Aby z kolei odczytać wartość pola `x` i zapisać ją w zmiennej `liczba`, trzeba skorzystać z instrukcji:

```
int liczba = nowyPunkt.x;
```

¹ Ścisłej: referencja do obiektu typu Punkt.

Metody

Metody, jak już zostało wspomniane, zawierają kod operujący na polach danej klasy bądź też na dostarczonych z zewnątrz danych (ogólniej: mogą wykonywać dowolnie zaprogramowane zadanie). Metody wywołujemy za pomocą operatora `.` (znak kropki), poprzedzając go nazwą zmiennej odnośnikowej (referencyjnej). Wygląda to następująco:

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

W nawiasie okrągłym po nazwie metody podaje się argumenty, o ile są wymagane (w ten sposób można przekazać metodzie jakieś dane). W klasie `Punkt` można np. utworzyć dwie metody, które zwracałyby odpowiednio współrzędną `x` i współrzędną `y` punktu.

ĆWICZENIE

4.2 Metody zwracające wartości pól

Do klasy `Punkt` dodaj metody podające współrzędną `x` oraz współrzędną `y`.

```
public class Punkt
{
    public int x;
    public int y;
    public int GetX()
    {
        return x;
    }
    public int GetY()
    {
        return y;
    }
}
```

Wewnątrz klasy, oprócz pól `x` i `y`, zostały umieszczone metody `GetX` i `GetY`. Pierwsza zwraca wartość pola `x`, a druga — wartość pola `y`. Wartości pól są zwracane za pomocą instrukcji `return` (instrukcja `return` powoduje przerwanie wykonywania danej metody i, ewentualnie, zwrócenie przez nią wartości wymienionej po `return`). Jeśli mając taką klasę, utworzymy nowy obiekt typu `Punkt` i zapiszemy go w zmiennej `punkt`:

```
Punkt punkt = new Punkt();
```


to aby uzyskać informację o wartości współrzędnej x, będziemy mogli napisać:

```
punkt.x;
```

lub też:

```
punkt.GetX();
```

Na przykład tak:

```
int współrzędnaX = punkt.x;  
int wspX = punkt.GetX();
```

Należy zwrócić uwagę, że po wywołaniu metody (zapis `punkt.GetX()`) oznacza wywołanie metody `GetX` należącej do obiektu wskazywanego przez zmienną `punkt`) w miejsce wystąpienia tego wywołania jest poddawiana wartość zwrócona przez tę metodę za pomocą instrukcji `return`. To dlatego możliwe jest przypisanie `wspX = punkt.GetX()`.

Ć W I C Z E N I E

4.3 Ustawianie i pobieranie wartości pól

Do klasy `Punkt` dodaj metodę ustawiającą współrzędne oraz metodę zwracającą współrzędne.

```
public class Punkt  
{  
    public int x;  
    public int y;  
    public void UstawWspolrzedne(int wspX, int wspY)  
    {  
        x = wspX;  
        y = wspY;  
    }  
    public Punkt PobierzWspolrzedne()  
    {  
        Punkt punkt = new Punkt();  
        punkt.x = x;  
        punkt.y = y;  
        return punkt;  
    }  
}
```

Przedstawiona tu wersja klasy zawiera dwa pola typu `int` o nazwach `x` i `y`, które służą do przechowywania współrzędnych danego punktu, oraz dwie metody. Pierwsza zajmuje się ustawieniem pól danego obiektu, druga pobiera wartości. Przed metodą `UstawWspolrzedne()` znajduje

się słowo `void`, co oznacza, że metoda nie zwraca żadnej wartości. Ma natomiast dwa argumenty, `wspX` i `wspY`, oba typu `int`. W ciele (we wnętrzu) tej metody polu `x` przypisywana jest wartość parametru `wspX`, a polu `y` — wartość parametru `wspY`. Zatem po wykonaniu następującej instrukcji:

```
punkt.UstawWspolrzedne (1, 10);
```

pole `x` obiektu `punkt` przyjmie wartość 1, a pole `y` — wartość 10.

Przed drugą metodą — `pobierzPunkt` — znajduje się słowo `Punkt`. Oznacza to, że zwraca ona referencję do obiektu typu `Punkt`. Można zatem napisać taką oto instrukcję:

```
Punkt innyPunkt = punkt.PobierzWspolrzedne();
```

W ciele tej metody najpierw tworzony jest nowy obiekt typu `Punkt`, a następnie odpowiednim polom tego obiektu przypisywane są wartości pól `x` i `y` obiektu bieżącego. Obiekt `punkt` (a dokładniej referencja do tego obiektu) jest zwracany za pomocą instrukcji `return`.

Najwyższy czas wykorzystać tak utworzony kod w konkretnym przykładzie. Napiszemy program korzystający z klasy `Punkt`. Będzie się składał z dwóch klas: `Punkt` i `Program`. Można je zapisać albo w jednym pliku (np. o nazwie *program.cs*), albo też w dwóch. Wybierzemy to drugie rozwiązanie. Kod klasy `Punkt` z ćwiczenia 4.3 zapiszemy w pliku *Punkt.cs*, natomiast kod programu korzystającego z tej klasy (kod z ćwiczenia 4.4) — w pliku *Program.cs* (nazwy plików rozpoczynają się wielką literą, jako że będą to też nazwy jedynych klas zawartych w tych plikach; nie ma to jednak formalnego znaczenia, nazwy plików można dobrać dowolnie).

Ć W I C Z E N I E

4.4 Wykorzystanie klasy Punkt

Napisz program korzystający z obiektów klasy `Punkt`.

```
using System;

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

```
Punkt pomocniczyPunkt;  
pomocniczyPunkt = punkt.PobierzWspolrzedne();  
  
Console.WriteLine("Przed ustawieniem wartości:");  
Console.WriteLine("Współrzędna x = " + pomocniczyPunkt.x);  
Console.WriteLine("Współrzędna y = " + pomocniczyPunkt.y);  
  
punkt.UstawWspolrzedne(1, 2);  
pomocniczyPunkt = punkt.PobierzWspolrzedne();  
  
Console.WriteLine("\nPo ustawieniu wartości:");  
Console.WriteLine("Współrzędna x = " + pomocniczyPunkt.x);  
Console.WriteLine("Współrzędna y = " + pomocniczyPunkt.y);  
}  
}
```

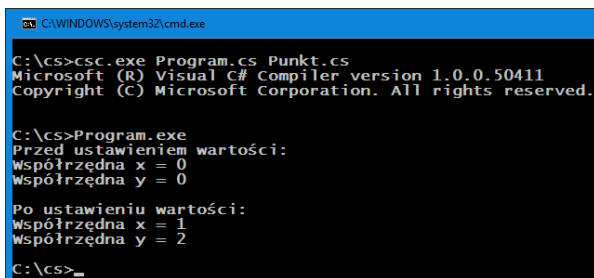
Oba pliki (*Program.cs* oraz *Punkt.cs*) umieszczamy w jednym katalogu i kompilujemy w wierszu poleceń, wydając komendę:

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

Powstanie wtedy plik wynikowy o nazwie *Program.exe*, który będzie można uruchomić. W efekcie działania programu na ekranie zobaczymy widok zaprezentowany na rysunku 4.1.

Rysunek 4.1.

Wynik działania
programu
z ćwiczenia 4.4



```
C:\WINDOWS\system32\cmd.exe  
  
C:\cs>csc.exe Program.cs Punkt.cs  
Microsoft (R) Visual C# Compiler version 1.0.0.50411  
Copyright (C) Microsoft Corporation. All rights reserved.  
  
C:\cs>Program.exe  
Przed ustawieniem wartości:  
Współrzędna x = 0  
Współrzędna y = 0  
  
Po ustawieniu wartości:  
Współrzędna x = 1  
Współrzędna y = 2  
  
C:\cs>_
```

Jak działa przedstawiony program? Na początku tworzymy zmienną o nazwie *punkt* i przypisujemy jej odniesienie (referencję) do nowo powstałego obiektu klasy *Punkt* oraz drugą zmienną — *pomocniczyPunkt*. Ponieważ metoda *PobierzWspolrzedne()* zwraca referencję do obiektu typu *Punkt*, można wykonać przypisanie:

```
pomocniczyPunkt = punkt.PobierzWspolrzedne();
```

Wartości pól *x* i *y* obiektu *pomocniczyPunkt* są wyświetlane na ekranie. Będą to dwa zera. Wynika to z tego, że pola te nie zostały zainicjalizowane (zainicjowane), a niezainicjalizowane pola typu *int* przyjmują wartość 0. Ogólniej rzecz ujmując: każde niezainicjalizowane pole

typu arytmetycznego przyjmuje wartość 0. Znaczy to, że można takiego pola użyć (to różnica w stosunku do zmiennych stosowanych w rozdziale 3.; niezainicjalizowanej zmiennej nie można było wykonać).

Następnie została użyta metoda `UstawWspolrzedne` obiektu `punkt`:

```
punkt.UstawWspolrzedne(1, 2);
```

Oznacza to, że polu `x` została przypisana wartość 1, a polu `y` — wartość 2. Te wartości zostały pobrane w postaci obiektu typu `Punkt`, przypisane zmiennej pomocniczy `Punkt` i wyświetlone na ekranie (użyta w jednej z instrukcji sekwencja `\n` oznacza przejście do nowego wiersza; por. tabela 2.3 w rozdziale 2.).

Analizując kod z ćwiczenia 4.4, można zauważyć, że zmienna pomocniczy `Punkt` wcale nie jest konieczna do prawidłowego działania programu. Z równie dobrym skutkiem można wykonać odwołanie bezpośrednio. Modyfikacji należałoby oczywiście poddać metodę `Main`. Będzie to jednak skutkowało zmniejszeniem czytelności kodu.

Ć W I C Z E N I E

4.5 Odwołania do pól z pominięciem zmiennej pomocniczej

Zmodyfikuj kod metody `Program` z ćwiczenia 4.4 tak, aby nie trzeba było używać zmiennej pomocniczy `Punkt`.

```
using System;

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

        Console.WriteLine("Przed ustawieniem wartości:");
        Console.WriteLine(
            "Współrzędna x = " + punkt.PobierzWspolrzedne().x);
        Console.WriteLine(
            "Współrzędna y = " + punkt.PobierzWspolrzedne().y);

        punkt.UstawWspolrzedne (1, 2);

        Console.WriteLine("\nPo ustawieniu wartości:");
```

```
Console.WriteLine(  
    "Współrzędna x = " + punkt.PobierzWspolrzedne().x);  
Console.WriteLine(  
    "Współrzędna y = " + punkt.PobierzWspolrzedne().y);  
}  
}
```

Ponieważ w miejsce wywołania metody `PobierzWspolrzedne` jest podstawiany wynik jej działania, czyli zwrócona referencja do obiektu klasy `Punkt`, można zastosować odwołanie typu:

```
punkt.PobierzWspolrzedne().x
```

Trzeba jednak pamiętać, że w tym przykładzie znajduje się dwukrotnie więcej wywołań metody `PobierzWspolrzedne` niż w ćwiczeniu 4.4, co nie pozostaje bez wpływu na wydajność działania kodu. Oczywiście w tak prostym przypadku nie ma to znaczenia, jednak przy bardzo dużej liczbie wywołań można się spodziewać pewnego spadku wydajności.

Dlaczego jednak wykonujemy tę na pozór karkołomną konstrukcję, tworząc najpierw w metodzie `PobierzWspolrzedne()` nowy obiekt typu `Punkt` (ćwiczenie 4.3), przypisując mu odpowiednie wartości `x` oraz `y` i dopiero potem zwracając nowy byt? Odpowiedź jest prosta. Nie można jednocześnie zwrócić współrzędnej `x` i współrzędnej `y`. Metoda może zwracać tylko jedną wartość typu podstawowego (wartościowego) lub referencyjnego (odnośnikowego).

Można co najwyżej napisać dwie dodatkowe metody, które będą osobno zwracały wartość `x`, a osobno wartość `y`, tak jak zostało to wykonane w ćwiczeniu 4.2. Podobnie można utworzyć dwie metody ustawiające osobno wartości `x` i `y`.

Może się to okazać bardzo przydatne, gdy gdzieś w programie będziemy chcieli zmodyfikować tylko jedną ze współrzędnych. Załóżmy, że chcemy zmodyfikować tylko `x`, ustawiając jego wartość na 5. W obecnej sytuacji, gdyby do dyspozycji były tylko metody z ćwiczenia 4.3 operujące na polach (przy założeniu, że nie chcemy lub nie możemy odwołać się bezpośrednio do pól), musielibyśmy zastosować konstrukcję:

```
punkt.UstawWspolrzedne (5, punkt.PobierzWspolrzedne().y);
```

Dopiszmy więc brakujące metody, niech będą to: `UstawX`, `UstawY`, `PobierzX`, `PobierzY`.

Ć W I C Z E N I E

4.6 Brakujące metody klasy Punkt

Zmodyfikuj klasę Punkt, dodając metody umożliwiające niezależne ustawianie i odczytywanie współrzędnych.

```
public class Punkt
{
    public int x;
    public int y;
    public void UstawWspolrzedne(int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
    }
    public Punkt PobierzWspolrzedne()
    {
        Punkt punkt = new Punkt();
        punkt.x = x;
        punkt.y = y;
        return punkt;
    }
    public void UstawX(int wspX)
    {
        x = wspX;
    }
    public void UstawY(int wspY)
    {
        y = wspY;
    }
    public int PobierzX()
    {
        return x;
    }
    public int PobierzY()
    {
        return y;
    }
}
```

Metody UstawWspolrzedne i PobierzWspolrzedne mają taką samą postać jak w ćwiczeniu 4.3. Metoda UstawX przyjmuje jeden argument typu int (wspX) i przypisuje jego wartość polu x. Metoda UstawY działa analogicznie, z tym że przypisuje wartość otrzymanego argumentu polu y. Metody PobierzX i PobierzY po prostu zwracają wartości odpowiednich pól za pomocą instrukcji return (podobnie jak w ćwiczeniu 4.2).

Istnieje jeszcze jedna możliwość równoczesnego uzyskania wartości x i y — można metodzie `pobierzPunkt()` przekazać argument. Wygląda to następująco:

```
public void PobierzWspolrzedne(Punkt punkt)
{
    punkt.x = x;
    punkt.y = y;
}
```

Wtedy po wykonaniu metody pola obiektu wskazywanego przez argument `punkt` przyjmą wartości z pól x i y .

Zatrzymajmy się tu na chwilę. Przecież jeżeli dopiszemy do klasy `Punkt` taką metodę, będzie ona zawierała dwie metody o takiej samej nazwie — `PobierzWspolrzedne`. Czy jest to dopuszczalne? Otóż tak, ale pod jednym warunkiem — metody te muszą się od siebie różnić argumentami wywołania. Sytuację taką nazywamy przeciążaniem metod (ang. *method overloading*) lub funkcji (ang. *function overloading*). W naszym przypadku jedna metoda nie ma żadnych argumentów wywołania, druga jako argument przyjmuje referencję do typu `Punkt`. Wszystko zatem jest zgodne z zasadami.

Ć W I C Z E N I E

4.7 Przeciążanie metod

Dołącz do klasy `Punkt` przeciążoną metodę `PobierzWspolrzedne`.

```
public class Punkt
{
    public int x;
    public int y;
    public void UstawWspolrzedne(int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
    }
    public Punkt PobierzWspolrzedne()
    {
        Punkt punkt = new Punkt();
        punkt.x = x;
        punkt.y = y;
        return punkt;
    }
    public void PobierzWspolrzedne(Punkt punkt)
    {
        punkt.x = x;
```

```
punkt.y = y;
}
// Tutaj pozostałe metody klasy Punkt z ćwiczenia 4.6
}
```

Aby się przekonać, że dwie metody o tej samej nazwie faktycznie mogą się znajdować w jednej klasie i nie spowoduje to kolizji w ich działaniu, napiszemy testową klasę Program, która będzie korzystała z najnowszej wersji klasy Punkt z ćwiczenia 4.7. Odpowiedni kod znajduje się w ćwiczeniu 4.8.

Ć W I C Z E N I E

4.8

Wykorzystanie przeciążonych metod

Napisz klasę Program korzystającą z przeciążonych metod klasy Punkt.

```
using System;

public class Program
{
    public static void Main()
    {
        Punkt punkt = new Punkt();
        Punkt pomocniczyPunkt = new Punkt();
        punkt.PobierzWspolrzedne(pomocniczyPunkt);

        Console.WriteLine("Przed ustawieniem wartości:");
        Console.WriteLine(
            "Współrzędna x = " + pomocniczyPunkt.PobierzX());
        Console.WriteLine(
            "Współrzędna y = " + pomocniczyPunkt.PobierzY());

        punkt.UstawWspolrzedne(100, 200);
        pomocniczyPunkt = punkt.PobierzWspolrzedne();

        Console.WriteLine("\nPo ustawieniu wartości:");
        Console.WriteLine(
            "Współrzędna x = " + pomocniczyPunkt.PobierzX());
        Console.WriteLine(
            "Współrzędna y = " + pomocniczyPunkt.PobierzY());
    }
}
```

Na początku powstały dwa obiekty typu Punkt, a referencje do nich zostały przypisane zmiennym punkt i pomocniczyPunkt. Instrukcja:

```
punkt.PobierzWspolrzedne(pomocniczyPunkt);
```


spowodowała wywołanie metody `PobierzWspolrzedne` obiektu wskazywanego przez zmienną `punkt` i przekazanie jej w postaci argumentu referencji do obiektu wskazywanego przez zmienną `pomocniczyPunkt`. To oznacza, że bieżące wartości pól `x` i `y` obiektu `punkt` zostaną przypisane polom `x` i `y` obiektu `pomocniczyPunkt`. Metoda `PobierzWspolrzedne` została też ponownie wywołana po zmianie współrzędnych obiektu `punkt` (za pomocą metody `UstawWspolrzedne`). W tym drugim przypadku była to przeciążona wersja metody nieprzyjmująca argumentów, za to zwracająca wynik typu `Punkt` (warto zwrócić uwagę, że tym samym pierwotna zawartość zmiennej `pomocniczyPunkt` została utracona, została jej natomiast przypisana referencja do obiektu typu `Punkt` powstałego w metodzie `pobierzPunkt`; to inna sytuacja niż przy pierwszym wywołaniu tej metody, gdy obiekt pozostał ten sam, ale zmieniły się wartości jego pól).

W obu przypadkach bieżące wartości współrzędnych zapisanych w zmiennej `pomocniczyPunkt` są wyświetlane na ekranie, dzięki czemu można się przekonać, że wszystko działa zgodnie z opisem. Jak więc widać, metoda `pobierzPunkt` przyjmująca argument działa bez problemów, mimo że w klasie `Punkt` istnieje jej wersja nieprzyjmująca argumentów i zachowująca się w nieco inny sposób.

Na zakończenie tego podrozdziału spróbujmy napisać metodę, która przyjmie argument, wykona na nim jakąś operację i zwróci wynik, oraz użyjmy jej w programie. Żeby nie komplikować przykładu, przyjmijmy, że jej działanie będzie polegało na pomnożeniu otrzymanej wartości przez dwa, program zaś będzie oczekiwał argumentu wywołania w postaci łańcucha znaków reprezentującego wartość rzeczywistą. Sprawdzimy, jak takie zadanie zrealizować w C# 5.0 i niższych wersjach, a także jak by to mogło wyglądać przy wykorzystaniu nowych konstrukcji dostępnych w C# 6.0 — **wyrażeń lambda** (ang. *lambda expression*) i **interpolacji łańcuchów znakowych** (ang. *string interpolation*).

Ć W I C Z E N I E

4.9 Metoda wykonująca operacje na argumencie

Napisz program, który przyjmie jeden argument wywołania, przetworzy go na wartość liczbową oraz wyświetli napis informujący, jaki jest wynik mnożenia tej wartości przez 2. Mnożenie powinno być wyko-

nywane przez osobną metodę, np. o nazwie `RazyDwa`. Ma ona przyjmować jeden argument i zwracać wynik operacji w postaci liczby typu `Double`.

```
using System;

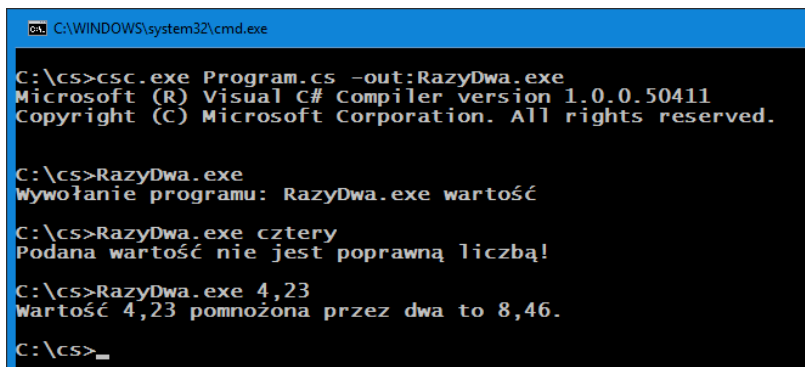
public class Program
{
    public static Double RazyDwa(Double arg)
    {
        return arg * 2;
    }
    public static void Main(string[] args)
    {
        Double liczba;
        if(args.Length < 1)
        {
            Console.WriteLine("Wywołanie programu: RazyDwa.exe wartość");
            return;
        }

        try
        {
            liczba = Double.Parse(args[0]);
        }
        catch(Exception)
        {
            Console.WriteLine("Podana wartość nie jest poprawną liczbą!");
            return;
        }

        Double wynik = RazyDwa(liczba);
        Console.WriteLine("Wartość {0} pomnożona przez dwa to {1}.",
            ↪liczba, wynik);
    }
}
```

Za wykonanie mnożenia odpowiada metoda `RazyDwa` przyjmująca jeden argument typu `Double` (`arg`) i zwracająca wynik takiego samego typu. Wewnątrz tej metody wartość argumentu jest mnożona przez liczbę dwa, a rezultat tej operacji jest zwracany za pomocą instrukcji `return`. W metodzie `Main`, od której zaczyna się wykonywanie kodu aplikacji, zadeklarowana została zmienna `liczba`, do której zostanie zapisana wartość wynikająca z przetworzenia pierwszego argumentu wywołania na typ `Double`. Odczyt danych i sprawdzenie liczby argumentów wywołania odbywa się na takiej samej zasadzie jak w przykładzie z ćwiczenia 3.19 (rozdział 3.), a konwersja ciągu znaków na liczbę przebiega tak jak w przykładzie z ćwiczenia 3.20.

Na końcu kodu znajduje się zmienna pomocnicza wynik, która będzie tymczasowo przechowywała wartość zwróconą przez metodę RazyDwa. W wywołaniu RazyDwa przekazywany jest argument o wartości zapisanej w zmiennej liczba. Ostatecznie za pomocą instrukcji Console.WriteLine na ekranie wyświetlany jest komunikat informujący o tym, jaki jest wynik mnożenia podanej wartości przez 2 (rysunek 4.2).



```
C:\WINDOWS\system32\cmd.exe

C:\cs>csc.exe Program.cs -out:RazyDwa.exe
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

C:\cs>RazyDwa.exe
Wywołanie programu: RazyDwa.exe wartość

C:\cs>RazyDwa.exe cztery
Podana wartość nie jest poprawną liczbą!

C:\cs>RazyDwa.exe 4,23
Wartość 4,23 pomnożona przez dwa to 8,46.

C:\cs>
```

Rysunek 4.2. Efekt działania programu mnożącego podane wartości przez 2

Program z ćwiczenia 4.9 w C# 6.0 można zapisać nieco prościej, a to dzięki możliwości użycia operatora lambda (wyrażenia lambda) oraz interpolacji łańcuchów znakowych. Operator lambda ma postać => i pozwala na zbudowanie wyrażenia lambda. Po lewej stronie takiego operatora występuje pewien parametr, a po prawej stronie wyrażenie lub blok instrukcji². Jeśli na przykład mamy metodę wykonującą prostą operację, taką jak metoda RazyDwa z ćwiczenia 4.9, to przypisanie treści funkcji może być wykonane następująco:

```
public static Double razyDwa(Double arg) => arg * 2;
```

To uproszczenie i skrócenie zapisu. Tak powstała metoda może być dalej używana w identyczny sposób jak metoda utworzona w klasyczny sposób.

² To bardzo uproszczony przykład. Wyrażenia lambda pozwalają na tworzenie funkcji lokalnych (anonimowych), które mogą być przekazywane jako argumenty lub też zwracane jako rezultat działania metod. Ułatwiają dzięki temu tworzenie bardziej złożonych konstrukcji programistycznych.

Kolejne ułatwienie dla programistów to tzw. interpolacja łańcuchów znakowych. W C# 6.0 i wyższych wersjach w łańcuchu znakowym można zawrzeć wyrażenia ujęte w znaki nawiasu klamrowego. Zostaną one wyliczone, a rezultat obliczeń będzie wstawiony do łańcucha. Taki ciąg znaków musi być poprzedzony znakiem \$. Ogólnie:

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

W prostym przypadku wyrażeniem może być nazwa zmiennej, wtedy w odpowiednim miejscu zostanie wstawiona wartość tej zmiennej, np.:

```
$"łańcuch {zmienna1}znakowy {zmienna2}łańcuch znakowy{zmiennaN}"
```

Praktyczne zastosowanie zostało przedstawione w ćwiczeniu 4.10.

Ć W I C Z E N I E

4.10 Wyrażenia lambda i interpolacja ciągów znaków

Zmień kod z ćwiczenia 4.9 w taki sposób, aby do definicji metody `RazyDwa` zostało użyte wyrażenie lambda, a wyświetlanie danych na ekranie odbywało się dzięki technice interpolacji łańcuchów znakowych.

```
using System;

public class Program
{
    public static Double RazyDwa(Double arg) => arg * 2;
    public static void Main(string[] args)
    {
        Double liczba;
        if(args.Length < 1)
        {
            Console.WriteLine("Wywołanie programu: RazyDwa.exe wartość");
            return;
        }

        try
        {
            liczba = Double.Parse(args[0]);
        }
        catch(Exception)
        {
            Console.WriteLine("Podana wartość nie jest poprawną liczbą!");
            return;
        }

        Double wynik = RazyDwa(liczba);
        Console.WriteLine($"Wartość {liczba} pomnożona przez dwa to {wynik}.");
    }
}
```

Konstrukcja programu pozostała tu taka sama jak w przypadku ćwiczenia 4.9. Różnice dotyczą sposobu utworzenia metody `RazyDwa` — tym razem przy użyciu operatora `=>` — oraz sposobu wpłatania wartości zmiennych `liczba` oraz `wynik` do ciągu znaków wyświetlanego na ekranie. Należy zwrócić uwagę na znak `$` znajdujący się przed ciągiem przekazywanym metodzie `WriteLine`. Jego pominięcie spowoduje literalne potraktowanie ciągu, a zatem brak zamiany sekwencji `{zmienna}` na odpowiednie wartości.

Konstruktory

Zwykle połom danego obiektu chcemy przypisać jakieś wartości początkowe. Jeżeli pola te zawierają zmienne referencyjne, trzeba wręcz utworzyć przypisane im obiekty. Czasami też przed użyciem danego obiektu należy wykonać bardziej złożony kod. Można oczywiście napisać w tym celu dodatkową zwykłą metodę i używać jej np. następująco:

```
Klasa obiekt = new Klasa();  
obiekt.InicjujZmienne();
```

Zakładając, że nowo tworzonemu obiektom znanej już dobrze klasy `Punkt` miałyby być przypisywane początkowe wartości `x = 800`, `y = 600`, metoda ta wyglądałaby tak:

```
void InicjujZmienne()  
{  
    x = 800;  
    y = 600;  
}
```

Gdybyśmy chcieli nadawać tym polom różne wartości w zależności od obiektu, zapewne skorzystalibyśmy z metody `UstawWspolrzedne` wywoływanej tuż po powołaniu obiektu do życia. Zatem wyglądałoby to tak:

```
Punkt punkt = new Punkt();  
punkt.UstawWspolrzedne(800, 600);
```

Czynności te można jednak wykonywać automatycznie. Służą do tego specjalne metody zwane **konstruktorami** (ang. *constructors*). Metody te są wykonywane *zawsze* przy tworzeniu nowego obiektu. Konstruktory

są publiczne i bezrezultatowe (tzn. nie mogą zwracać wyniku) oraz mają nazwę identyczną z nazwą klasy, której dotyczą. Mogą mieć różne argumenty, może też istnieć kilka konstruktorów dla danej klasy.

Schemat budowy klasy z konstruktorem jest następujący:

```
public class nazwa_klasy
{
    public nazwa_klasy(argumenty)
    {
        // kod konstruktora
    }
}
```

Napiszmy dwa konstruktory dla klasy Punkt. Jeden będzie bezargumentowy i będzie ustawiał wartości *x* i *y* odpowiednio na 800 i 600. Drugi ustawi współrzędne na wartości podane przez użytkownika w postaci argumentów.

Ć W I C Z E N I E

4.11 Konstruktory dla klasy Punkt

Utwórz dwa konstruktory dla klasy Punkt. Jeden powinien być bezargumentowy i przypisywać polom klasy wartości 800 i 600. Drugi ma przyjmować dwa argumenty typu *int*.

```
public class Punkt
{
    public int x;
    public int y;
    public Punkt()
    {
        x = 800;
        y = 600;
    }
    public Punkt(int wspX, int wspY)
    {
        x = wspX;
        y = wspY;
    }
    // Tutaj pozostałe metody klasy Punkt
}
```

Pierwszy konstruktor ma bardzo prostą budowę. Zgodnie z podanym wyżej schematem nie zwraca żadnej wartości (nie ma też przed nim słowa *void*!). We wnętrzu polu *x* przypisywana jest wartość 800, a polu *y* — wartość 600. Tak więc będą to początkowe wartości każdego obiektu

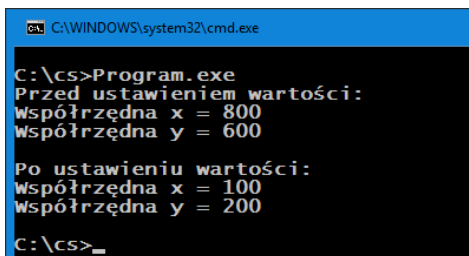
typu Punkt. Drugi konstruktor działa na takiej samej zasadzie jak metoda `UstawWspolrzedne` prezentowana we wcześniejszych ćwiczeniach. Otrzymuje argumenty `wspX` oraz `wspY` i przypisuje ich wartości polom `x` i `y`.

Pozostałe metody pozostają takie same jak w poprzednich ćwiczeniach, dlatego też nie zostały ponownie zaprezentowane.

Należy zauważyć, że jeśli przetestujemy tak przygotowaną klasę Punkt, korzystając z klasy Program powstałej w ćwiczeniu 4.8, to dwoma pierwszymi wynikami będą liczby 800 i 600 (rysunek 4.3), a nie dwa zera, jak miało to miejsce, kiedy użyliśmy klasy Punkt z ćwiczenia 4.7. Widać więc wyraźnie, że konstruktor faktycznie został wykonany.

Rysunek 4.3.

*Wykorzystanie
klasy Program
z ćwiczenia 4.8
w połączeniu
z klasą Punkt
z ćwiczenia 4.11*



```
C:\WINDOWS\system32\cmd.exe
C:\cs>Program.exe
Przed ustawieniem wartości:
Współrzędna x = 800
Współrzędna y = 600
Po ustawieniu wartości:
Współrzędna x = 100
Współrzędna y = 200
C:\cs>_
```

Warto też zwrócić uwagę, że oba zaprezentowane konstruktory w rzeczywistości dublują kod przeciążonych metod `UstawWspolrzedne`. Skoro tak, to dobrym rozwiązaniem jest po prostu wywoływanie odpowiednich wersji tych metod w konstruktorze. Nie jest bowiem dobrym pomysłem powtarzanie instrukcji wykonujących to samo zadanie w różnych miejscach programu — może to w łatwy sposób doprowadzić do niespójności kodu.

Ć W I C Z E N I E

4.12 Wywoływanie metod w konstruktorach

Zmodyfikuj kod konstruktorów z ćwiczenia 4.11 w taki sposób, aby korzystały z metody `UstawWspolrzedne`.

```
public class Punkt
{
    public int x;
    public int y;
    public Punkt()
    {
```

```
        UstawWspolrzedne(800, 600);
    }
    public Punkt(int x, int y)
    {
        UstawWspolrzedne(x, y);
    }
    // Tutaj pozostale metody klasy Punkt
}
```

Na zakończenie tego podrozdziału wykonajmy jeszcze jeden przykład, w którym zostaną wywołane oba gotowe konstruktory klasy Punkt, te z ćwiczenia 4.11 lub 4.12. Efekt ich działania będzie taki sam (czyli ustawienie pól x i y na odpowiednie wartości), mimo że w każdym z tych ćwiczeń osiągnięty został w nieco inny sposób.

ĆWICZENIE

4.13 Wykorzystanie konstruktorów

Napisz program korzystający z obu konstruktorów klasy Punkt powstałych w ćwiczeniu 4.11 lub 4.12.

```
using System;

public class Program
{
    public static void Main()
    {
        Punkt punkt1 = new Punkt();
        Punkt punkt2 = new Punkt(100, 100);

        Console.WriteLine(
            "Punkt1: Współrzędna x = " + punkt1.PobierzX());
        Console.WriteLine(
            "Punkt1: Współrzędna y = " + punkt1.PobierzY());
        Console.WriteLine(
            "Punkt2: Współrzędna x = " + punkt2.PobierzX());
        Console.WriteLine(
            "Punkt2: Współrzędna y = " + punkt2.PobierzY());
    }
}
```

Powstały dwa obiekty: punkt1 i punkt2. W pierwszym przypadku został użyty konstruktor bezargumentowy, a więc wartością pola x będzie 800, a pola y — 600. W drugim przypadku użyto konstruktora dwuargumentowego, a więc polom zostaną przypisane wartości prze

kazane w postaci argumentów (czyli x będzie równe 100 i y również będzie równe 100). Zatem po uruchomieniu programu zobaczymy widok przedstawiony na rysunku 4.4.

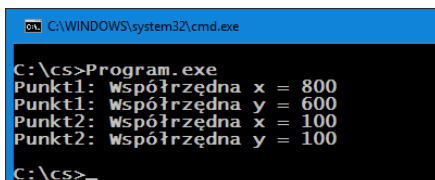
Rysunek 4.4.

Efekt użycia

konstruktorów

klasy Punkt

w ćwiczeniu 4.13



```
C:\WINDOWS\system32\cmd.exe
C:\cs>Program.exe
Punkt1: Współrzędna x = 800
Punkt1: Współrzędna y = 600
Punkt2: Współrzędna x = 100
Punkt2: Współrzędna y = 100
C:\cs>
```

Specyfikatory dostępu

W dotychczasowych ćwiczeniach przed słowem `class` pojawiało się zwykle słowo `public`. Jest to tzw. **specyfikator dostępu** (lub **modyfikator dostępu**, ang. *access modifier*), który oznacza, że dana klasa jest publiczna, czyli dostęp do niej nie jest ograniczony. Jeżeli modyfikator `public` nie pojawi się przed słowem `class`, taka klasa będzie wewnętrzna, czyli będzie się zachowywała tak, jakby znajdował się przed nią modyfikator `internal` (patrz niżej).

Specyfikatory dostępu pojawiają się jednak nie tylko przy klasach, ale także przy składowych klas (polach, metodach i innych). Każde pole oraz metoda (dotyczy to także innych rodzajów składowych, które nie były omawiane w książce) mogą być:

- ❑ publiczne (`public`),
- ❑ chronione (`protected`),
- ❑ wewnętrzne (`internal`),
- ❑ wewnętrzne chronione (`protected internal`),
- ❑ prywatne (`private`).

Publiczne składowe klasy określa się słowem `public`, co oznacza, że wszyscy mają do nich dostęp i są one dziedziczone przez klasy pochodne (dziedziczenie zostanie opisane w kolejnym podrozdziale). Do składowych prywatnych (`private`) można dostać się tylko z wnętrza danej klasy, natomiast do składowych chronionych (`protected`) można się dostać z wnętrza danej klasy oraz klas potomnych (pochodnych).

Znaczenie tych specyfikatorów dostępu jest praktycznie takie samo jak w innych językach obiektowych, np. w Javie.

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. Tymi dwoma specyfikatorami nie będziemy się bliżej zajmować, jako że nie będą przydatne przy dalszych ćwiczeniach. Spotkamy się natomiast ze specyfikatorami `public`, `private` i `protected`.

Zobaczmy, jak to wygląda w praktyce. Wróćmy do przykładowej klasy `Punkt`. Pola oznaczające współrzędne `x` i `y` dotychczas były określone jako publiczne. Oznacza to, że można się do nich odwoływać bezpośrednio. Co się jednak stanie, jeśli dostęp zmieni się na prywatny?

Ć W I C Z E N I E

4.14 Składowe prywatne

Zmodyfikuj kod klasy `Punkt` z ćwiczenia 4.2 tak, aby składowe `x` i `y` były zadeklarowane jako prywatne.

```
public class Punkt
{
    private int x;
    private int y;
    public int GetX()
    {
        return x;
    }
    public int GetY()
    {
        return y;
    }
}
```

Ć W I C Z E N I E

4.15 Próba odwołania do składowych prywatnych

Napisz klasę `Program`, w której nastąpi bezpośrednie odwołanie do składowych klasy `Punkt` z ćwiczenia 4.14. Spróbuj skompilować otrzymany kod.

```
using System;

public class Program
{
    public static void Main()
    {
        Punkt punkt = new Punkt();
        punkt.x = 10;
        punkt.y = 20;
        Console.WriteLine ("Współrzędna x = " + punkt.x);
        Console.WriteLine ("Współrzędna y = " + punkt.y);
    }
}
```

Kod klasy wygląda standardowo. Utworzony został obiekt typu `Punkt`, a następnie polom `x` i `y` zostały przypisane wartości oraz nastąpiło ich wyświetlenie na ekranie. Próba kompilacji tego programu (w połączeniu z klasą `Punkt` z ćwiczenia 4.14) spowoduje jednak wyłącznie wyświetlenie czterech komunikatów o błędach (widocznych na rysunku 4.5) — po jednym na każde z odwołań do pól klasy — bo skoro pola te są prywatne, to spoza klasy nie można się do nich odwoływać ani przy zapisie, ani przy odczycie.



Wyświetlane komunikaty o błędach (ang. *error*) mogą się nieco różnić w zależności od tego, która z wersji kompilatora C# została użyta. W nowych wersjach kompilatora mogą się również pojawić (jak na rysunku 4.5) ostrzeżenia (ang. *warning*) o niezainicjalizowanych polach `x` i `y`.

```
C:\WINDOWS\system32\cmd.exe
C:\cs>csc.exe Program.cs Punkt.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

Program.cs(8,11): error CS0122: 'Punkt.x' is inaccessible due to its protection level
Program.cs(9,11): error CS0122: 'Punkt.y' is inaccessible due to its protection level
Program.cs(10,51): error CS0122: 'Punkt.x' is inaccessible due to its protection level
Program.cs(11,51): error CS0122: 'Punkt.y' is inaccessible due to its protection level
Punkt.cs(3,15): warning CS0649: Field 'Punkt.x' is never assigned to, and will always have its default value 0
Punkt.cs(4,15): warning CS0649: Field 'Punkt.y' is never assigned to, and will always have its default value 0
C:\cs>
```

Rysunek 4.5. Próba bezpośredniego odwołania do prywatnych składowych klasy kończy się niepowodzeniem

Pamiętamy jednak, że w celu ustawiania oraz pobierania wartości pól w wykonywanych wcześniej ćwiczeniach zdefiniowaliśmy odpowiednie metody. Metody te zostały zadeklarowane jako publiczne, zatem mamy do nich dostęp z innych klas. Przy tym metody te (niezależnie od tego, czy są publiczne, prywatne, czy chronione) mają dostęp do wszystkich innych metod oraz pól klasy Punkt, ponieważ stanowią składowe tej klasy. To znaczy, że są możliwe i modyfikacja, i odczyt pól prywatnych, o ile tylko w klasie znajdują się odpowiednie metody publiczne.

Ć W I C Z E N I E

4.16 Uzyskanie dostępu do składowych prywatnych

Zmodyfikuj kod klasy Program z ćwiczenia 4.15 oraz klasy Punkt z ćwiczenia 4.14 w taki sposób, aby był możliwy dostęp do składowych *x* i *y*. Nie zmieniaj jednak poziomu dostępu do tych składowych (pozostaw modyfikator *private*).

```
public class Punkt
{
    private int x;
    private int y;
    public int GetX()
    {
        return x;
    }
    public int GetY()
    {
        return y;
    }
    public void SetX(int wspX)
    {
        x = wspX;
    }
    public void SetY(int wspY)
    {
        y = wspY;
    }
}

public class Program
{
    public static void Main()
    {
        Punkt punkt = new Punkt();
        punkt.SetX(10);
        punkt.SetY(20);
        Console.WriteLine ("Współrzędna x = " + punkt.GetX());
    }
}
```

```
        Console.WriteLine ("Współrzędna y = " + punkt.GetY());  
    }  
}
```

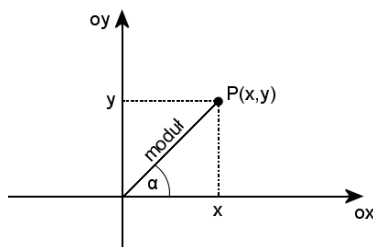
W klasie Punkt oprócz metod `GetX` i `GetY` (tzw. *getter*y) pozwalających na pobieranie wartości pól pojawiły się metody `SetX` i `SetY` (tzw. *setter*y) umożliwiające ustawianie wartości. Wszystkie te metody mają dostęp publiczny (`public`), a zatem nie będzie żadnych ograniczeń w dostępie do nich i będą mogły być wywoływane (uruchamiane) we wszystkich innych klasach, a więc także w klasie Program. Klasa Program wykonuje takie same zadania jak wersja z ćwiczenia 4.15 (utworzenie obiektu, ustawienie wartości pól, odczyt pól), jednak dzięki skorzystaniu z wyżej opisanych metod tym razem działa bezproblemowo.

Dlaczego nie korzystamy wyłącznie ze składowych publicznych? Otóż dlatego, aby nie było bezpośredniego dostępu do wnętrza danej klasy. Pozwala to ukryć wewnętrzną organizację obiektu, a na „zewnątrz” udostępnić jedynie interfejs umożliwiający wykonywanie operacji ściśle określonych przez programistę. Przydaje się to zwykle w bardziej skomplikowanych projektach, jednak nawet na przykładzie tak prostej klasy jak Punkt można pokazać, dlaczego takie rozwiązanie może być potrzebne.

Załóżmy, że mamy napisany program, ale z pewnych powodów zmieniliśmy reprezentację współrzędnych i teraz punkt identyfikujemy za pomocą kąta alfa oraz odległości punktu od początku układu współrzędnych (a więc za pomocą współrzędnych w tzw. układzie biegunowym; rysunek 4.6). Zatem w klasie Punkt nie ma już pól `x` i `y`, nie mają więc sensu odwołania do nich. Jeśli w takiej sytuacji dostęp do składowych `x` i `y` byłby publiczny, to nie dość, że we wszystkich innych klasach trzeba byłoby zmienić odwołania, to jeszcze przy każdym odwołaniu należałoby wykonywać niezbędne przeliczenia. Spowodowałoby to naprawdę duże komplikacje i konieczność wykonania sporej pracy nad adaptacją kodu.

Rysunek 4.6.

*Położenie punktu
reprezentowane
za pomocą
współrzędnych
biegunowych*



Jeżeli jednak pola x i y będą prywatne, trzeba będzie tylko przeddefiniować metody klasy Punkt. Cała reszta programu nawet nie „zauważy”, że coś się zmieniło! Przekonajmy się o tym!

Ć W I C Z E N I E

4.17 Zmiana sposobu reprezentacji współrzędnych

Zmień definicję klasy Punkt z ćwiczenia 4.12 (lub 4.11) w taki sposób, aby położenie punktu było reprezentowane w układzie biegunowym.

```
using System;

public class Punkt
{
    private double modul;
    private double sinalfa;
    public Punkt()
    {
        UstawWspolrzedne(800, 600);
    }
    public Punkt(int x, int y)
    {
        UstawWspolrzedne(x, y);
    }
    public void UstawWspolrzedne(int wspX, int wspY)
    {
        modul = Math.Sqrt(wspX * wspX + wspY * wspY);
        sinalfa = wspY / modul;
    }
    public Punkt PobierzWspolrzedne()
    {
        Punkt punkt = new Punkt();
        punkt.sinalfa = sinalfa;
        punkt.modul = modul;
        return punkt;
    }
    public void PobierzWspolrzedne(Punkt punkt)
    {
        punkt.sinalfa = sinalfa;
        punkt.modul = modul;
    }
    public void UstawX(int wspX)
    {
        UstawWspolrzedne (wspX, PobierzY());
    }
    public void UstawY(int wspY)
    {
        UstawWspolrzedne (PobierzX(), wspY);
    }
}
```

```
public int PobierzX()
{
    double x = modul * Math.Sqrt(1 - sinalfa * sinalfa);
    return (int) x;
}
public int PobierzY()
{
    double y = modul * sinalfa;
    return (int) y;
}
}
```

Przeliczenie współrzędnych kartezjańskich (tzn. w postaci x, y) na układ biegunowy (czyli kąt i modul) nie jest skomplikowane³. Najwygodniejsza jest tu funkcja sinus, dlatego też została użyta w rozwiązaniu ćwiczenia. Zatem sinus kąta alfa ($\sin(\alpha)$ dla oznaczeń jak na rysunku 4.6)⁴, reprezentowany przez pole o nazwie `sinalfa`, jest równy $\frac{y}{\text{modul}}$. Natomiast sam modul, reprezentowany przez pole o nazwie

`modul`, to $\sqrt{x^2 + y^2}$.

Przeliczenie współrzędnych biegunowych na kartezjańskie jest nieco trudniejsze. Co prawda y to po prostu:

$$\text{modul} \times \sin(\alpha),$$

za to x wynika ze wzoru:

$$\text{modul} \times \sqrt{1 - \sin^2(\alpha)}.$$

Zapis (*nazwaTypu*) *zmienna*, np. `(int) x`, oznacza konwersję zmiennej do podanego typu — w omawianym przykładzie konwersję wartości

³ W celu uniknięcia umieszczania w programie dodatkowych instrukcji warunkowych zaciemniających sedno zagadnienia przedstawiony kod i wzory są poprawne dla dodatnich współrzędnych x . Zaprezentowane rozwiązanie nie będzie także działało poprawnie dla punktu o współrzędnych $(0,0)$ — niezbędne byłoby wprowadzenie dodatkowych instrukcji warunkowych. Odpowiednie uzupełnienie klasy `Punkt` w taki sposób, aby usunąć te mankamenty, można potraktować jako ćwiczenie do samodzielnego wykonania.

⁴ Chodzi o kąt pomiędzy prostą przechodzącą przez reprezentowany punkt i środek układu współrzędnych a osią OX .

typu `double` (będącej wynikiem obliczeń) na typ `int` (niezbędny, aby zwrócić wartości `x` i `y`, które w pierwotnej klasie miały właśnie taki typ)⁵.

Warto też zwrócić uwagę, że w metodach `UstawX` i `UstawY` — jako że ustawiają wartość tylko jednego z argumentów — konieczne było uzyskanie bieżącej wartości drugiego argumentu. Dlatego też zostały w nich użyte metody `PobierzX` i `PobierzY`.

ĆWICZENIE

4.18 Testowanie nowej klasy

Wykorzystaj klasę `Program` z ćwiczenia 4.13 do przetestowania klasy `Punkt` z ćwiczenia 4.17.

Po kompilacji obu klas i uruchomieniu powstałego w ten sposób programu okaże się, że wynik działania jest identyczny z wynikiem z ćwiczenia 4.13, mimo że całkowicie zmieniliśmy reprezentację klasy. Co więcej, nie trzeba było dokonywać żadnych modyfikacji metody `Main` w klasie `Program`!

Dziedziczenie

Znamy już dosyć dobrze klasę `Punkt`, założmy jednak, że chcielibyśmy mieć klasę opisującą nie tylko współrzędne punktu, ale również jego kolor. Przyjmijmy, że kolor będzie reprezentowany przez wartość liczbową (typ `int`). Chcielibyśmy mieć również możliwość jednoczesnego korzystania z obu klas. Można oczywiście utworzyć nową klasę typu `KolorowyPunkt`, która mogłaby wyglądać np. tak:

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

⁵ 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 drobne nieścisłości. Żeby jednak nie zaciemniać przedstawianego zagadnienia dodatkowymi instrukcjami, trzeba się z tą drobną niedogodnością pogodzić.

Trzeba by jednak dopisać do niej wszystkie zdefiniowane wcześniej metody klasy Punkt oraz zapewne dodatkowo `UstawKolor()` i `PobierzKolor()`. W ten sposób dwukrotnie piszemy ten sam kod. Przecież klasy Punkt i KolorowyPunkt robią w dużej części to samo. Dokładniej mówiąc, klasa KolorowyPunkt jest swoistego rodzaju rozszerzeniem klasy Punkt, a zatem niech klasa KolorowyPunkt przejmie własności klasy Punkt, a dodatkowo dodajmy do niej pole określające kolor. Jest to dziedziczenie znane m.in. z Javy i C++. Wtedy klasa Punkt będzie **klasą bazową** (inaczej nadrzędną; ang. *base class*, *parent class*, *superclass*), a klasa KolorowyPunkt — **klasą potomną** (inaczej pochodną, podrzędną; ang. *derived class*, *child class*, *subclass*). Zobaczmy, jak wygląda to w C#. Schemat definicji jest następujący:

```
public class klasa_potomna : klasa_bazowa
{
    // wewnątrz klasy
}
```

ĆWICZENIE

4.19 Proste dziedziczenie

Utwórz klasę KolorowyPunkt rozszerzającą klasę Punkt (np. wersję z ćwiczenia 4.12) o możliwość przechowywania informacji o kolorze (możesz przyjąć przechowywanie tej informacji w postaci kodu koloru).

```
public class KolorowyPunkt : Punkt
{
    private int kolor;
    public KolorowyPunkt()
    {
        UstawKolor(100);
    }
    public KolorowyPunkt(int wspX, int wspY, int nowyKolor)
    {
        UstawX(wspX);
        UstawY(wspY);
        UstawKolor(nowyKolor);
    }
    public void UstawKolor (int nowyKolor)
    {
        kolor = nowyKolor;
    }
    public int PobierzKolor()
    {
        return kolor;
    }
}
```

Klasa `KolorowyPunkt` jest rozszerzeniem klasy `Punkt` (świadczy o tym umieszczenie nazwy `Punkt` po dwukropku za nazwą `KolorowyPunkt`). Znaczy to, że zawiera pola i metody klasy `Punkt` oraz dodatkowo pola i metody zdefiniowane w `KolorowyPunkt`. Dopisane zostały metody operujące na nowej składowej o nazwie `kolor`. Ponieważ jest ona prywatna, tylko dzięki nim możliwe będzie pobieranie tej wartości (`PobierzKolor`) oraz jej ustawianie (`UstawKolor`). Metody te działają analogicznie do używanych we wcześniejszych ćwiczeniach metod `PobierzX` i `UstawX`.

W klasie dostępne są również dwa konstruktory: bezargumentowy i trójargumentowy. Pierwszy ustawia wartość pola `kolor` na 100 (to wartość przykładowa), natomiast drugi ustawia wartości wszystkich pól na zgodne z przekazanymi argumentami. Używane są przy tym metody `UstawX` i `UstawY` (odziedziczone po klasie `Punkt`) oraz `UstawKolor` (z klasy `KolorowyPunkt`).

Ć W I C Z E N I E

4.20 Użycie klasy potomnej

Napisz klasę `Program` umożliwiającą przetestowanie działania klasy `KolorowyPunkt`.

```
using System;

public class Program
{
    public static void Main()
    {
        KolorowyPunkt punkt = new KolorowyPunkt(100, 200, 10);
        Console.WriteLine("współrzędna x = " + punkt.PobierzX());
        Console.WriteLine("współrzędna y = " + punkt.PobierzY());
        Console.WriteLine("kolor = " + punkt.PobierzKolor());
    }
}
```

Najpierw przy użyciu trójargumentowego konstruktora został utworzony nowy obiekt klasy `KolorowyPunkt` (o przykładowych wartościach składowych `x=100`, `y=200` i `kolor=10`), a referencja do niego została przypisana zmiennej `punkt`. Następnie wartości pól obiektu zostały wyświetlone na ekranie (do uzyskania wartości zostały użyte metody `PobierzX`, `PobierzY` oraz `PobierzKolor`).

Oczywiście z klasy `KolorowyPunkt` możemy wyprowadzić kolejną klasę, o nazwie `DwuKolorowyPunkt`, np. dla punktów, które przyjmują dwa różne kolory, w zależności od znaku wartości współrzędnej x . Po klasie `DwuKolorowyPunkt` może dziedziczyć kolejna klasa itd.

Należy w tym miejscu zwrócić uwagę na trójargumentowy konstruktor klasy `KolorowyPunkt`. Oprócz określenia koloru przyjmuje on parametry dotyczące współrzędnej x oraz współrzędnej y i przypisuje je odpowiednim polom. Spełnia swoje zadanie, jednak wygodniej byłoby po prostu wywołać konstruktor klasy bazowej (czyli dwuargumentowy konstruktor z klasy `Punkt`). W C# stosuje się w tym celu następującą konstrukcję:

```
KonstruktorKlasyPotomnej(arg1, arg2) : base(arg1);
```

gdzie `arg1` to argumenty (może być ich wiele) konstruktora klasy bazowej, a `arg2` to argumenty konstruktora klasy potomnej. W konkretnym przypadku klasy `KolorowyPunkt` wywołanie to powinno wyglądać następująco:

```
public KolorowyPunkt(int wspX, int wspY, int nowyKolor) : base(wspX, wspY)
```

Ć W I C Z E N I E

4.21 Wywołanie konstruktora klasy bazowej

Utwórz klasę `KolorowyPunkt` rozszerzającą klasę `Punkt` o możliwość przechowywania informacji o kolorze. W konstruktorze klasy potomnej wywołaj konstruktor klasy bazowej.

```
public class KolorowyPunkt : Punkt
{
    private int kolor;
    public KolorowyPunkt()
    {
        UstawKolor(100);
    }
    public KolorowyPunkt(int wspX, int wspY, int nowyKolor) : base
        (wspX, wspY)
    {
        UstawKolor(nowyKolor);
    }
    public void UstawKolor (int nowyKolor)
    {
        kolor = nowyKolor;
    }
    public int PobierzKolor()
    {

```

```
        return kolor;
    }
}
```

Klasa ma prawie taką samą postać jak w ćwiczeniu 4.19. Różnice dotyczą konstruktora trójargumentowego. Zostały z niego usunięte wywołania metod `UstawX` i `UstawY`, a zamiast nich, korzystając ze składni ze słowem `base`, zastosowano wywołanie konstruktora klasy bazowej.

Słowo kluczowe `this`

Żałujemy, że mamy dwuargumentowy konstruktor klasy `Punkt` taki jak w ćwiczeniu 4.11, czyli przyjmujący dwa argumenty, którymi są liczby typu `int`. Nazwami tych parametrów były `wspX` i `wspY`. Co by się stało, gdyby nazwy tych parametrów brzmiały `x` i `y`, czyli gdyby deklaracja tego konstruktora wyglądała tak jak poniżej?

```
Punkt(int x, int y) {
    // treść konstruktora
}
```

Powstałby problem, jako że nazwy parametrów byłyby tożsame z nazwami pól. W takich sytuacjach rozwiązaniem jest słowo kluczowe `this` będące odwołaniem do obiektu bieżącego (wskazaniem na obiekt bieżący). Można je traktować jako referencję do aktualnego obiektu. Jeśli chcemy zaznaczyć, że chodzi o składową klasy (pole, metodę), korzystamy z odwołania w postaci:

```
this.nazwa_składowej
```

Ć W I C Z E N I E


4.22 Nazwy parametrów tożsame z nazwami pól

Napisz taką wersję klasy `Punkt` z ćwiczenia 4.11, aby parametry konstruktora nazywały się `x` i `y`, parametr metody `UstawX` — `x`, a parametr metody `UstawY` — `y`.

```
public class Punkt
{
    public int x;
    public int y;
    public Punkt()
```

```
{
    UstawWspolrzedne(800, 600);
}
public Punkt (int x, int y)
{
    UstawWspolrzedne(x, y);
}
public void UstawWspolrzedne(int x, int y)
{
    this.x = x;
    this.y = y;
}
public void UstawX(int x)
{
    this.x = x;
}
public void UstawY(int y)
{
    this.y = y;
}
// tutaj pozostale metody klasy Punkt
}
```

W stosunku do ćwiczenia 4.11 zmianie uległy — co oczywiste — konstruktor dwuargumentowy oraz metody `PobierzX` i `PobierzY`. Instrukcję `this.x = x` należy rozumieć jako: przypisz polu `x` wartość przekazaną jako argument o nazwie `x`, a instrukcję `this.y = y` odpowiednio jako: przypisz polu `y` wartość przekazaną jako argument o nazwie `y`.

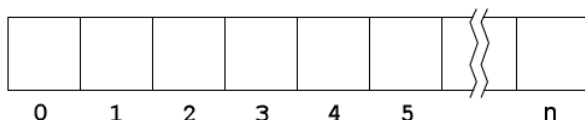


5

Tablice

Tablice to jedne z podstawowych struktur danych; znane są zapewne nawet początkującym programistom. Warto jednak w kilku słowach przypomnieć podstawowe wiadomości i pojęcia z nimi związane. Tablica to stosunkowo prosta struktura danych pozwalająca na przechowywanie uporządkowanego zbioru elementów danego typu — można ją sobie wyobrazić tak, jak zaprezentowano na rysunku 5.1. Składa się z ponumerowanych kolejno komórek, a każda taka komórka może przechowywać pewną porcję danych.

Rysunek 5.1.
*Schemat
struktury tablicy*



Jakiego rodzaju będą to dane, określa typ tablicy. Jeśli zatem zadeklarujemy tablicę typu całkowitoliczbowego (`int`), będzie mogła ona zawierać liczby całkowite, a jeśli będzie to typ znakowy (`char`), poszczególne komórki będą mogły zawierać różne znaki. Należy zwrócić uwagę, że w C# (podobnie jak w większości współczesnych popularnych języków programowania) numerowanie komórek zaczyna się od 0, czyli pierwsza komórka ma indeks 0, druga — indeks 1 itd.

Deklarowanie tablic

Przed skorzystaniem z tablicy należy zadeklarować zmienną tablicową. Ponieważ w C# tablice są obiektami, należy również utworzyć odpowiedni obiekt. Schematycznie robi się to w sposób następujący:

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

Deklarację zmiennej tablicowej oraz przypisanie jej nowo utworzonego elementu można przy tym wykonać w osobnych instrukcjach, np. w ten sposób:

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

Pisząc zatem:

```
int tablica[];
```

zadeklarujemy odniesienie do tablicy, która będzie mogła zawierać elementy typu `int`, czyli 32-bitowe liczby całkowite. Samej tablicy jednak jeszcze nie będzie (odwrotnie niż w przypadku prostych typów wartościowych, takich jak `int`, `byte` czy `char`) i konieczne jest jej utworzenie.

Ć W I C Z E N I E

5.1 Utworzenie tablicy

Zadeklaruj i zainicjalizuj tablicę elementów typu całkowitego. Przypisz pierwszemu elementowi tablicy dowolną wartość. Wyświetl zawartość tego elementu na ekranie.

```
using System;  
  
public class Program  
{  
    public static void Main()  
    {  
        int[] tablica = new int[5];  
        tablica[0] = 10;  
        Console.WriteLine("Pierwszy element tablicy: " + tablica[0]);  
    }  
}
```

Wyrażenie `new tablica[5]` oznacza utworzenie nowej, jednowymiarowej, 5-elementowej tablicy liczb typu `int`. Ta nowa tablica została przy-

pisana zmiennej odnośnikowej o nazwie `tablica`. Od miejsca tego przypisania można odwoływać się do kolejnych elementów tej tablicy, pisząc:

```
tablica[index]
```

W tym przypadku pierwszemu elementowi (o indeksie 0) została przypisana wartość 10. O tym, że takie przypisanie faktycznie miało miejsce, przekonaliśmy się, wyświetlając wartość tej komórki na ekranie.

Warto w tym miejscu ponownie przypomnieć, że elementy tablicy numerowane są od 0, a nie od 1. Oznacza to, że pierwszy element tablicy 10-elementowej ma indeks 0, a ostatni 9 (nie 10!). Co się stanie, jeśli nieprzyzwyczajeni do takiego sposobu indeksowania odwołamy się do indeksu o numerze 10?

Ć W I C Z E N I E

5.2 Odwołanie do nieistniejącego elementu tablicy

Zadeklaruj i zainicjalizuj tablicę 10-elementową. Spróbuj przypisać elementowi o indeksie 10 dowolną liczbę całkowitą.

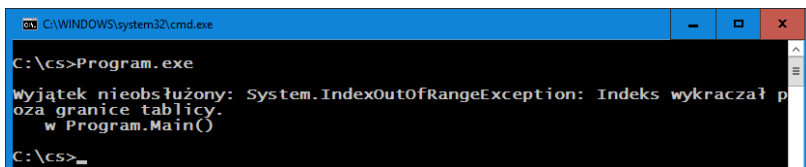
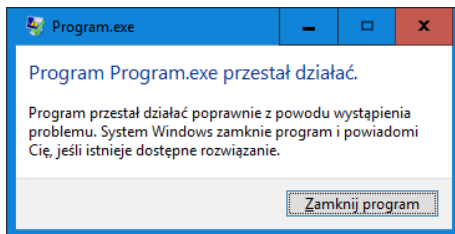
```
using System;

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

Powyższy kod da się bez problemu skompilować, jednak przy próbie uruchomienia takiego programu na ekranie zobaczymy okno z informacją o wystąpieniu błędu. Może ono mieć różną postać, w zależności od tego, w jakiej wersji systemu została uruchomiona aplikacja. Na rysunku 5.2 jest widoczne okno z systemu Windows 8. Również na konsoli (w Windows XP dopiero po zamknięciu okna dialogowego) ujrzemy komunikat podający konkretne informacje o typie błędu oraz miejscu programu, w którym wystąpił (rysunek 5.3).

Rysunek 5.2.

Próba odwołania się do nieistniejącego elementu tablicy powoduje błąd aplikacji

**Rysunek 5.3.** Systemowa informacja o błędzie

Wbrew pozorom nie stało się nic strasznego. Program co prawda nie działa, ale błąd został wychwycony przez środowisko uruchomieniowe. Konkretnie mówiąc, został wygenerowany tzw. wyjątek i aplikacja zakończyła działanie. Taki wyjątek można jednak przechwycić i tym samym zapobiec niekontrolowanemu zakończeniu wykonywania kodu. To jednak odrębny temat, który zostanie przedstawiony w rozdziale 6. Ważne jest to, że próba odwołania się do nieistniejącego elementu została wykryta i to odwołanie nie wystąpiło! Program nie naruszył więc obszaru pamięci niezarezerwowanej dla niego.

Inicjalizacja tablic

Tablicę można zainicjalizować już w momencie jej tworzenia. Dane, które mają się znaleźć w poszczególnych komórkach, podaje się w nawiasie klamrowym po deklaracji tablicy. Schematycznie wygląda to następująco:

```
typ[] nazwa = new typ [liczba_elementów]{dana1, dana2,...,danaN}
```

Jeśli zatem chcielibyśmy utworzyć 5-elementową tablicę liczb całkowitych i od razu zainicjalizować ją liczbami od 1 do 5, możemy zrobić to w taki sposób:

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

Ć W I C Z E N I E

5.3 Inicjalizacja tablicy

Zadeklaruj tablicę 5-elementową typu `int` i zainicjalizuj ją liczbami od 1 do 5. Zawartość tablicy wyświetl na ekranie.

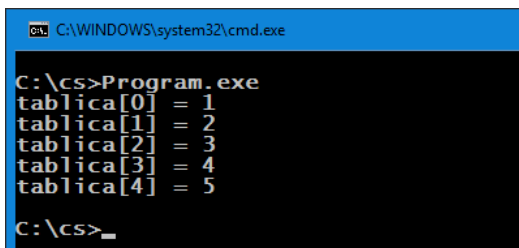
```
using System;

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

Wynik działania kodu z powyższego ćwiczenia widoczny jest na rysunku 5.4. Nie jest niespodzianką, że wyświetlone zostały liczby od 1 do 5, natomiast indeksy kolejnych komórek zmieniają się od 0 do 4. Powstała tu bowiem 5-elementowa tablica liczb typu `int`. Skoro ma 5 elementów, to pierwszy z nich ma indeks 0, a ostatni — 4. Dlatego zmienna sterująca pętlą `for`, która odczytuje dane z tablicy, ma początkową wartość 0, a warunek zakończenia pętli to `i < 5`. Tym samym `i` zmienia się też od 0 do 4.

Rysunek 5.4.

*Zawartość
kolejnych
komórek tablicy
utworzonej
w ćwiczeniu 5.3*



Kiedy inicjalizowana jest tablica o z góry znanej liczbie elementów, dopuszcza się pominięcie fragmentu kodu związanego z tworzeniem obiektu. Kompilator sam wykona odpowiednie uzupełnienia. Zamiast pisać:

```
typ[] nazwa = new typ [liczba_elementów]{dana1, dana2,...,danaN}
```

można zatem równie dobrze użyć konstrukcji:

```
typ[] nazwa = {dana1, dana2,...,danaN}
```

Oba sposoby są równoważne i należy używać tego, który jest wygodniejszy.

Ć W I C Z E N I E

5.4 Bezpośrednia inicjalizacja tablicy

Zadeklaruj tablicę 5-elementową typu `int` i zainicjalizuj ją liczbami od 6 do 2. Użyj drugiego z poznanych sposobów inicjalizacji. Zawartość tablicy wyświetl na ekranie.

```
using System;

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

Rozmiar tablicy

Każda tablica posiada właściwość `Length`, która określa bieżącą liczbę komórek. Aby uzyskać tę informację, piszemy:

```
tablica.Length
```

Przy tym dopuszczalny jest tylko odczyt, czyli prawidłowa jest konstrukcja:

```
int rozmiar = tablica.Length;
```

ale nieprawidłowy jest zapis:

```
tablica.Length = 10;
```

Ć W I C Z E N I E

5.5 Odczyt rozmiaru tablicy

Utwórz tablicę o dowolnym rozmiarze. Odczytaj wartość właściwości `Length` i wyświetl ją na ekranie.

```
using System;

public class Program
{
    public static void Main()
    {
        int[] tablica =
        {
            10, 9, 8, 7, 6, 5, 4, 3, 2, 1
        };
        Console.Write("Liczba elementów tablicy: ");
        Console.WriteLine(tablica.Length);
    }
}
```

Ć W I C Z E N I E

5.6 Właściwość `Length` i pętla `for`

Utwórz tablicę zawierającą pewną liczbę wartości całkowitych. Zawartość tablicy wyświetl na ekranie za pomocą pętli `for`. Do określenia rozmiaru tablicy użyj właściwości `Length`.

```
using System;

public class Program
{
    public static void Main()
    {
        int[] tab =
        {
            10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
            1, 2, 3, 4, 5, 6, 7, 8, 9, 10
        };
        for(int i = 0; i < tab.Length; i++)
        {
            Console.WriteLine("tab[" + i + "] = " + tab[i]);
        }
    }
}
```

Zasada odczytu danych w tym przykładzie jest taka sama jak w ćwiczeniach 5.3 i 5.4, z tą różnicą, że rozmiar tablicy jest określany za

pomocą właściwości `Length` (`tab.Length`). Dzięki temu można np. dopisać dowolną liczbę nowych danych w instrukcji inicjalizującej tablicę, a kod pętli `for` nie będzie wymagał żadnych zmian. Nowy rozmiar zostanie uwzględniony automatycznie.

Do zapisywania danych (podobnie jak do odczytu) w tablicach często używa się pętli (przedstawionych w rozdziale 2.). Jest to wręcz niezbędne, gdyż trudno się spodziewać, że można byłoby „ręcznie” zapisać wartości z więcej niż kilkunastu czy kilkudziesięciu komórek. Wielkość tablicy nie musi też być z góry znana, może wynikać z danych uzyskanych w trakcie działania programu. Z tablicami mogą współpracować dowolne rodzaje pętli. W niektórych przypadkach bardzo wygodna jest omówiona w kolejnym podrozdziale pętla `foreach`.

ĆWICZENIE

5.7 Użycie pętli do zapisu danych w tablicy

Użyj pętli `for` do zapisania w 10-elementowej tablicy 10 kolejnych liczb całkowitych.

```
using System;

public class Program
{
    public static void Main()
    {
        int[] tab = new int[10];
        for(int i = 0; i < tab.Length; i++)
        {
            tab[i] = i + 1;
        }
        Console.WriteLine("Zawartość tablicy:");
        for(int i = 0; i < tab.Length; i++)
        {
            Console.WriteLine("tab[{0}] = {1}", i, tab[i]);
            // w C# 6.0 można również tak
            // Console.WriteLine($"{tab[{i}]} = {tab[i]}");
        }
    }
}
```

Powstała 10-elementowa tablica liczb typu `int`. Mamy w niej zapisać wartości od 1 do 10, czyli komórka o indeksie 0 ma mieć wartość 1, o indeksie 1 — wartość 2 itd. A zatem wartość komórki ma być zawsze

o 1 większa niż wartość indeksu (zmiennej *i*). Dlatego instrukcja wewnątrz pętli ma postać:

```
tab[i] = i + 1;
```

Druga pętla `for` służy tylko do wyświetlania danych zawartych w tablicy. Jej konstrukcja jest taka sama jak w pierwszym przypadku. Wewnątrz pętli znajduje się instrukcja wyświetlająca wartości kolejnych komórek.

Pętla `foreach`

Dotychczas poznaliśmy trzy rodzaje pętli: `for`, `while` i `do...while` (była o nich mowa w rozdziale 3.). W przypadku tablic (jak również kolekcji, które w tej książce nie były omawiane¹) można również skorzystać z pętli typu `foreach`. Jest ona bardzo wygodna, gdyż umożliwia prostą iterację po wszystkich elementach tablicy; nie trzeba wtedy wprowadzać dodatkowej zmiennej iteracyjnej. Pętla `foreach` ma następującą postać:

```
foreach(typ identyfikator in wyrażenie)  
{  
    // instrukcje  
}
```

Jeżeli zatem mamy tablicę o nazwie `tab` zawierającą liczby typu `int`, możemy zastosować konstrukcję:

```
foreach(int val in tab)  
{  
    // instrukcje  
}
```

Wtedy w kolejnych przebiegach pętli pod `val` będą podstawiane kolejne elementy tablicy. Słowo `val` jest tu identyfikatorem odczytywanej wartości (można je traktować jak zmienną). Oczywiście nic nie stoi na przeszkodzie, aby zmienić je na dowolne inne.

¹ Ścisłej rzecz ujmując, pętli `foreach` można użyć z każdym obiektem udostępniającym tzw. iterator. Ten temat nie będzie jednak poruszany w książce.

Ć W I C Z E N I E

5.8 Użycie pętli foreach do wyświetlenia zawartości tablicy

Wykorzystaj pętlę foreach do wyświetlenia wszystkich elementów tablicy przechowującej liczby całkowite.

```
using System;

public class Program
{
    public static void Main()
    {
        int[] tab = new int[10];
        for(int i = 0; i < 10; i++)
        {
            tab[i] = i;
        }
        foreach(int i in tab)
        {
            Console.WriteLine(i);
        }
    }
}
```

Tablica `tab` została zainicjalizowana w pętli `for` kolejnymi liczbami od 0 do 9. Do wyświetlenia danych została natomiast użyta pętla `foreach`. W każdym jej przebiegu pod identyfikator `i` jest podstawiana wartość kolejnego elementu tablicy. W pierwszym przebiegu jest to pierwszy element (o indeksie 0), w drugim — drugi element (o indeksie 1) itd. Pętla kończy się po osiągnięciu ostatniego elementu (o indeksie 9).

Ć W I C Z E N I E

5.9 Zliczanie wartości w pętli foreach

Wypełnij tablicę losowymi liczbami całkowitymi typu `int`. Wykorzystaj pętlę `foreach` do sprawdzenia, ile jest w tej tablicy liczb parzystych, a ile nieparzystych.

```
using System;

public class Program
{
    public static void Main()
    {
        int[] tab = new int[100];
        int parzyste = 0, nieparzyste = 0;
        Random rand = new Random();
```



```
for(int i = 0; i < 100; i++)
{
    tab[i] = rand.Next();
}

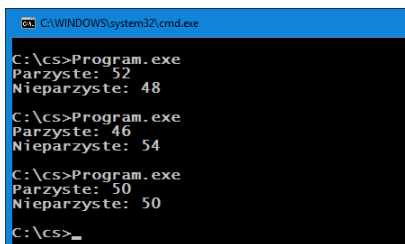
foreach(int i in tab)
{
    if(i % 2 == 0)
    {
        parzyste++;
    }
    else
    {
        nieparzyste++;
    }
}

Console.WriteLine("Parzyste: {0}", parzyste);
Console.WriteLine("Nieparzyste: {0}", nieparzyste);
}
```

Powstała tablica `tab` typu `int`, 100-elementowa. Do wypełnienia jej danymi zostały użyte pętle `for` oraz obiekt `rand` typu `Random`, za pomocą którego uzyskujemy wartości pseudolosowe. Dokładniej rzecz ujmując, kolejną pseudolosową liczbę całkowitą otrzymujemy, wywołując metodę `Next` tego obiektu. W pętli `foreach` badamy, które z komórek tablicy `tab` zawierają wartości parzyste, a które — nieparzyste. Aby to stwierdzić, używamy operatora dzielenia modulo `%` (reszty z dzielenia; por. tabela 2.4 z rozdziału 2. i ćwiczenia 3.10 – 3.11 z rozdziału 3.). Gdy wynikiem tego dzielenia jest 0, dana komórka zawiera liczbę parzystą (jest wtedy zwiększana wartość pomocniczej zmiennej `parzyste`), natomiast gdy wynik dzielenia jest różny od 0, komórka zawiera wartość nieparzystą (jest wtedy zwiększana wartość pomocniczej zmiennej `nieparzyste`). Po zakończeniu pętli na ekranie wyświetlany jest komunikat z poszukiwaną informacją, co pokazano na rysunku 5.5 (w komunikatach używane są wartości pobrane ze zmiennych `parzyste` i `nieparzyste`).

Rysunek 5.5.

Wynik kilku uruchomień programu zliczającego liczbę wartości parzystych i nieparzystych



```
C:\WINDOWS\system32\cmd.exe
C:\cs>Program.exe
Parzyste: 52
Nieparzyste: 48

C:\cs>Program.exe
Parzyste: 46
Nieparzyste: 54

C:\cs>Program.exe
Parzyste: 50
Nieparzyste: 50

C:\cs>
```

Tablice wielowymiarowe

Tablice nie muszą być jednowymiarowe, jak było w dotychczas prezentowanych przykładach. Tych wymiarów może być więcej, np. dwa — otrzymujemy wtedy strukturę widoczną na rysunku 5.6, czyli rodzaj tabeli o zadanej liczbie wierszy i kolumn. W tym przypadku są dwa wiersze oraz cztery kolumny. Łatwo zauważyć, że aby w takiej sytuacji jednoznacznie wyznaczyć komórkę, trzeba podać dwie liczby: indeks wiersza i indeks kolumny.

Rysunek 5.6.

Przykład tablicy dwuwymiarowej

	0	1	2	3	4
0					
1					

W jaki sposób można zadeklarować tego typu tablicę? Zaczniemy od deklaracji samej zmiennej tablicowej. Dla tablicy dwuwymiarowej ma ona postać:

```
typ_tablicy[,] nazwa_tablicy;
```

Samą tablicę tworzy się za pomocą instrukcji:

```
new int[wiersze, kolumny];
```

Przykładową dwuwymiarową tablicę widoczną na rysunku 5.6 utworzymy następująco (przy założeniu, że ma przechowywać liczby całkowite):

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

Inicjalizacja komórek może odbywać się — podobnie jak było w przypadku tablic jednowymiarowych — już w trakcie deklaracji:

```
typ_tablicy[,] nazwa_tablicy =  
{  
    (dana1, dana2),  
    (dana3, dana4),  
    ...,  
    (danaM, danaN)  
};
```

Zobaczmy na konkretnym przykładzie, jak będzie to wyglądało.

Ć W I C Z E N I E

5.10 Tworzenie tablicy dwuwymiarowej

Zadeklaruj tablicę dwuwymiarową typu `int` o dwóch wierszach i pięciu kolumnach i zainicjalizuj ją kolejnymi liczbami całkowitymi. Zawartość tablicy wyświetl na ekranie.

```
using System;

public class Program
{
    public static void Main()
    {
        int[,] tablica = new int[2, 5];
        int licznik = 0;
        for(int i = 0; i < 2; i++)
        {
            for(int j = 0; j < 5; j++)
            {
                tablica[i, j] = licznik++;
            }
        }
        for(int i = 0; i < 2; i++)
        {
            for(int j = 0; j < 5; j++)
            {
                Console.WriteLine(
                    "tablica[{0}, {1}] = {2}", i, j, tablica[i, j]);
            }
        }
    }
}
```

Jak widać, do wypełniania tablicy użyto dwóch zagnieżdżonych pętli `for`. Pierwsza, zewnętrzna, odpowiada za iterację po indeksach wierszy tablicy, druga — za iterację po indeksach kolumn. Zmienna `licznik` służy za licznik i jest w każdym przebiegu zwiększana o jeden, dzięki czemu w kolejnych komórkach uzyskujemy kolejne liczby całkowite od 0 do 9. Po wypełnieniu danymi tablica przyjmie postać widoczną na rysunku 5.7.

Do wyświetlenia danych używana jest analogiczna konstrukcja z dwiema zagnieżdżonymi pętlami. Po uruchomieniu kodu na ekranie zobaczymy widok przedstawiony na rysunku 5.8. Jak widać, dane są zgodne ze strukturą przedstawioną na rysunku 5.7.

Rysunek 5.7.

Tablica
z ćwiczenia 5.10
po wypełnieniu
danymi

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

Rysunek 5.8.

Wynik działania
programu
z ćwiczenia 5.10

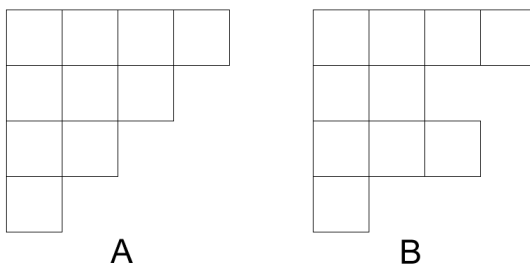
```
C:\WINDOWS\system32\cmd.exe

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

Tablica dwuwymiarowa nie musi mieć (tak jak w poprzednich przykładach) kształtu prostokątnego, tzn. takiego, że liczba komórek w każdym wierszu i każdej kolumnie jest stała. Równie dobrze można utworzyć np. tablicę o kształcie trójkąta (rysunek 5.9 A) lub zupełnie nieregularną (rysunek 5.9 B). Przy tworzeniu struktur nieregularnych trzeba się jednak więcej napracować, gdyż każdy wiersz zazwyczaj należy tworzyć ręcznie, pisząc odpowiednią linię kodu.

Rysunek 5.9.

Przykłady bardziej
skomplikowanych
tablic
dwuwymiarowych



Postarajmy się utworzyć strukturę przedstawioną na rysunku 5.9 B. Należy zauważyć, że każdy wiersz można traktować jak oddzielną tablicę jednowymiarową. Zatem jest to jednowymiarowa tablica, której poszczególne komórki zawierają inne jednowymiarowe tablice. Inaczej mówiąc, jest to tablica tablic. Wystarczy więc zadeklarować zmienną

tablicową o odpowiednim typie, a następnie poszczególnym jej elementom przypisać nowo utworzone tablice jednowymiarowe o zadanej długości. Oto całe rozwiązanie problemu.

Jednak co znaczy określenie „odpowiedni typ tablicy”? Pomyślmy — jeśli w tablicy (jednowymiarowej) miały być przechowywane liczby całkowite typu `int`, typem tej tablicy był `int`. Pisaliśmy wtedy:

```
int[] tablica;
```

Jeśli zatem typem nie jest `int`, ale tablica typu `int`, którą oznacza się jako `int[]`, należy napisać:

```
int[] [] tablica;
```

Z kolei utworzenie 4-elementowej tablicy zawierającej tablice z liczbami całkowitymi wymaga zapisu:

```
new tablica[4] [];
```

Te wiadomości powinny wystarczyć do wykonania kolejnego ćwiczenia.

Ć W I C Z E N I E

5.11 Budowa tablicy nieregularnej

Napisz kod tworzący tablicę o strukturze pokazanej na rysunku 5.9 B przechowującą liczby całkowite. W kolejnych komórkach powinny znaleźć się kolejne liczby całkowite, zaczynając od 1.

```
public class Program
{
    public static void Main()
    {
        int[] [] tablica = new int[4] [];
        tablica[0] = new int[4]{1, 2, 3, 4};
        tablica[1] = new int[2]{5, 6};
        tablica[2] = new int[3]{7, 8, 9};
        tablica[3] = new int[1]{10};
    }
}
```

Po wypełnieniu danymi tablica z ćwiczenia będzie miała postać przedstawioną na rysunku 5.10. Jak sobie poradzić z wyświetleniem jej zawartości na ekranie? Oczywiście można zrobić to ręcznie, pisząc kod oddzielnie dla każdego wiersza. Przy tak małej tablicy nie będzie to problemem. Czy jednak tej czynności nie da się zautomatyzować?

Rysunek 5.10.

Tablica
z ćwiczenia 5.11
wypełniona
przykładowymi
danymi

1	2	3	4
5	6		
7	8	9	
10			

Najwygodniej byłoby przecież wyprowadzać dane na ekran w zagnieżdżonych pętlach, tak jak w ćwiczeniu 5.10.

Jest to jak najbardziej możliwe, a z nieregularnością tablicy można sobie poradzić w bardzo prosty sposób. Przecież każda tablica ma (omówioną wcześniej w tym rozdziale) właściwość `Length`, przy użyciu której da się sprawdzić jej długość. To całkowicie rozwiązuje problem wyświetlenia danych nawet z tak nieregularnej struktury jak obecnie opisywana.

Ć W I C Z E N I E

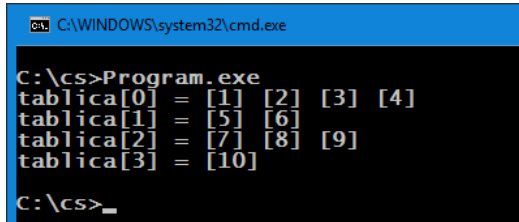
5.12 Wyświetlanie danych z tablicy nieregularnej

Zmodyfikuj kod z ćwiczenia 5.11 w taki sposób, aby dane zawarte w tablicy zostały wyświetlone na ekranie (rysunek 5.11). W tym celu użyj zagnieżdżonych pętli `for`.

```
using System;

public class Program
{
    public static void Main()
    {
        int[] [] tablica = new int[4] [];
        tablica[0] = new int[4] {1, 2, 3, 4};
        tablica[1] = new int[2] {5, 6};
        tablica[2] = new int[3] {7, 8, 9};
        tablica[3] = new int[1] {10};
        for(int i = 0; i < tablica.Length; i++)
        {
            Console.Write("tablica[{0}] = ", i);
            for(int j = 0; j < tablica[i].Length; j++)
            {
                Console.Write("[{0}] ", tablica[i][j]);
            }
        }
    }
}
```

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

Rysunek 5.11.*Wyświetlenie**danych**z nieregularnej**tablicy**w ćwiczeniu 5.12*

```
C:\WINDOWS\system32\cmd.exe  
  
C:\cs>Program.exe  
tablica[0] = [1] [2] [3] [4]  
tablica[1] = [5] [6]  
tablica[2] = [7] [8] [9]  
tablica[3] = [10]  
  
C:\cs>
```

Do wyświetlenia danych również zostały użyte dwie zagnieżdżone pętle for. W pętli zewnętrznej jest umieszczona instrukcja `Console.WriteLine("tablica[{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.



6

Wyjątki i obsługa błędów

Obsługa błędów

W każdym większym programie występują jakieś błędy. Oczywiście staramy się ich wystrzegać, nigdy jednak nie uda się ich całkowicie wyeliminować. Co więcej, aby program wychwytywał przynajmniej część błędów, których przyczyną jest np. wprowadzenie złych wartości przez użytkownika, trzeba napisać wiele wierszy dodatkowego kodu, który zaciemnia główny kod programu. Jeżeli na przykład zadeklarowaliśmy tablicę 5-elementową, należałoby sprawdzić, czy nie odwołujemy się do nieistniejącego elementu.

Ć W I C Z E N I E

6.1 Błędne odwołanie do tablicy

W klasie Main zadeklaruj tablicę 5-elementową. Spróbuj odczytać wartość nieistniejącego 20. elementu tej tablicy.

```
using System;

public class Program
{
    public static void Main()
    {
        int[] tab = new int[5];
        int value = tab[19];
        Console.WriteLine("Element nr 20 ma wartość: " + value);
    }
}
```

Oczywiście próba wykonania powyższego kodu spowoduje błąd, ponieważ nie ma w tablicy tab elementu o indeksie 19 (czyli dwudziestego; elementy są numerowane od 0). Zobaczymy wyświetlone przez środowisko uruchomieniowe (CLR) okno z informacją, że aplikacja wygenerowała błąd, a na konsoli pojawi się komunikat podający dokładny typ błędu. Tego typu reakcję spotkaliśmy już w rozdziale 5. (rysunki 5.2 i 5.3).

W powyższym przykładzie popełniony błąd jest ewidentnie widoczny. Skoro tablica miała 5 elementów, na pewno nie można odwołać się do elementu o indeksie 19. Gdyby jednak deklaracja tablicy znajdowała się w jednej klasie, a odwołanie do niej w drugiej klasie, nie byłoby to tak oczywiste.

Ć W I C Z E N I E

6.2 Odwołania do tablicy pomiędzy klasami

Napisz takie dwie klasy, aby w jednej znajdowała się tablica 5-elementowa, a w drugiej następowało odwołanie do tej tablicy.

```
using System;

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

```
        Tablica tab = new Tablica();
        int value = tab.getElement(20);
        Console.WriteLine("Element nr 20 ma wartość: " + value);
    }
}

public class Tablica
{
    int[] tab;
    public Tablica()
    {
        tab = new int[5];
    }
    public int getElement(int index)
    {
        return tab[index];
    }
}
```

W klasie `Tablica` znajduje się pole typu tablicowego, któremu w konstruktorze przypisywana jest nowa tablica (przechowująca wartości całkowite typu `int`) o wielkości 5 elementów. Do pobierania danych z tej tablicy służy metoda `getElement` przyjmująca jako argument indeks poszukiwanego elementu. W kodzie klasy `Program` następuje utworzenie nowego obiektu typu `Tablica` oraz wywołanie metody `getElement` tego obiektu (poprzez zmienną referencyjną `tab`). W kodzie klasy `Program` nie widzimy jednak, jaki rozmiar ma tablica znajdująca się w obiekcie wskazywanym przez `tab` (klasy `Program` i `Tablica` mogą się przecież znajdować w różnych plikach). Bardzo łatwo można więc przekroczyć maksymalny indeks i spowodować błąd. Tak też dzieje się w tym przypadku. Efekt działania programu będzie więc bardzo podobny do tego z ćwiczenia 6.1.

Błędów z ćwiczenia 6.2 można uniknąć, sprawdzając, czy wartość podawana jako argument metody `getElement` nie przekracza dopuszczalnego zakresu, czyli przedziału 0 – 4. Takie sprawdzenie można wykonać, stosując znaną już instrukcję `if` (rozdział 3., ćwiczenia 3.1 – 3.3).

Ć W I C Z E N I E

6.3 Sprawdzenie zakresu indeksów tablicy

Popraw kod z ćwiczenia 6.2 tak, aby po przekroczeniu dopuszczalnego indeksu tablicy nie występował błąd w programie. Wprowadź do metody getElement instrukcję sprawdzającą, czy nie następuje przekroczenie zakresu indeksów tablicy.

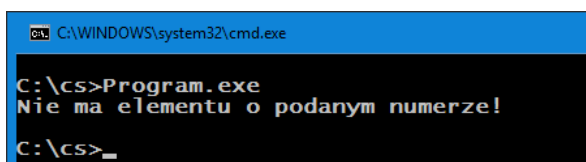
```
using System;

public class Program
{
    public static void Main()
    {
        Tablica tab = new Tablica();
        int value = tab.getElement(20);
        if (value == -1)
        {
            Console.WriteLine("Nie ma elementu o podanym numerze!");
        }
        else
        {
            Console.WriteLine("Element nr 20 ma wartość: " + value);
        }
    }
}

public class Tablica
{
    int[] tab;
    public Tablica()
    {
        tab = new int[5];
    }
    public int getElement(int index)
    {
        if ((index >= 0) && (index < 5))
        {
            return tab[index];
        }
        else
        {
            return -1;
        }
    }
}
```

Tak przygotowany kod reaguje poprawnie na próbę odwołania się do nieistniejącego elementu tablicy (rysunek 6.1). W metodzie `getElement` klasy `Tablica` następuje sprawdzenie, czy indeks przekazany jako argument jest właściwy, czyli czy mieści się w zakresie 0 – 4. W tym celu jest używana instrukcja warunkowa `if`. Gdy zakres jest prawidłowy, zwracana jest wartość zapisana w tablicy pod tym indeksem, gdy natomiast jest nieprawidłowy, zwracana jest wartość -1. Można by oczywiście zakończyć wykonywanie programu już w metodzie `getElement`, zwykle jednak trzeba poinformować funkcję wywołującą o wystąpieniu błędu.

Rysunek 6.1.
*Odwołanie do
nieistniejącego
elementu nie
powoduje już
błędu aplikacji*



W metodzie `Main` klasy `Program` badana jest wartość zwrócona przez wywołanie `tab.getElement(20)`. Jeśli jest nią -1, oznacza to wystąpienie błędu. Wtedy wyświetlany jest komunikat. Gdy otrzymana wartość jest różna od -1, jest to wartość pobranego elementu. I ta wartość jest wyświetlana.

Sposób przedstawiony w ćwiczeniu 6.3 sprawdzi się w praktyce, ma jednak wadę, która w pewnych sytuacjach może być dyskwalifikująca. Otóż żaden z elementów tablicy nie może mieć wartości równej -1. To ograniczenie funkcjonalności programu. Można jednak poradzić sobie z tym problemem, dodając do klasy `Tablica` pole typu `boolean` (np. o nazwie `isError`) sygnalizujące wystąpienie błędu.

Ć W I C Z E N I E

6.4 Sygnalizacja wystąpienia błędu

Zmodyfikuj kod z ćwiczenia 6.3 tak, aby o wystąpieniu błędu informowało dodatkowe pole umieszczone w klasie `Tablica`.

```
using System;

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

```
{
    Tablica tab = new Tablica();
    int value = tab.getElement(20);
    if (tab.isError)
    {
        Console.WriteLine("Nie ma elementu o podanym numerze!");
    }
    else
    {
        Console.WriteLine("Element nr 20 ma wartość: " + value);
    }
}

}

public class Tablica
{
    int[] tab;
    public bool isError;
    public Tablica()
    {
        tab = new int[5];
        isError = false;
    }
    public int getElement(int index)
    {
        if ((index >= 0) && (index < 5))
        {
            isError = false;
            return tab[index];
        }
        else
        {
            isError = true;
            return -1;
        }
    }
}
```

Tym razem informacja o tym, czy wystąpił błąd, jest zawarta w polu `isError`, któremu w konstruktorze przypisywana jest początkowa wartość `false`. W metodzie `getElement` w standardowy sposób bada się zakres wartości argumentu `index`. Jeżeli zawiera się on w prawidłowym przedziale, pole `isError` otrzymuje wartość `false` i zwracana jest wartość elementu pobranego z tablicy. W przeciwnym razie pole `isError` otrzymuje wartość `true`, co sygnalizuje wystąpienie błędu, i zwracana jest wartość `-1`. Co prawda jest ona w takim przypadku bezużyteczna, ale instrukcja `return` nie może się obyć bez podania wartości typu `int`, gdyż taką wartość musi zwrócić metoda `getElement`.

W klasie Program, korzystającej z obiektów typu Tablica, po wywołaniu metody getElement badany jest stan pola isError obiektu tab. Gdy jest równy true (if(tab.isError)), wypisywany jest komunikat o błędzie, a gdy jest równy false, wyświetlana jest wartość pobranego elementu.

Jak pokazano w dotychczasowych ćwiczeniach, z błędami można radzić sobie na różne sposoby. Nie da się jednak nie zauważyć, że tego typu rozwiązania powodują dodawanie coraz to większej liczby nowych zmiennych i warunków. Warto więc skorzystać z jakiejś innej metody obsługi błędów. Z pomocą przychodzą tutaj tzw. **wyjątki** (ang. *exceptions*). Wyjątek (ang. *exception*) jest to byt programistyczny, który powstaje w chwili wystąpienia błędu. Możemy go jednak przechwycić i wykonać nasz własny kod obsługi błędu. Jeżeli tego nie zrobimy, wyjątek zostanie obsłużony przez system, a na konsoli (o ile program został uruchomiony na konsoli) pojawi się komunikat z informacją, gdzie i jakiego typu błąd wystąpił. Przykładowo w ćwiczeniu 6.1 występował wyjątek o nazwie IndexOutOfRangeException (podobnie jak w ćwiczeniu 5.2 w rozdziale 5., co widać było wyrażenie na rysunku 5.3). Oznaczał on, że został przekroczony dopuszczalny zakres indeksu w tablicy.

Blok try...catch

Do obsługi wyjątków służy blok try...catch, którego schemat wykorzystania wygląda następująco:

```
try
{
    // blok instrukcji mogący spowodować wyjątek
}
catch (TypWyjtku1 identyfikatorWyjtku1)
{
    // obsługa wyjątku 1
}
catch (TypWyjtku2 identyfikatorWyjtku2)
{
    // obsługa wyjątku 2
}
catch (TypWyjtkuN identyfikatorWyjtkuN)
{
    // obsługa wyjątku N
}
```

Po try następuje blok instrukcji mogących spowodować wyjątek. Jeżeli podczas ich wykonywania wyjątek zostanie wygenerowany, wykonywanie bieżącego kodu zostanie przerwane, a sterowanie przekazane do bloku instrukcji catch. Tu z kolei nastąpi sprawdzenie, czy któryś z bloków catch odpowiada wygenerowanemu wyjątkowi. Jeśli tak, wykonany zostanie kod danego bloku catch. Dopuszczalne jest przy tym pominięcie identyfikatorów wyjątków, o ile nie będą używane w blokach catch.

Ć W I C Z E N I E

6.5 Wykorzystanie bloku try...catch

Zastosuj instrukcje try...catch do przechwycenia wyjątku generowanego przez system w ćwiczeniu 6.2. Wyświetl własny komunikat o błędzie.

```
using System;

public class Program
{
    public static void Main()
    {
        Tablica tab = new Tablica();
        try
        {
            int value = tab.getElement(20);
            Console.WriteLine("Element nr 20 ma wartość: " + value);
        }
        catch(IndexOutOfRangeException)
        {
            Console.WriteLine("Nie ma elementu o numerze 20!");
        }
    }
}

public class Tablica
{
    int[] tab;
    public Tablica()
    {
        tab = new int[5];
    }
    public int getElement(int index)
    {
        return tab[index];
    }
}
```


Klasa `Tablica` pozostała w wersji z ćwiczenia 6.2, a zatem nie zawiera żadnych instrukcji weryfikujących poprawność danych. W klasie `Program` został natomiast użyty blok `try...catch`, w którym znalazły się instrukcje. Jedna z nich pobiera wartość elementu tablicy o numerze 20, a druga wyświetla pobraną wartość na ekranie. Ponieważ tablica zawarta wewnątrz obiektu `tab` ma tylko 5 elementów i nie istnieje element o numerze 20, powstanie wyjątek. Wyjątek będzie przechwycony w bloku `catch`. Działanie zatem przebiegnie tak: wykonywanie instrukcji:

```
int value = tab.getElement(20);
```

zostanie przerwane z powodu wystąpienia błędu i sterowanie będzie przekazane do bloku `catch`. Tym samym instrukcja:

```
Console.WriteLine("Element nr 20 ma wartość: " + value);
```

zostanie pominięta, a zamiast niej będzie wykonana znajdująca się w bloku `catch` instrukcja:

```
Console.WriteLine("Nie ma elementu o numerze 20!");
```

Na ekranie pojawi się zatem napis `Nie ma elementu o numerze 20`. Widać wyraźnie, że wyjątek faktycznie został przechwycony i prawidłowo obsłużony.

Należy zwrócić uwagę, że kod z ćwiczenia 6.5 nieco odbiega od przedstawionego wcześniej schematu, choć zgodnie z przedstawionym wyżej opisem jest jak najbardziej poprawny. Otóż instrukcja `catch` w tym przykładzie ma postać:

```
catch(TypWyjatkku)
```

zamiast:

```
catch(Wyjatkku identyfikatorWyjatkku)
```

Czym jest `identyfikatorWyjatkku`? Wyjaśnienie trzeba zacząć od tego, że wyjątki są obiektami. W momencie wygenerowania błędu tworzony jest również odpowiadający mu obiekt, zatem `identyfikatorWyjatkku` to zmienna referencyjna wskazująca na obiekt wyjątku. Do czego może się ona przydać? Na przykład do wyświetlania wygenerowanej przez system informacji o błędzie.

Informację taką można otrzymać na dwa różne sposoby. Pierwszy z nich to użycie metody ToString(), zapewniający najpełniejszy komunikat. Drugi to skorzystanie z właściwości Message. A zatem blok catch mógłby wyglądać następująco:

```
catch(TypWyjątku identyfikatorWyjątku)
{
    Console.WriteLine("Komunikat 1: " + identyfikatorWyjątku.ToString());
    Console.WriteLine("Komunikat 2: " + identyfikatorWyjątku.Message);
}
```

Ta metoda zostanie użyta w kolejnym ćwiczeniu.

Ć W I C Z E N I E

6.6 Wykorzystanie obiektu wyjątku

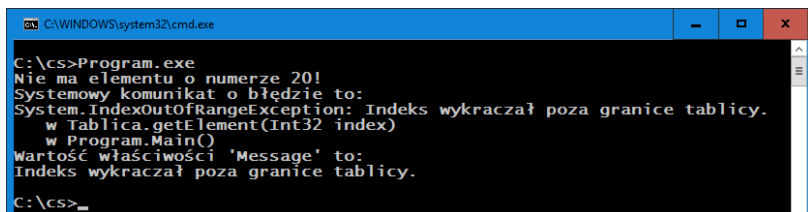
Zmodyfikuj kod z ćwiczenia 6.5 w taki sposób, aby po wystąpieniu wyjątku na ekranie pojawiały się również systemowe komunikaty o błędzie.

```
using System;

public class Program
{
    public static void Main()
    {
        Tablica tab = new Tablica();
        try
        {
            int value = tab.getElement(20);
            Console.WriteLine("Element nr 20 ma wartość: " + value);
        }
        catch(IndexOutOfRangeException e)
        {
            Console.WriteLine("Nie ma elementu o numerze 20!");
            Console.WriteLine("Systemowy komunikat o błędzie to: ");
            Console.WriteLine(e.ToString());
            Console.WriteLine("Wartość właściwości 'Message' to: ");
            Console.WriteLine(e.Message);
        }
    }
}

public class Tablica
{
    // W tym miejscu kod klasy Tablica z ćwiczenia 6.5.
}
```

Identyfikator wyjątku w tym przykładzie to nazwa `e` (to często stosowane określenie). W celu wykonania ćwiczenia wystarczyło dodać w bloku `catch` wiersze powodujące wyświetlanie wartości (ciągu znaków) zwróconej przez wywołanie `e.ToString()`¹ oraz treść zawartą w polu `Message` obiektu wskazywanego przez `e`. Po kompilacji i uruchomieniu programu zobaczymy widok przedstawiony na rysunku 6.2.



Rysunek 6.2. Systemowe komunikaty związane z przekroczeniem dopuszczalnego indeksu tablicy

Spróbujemy teraz wychwycić dosyć często występujący błąd; jest to dzielenie przez zero. Gdy nie kontrolujemy stanu zmiennych, taka sytuacja może doprowadzić do poważnej awarii aplikacji. Sprawdźmy najpierw, czy kompilator pozwoli wykonać jawne dzielenie przez zero, wprowadzone bezpośrednio w kodzie źródłowym aplikacji.

Ć W I C Z E N I E

6.7 Próba dzielenia przez zero

Napisz program, w którym występuje jawne dzielenie przez zero. Spróbuj wykonać kompilację kodu.

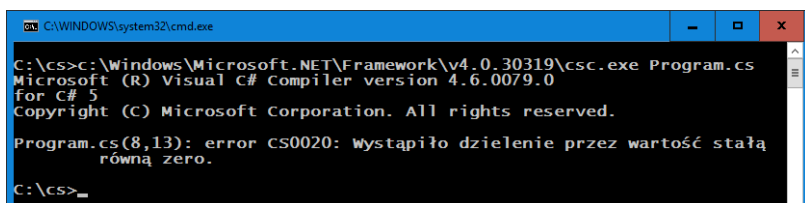
```
using System;

public class Program
{
    public static void Main()
    {
        int x = 2;
        int y = x / 0;
    }
}
```

¹ Prawidłowym zapisem byłby też `Console.WriteLine(e);`, gdyż w takim przypadku zostałaby wykonana niejawna konwersja do typu `string` polegająca właśnie na wykonaniu metody `ToString`. To jednak temat wykraczający poza ramy tego rozdziału.

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

Szybko przekonamy się, że kompilacja kodu z powyższego ćwiczenia się nie uda. Kompilator wykrywa, że chcemy dzielić przez zero i, jako że jest to operacja niedozwolona, wyświetla komunikat o błędzie. Jest to widoczne na rysunku 6.3. (Uwaga. Zależy to od wersji kompilatora. Wczesne wersje narzędzi dla C# 6.0 standardowo nie wykrywały tego błędu. Na rysunku widoczny jest komunikat z kompilatora pochodzącego z platformy .NET Framework w wersji 4.6).



Rysunek 6.3. Przy próbie jawnego dzielenia przez zero kompilator generuje błąd

Niestety, mądrość kompilatora kończy się w sytuacji, kiedy najpierw zero przypiszemy jakiejś zmiennej i dopiero tej zmiennej użyjemy jako dzielnika. W takim przypadku kompilacja uda się bez problemów, natomiast program nie będzie działał. Można się zabezpieczyć przed taką sytuacją, stosując odpowiedni blok try...catch.

Ć W I C Z E N I E

6.8 Przechwytywanie wyjątku DivideByZeroException

Napisz program, w którym występuje ukryte dzielenie przez zero. Przechwyc wygenerowany przez środowisko uruchomieniowe wyjątek i wyświetl odpowiedni komunikat na ekranie.

```
using System;  
  
public class Program  
{  
    public static void Main()  
    {  
        int x = 0;  
        int y = 6;
```

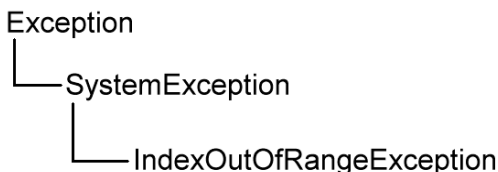
```
try
{
    int z = y / x;
    Console.WriteLine("x = " + x);
    Console.WriteLine("y = " + y);
    Console.WriteLine("z = " + z);
}
catch(DivideByZeroException)
{
    Console.WriteLine("Błąd! Dzielenie przez 0.");
}
}
```

W bloku try zmiennej z został przypisany wynik dzielenia zmiennych y i x. Ponieważ w x została zapisana wartość zero, dzielenie nie może się udać. Powstanie zatem wyjątek typu DivideByZeroException. Jest on przechwytywany w bloku catch. Dlatego po uruchomieniu powyższego kodu na ekranie pojawi się komunikat o treści: Błąd! Dzielenie przez 0.

Hierarchia wyjątków

Wyjątki w C# tworzą strukturę hierarchiczną, na czele której znajduje się klasa Exception. Z klasy tej wyprowadzone są kolejne klasy potomne obsługujące różne typy błędów. Na przykład dla znanego już wyjątku o nazwie IndexOutOfRangeException hierarchia wygląda tak jak na rysunku 6.4.

Rysunek 6.4.
*Hierarchia klas
dla wyjątku
IndexOutOfRangeException
↳ Exception*



Skoro wyjątki są strukturą hierarchiczną, nie jest obojętne, w jakiej kolejności będą przechwytywane. W jednym bloku try...catch można przecież obsłużyć wiele różnych błędów. Jaka obowiązuje kolejność? Zasada jest prosta — najpierw należy przechwytywać wyjątki bardziej szczegółowe, a dopiero później bardziej ogólne.

W przykładzie widocznym na rysunku 6.4 `IndexOutOfRangeException` jest wyjątkiem najbardziej szczegółowym, `SystemException` ogólniejszym, a `Exception` najbardziej ogólnym. Aby zobaczyć, jakie są tego konsekwencje w praktyce, wykonajmy odpowiednie ćwiczenie.

ĆWICZENIE

6.9 Nieprawidłowa kolejność wyjątków

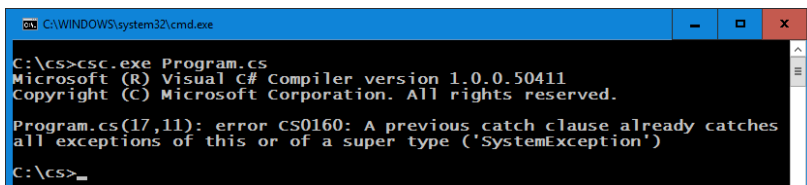
Zmodyfikuj kod z ćwiczenia 6.5 w taki sposób, aby najpierw przechwytywany był wyjątek ogólny `Exception`, a następnie wyjątek `IndexOutOfRangeException`. Spróbuj wykonać kompilację otrzymanego kodu.

```
using System;

public class Program
{
    public static void Main()
    {
        Tablica tab = new Tablica();
        try
        {
            int value = tab.getElement(20);
            Console.WriteLine("Element nr 20 ma wartość: " + value);
        }
        catch(SystemException)
        {
            Console.WriteLine("Wyjątek SystemException");
        }
        catch(IndexOutOfRangeException)
        {
            Console.WriteLine("Wyjątek IndexOutOfRangeException");
        }
    }
}

public class Tablica
{
    // Tutaj treść klasy Tablica z ćwiczenia 6.5.
}
```

Kompilacja się nie uda. Na ekranie zobaczymy komunikat pokazany na rysunku 6.5. Nie należy się dziwić: skoro najpierw przechwyciliśmy wyjątek ogólny `SystemException`, kod występujący w bloku `catch(IndexOutOfRangeException)` nigdy nie zostanie osiągnięty, kompilator zatem protestuje. Taki błąd łatwo popełnić, jeśli nie pamięta się o hierarchii wyjątków.

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system32\cmd.exe'. The command prompt shows the following text:

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

Program.cs(17,11): error CS0160: A previous catch clause already catches
all exceptions of this or of a super type ('SystemException')
C:\>cs>
```

Rysunek 6.5. Próba kompilacji kodu z błędną hierarchią wyjątków

Aby poprawić kod z ćwiczenia 6.9 tak, żeby można było wykonać kompilację, a program działał poprawnie, należy zamienić miejscami bloki catch; wtedy najpierw będzie przechwytywany wyjątek `IndexOutOfRangeException` (bardziej szczegółowy), a dopiero za nim wyjątek `SystemException` (bardziej ogólny).

Ć W I C Z E N I E

6.10 Uwzględnienie hierarchii wyjątków

Popraw kod klasy `Program` z poprzedniego ćwiczenia tak, aby wyjątki były przechwytywane w odpowiedniej kolejności.

```
using System;

public class Program
{
    public static void Main()
    {
        Tablica tab = new Tablica();
        try
        {
            int value = tab.getElement(20);
            Console.WriteLine("Element nr 20 ma wartość: " + value);
        }
        catch(IndexOutOfRangeException)
        {
            Console.WriteLine("Wyjątek IndexOutOfRangeException");
        }
        catch(SystemException){
            Console.WriteLine("Wyjątek SystemException");
        }
    }
}
```

Własne wyjątki

Stosowanie wyjątków nie ogranicza się tylko do przechwytywania tych, które są generowane przez system. Można je również zgłaszać samodzielnie. Powróćmy zatem do ćwiczenia 6.5. Konstrukcja klasy *Tablica* mogłaby być zupełnie inna. Dobrze byłoby sprawdzać, czy parametr przekazywany metodzie *getElement* przekracza dopuszczalne wartości, a jeśli tak, samodzielnie wygenerować wyjątek. Jest to możliwe; służy do tego instrukcja *throw* w postaci:

```
throw obiekt_wyjatku;
```

Jeśli zatem chcemy wygenerować (używa się również terminów „zgłosić” lub — żargonowo — „wyrzucić”) nowy wyjątek *IndexOutOfRangeException*, należy zastosować konstrukcję:

```
throw new IndexOutOfRangeException();
```

Ć W I C Z E N I E

6.11 Zgłoszenie wyjątku

Zmodyfikuj kod z ćwiczenia 6.5 w taki sposób, aby metoda *getElement* sprawdzała, czy następuje przekroczenie zakresu tablicy, i w takiej sytuacji generowała wyjątek.

```
using System;

public class Program
{
    public static void Main()
    {
        Tablica tab = new Tablica();
        try
        {
            int value = tab.getElement(20);
            Console.WriteLine("Element nr 20 ma wartość: " + value);
        }
        catch(IndexOutOfRangeException)
        {
            Console.WriteLine("Nie ma elementu o numerze 20!");
        }
    }
}

public class Tablica
{
    int[] tab;
```



```
public Tablica()
{
    tab = new int[5];
}
public int getElement(int index)
{
    if(index < 0 || index > tab.Length - 1)
    {
        throw new IndexOutOfRangeException();
    }
    else
    {
        return tab[index];
    }
}
```

Do sprawdzenia, czy został przekroczony indeks tablicy, wykorzystano zwykłą instrukcję warunkową `if`. Jeśli przekroczenie nastąpiło, powstaje nowy obiekt typu `IndexOutOfRangeException`, który jest zgłaszany za pomocą instrukcji `throw`. Dzięki temu można samodzielnie sterować tym, kiedy i jaki wyjątek zostanie zgłoszony.

Co jednak zrobić w sytuacji, gdy chcemy utworzyć własny wyjątek, np. o nazwie `NieMaTakiegoElementu`, i tenże wyjątek przechwytywać w bloku instrukcji `catch`? To żaden problem — trzeba po prostu utworzyć nową klasę, pochodną klasy `Exception` (lub jednej z klas pochodnych od `Exception`), i nazwać ją `NieMaTakiegoElementu`. W najprostszym przypadku mogłaby ona wyglądać następująco:

```
public class NieMaTakiegoElementu : Exception
{
}
```

To wszystko. Od tej chwili klasa `NieMaTakiegoElementu` będzie opisywała pełnoprawny wyjątek, który możemy wygenerować z zastosowaniem instrukcji `throw`, czyli pisząc:

```
throw new NieMaTakiegoElementu();
```

Ć W I C Z E N I E

6.12 Wykorzystanie własnego wyjątku

Zdefiniuj klasę wyjątku o nazwie `NieMaTakiegoElementu` i zmodyfikuj kod z ćwiczenia 6.11 tak, aby z niej korzystał.

```
using System;

public class NieMaTakiegoElementu : Exception
{
}

public class Program
{
    public static void Main()
    {
        Tablica tab = new Tablica();
        try
        {
            int value = tab.getElement(20);
            Console.WriteLine("Element nr 20 ma wartość: " + value);
        }
        catch(NieMaTakiegoElementu)
        {
            Console.WriteLine("Nie ma elementu o numerze 20!");
        }
    }
}

public class Tablica
{
    int[] tab;
    public Tablica()
    {
        tab = new int[5];
    }
    public int getElement(int index)
    {
        if(index < 0 || index > tab.Length - 1)
        {
            throw new NieMaTakiegoElementu();
        }
        else
        {
            return tab[index];
        }
    }
}
```

Struktura kodu pozostała tu taka sama jak w ćwiczeniu 6.11. Różnice są takie, że na początku pojawiła się definicja klasy `NieMaTakiegoElementu` (pochodnej od `Exception`) i obiekty tej klasy są używane przy generowaniu oraz przechwytywaniu wyjątków.

Sekcja finally

Do bloku `try...catch`, a także do samego bloku `try` można dołączyć sekcję `finally`. Kod tej sekcji zostanie wykonany zawsze, niezależnie od tego, czy wyjątek wystąpi, czy też nie. W pierwszym przypadku schemat tej konstrukcji będzie wyglądał następująco:

```
try
{
    // blok instrukcji mogący spowodować wyjątek
}
catch (TypWyjtku1 identyfikatorWyjtku1)
{
    // obsługa wyjątku 1
}
// inne ewentualne bloki catch
finally{
    // kod wykonywany zawsze, niezależnie od warunków
}
```

W drugim przypadku całą konstrukcję można zapisać następująco:

```
try
{
    // dowolne instrukcje
}
finally{
    // kod wykonywany zawsze, niezależnie od warunków
}
```

O tym, że blok `finally` faktycznie jest wykonywany w każdym przypadku, można się przekonać, wykonując ćwiczenie 6.13.

ĆWICZENIE

6.13 Użycie bloku finally

Napisz przykładowy kod programu, w którym zawarty będzie blok `try...finally`. W sekcji `try` zamieść instrukcję przerywającą działanie aplikacji, a w sekcji `finally` instrukcję wyświetlającą komunikat. Sprawdź, czy całość działa zgodnie z założeniami.

```
using System;

public class Program
{
    public static void Main()
    {
        Console.WriteLine("[1]Przed blokiem try...finally");
        try
        {
            Console.WriteLine("[2]Początek bloku try");
            return;
            Console.WriteLine("[3]Koniec bloku try");
        }
        finally
        {
            Console.WriteLine("[4]Blok finally");
        }
        Console.WriteLine("[5]Po bloku try...finally");
    }
}
```

W programie znajduje się instrukcja `try...finally` oraz są instrukcje `Console.WriteLine` wyświetlające komunikaty o kolejnych fazach działania aplikacji. W bloku `try` znajduje się instrukcja `return` powodująca natychmiastowe przerwanie działania metody `Main` — a tym samym zakończenie działania programu. Nie zostanie więc wykonana instrukcja `Console.WriteLine` oznaczona numerem 3, a skoro program się zakończy, pominięta zostanie również instrukcja oznaczona numerem 5. Inaczej jest jednak z instrukcją nr 4 znajdującą się w bloku `finally`. Ponieważ treść tego bloku jest wykonywana niezależnie od tego, co się dzieje w boku `try` (nawet jeśli jest to zakończenie działania aplikacji), instrukcja:

```
Console.WriteLine("[4]Blok finally");
```

będzie zawsze wykonywana (rysunek 6.6).

```
C:\WINDOWS\system32\cmd.exe

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

Program.cs(12,7): warning CS0162: Unreachable code detected
Program.cs(18,5): warning CS0162: Unreachable code detected

C:\cs>Program.exe
[1]Przed blokiem try...finally
[2]Początek bloku try
[4]Blok finally

C:\cs>_
```

Rysunek 6.6. Blok finally został wykonany mimo przerwania działania programu

Warto przy tym zauważyć, że kompilator potrafi wykryć nieosiągalne części kodu (czyli takie, które ze względu na konstrukcję programu nie mogą zostać w ogóle wykonane) i w trakcie kompilacji przedstawionego programu na ekranie pojawiają się związane z tym ostrzeżenia (widoczne na rysunku 6.6).

Filtrowanie wyjątków

W C# 6.0 pojawiła się dodatkowa możliwość filtrowania wyjątków. Konkretnie chodzi o to, że wykonywanie dowolnego bloku catch może być ograniczone tylko do sytuacji, w której spełniony jest pewien warunek. Ogólna konstrukcja bloku ma wtedy następującą postać:

```
try
{
    // blok instrukcji mogący spowodować wyjątek
}
catch(TypWyjątku1 identyfikatorWyjątku1) when(wyrażenie)
{
    // obsługa wyjątku 1
}
// inne bloki catch
```

Wyrażenie znajdujące się w nawiasie okrągłym występującym za słowem when powinno dawać wartość true, jeżeli dany blok catch ma być wykonany, a false — w przeciwnym wypadku.

Ć W I C Z E N I E

6.14 Warunkowe bloki catch

Napisz przykładowy program, w którym zostanie użyta technika filtrowania wyjątków.

```
using System;

public class Program
{
    public static void Main()
    {
        bool detale = true;
        int[] tab = {0,10,100};
        try
        {
            Console.WriteLine($"tab[4] = {tab[4]}");
            int wynik = 100 / tab[0];
            Console.WriteLine($"A wynik działania to {wynik}.");
        }
        catch(IndexOutOfRangeException) when(detale)
        {
            Console.WriteLine("Nieprawidłowy indeks tablicy.");
        }
        catch(DivideByZeroException) when(detale)
        {
            Console.WriteLine("Dzielenie przez zero.");
        }
        catch(Exception){
            Console.WriteLine("Jakiś błąd.");
        }
    }
}
```

W metodzie Main została zadeklarowana zmienna `detale` typu `bool`, która będzie decydowała o tym, czy mają być wyświetlane szczegółowe informacje o błędach (`detale = true`), czy też nie (`detale = false`), zadeklarowano też tablicę liczb typu całkowitego (`tab`) zainicjalizowaną trzema wartościami. W bloku `try` znalazła się instrukcja wyświetlająca wartość piątego elementu tablicy (o indeksie 4) oraz instrukcja wykonująca dzielenie przez zero. Obie są nieprawidłowe — nie ma piątego elementu tablicy i nie można dzielić przez zero. Próba wykonania którejkolwiek z nich spowoduje zatem powstanie wyjątku. W pierwszym przypadku będzie to `IndexOutOfRangeException`, a w drugim — `DivideByZeroException`.

Wyjątki są przechwytywane w blokach `catch`. W odróżnieniu od dotychczas stosowanych konstrukcji tym razem są to jednak przechwytywania warunkowe. Będą miały miejsce tylko wtedy, gdy zmienna `detail` będzie miała wartość `true`. W przypadku wartości `false` zostanie wykonany ostatni blok `catch`, w którym przechwytywany jest wyjątek ogólny `Exception`. Tym samym w zależności od stanu tej zmiennej otrzymamy albo komunikat szczegółowy, albo też ogólny. Oczywiście komunikat o dzieleniu przez zero można otrzymać dopiero po usunięciu lub ujęciu w komentarz instrukcji powodującej wcześniejszy błąd (czyli nieprawidłowy indeks tablicy).

7

Interfejsy

Prosty interfejs

Interfejsy są konstrukcjami takimi jak klasy, z tą różnicą, że zawierają jedynie deklaracje metod, a nie ich definicje. Inaczej mówiąc, są klasami abstrakcyjnymi. Każda zwykła klasa może implementować dany interfejs, co oznacza, że musi zawierać definicje wszystkich metod umieszczonych w tym interfejsie.

Założmy, że mamy zdefiniowaną klasę `Point` w najprostszej postaci, zawierającą jedynie dwa pola typu `int` — jedno dla współrzędnej `x`, drugie dla współrzędnej `y`:

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

Pierwszym zadaniem będzie napisanie interfejsu o nazwie `IShow`, w którym znajdzie się tylko jedna metoda o nazwie `Show`.


Ć W I C Z E N I E

7.1 Pierwszy interfejs

Napisz interfejs o nazwie IShow, w którym zadeklarowana będzie metoda o nazwie Show.

```
public interface IShow
{
    void Show();
}
```

Jak widać, zadanie to nie było skomplikowane. Interfejs przypomina definicję klasy, ale zamiast słowa kluczowego `class` używa się słowa kluczowego `interface`. Metoda `show` ma też tylko deklarację, a nie ma definicji (treści).



Przekonajmy się teraz, że faktycznie w interfejsie nie wolno zdefiniować ciała (treści) metody. W tym celu wystarczy wykonać ćwiczenie 7.2.

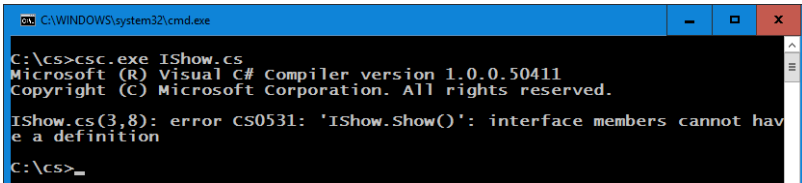
Ć W I C Z E N I E

7.2 Umieszczenie w interfejsie treści metody

Dodaj do metody `Show` z interfejsu z ćwiczenia 7.1 jej implementację. Spróbuj wykonać kompilację kodu.

```
public interface IShow
{
    void Show()
    {
        int a;
    }
}
```

Próba kompilacji powyższego fragmentu kodu spowoduje wygenerowanie błędu kompilacji CS0531 z komunikatem widocznym na rysunku 7.1 (lub podobnym, w zależności od wersji użytego kompilatora i pakietu językowego). Tak więc metody interfejsu z pewnością nie mogą zawierać implementacji.



```
C:\WINDOWS\system32\cmd.exe
C:\cs>csc.exe IShow.cs
Microsoft (R) Visual C# Compiler version 1.0.0.50411
Copyright (C) Microsoft Corporation. All rights reserved.

IShow.cs(3,8): error CS0531: 'IShow.Show()': interface members cannot have a definition

C:\cs>
```

Rysunek 7.1. Próba kompilacji interfejsu zawierającego implementację metody *Show*

Powróćmy do klasy *Point* oraz interfejsu *IShow* i sprawdźmy, jak wykorzystać je w praktyce. Przede wszystkim trzeba się dowiedzieć, jak spowodować, aby klasa implementowała interfejs. Okazuje się, że należy zastosować konstrukcję analogiczną do dziedziczenia, zatem schemat definicji klasy implementującej interfejs wyglądać będzie następująco:

```
public class nazwa_klasy : nazwa_interfejsu
{
    // definicja klasy
}
```

Ć W I C Z E N I E

7.3 Implementacja interfejsu przez klasę

Napisz kod, w którym klasa *Point* implementuje interfejs *IShow*.

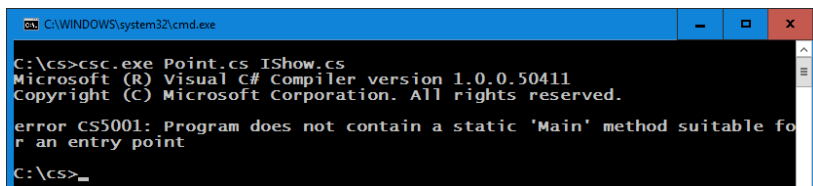
```
using System;

public interface IShow
{
    void Show();
}

public class Point : IShow
{
    public int x;
    public int y;
    public void Show()
    {
        Console.WriteLine("Dane dotyczące punktu:");
        Console.WriteLine("współrzędna x = {0}", x);
        Console.WriteLine("współrzędna y = {0}", y);
    }
}
```

Jak widać, w tym przypadku implementacja interfejsu polega na zadeklarowaniu oraz zdefiniowaniu w klasie *Point* metody *Show*. Treść tej metody może być dowolna. W prezentowanym przykładzie metoda *Show* powoduje wyświetlenie na ekranie bieżących wartości *x* oraz *y*. Przedstawiony kod można zapisać w jednym pliku lub też w dwóch osobnych. To drugie rozwiązanie wydaje się bardziej uniwersalne, dlatego też treść interfejsu *IShow* najlepiej zapisać w pliku *IShow.cs*, a treść klasy *Point* — w pliku *Point.cs*.

Oczywiście standardowa próba kompilacji kodu z ćwiczenia 7.3 (niezależnie od tego, czy został zapisany w jednym pliku, czy też w dwóch) nie powiedzie się, a na ekranie zobaczymy komunikat widoczny na rysunku 7.2¹. Powód jest chyba jasny — nie została zdefiniowana metoda *Main*, od której mogłoby się zacząć wykonywanie programu. Informacja ta zawarta jest w komunikacie zgłoszonym przez kompilator. Napiszmy zatem program, który skorzysta z klasy *Point*.



Rysunek 7.2. Niezdefiniowanie metody *Main* powoduje błąd kompilacji

Ć W I C Z E N I E

7.4 Wykorzystanie klasy *Point* i interfejsu *IShow*

Napisz program, w którym użyta będzie metoda *Show* z klasy *Point*.

```
using System;

public class Program
{
    public static void Main()
    {
        Point punkt = new Point();
        punkt.x = 100;
```

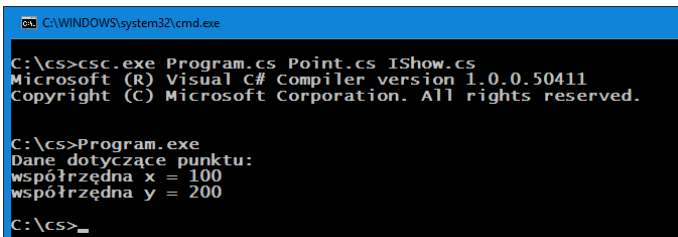
¹ Chyba że zastosujemy opcję `/target` kompilatora i skompilujemy kod np. jako bibliotekę.

```
punkt.y = 200;  
punkt.Show();  
}  
}
```

Powstała klasa Program zawierająca publiczną i statyczną metodę Main, od której zaczynię się wykonywanie programu. Ten schemat jest dobrze znany, gdyż był już wykorzystywany wiele razy. W metodzie Main deklarujemy zmienną referencyjną punkt i przypisujemy jej nowo utworzony obiekt klasy Point. Następnie inicjalizujemy pola x i y tego obiektu odpowiednio wartościami 100 i 200. Na zakończenie wywołujemy metodę Show. Jeżeli klasę Program zapiszemy w pliku *Program.cs*, klasę Point — w pliku *Point.cs*, a interfejs IShow — w pliku *IShow.cs*, do kompilacji programu niezbędne będzie wykonanie komendy:

```
csc Program.cs Point.cs IShow.cs
```

Powstanie wtedy plik *Program.exe*, a wynikiem jego działania, co nie powinno być żadnym zaskoczeniem, będzie pojawienie się na ekranie tekstu zaprezentowanego na rysunku 7.3.



```
C:\WINDOWS\system32\cmd.exe  
C:\cs>csc.exe Program.cs Point.cs IShow.cs  
Microsoft (R) Visual C# Compiler version 1.0.0.50411  
Copyright (C) Microsoft Corporation. All rights reserved.  
  
C:\cs>Program.exe  
Dane dotyczące punktu:  
współrzędna x = 100  
współrzędna y = 200  
C:\cs>
```

Rysunek 7.3. Wynik działania kodu z ćwiczenia 7.4

Interfejsy w klasach potomnych

Dotychczas implementowaliśmy interfejs jedynie w klasie bazowej, tzn. takiej, która nie dziedziczy jawnie po innej klasie². Czy zatem jest możliwa implementacja interfejsu w klasie potomnej? Jak będzie wyglądała taka konstrukcja? Na szczęście jest to bardzo proste. Wystarczy nazwę interfejsu oddzielić przecinkiem od nazwy klasy bazowej:

² Dziedziczy jednak domyślnie po klasie Object.

```
class klasa_potomna : klasa_bazowa, interfejs
{
    // definicja klasy
}
```

Zbudujmy zatem jedną klasę bazową o nazwie Shape i wyprowadźmy z niej dwie klasy potomne Rectangle i Triangle. Klasa Shape będzie klasą ogólną służącą do opisu figur geometrycznych, Rectangle będzie klasą opisującą prostokąty, Triangle klasą opisującą trójkąty. Odświeżymy przy okazji wiadomości o dziedziczeniu.

Ć W I C Z E N I E

7.5 Budowa klas reprezentujących figury geometryczne

Zbuduj klasę Shape służącą do przechowywania informacji na temat figur geometrycznych i wyprowadź z niej dwie klasy potomne: Rectangle dla prostokątów i Triangle dla trójkątów.

```
public class Shape
{
    public int color;
}

public class Point : Shape
{
    public int x;
    public int y;
}

public class Rectangle : Shape
{
    public int x;
    public int y;
    public int width;
    public int height;
}

public class Triangle : Shape
{
    public Point a;
    public Point b;
    public Point c;
}
```

Klasa Shape zawiera tylko jedną składową oznaczającą kolor figury. W klasie Rectangle zdefiniowane zostały pola x i y wyznaczające położenie górnego lewego wierzchołka oraz pola width i height określające

odpowiednio jego długość i wysokość (warto zauważyć, że zamiast pól x i y można by też użyć pola typu `Point`). Klasa `Triangle` zawiera z kolei trzy pola, wszystkie typu `Point`, które wyznaczają położenie opisywanego trójkąta.

Spróbujmy teraz zaimplementować w klasach `Rectangle` i `Triangle` znany już interfejs `IShow`. Zgodnie z wcześniejszymi wyjaśnieniami w każdej z tych klas trzeba będzie zawrzeć odpowiednią metodę `Show`. Metoda będzie musiała uwzględniać specyfikę danej figury. Co innego będzie wyświetlane w przypadku prostokąta, a co innego w przypadku trójkąta.

ĆWICZENIE

7.6 Interfejs `IShow` dla klas `Rectangle` i `Triangle`

Zaimplementuj interfejs `IShow` dla klas `Rectangle` i `Triangle` zdefiniowanych w ćwiczeniu 7.5.

```
using System;

public interface IShow
{
    void Show();
}

public class Shape
{
    public int color;
}

public class Point:Shape
{
    public int x;
    public int y;
}

public class Rectangle : Shape, IShow
{
    public int x;
    public int y;
    public int width;
    public int height;
    public void Show()
    {
        Console.WriteLine("Parametry prostokąta:");
    }
}
```

```
        Console.WriteLine("współrzędna x = {0}", x);
        Console.WriteLine("współrzędna y = {0}", y);
        Console.WriteLine("długość = {0}", width);
        Console.WriteLine("szerokość = {0}", height);
    }
}

public class Triangle : Shape, IShow
{
    public Point a;
    public Point b;
    public Point c;
    public void Show()
    {
        Console.WriteLine("Parametry trójkąta:");
        Console.WriteLine("punkt a = ({0}, {1})", a.x, a.y);
        Console.WriteLine("punkt b = ({0}, {1})", b.x, b.y);
        Console.WriteLine("punkt c = ({0}, {1})", c.x, c.y);
    }
}
```

W klasach `Rectangle` oraz `Triangle` pojawiła się metoda `Show`. W pierwszym przypadku wyświetla ona współrzędne lewego górnego rogu oraz szerokość i wysokość, a w drugim — współrzędne wszystkich wierzchołków. Obie metody mają typ zwracany `void` i nie przyjmują argumentów, zatem są zgodne z deklaracją zawartą w interfejsie `IShow`, który jest implementowany w obu wymienionych klasach.

Ć W I C Z E N I E

7.7 Użycie klas `Rectangle` i `Triangle`

Napisz program, w którym będą używane obiekty typów `Rectangle` i `Triangle`.

```
using System;

public class Program
{
    public static void Main()
    {
        Rectangle prostokat = new Rectangle();
        prostokat.x = 100;
        prostokat.y = 200;
        prostokat.width = 50;
        prostokat.height = 80;

        Triangle trojkat = new Triangle();
        trojkat.a = new Point();
    }
}
```



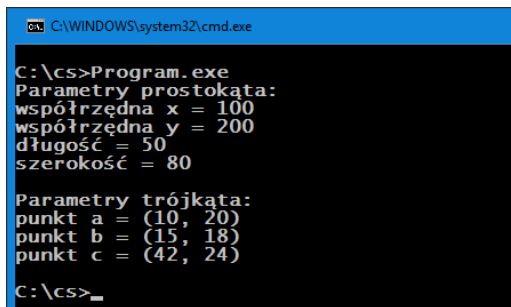
```
trojkat.b = new Point();
trojkat.c = new Point();
trojkat.a.x = 10;
trojkat.a.y = 20;
trojkat.b.x = 15;
trojkat.b.y = 18;
trojkat.c.x = 42;
trojkat.c.y = 24;

prostokat.Show();
Console.WriteLine("");
trojkat.Show();
}
}
```

W metodzie Main klasy Program tworzymy obiekt typu Rectangle i przypisujemy jego polom przykładowe wartości. Podobnie postępujemy z obiektem typu Triangle. Ponieważ pola w tej klasie są typu Point (a klasa Triangle nie ma odpowiedniego konstruktora), trzeba dodatkowo utworzyć obiekty klasy Point. Po przypisaniu wszystkich wartości wywołujemy odpowiednie metody Show, co powoduje pojawienie się na ekranie informacji zaprezentowanych na rysunku 7.4.

Rysunek 7.4.

*Przykład
działania
metody Show
dla obiektów
różnych klas*



```
C:\WINDOWS\system32\cmd.exe

C:\cs>Program.exe
Parametry prostokata:
współrzędna x = 100
współrzędna y = 200
długość = 50
szerokość = 80

Parametry trójkąta:
punkt a = (10, 20)
punkt b = (15, 18)
punkt c = (42, 24)
C:\cs>
```

Warto tu zwrócić uwagę, że o ile we wcześniejszych ćwiczeniach (7.3, 7.4) implementowaliśmy interfejs IShow dla klasy Point, tym razem (ćwiczenia 7.6 i 7.7) wyświetlanie parametrów punktu obsługujemy z poziomu klasy Triangle, np. w wierszu:

```
Console.WriteLine("punkt a = ({0}, {1})", a.x, a.y);
```

Jest to pewna niespójność w kodzie, jako że punkt też jest figurą geometryczną i nie ma powodu, aby pozbawiać klasę Point możliwości samodzielnego wyświetlania parametrów na ekranie. Warto naprawić

to małe niedopatrzenie i dodać metodę Show, tak aby dane wyświetlane były np. w następującej postaci:

(wartość_x, wartość_y)

Da się też zauważyć, że klasom Rectangle i Triangle przydałyby się konstruktory, tak aby parametry opisujące prostokąty i trójkąty przekazywane były już w trakcie tworzenia danych obiektów. W obecnej postaci łatwo zapomnieć o zainicjalizowaniu jednego z pól. Cały kod jest też dużo mniej czytelny.

Wszystkie te poprawki wprowadzimy w kolejnym ćwiczeniu.

Ć W I C Z E N I E

7.8

Dodatkowe konstruktory i implementacje interfejsu

Zaimplementuj interfejs IShow dla klasy Point, dodaj konstruktory dla klas Point, Rectangle i Triangle.

```
using System;

interface IShow
{
    void Show();
}

public class Shape
{
    public int color;
}

public class Rectangle : Shape, IShow
{
    public int x;
    public int y;
    public int width;
    public int height;
    public Rectangle(int x, int y, int width, int height)
    {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    public void Show()
    {
        Console.WriteLine("Parametry prostokąta:");
    }
}
```

```
        Console.WriteLine("współrzędna x = {0}", x);
        Console.WriteLine("współrzędna y = {0}", y);
        Console.WriteLine("długość = {0}", width);
        Console.WriteLine("szerokość = {0}\n", height);
    }
}

public class Triangle : Shape, IShow
{
    public Point a;
    public Point b;
    public Point c;
    public Triangle(Point a, Point b, Point c)
    {
        this.a = a;
        this.b = b;
        this.c = c;
    }
    public void Show()
    {
        Console.WriteLine("Parametry trójkąta:");
        Console.Write("punkt a = ");
        a.Show();
        Console.Write("\npunkt b = ");
        b.Show();
        Console.Write("\npunkt c = ");
        c.Show();
        Console.WriteLine("");
    }
}

public class Point : Shape, IShow
{
    public int x;
    public int y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public void Show()
    {
        Console.WriteLine("{0}, {1}", x, y);
    }
}
```

Klasa `Rectangle` otrzymała czteroargumentowy konstruktor, w którym przekazywane są wartości dla wszystkich zawartych w niej pól (`x`, `y`, `width`, `height`). W jego wnętrzu wartości argumentów przypisywane są odpowiednim polom. W klasie `Triangle` pojawił się konstruktor

trójargumentowy, przyjmujący argumenty typu `Point` określające współrzędne wierzchołków trójkąta. Najwięcej zmian zaszło w klasie `Point`. Implementuje ona teraz interfejs `IShow`, a więc zawiera również metodę `Show` wyświetlającą wartości współrzędnych. Dodatkowo jest również konstruktor przyjmujący dwie wartości typu `int` i przypisujący je polom `x` i `y`.

Ć W I C Z E N I E

7.9 Zastosowanie nowych wersji klas

Napisz klasę `Program` testującą zachowanie klas z ćwiczenia 7.8.

```
using System;

public class Program
{
    public static void Main()
    {
        Rectangle prostokat = new Rectangle(100, 200, 50, 80);
        Triangle trojkat = new Triangle
        (
            new Point(10, 20),
            new Point(15, 18),
            new Point(42, 24)
        );
        prostokat.Show();
        trojkat.Show();
    }
}
```

W porównaniu z ćwiczeniem 7.7 kod klasy `Program` bardzo się uprościł i jest o wiele czytelniejszy. Obiekty przypisywane zmiennym `prostokat` i `trojkat` tworzone są za pomocą nowych konstruktorów, dzięki temu nie trzeba dokonywać bezpośrednich przypisań do pól. W wywołaniu konstruktora klasy `Triangle` używany jest też nowo powstały konstruktor klasy `Point`.

Czy to interfejs?

Wiemy już sporo o interfejsach w C#, przydałyby się jednak jeszcze wiadomości o tym, w jaki sposób stwierdzić, czy dany obiekt implementuje konkretny interfejs. W najprostszym przypadku taką wiedzę

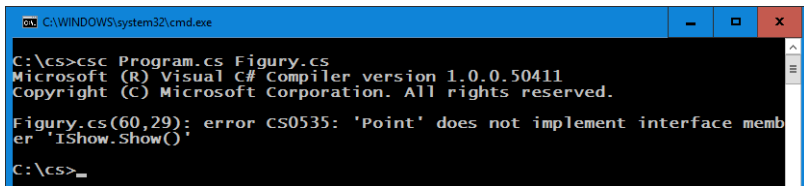
dysponować będziemy już na etapie kompilacji. Założmy, że operujemy na klasach `Rectangle` i `Triangle` utworzonych w ćwiczeniu 7.8 i klasie `Point` z ćwiczenia 7.5 uzupełnionej konstruktorem z ćwiczenia 7.8. Klasy `Rectangle` i `Triangle` implementują interfejs `IShow`, natomiast klasa `Point` — nie.

Co się zatem stanie, jeśli spróbujemy skompilować następujący fragment kodu:

```
using System;

public class Program
{
    public static void Main()
    {
        Rectangle prostokat = new Rectangle(100, 200, 50, 80);
        Triangle trojkat = new Triangle
        (
            new Point(10, 20),
            new Point(15, 18),
            new Point(42, 24)
        );
        Point punkt = new Point(150, 300);
        prostokat.Show();
        trojkat.Show();
        punkt.Show();
    }
}
```

Oczywiście kompilacja się nie uda, jako że w klasie `Point` nie ma metody `Show`, a na ekranie zobaczymy komunikat o błędach widoczny na rysunku 7.5 (komunikatów może być więcej, zależy to od konkretnej wersji kompilatora).



Rysunek 7.5. Próba wywołania nieistniejącej metody powoduje błąd kompilacji

W tym przypadku nie było żadnych problemów, nie zawsze jednak mamy tak klarowną sytuację. Rozpatrzmy nieco bardziej skomplikowany przykład (czytelnicy nieznający pojęć klas abstrakcyjnych oraz

metod wirtualnych mogą pominąć dalszy fragment i przejść od razu do rozdziału 8.). W tym celu wykonamy pewne modyfikacje utworzonych dotychczas klas Shape, Rectangle, Triangle i Point.

W klasie Shape zdefiniujemy wirtualną metodę Show. Zmiana ta wymusi przesłonięcie (inaczej nadpisanie, ang. *override*) metod Show w klasach Rectangle i Triangle, jako że wciąż będą one implementować interfejs IShow. W klasie Point metody Show nie zadeklarujemy.

Ć W I C Z E N I E

7.10 Interfejsy i metody wirtualne

Zmodyfikuj kod dotyczący klas Shape, Rectangle, Triangle i Point w taki sposób, aby w klasie Shape znalazła się wirtualna metoda Show.

```
interface IShow
{
    void Show();
}

public abstract class Shape
{
    public int color;
    public virtual void Show()
    {
        Console.WriteLine("Metoda Show z klasy bazowej");
    }
}

public class Rectangle : Shape, IShow
{
    public int x;
    public int y;
    public int width;
    public int height;
    public Rectangle(int x, int y, int width, int height)
    {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    public override void Show()
    {
        Console.WriteLine("Parametry prostokąta:");
        Console.WriteLine("współrzędna x = {0}", x);
        Console.WriteLine("współrzędna y = {0}", y);
        Console.WriteLine("długość = {0}", width);
        Console.WriteLine("szerokość = {0}\n", height);
    }
}
```

```
    }  
}  
  
public class Triangle : Shape, IShow  
{  
    public Point a;  
    public Point b;  
    public Point c;  
    public Triangle(Point a, Point b, Point c)  
    {  
        this.a = a;  
        this.b = b;  
        this.c = c;  
    }  
    public override void Show()  
    {  
        Console.WriteLine("Parametry trójkąta:");  
        Console.WriteLine("punkt a = ({0}, {1})", a.x, a.y);  
        Console.WriteLine("punkt b = ({0}, {1})", b.x, b.y);  
        Console.WriteLine("punkt c = ({0}, {1})\n", c.x, c.y);  
    }  
}  
  
public class Point : Shape  
{  
    public int x;  
    public int y;  
    public Point(int x, int y)  
    {  
        this.x = x;  
        this.y = y;  
    }  
}
```

Klasa Shape stała się klasą abstrakcyjną, co oznacza, że nie można będzie tworzyć jej instancji (obiektów typu Shape). Z kolei umieszczona w niej metoda Show stała się metodą wirtualną (oznacza to sprawdzanie typu obiektu w trakcie wykonania programu, tzw. późne wiązanie lub wiązanie czasu wykonania, ang. *late binding*, *runtime binding*). W klasach Rectangle oraz Triangle metoda Show została zadeklarowana z użyciem słowa `override`. To pozwala na przesłonięcie metody Show z klasy bazowej (Shape). Ostatnia z klas (Point) nie implementuje interfejsu IShow, zatem nie musi zawierać metody Show. Dlatego też zawiera tylko konstruktor (oraz definicję pól).



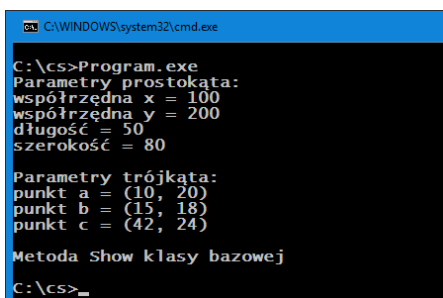
Nietrudno się domyślić, że po zdefiniowaniu klas, takim jak w ćwiczeniu 7.10, kod przedstawiony na początku tego podrozdziału będzie w pełni poprawny, a więc uda się jego kompilacja oraz wykonanie. Szczególnie zaś możliwe będzie wywołanie metody `Show` dla obiektu `Punkt`:

```
punkt.Show();
```

Efekt działania takiego kodu jest widoczny na rysunku 7.6. Widać wyraźnie, że dla obiektów prostokąt i trójkąt zostały wywołane metody `Show` klas `Rectangle` i `Triangle`, natomiast dla obiektu `punkt` wywołana została metoda pochodząca z klasy bazowej (`Shape`).

Rysunek 7.6.

*Dla obiektu
punkt została
wywołana
metoda Show
klasy Shape*



```
C:\WINDOWS\system32\cmd.exe
C:\cs>Program.exe
Parametry prostokąta:
współrzędna x = 100
współrzędna y = 200
długość = 50
szerokość = 80
Parametry trójkąta:
punkt a = (10, 20)
punkt b = (15, 18)
punkt c = (42, 24)
Metoda Show klasy bazowej
C:\cs>_
```

Problem pojawi się w sytuacji, gdy trzeba będzie wywoływać wyłącznie metody implementowane ze względu na interfejs, a nie będzie wiadomo dokładnie, jakiego typu jest obiekt. Inaczej mówiąc, metoda `Show` ma być wywołana tylko wtedy, kiedy dany obiekt implementuje interfejs `IShow`. W pozostałych sytuacjach nie powinna być wywoływana. W powyższym przypadku teoretycznie wiadomo, które obiekty implementują ten interfejs — wystarczy zajrzeć do kodu źródłowego danej klasy. Nie zawsze jednak będzie można tak łatwo to sprawdzić.

Ć W I C Z E N I E

7.11 Wywołanie metody niezgodne z założeniami

Zadeklaruj tablicę obiektów typu `Shape` i przypisz jej komórkom wskazania na obiekty klas `Rectangle`, `Trinangle` i `Point`. Dla każdego z elementów tablicy w pętli `for` wywołaj metodę `Show`.

```
using System;

public class Program
{
```



```
public static void Main()
{
    Rectangle prostokat = new Rectangle(100, 200, 50, 80);
    Triangle trojkat = new Triangle
    (
        new Point(10, 20),
        new Point(15, 18),
        new Point(42, 24)
    );
    Point punkt = new Point(150, 300);

    Shape[] tab = new Shape[3];
    tab[0] = prostokat;
    tab[1] = trojkat;
    tab[2] = punkt;

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

W kodzie zostały utworzone trzy obiekty typów `Rectangle`, `Triangle` oraz `Point`. Powstała również 3-elementowa tablica typu `Shape`, w której kolejnych komórkach zostały zapisane obiekty: prostokat, trojkat i punkt. Przypisanie było możliwe, ponieważ wymienione klasy dziedziczą po klasie `Shape`. Na zakończenie w pętli `for` dla każdego obiektu zawartego w tablicy została wywołana metoda `Show` (`tab[i].Show()`).

Program zaprezentowany w ćwiczeniu 7.11 zadziała, choć nie do końca zgodnie z pierwotnymi założeniami. Ponieważ nie sprawdzamy, jaki jest rzeczywisty typ obiektu zawartego w danej komórce tablicy, wywołamy metodę `Show` również dla komórki numer trzy (o indeksie 2), a znajdujący się tam obiekt jest typu `Point`. Ta klasa nie dziedziczy po interfejsie `IShow` i nie zawiera metody `Show`. Dlatego też wywołana będzie metoda klasy bazowej. Tymczasem metoda `Show` miała być wywoływana jedynie dla obiektów implementujących interfejs `IShow`.

Jak zatem stwierdzić, czy dany obiekt implementuje określony interfejs, czy też nie? Do dyspozycji są trzy metody:

- ❑ rzutowanie typu,
- ❑ konstrukcja ze słowem `is`,
- ❑ konstrukcja ze słowem `as`.

Rzutowanie

Pierwsza metoda to dokonanie rzutowania obiektu na typ interfejsu. Jeśli w tablicy znajduje się obiekt typu Shape, można wykonać próbę rzutowania go na typ IShow, czyli użyć konstrukcji w postaci:

```
(IShow) tablica[indeks];
```

Jeśli obiekt znajdujący się w tablicy implementuje interfejs IShow, rzutowanie zakończy się sukcesem. W przeciwnym razie wygenerowany zostanie wyjątek `InvalidCastException`. Wyjątek ten można przechwycić i odpowiednio zareagować. Zastosujmy zatem tę metodę i poprawmy kod z ćwiczenia 7.11 tak, aby działał zgodnie z pierwotnymi założeniami.

Ć W I C Z E N I E

7.12 Rzutowanie na typ interfejsu

Zmień kod z ćwiczenia 7.11 w taki sposób, aby metoda `Show` była wywoływana tylko w stosunku do obiektów implementujących interfejs `IShow`. Zastosuj metodę rzutowania i przechwytywania wyjątków.

```
using System;

public class Program
{
    public static void Main()
    {
        Rectangle prostokat = new Rectangle(100, 200, 50, 80);
        Triangle trojkat = new Triangle
        (
            new Point(10, 20),
            new Point(15, 18),
            new Point(42, 24)
        );
        Point punkt = new Point(150, 300);
        Shape[] tab = new Shape[3];
        tab[0] = prostokat;
        tab[1] = trojkat;
        tab[2] = punkt;
        for(int i = 0; i < tab.Length; i++)
        {
            try
            {
                ((IShow)tab[i]).Show();
            }
            catch(InvalidCastException)
            {
            }
        }
    }
}
```

```

    {
        Console.WriteLine("tab[{0}]: brak wywołania Show.", i);
    }
}
}

```

Zmodyfikowana została pętla `for` przetwarzająca elementy tablicy `tab`. Każdy element poddawany jest rzutowaniu na typ interfejsowy `IShow`. To oznacza, że jeśli dany obiekt implementuje interfejs `IShow`, konwersja zostanie wykonana i będzie mogła być wywołana metoda `Show`. W przeciwnym razie (obiekt nie implementuje interfejsu `IShow`) będzie wygenerowany wyjątek `InvalidCastException` przechwytywany w bloku `catch`. A zatem metoda `Show` zostanie wywołana jedynie dla obiektów klas `Rectangle` i `Triangle`, natomiast dla obiektu typu `Point` zostanie wyświetlony osobny komunikat informacyjny. Widać to wyraźnie na rysunku 7.7.

Rysunek 7.7.

Metoda `Show`
została
wywołana
jedynie dla
obiektów
`Rectangle`
i `Triangle`

```

C:\cs>Program.exe
Parametry prostokąta:
współrzędna x = 100
współrzędna y = 200
długość = 50
szerokość = 80

Parametry trójkąta:
punkt a = (10, 20)
punkt b = (15, 18)
punkt c = (42, 24)

tab[2]: brak wywołania Show.
C:\cs>_

```

Kod bloku `try` z ćwiczenia 7.12 można też zapisać w nieco czytelniejszej, choć bardziej rozwlekłej postaci:

```

try
{
    IShow tempObj = (IShow)tab[i];
    tempObj.Show();
}

```

Możliwe byłoby również umieszczenie instrukcji `tempObj.Show()` poza blokiem `try...catch`. Pętla `for` wyglądałaby wtedy tak:

```
for(int i = 0; i < tab.Length; i++)
{
    IShow tempObj;
    try
    {
        tempObj = (IShow)tab[i];
    }
    catch(InvalidCastException)
    {
        Console.WriteLine("tab[{0}]: brak wywołania Show.", i);
        continue;
    }
    tempObj.Show();
}
```

Słowo kluczowe as

Sposób drugi to użycie operatora `as`, dla którego ogólny schemat zastosowania jest następujący:

wyrażenie as typ

Powoduje to konwersję wyrażenia do typu podanego z prawej strony operatora, a jeśli taka konwersja jest niemożliwa — zwrócenie wartości `null`. W przypadku naszych ćwiczeń z interfejsami można to wykorzystać w sposób następujący:

obiekt as interfejs;

Na przykład:

```
IShow obj = tablica[indeks] as IShow;
```

Gdy obiekt nie implementuje danego interfejsu, konstrukcja `as` zwraca wartość `null`. Jeśli w powyższym przypadku w tablicy `tablica` na pozycji `indeks` nie ma obiektu implementującego interfejs `IShow`, zmiennej `obj` zostanie przypisana wartość `null`.

Konstrukcja ze słowem kluczowym `as` to również rodzaj rzutowania, realizowany tylko nieco inaczej niż w przykładzie z ćwiczenia 7.12.

Ć W I C Z E N I E

7.13 Użycie operatora as

Zmień kod z ćwiczenia 7.12 w taki sposób, aby metoda Show była wywoływana tylko w stosunku do obiektów implementujących interfejs IShow. Wykorzystaj konstrukcję ze słowem kluczowym as.

```
using System;

public class Program
{
    public static void Main()
    {
        // Tutaj początek kodu z ćwiczenia 7.11.

        for(int i = 0; i < tab.Length; i++)
        {
            IShow tempObj = tab[i] as IShow;
            if(tempObj != null)
            {
                tempObj.Show();
            }
            else
            {
                Console.WriteLine("tab[{0}]: brak wywołania Show.", i);
            }
        }
    }
}
```

Słowo kluczowe is

Ostatnim z prezentowanych sposobów na stwierdzenie, czy obiekt implementuje dany interfejs, jest użycie operatora is. Schemat zastosowania jest następujący:

obiekt is typ

Takie wyrażenie zwraca wartość true, gdy *obiekt* jest typu *typ* (a dokładniej: gdy może być wykonane rzutowanie tego obiektu na wskazany typ).

Wyrażenia is najczęściej używa się w połączeniu z klasyczną konstrukcją warunkową if. Schemat postępowania jest następujący:

```
if(obiekt is interfejs)
{
    // instrukcje do wykonania, gdy obiekt implementuje interfejs
}
```

Widać więc wyraźnie, że i ten sposób doskonale nadaje się do użycia w omawianym przypadku. Wykonajmy zatem ostatnie ćwiczenie w tym rozdziale.

Ć W I C Z E N I E

7.14 Wykorzystanie operatora is

Zmień kod z ćwiczenia 7.11 w taki sposób, aby metoda Show była wywoływana tylko w stosunku do obiektów implementujących interfejs IShow. Wykorzystaj konstrukcję ze słowem kluczowym is.

```
using System;

public class Program
{
    public static void Main()
    {
        // Tutaj początek kodu z ćwiczenia 7.11

        for(int i = 0; i < tab.Length; i++)
        {
            if(tab[i] is IShow)
            {
                tab[i].Show();
            }
            else
            {
                Console.WriteLine("tab[{0}]: brak wywołania Show.", i);
            }
        }
    }
}
```

Część III

Programowanie w Windows

8

Pierwsze okno

Utworzenie okna

W pierwszych dwóch częściach książki znalazły się przykłady bardzo wielu programów konsolowych, zatem najwyższy czas zająć się tworzeniem aplikacji z graficznym interfejsem użytkownika. Podobnie jak w części pierwszej kod będziemy pisać „ręcznie”, nie korzystając z pomocy edytora Visual C#. To pozwoli na dobre poznanie mechanizmów rządzących aplikacjami okienkowymi.

Podstawowy szablon kodu pozostanie taki sam jak w dotychczasowych przykładach. Dodatkowo trzeba będzie poinformować kompilator o tym, że chcemy korzystać z klas pakietu Windows.Forms. Na początku kodu będzie więc umieszczana dyrektywa `using` w postaci `using Windows.Forms;`

```
using System;
using Windows.Forms;

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

```
{  
    // tutaj instrukcje do wykonania  
}
```

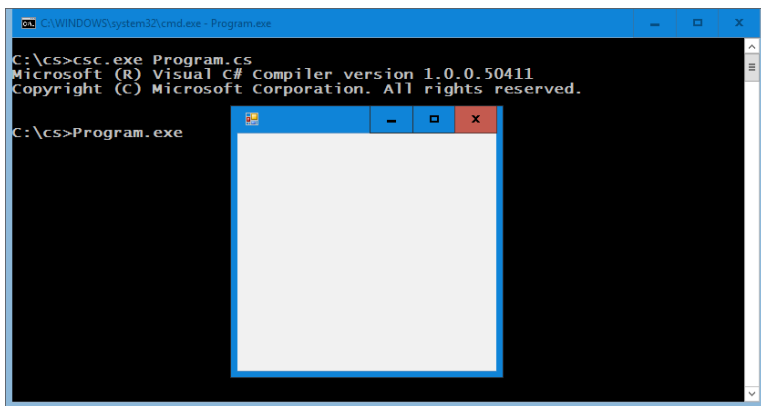
Do utworzenia podstawowego okna będzie potrzebna klasa `Form` z platformy .NET. Należy utworzyć jej instancję oraz przekazać ją jako parametr w wywołaniu instrukcji `Application.Run()`. A zatem w najprostszym przypadku w metodzie `Main` powinna znaleźć się linia: `Application.Run(new Form());`.

Ć W I C Z E N I E

8.1 Wyświetlenie okna aplikacji

Napisz aplikację okienkową, której jedynym zadaniem będzie wyświetlenie na ekranie okna (rysunek 8.1).

```
using System;  
using System.Windows.Forms;  
  
public class Program  
{  
    public static void Main()  
    {  
        Application.Run(new Form());  
    }  
}
```



Rysunek 8.1. Wynik działania kodu z ćwiczenia 8.1

Okno z ćwiczenia 8.1 nie robi nic pożytecznego, warto jednak zauważyć, że są w nim do dyspozycji działające przyciski służące do minimalizacji, maksymalizacji oraz zamykania, a także typowe menu systemowe. Osoby programujące w Javie powinny zwrócić też uwagę, że faktycznie taką aplikację można zamknąć, klikając odpowiedni przycisk, mimo że w kodzie nie ma żadnych instrukcji odpowiedzialnych za wykonanie tej operacji.

Nie będziemy jednak zadowoleni z jednej rzeczy. Otóż niezależnie od tego, czy tak skompilowany program zostanie uruchomiony z poziomu wiersza poleceń, czy też przez kliknięcie jego ikony, zawsze w tle pojawiać się będzie okno konsoli. Jest to widoczne na rysunku 8.1. Powód takiego zachowania jest prosty. Domyślnie kompilator zakłada, że tworzymy aplikację konsolową. Dopiero ustawienie odpowiedniej opcji kompilacji zmienia 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 jego formy skróconej:

```
csc /t:winexe Program.cs
```

Klasa `Form` udostępnia duży zbiór właściwości, które wpływają na jej zachowanie. Część z nich zaprezentowana jest w tabeli 8.1. Pozwalają one m.in. na zmianę rozmiarów, kolorów czy przypisanego kroju czcionki.

Tabela 8.1. Wybrane właściwości klasy `Form`

Typ	Nazwa właściwości	Znaczenie
bool	<code>AutoScaleMode</code>	Ustala tryb automatycznego skalowania okna.
bool	<code>AutoScroll</code>	Określa, czy w oknie mają się automatycznie pojawiać paski przewijania.
bool	<code>AutoSize</code>	Określa, czy forma (okno) może automatycznie zmieniać rozmiary zgodnie z trybem określonym przez <code>AutoSizeMode</code> .
<code>AutoSizeMode</code>	<code>AutoSizeMode</code>	Określa tryb automatycznej zmiany rozmiarów formy.
<code>Color</code>	<code>BackColor</code>	Określa aktualny kolor tła.

Tabela 8.1. Wybrane właściwości klasy *Form* — ciąg dalszy

Typ	Nazwa właściwości	Znaczenie
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.
↪ Style		
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 zadań.
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.
↪ State		

Ć W I C Z E N I E

8.2 Wyświetlenie okna o zadanym rozmiarze

Napisz aplikację wyświetlającą okno o rozmiarze 320×200 pikseli. Zdefiniuj tytuł okna wyświetlany na pasku tytułu.

```
using System;
using System.Windows.Forms;

public class MainForm : Form
{
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;
        this.Text = "Okno z tytułem";
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

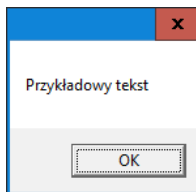
Powstała tu klasa MainForm dziedzicząca po klasie Form, a więc przejmująca jej cechy i właściwości. Jest to bardzo prosta konstrukcja zawierająca jedynie konstruktor oraz metodę Main. W konstruktorze właściwościom width i height przypisywane są wartości 320 i 200, a właściwości Text — dowolny tekst, który stanie się tytułem okna. W metodzie Main tworzony jest nowy obiekt klasy MainForm, który staje się argumentem metody Run klasy Application. Dzięki temu na ekranie pojawia się okno o wymiarach 320×200 i zdefiniowanym tytule.

Wyświetlanie komunikatu

Aby wyświetlić okno z komunikatem (rysunek 8.2), należy skorzystać z klasy MessageBox. Udostępnia ona metodę Show wyświetlającą ciąg znaków podany jako argument. Ponieważ jest to metoda statyczna, nie trzeba wcześniej tworzyć obiektu MessageBox, wystarczy wywołanie:

```
MessageBox.Show("tekst");
```

Rysunek 8.2.
*Wynik działania
metody
Show klasy
MessageBox*



Ć W I C Z E N I E

8.3 Wyświetlenie okna dialogowego z zadany komunikatem

Wyświetl okno z dowolnym komunikatem.

```
using System;
using System.Windows.Forms;

public class Aplikacja
{
    public static void Main()
    {
        MessageBox.Show("Przykładowy tekst");
    }
}
```

Okno z komunikatem wyświetlane za pomocą metody zaprezentowanej w ćwiczeniu 8.3 jest niezależne od okna aplikacji. W prezentowanym przykładzie okno aplikacji nawet nie powstało. Ten fakt można wykorzystać np. do wyświetlania krótkiego tekstu licencji przed uruchomieniem programu głównego.

Ć W I C Z E N I E

8.4 Okno dialogowe przed oknem głównym

Wyświetl okno z komunikatem, które pojawi się przed głównym oknem aplikacji.

```
using System;
using System.Windows.Forms;

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

```
        MessageBox.Show("To jest przykładowy tekst licencji.");  
        Application.Run(new Form());  
    }  
}
```

Zdarzenie ApplicationExit

Co zrobić w sytuacji, kiedy chce się wyświetlić komunikat w czasie, gdy aplikacja jest zamykana? W jaki sposób wychwycić ten moment? Z pomocą przychodzi zdarzenie `ApplicationExit` (więcej informacji o zdarzeniach podano w kolejnym rozdziale). Jeśli połączy się je odpowiednio z zawartą w programie metodą, będzie można wykonać dowolny kod przed wyjściem z programu. Schemat postępowania jest zatem następujący.

Tworzymy metodę o dowolnej nazwie, np. `OnExit`, i następującej deklaracji:

```
private void nazwa_metody(object sender, EventArgs ea)  
{  
    // tutaj kod metody  
}
```

Następnie w konstruktorze klasy pochodnej od `Form` umieszczamy instrukcję:

```
Application.ApplicationExit += new EventHandler(this.nazwa_metody);
```

Ć W I C Z E N I E

8.5 Obsługa zdarzenia `ApplicationExit`

Napisz program wyświetlający okno główne. Przy próbie zamknięcia aplikacji wyświetl okno z dowolnym komunikatem.

```
using System;  
using System.Windows.Forms;  
  
public class MainForm : Form  
{  
    public MainForm()  
    {  
        Application.ApplicationExit += new EventHandler(this.onExit);  
    }  
    private void onExit(object sender, EventArgs ea)
```

```
{
    MessageBox.Show("Aplikacja zostanie zamknięta!");
}
public static void Main()
{
    Application.Run(new MainForm());
}
}
```



9

Delegacje i zdarzenia

Czym są delegacje?

Przed przystąpieniem do budowy aplikacji okienkowych bardziej złożonych niż przedstawione w rozdziale 8. trzeba zapoznać się z tematem delegacji i zdarzeń. Wiadomości te będą niezbędne, aby można było reagować na zdarzenia generowane przez użytkownika, takie jak kliknięcie przycisku czy też wybranie pozycji z menu. Czym zatem są wspomniane delegacje? Można powiedzieć, że historycznie wywodzą się ze wskaźników do funkcji znanych jeszcze z C i C++, choć są konstrukcjami bardziej zaawansowanymi.

Rozważmy taki przykład: mamy dowolną, utworzoną przez nas klasę. Klasa ta będzie zawierała dwa pola o nazwach *x* i *y* oraz metodę *Show* wyświetlającą dane na ekranie. Realizacja takiego zadania z pewnością nie będzie dla nikogo problemem, gdyż wszystkie potrzebne wiadomości zostały podane w rozdziale 4. Napiszmy więc krótki fragment kodu.

Ć W I C Z E N I E

9.1 Prosta klasa wyjściowa

Napisz kod klasy zawierającej dwa pola typu `int` o nazwach `x` i `y` oraz metodę `Show` wyświetlającą dane na ekranie.

```
using System;

public class Kontener
{
    public int x;
    public int y;
    public Kontener(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public void Show()
    {
        Console.WriteLine("x = {0}", x);
        Console.WriteLine("y = {0}", y);
    }
}
```

Zgodnie z poleceniem w klasie zostały umieszczone dwa pola typu `int`, tj. `x` i `y`. Ze względu na użycie modyfikatora `public` dostęp do nich nie jest ograniczony. Pierwotny stan pól może być ustalony za pomocą dwuargumentowego konstruktora, natomiast ich wartości mogą być wyświetlone na konsoli przy użyciu metody `Show`.

Załóżmy teraz, że chcemy mieć możliwość decydowania o kształcie metody `Show` i o tym, co robi w zupełnie innym miejscu programu. Konkretniej mówiąc, klasa `Kontener` ma zawierać metodę `Show`, ale implementacja tej metody mogłaby być zmieniana w trakcie działania programu. Raz wypisywalibyśmy na ekranie tylko wartość `x`, innym razem tylko `y`, a czasem chcielibyśmy, aby na ekranie pojawiało się pełne zdanie `Wartość x wynosi ...`, a wartość `y` Jak to zrobić? Najprościej użyć właśnie delegacji.

Ć W I C Z E N I E

9.2 Tworzenie delegacji

Zmodyfikuj klasę z ćwiczenia 9.1 w taki sposób, aby wywoływanie funkcji Show odbywało się przez delegację.

```
public class Kontener
{
    public int x;
    public int y;
    public Kontener(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public delegate void ShowCallBack(Kontener k);
    public void Show(ShowCallBack k)
    {
        k(this);
    }
}
```

Zaprezentowany kod początkowo może wydawać się skomplikowany. Tak jednak wcale nie jest. Jak go interpretować? Po pierwsze, została zadeklarowana delegacja o nazwie ShowCallBack, której parametrem jest obiekt typu Kontener. Po drugie, parametrem metody Show jest obiekt delegacji. Po trzecie, metoda Show wywołuje delegację, przekazując jej jako parametr instancję obiektu, na rzecz którego jest wywoływana.

Ponieważ dla osób, które nie znają tematyki delegacji, wyjaśnienie ćwiczenia 9.2 wciąż może być zawile, najlepiej po prostu od razu sprawdzić, jak w praktyce użyć tego typu konstrukcji. To z pewnością się pozwoli wyjaśnić wszelkie wątpliwości.

Ć W I C Z E N I E

9.3 Proste użycie delegacji

Napisz program korzystający z klasy Kontener z ćwiczenia 9.2. Użyj w nim mechanizmu delegacji.

```
using System;

public class Program
{
    public static void Show(Kontener k)
```

```
{
    Console.WriteLine("x = {0}", k.x);
    Console.WriteLine("y = {0}", k.y);
}
public static void Main()
{
    Kontener kont = new Kontener(100, 200);
    Kontener.ShowCallback callBack =
        new Kontener.ShowCallback(Program.Show);
    kont.Show(callBack);
}
}
```

W tym ćwiczeniu powstała klasa Program zawierająca metodę Show przyjmującą jako argument obiekt klasy Kontener. We wnętrzu metody następuje odczytanie wartości pól x i y tego obiektu oraz wyświetlenie ich na ekranie. Wykonywanie kodu zaczyna się, jak wiadomo, od funkcji Main. Powstaje w niej nowy obiekt kont klasy Kontener:

```
Kontener kont = new Kontener(100, 200);
```

Następnie tworzymy nową delegację callBack powiązaną z metodą Show:

```
Kontener.ShowCallback callBack =
    new Kontener.ShowCallback(Program.Show);
```

Ostatnim krokiem jest wywołanie metody Show obiektu kont i przekazanie jej w postaci argumentu delegacji callBack. Tym samym o sposobie wyświetlenia wartości pól x i y obiektu kont będzie decydowała metoda Show z klasy Program.

Ć W I C Z E N I E

9.4 Wykorzystanie kilku delegacji

Napisz i przetestuj kilka delegacji dla metody Show klasy Kontener.

```
using System;

public class Program
{
    public static void Show1(Kontener k)
    {
        Console.WriteLine("x = {0}", k.x);
        Console.WriteLine("y = {0}\n", k.y);
    }
}
```

```
public static void Show2(Kontener k)
{
    Console.WriteLine("Wartość x wynosi {0}", k.x);
    Console.WriteLine("Wartość y wynosi {0}\n", k.y);
}
public static void Show3(Kontener k)
{
    Console.WriteLine("Obiekt zawiera następujące pola:");
    Console.WriteLine("Pole x o wartości {0}", k.x);
    Console.WriteLine("Pole y o wartości {0}\n", k.y);
}
public static void Main()
{
    Kontener kont= new Kontener(100, 200);
    Kontener.ShowCallBack callBack1 =
        new Kontener.ShowCallBack(Program.Show1);
    Kontener.ShowCallBack callBack2 =
        new Kontener.ShowCallBack(Program.Show2);
    Kontener.ShowCallBack callBack3 =
        new Kontener.ShowCallBack(Program.Show3);

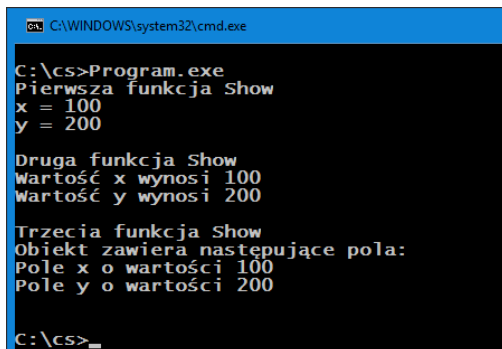
    Console.WriteLine("Pierwsza funkcja Show");
    kont.Show(callBack1);
    Console.WriteLine("Druga funkcja Show");
    kont.Show(callBack2);
    Console.WriteLine("Trzecia funkcja Show");
    kont.Show(callBack3);
}
}
```

Tym razem zdefiniowane zostały trzy delegacje (callBack1, callBack2, callBack3) i trzy funkcje Show (Show1, Show2, Show3), które w różny sposób wyświetlają dane z obiektu typu Kontener. Sposób postępowania jest tu taki sam jak w ćwiczeniu 9.3. W tym przypadku do obsługi wyświetlania danych z obiektu typu Kontener używane są po prostu trzy różne funkcje zdefiniowane w klasie Program. Najważniejsze jest jednak to, że wywołania metody Show pozwalają na wykonanie praktycznie dowolnego kodu, a nie wymagają przy tym żadnych zmian w klasie Kontener.

Efekt działania kodu z powyższego ćwiczenia widoczny jest na rysunku 9.1.



Rysunek 9.1.
Wynik działania
trzech różnych
delegacji dla
funkcji Show



```
C:\WINDOWS\system32\cmd.exe

C:\cs>Program.exe
Pierwsza funkcja Show
x = 100
y = 200

Druga funkcja Show
Wartość x wynosi 100
Wartość y wynosi 200

Trzecia funkcja Show
Obiekt zawiera następujące pola:
Pole x o wartości 100
Pole y o wartości 200

C:\cs>_
```

Jak obsługiwać zdarzenia?

Mechanizm zdarzeń (ang. *events*) oparty jest na delegacjach i pozwala, aby obiekt mógł informować o swoim stanie czy też właśnie o wystąpieniu pewnego zdarzenia. Najczęściej wykorzystuje się tę możliwość podczas tworzenia aplikacji z interfejsem graficznym. Właśnie przy użyciu zdarzeń można reagować na kliknięcie myszą czy wybranie pozycji z menu. Zanim jednak przejdziemy do obsługi zdarzeń w środowisku okienkowym (co zostanie pokazane w rozdziale 10.), zobaczmy, w jaki sposób te mechanizmy działają.

Do definiowania zdarzeń służy słowo kluczowe `event`. Jednak wcześniej należy utworzyć odpowiadającą zdarzeniu delegację, a zatem postępowanie jest dwuetapowe.

1. Definiujemy delegację w postaci:

```
public delegate typ_zwracany nazwa_delegacji(parametry);
```

2. Definiujemy jedno lub więcej zdarzeń w postaci:

```
public static event nazwa_delegacji nazwa_zdarzenia
```

Spróbujmy wykonać praktyczny przykład. Założmy, że mamy klasę `Kontener` w postaci:

```
public class Kontener
{
    private int x;
    public Kontener()
    {
        SetX(0);
    }
}
```

```
public void SetX(int x)
{
    this.x = x;
}

public int GetX()
{
    return x;
}
}
```

Pole `x` jest prywatne, nie ma do niego bezpośredniego dostępu. Do ustawiania służyć będzie zatem metoda `setX`. Pobieranie danych zapewnia metoda `getX`. Wyobraźmy sobie teraz, że ta klasa „bardzo nie lubi”, kiedy wartość `x` jest mniejsza od zera. Jeżeli więc wykryje, że polu `x` chcemy przypisać liczbę ujemną, wyśle, poprzez zdarzenie, stosowną informację. Czas zatem na utworzenie odpowiedniej delegacji i zdarzenia.

Ć W I C Z E N I E

9.5 Wiązanie delegacji i zdarzeń

Zmodyfikuj klasę `Kontener` w taki sposób, aby po przypisaniu wartości ujemnej polu `x` generowała odpowiednie zdarzenie.

```
public class Kontener
{
    private int x;
    public Kontener()
    {
        SetX(0);
    }
    public delegate void EventHandler();
    public static event EventHandler XJestUjemne;
    public void SetX(int x)
    {
        this.x = x;
        if(x < 0)
        {
            if(XJestUjemne != null)
                XJestUjemne();
        }
    }
    public int GetX()
    {
        return x;
    }
}
```

Zgodnie z podanymi wyżej zasadami została tu zdefiniowana zarówno delegacja:

```
public delegate void EventHandler();
```

jak i powiązane z nią zdarzenie:

```
public static event EventHandler XJestUjemne;
```

W metodzie setX następuje sprawdzanie, czy przekazany parametr ma wartość mniejszą od zera. Jeśli tak, wywoływane jest zdarzenie:

```
XJestUjemne();
```

Niezbędne jest jednak wcześniejsze sprawdzenie, czy do tego zdarzenia została przypisana jakaś procedura obsługi. Służy do tego instrukcja:

```
if(XJestUjemne != null)
```

Ujmując rzecz nieco inaczej: jeżeli istnieje powiązana ze zdarzeniem delegacja, to za pomocą tej delegacji można wywołać odpowiadającą jej metodę. Zostanie to pokazane w kolejnym ćwiczeniu.

Sprawdźmy teraz, w jaki sposób przypisać procedurę obsługi zdarzenia do obiektu. Należy skorzystać z operatora += w postaci:

```
NazwaKlasy.NazwaZdarzenia += new  
    NazwaKlasy.NazwaDelegacji(ProceduraObslugi);
```

Ć W I C Z E N I E

9.6 Testowanie zdarzeń i delegacji

Napisz procedurę obsługi zdarzenia dla klasy Kontener. Spróbuj przypisać polu x wartość ujemną i sprawdź, czy program działa zgodnie z założeniami.

```
using System;  
  
public class Program  
{  
    public static void OnUjemne()  
    {  
        Console.WriteLine("X jest ujemne!");  
    }  
    public static void Main()  
    {  
        Kontener k = new Kontener();  
        Kontener.XJestUjemne += new Kontener.EventHandler(OnUjemne);  
    }  
}
```



```
k.SetX(-1);  
k.SetX(8);  
k.SetX(-25);  
}  
}
```

Procedurą obsługi zdarzenia `XJestUjemne`, powstającego, gdy wartość pola `x` obiektu typu `Kontener` spadnie poniżej zera, jest tu metoda `OnUjemne` z klasy `Program`. Wiążemy ją ze zdarzeniem za pomocą delegacji w wierszu:

```
Kontener.XJestUjemne += new Kontener.EventHandler(OnUjemne);
```

Dzięki temu po wykonaniu instrukcji:

```
k.SetX(-1);
```

oraz:

```
k.SetX(-25);
```

na ekranie pojawi się komunikat zdefiniowany w metodzie `OnUjemne`. Natomiast wywołanie:

```
k.SetX(8);
```

nie spowoduje pojawienia się komunikatu, bo w tym przypadku wartość przekazana metodzie `setX` jest dodatnia.

Co jednak zrobić w sytuacji, kiedy chcemy w procedurze obsługi mieć dostęp do obiektu, który zdarzenie wygenerował? Nie ma najmniejszego problemu. Wystarczy odpowiednio skonstruować delegację. A gdyby trzeba było jednemu zdarzeniu przypisać kilka procedur obsługi? To też jest możliwe. Wystarczy kilkakrotnie użyć operatora `+=`.

Ć W I C Z E N I E

9.7 Dostęp do obiektu generującego zdarzenie

Zmodyfikuj utworzone w poprzednich ćwiczeniach klasy `Program` oraz `Kontener` w taki sposób, aby procedura obsługi zdarzenia otrzymywała jako parametr obiekt generujący zdarzenie. Napisz dwie różne procedury obsługi zdarzenia.

```
using System;  
  
public class Kontener  
{
```

```
private int x;
public Kontener()
{
    SetX(0);
}
public delegate void EventHandler(Kontener k);
public static event EventHandler XJestUjemne;
public void SetX(int x)
{
    this.x = x;
    if(x < 0)
    {
        if(XJestUjemne != null)
            XJestUjemne(this);
    }
}
public int GetX()
{
    return x;
}
}

public class Program
{
    public static void OnUjemne1(Kontener k)
    {
        Console.WriteLine("X jest równe {0}.", k.GetX());
    }
    public static void OnUjemne2(Kontener k)
    {
        Console.WriteLine("Druga procedura obsługi zdarzenia!");
    }
    public static void Main()
    {
        Kontener kont = new Kontener();
        Kontener.XJestUjemne += new Kontener.EventHandler(OnUjemne1);
        Kontener.XJestUjemne += new Kontener.EventHandler(OnUjemne2);
        kont.SetX(-1);
    }
}
```

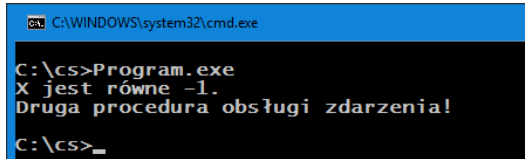
W klasie Kontener zmienił się typ delegacji. Obecnie przyjmuje ona jeden argument typu Kontener i nie zwraca wyniku. Dlatego też metody będące procedurami obsługi zdarzenia XJestUjemne będą musiały mieć deklaracje zgodne z powyższym opisem. Tak właśnie jest z metodami OnUjemne1 i OnUjemne2 w klasie Program. Wiązanie metod ze zdarzeniem odbywa się za pomocą instrukcji:

```
Kontener.XJestUjemne += new Kontener.EventHandler(OnUjemne1);
Kontener.XJestUjemne += new Kontener.EventHandler(OnUjemne2);
```

Tym samym zdarzenie otrzymuje dwie procedury obsługi i po przypisaniu ujemnej wartości polu `x` obie zostaną wykonane. Zatem po uruchomieniu programu na ekranie pojawią się komunikaty widoczne na rysunku 9.2.

Rysunek 9.2.

*Wykonanie
dwóch różnych
metod
w odpowiedzi
na zdarzenie*



```
C:\WINDOWS\system32\cmd.exe

C:\cs>Program.exe
X jest równe -1.
Druga procedura obsługi zdarzenia!
C:\cs>_
```


10

Komponenty

Etykiety (Label)

Klasa `Label` służy do tworzenia etykiet, czyli kontroltek używanych do wyświetlania krótkich napisów. Tekst etykiety może być zmieniany przez aplikację, użytkownik nie ma natomiast możliwości bezpośredniej jego edycji. Aby skorzystać z etykiety, należy utworzyć obiekt klasy `Label` oraz dodać go do kolekcji `Controls` okna, w którym ta etykieta ma się znajdować. W tym celu stosuje się instrukcję `Controls.Add(nazwa_etykiety)`. Wygląd oraz zachowanie etykiety można modyfikować, korzystając z jej właściwości, które zebrane są w tabeli 10.1.

Tabela 10.1. Wybrane właściwości klasy *Label*

Typ	Nazwa właściwości	Znaczenie
bool	AutoSize	Określa, czy etykieta ma automatycznie dopasowywać swój rozmiar do zawartego w niej tekstu.
Color	BackColor	Określa kolor tła etykiety.
BorderStyle	BorderStyle	Określa styl ramki otaczającej etykietę.
Bounds	Bounds	Określa rozmiar oraz położenie etykiety.
Cursor	Cursor	Określa rodzaj kursora wyświetlanego, kiedy wskaźnik myszy znajdzie się nad etykietą.
Font	Font	Określa rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się na etykiecie.
Color	ForeColor	Określa kolor tekstu etykiety.
int	Height	Określa wysokość etykiety.
Image	Image	Obraz wyświetlany na etykiecie.
int	Left	Określa położenie lewego górnego rogu w poziomie, w pikselach.
Point	Location	Określa współrzędne lewego górnego rogu etykiety.
string	Name	Nazwa etykiety.
Control	Parent	Referencja do obiektu zawierającego etykietę (nadrzędnego).
Size	Size	Określa wysokość i szerokość etykiety.
string	Text	Określa tekst wyświetlany na etykiecie.
Content ↪Alignment	TextAlign	Określa położenie tekstu na etykiecie.
int	Top	Określa położenie lewego górnego rogu w pionie, w pikselach.
bool	Visible	Określa, czy etykieta ma być widoczna.
int	Width	Określa rozmiar etykiety w poziomie.

Ć W I C Z E N I E

10.1 Dodanie etykiety do okna aplikacji

Dodaj do okna aplikacji komponent `Label` z dowolnym tekstem. Etykieta powinna znaleźć się w centrum okna aplikacji.

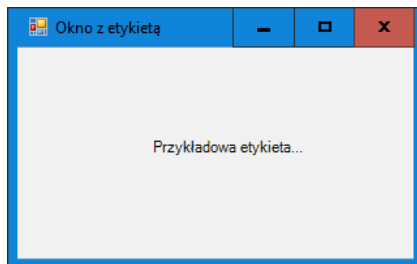
```
using System;
using System.Windows.Forms;

public class MainForm : Form
{
    Label label = new Label();
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;
        this.Text = "Okno z etykietą";
        label.Text = "Przykładowa etykieta...";
        label.AutoSize = true;
        label.Left = (this.ClientSize.Width - label.Width) / 2;
        label.Top = (this.ClientSize.Height - label.Height) / 2;
        this.Controls.Add(label);
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Wynikiem działania kodu z powyższego ćwiczenia jest okno widoczne na rysunku 10.1. Jak widać, tekst faktycznie znajduje się w centrum okna. Takie umiejscowienie etykiety uzyskujemy poprzez modyfikację właściwości `Top` oraz `Left`. Aby uzyskać odpowiednie wartości, wykonywane są proste działania matematyczne:

Współrzędna $X = (\text{długość okna} - \text{długość etykiety}) / 2$
Współrzędna $Y = (\text{wysokość okna} - \text{wysokość etykiety}) / 2$

Rysunek 10.1.
*Etykieta tekstowa
umiejscowiona
w centrum okna*



Oczywiście działania te należy wykonać po przypisaniu etykiety tekstu, inaczej nie będą uwzględniały jej prawidłowej wielkości. Do uzyskania szerokości i wysokości wewnętrznego obszaru okna aplikacji należy wykorzystać właściwości `ClientWidth` oraz `ClientHeight`. Podają one rozmiar okna po odliczeniu okalającej ramki oraz paska tytułowego i ewentualnego menu, czyli po prostu wielkość okna, które mamy do dyspozycji i w którym możemy umieszczać inne obiekty.

Tekst wyświetlany na etykiecie może być wyświetlany dowolną czcionką zainstalowaną w systemie. Wystarczy zmodyfikować właściwość `Font` obiektu `Label`. Aby jednak taką modyfikację wykonać, trzeba najpierw utworzyć obiekt klasy `Font` (dostępnej w przestrzeni nazw `System.Drawing`). Klasa ta zawiera kilka konstruktorów, można użyć konstruktora o postaci:

```
public Font(FontFamily family, float emSize, FontStyle style);
```

Parametr `emSize` określa wielkość czcionki, natomiast `FontStyle` to typ wyliczeniowy, w którym zdefiniowane są wartości przedstawione w tabeli 10.2.

Tabela 10.2. Wartości typu `FontStyle`

Nazwa	Znaczenie
Bold	tekst pogrubiony
Italic	tekst pochylony
Regular	tekst zwyczajny
Strikeout	tekst przekreślony
Underline	tekst podkreślony

`FontFamily` jest to klasa reprezentująca rodzinę czcionek. Udostępnia ona również statyczną właściwość `Families`, która zwraca tablicę z listą wszystkich fontów dostępnych w systemie. To, jak uzyskać taką listę, zostało pokazane w ćwiczeniu 10.2.

ĆWICZENIE

10.2 Lista dostępnych czcionek

Wypisz na ekranie listę wszystkich dostępnych czcionek.

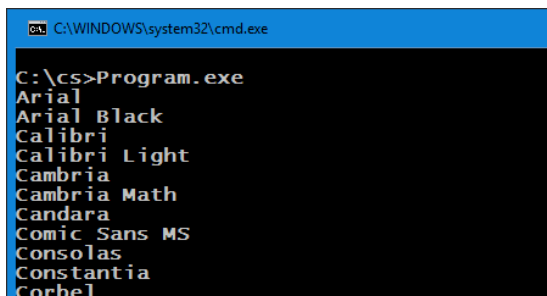
```
using System;
using System.Drawing;

public class Program
{
    public static void Main()
    {
        foreach(FontFamily font in FontFamily.Families)
        {
            Console.WriteLine(font.Name);
        }
    }
}
```

Przykładowa lista czcionek uzyskana za pomocą tego programu widoczna jest na rysunku 10.2. Kod jest bardzo prosty — została użyta zwyczajna pętla typu `foreach` (rozdział „Tablice”), która przebiega przez wszystkie elementy tablicy `FontFamily.Families`. W celu uzyskania nazwy danego fontu odwołujemy się do właściwości `Name` klasy obiektu podstawionego pod zmienną `font` (jej typem jest `Font`). Należy zwrócić uwagę na znajdującą się na początku kodu dyrektywę `using System.Drawing`, dzięki której można korzystać z przestrzeni nazw `System.Drawing`, gdzie zdefiniowane są m.in. klasy `Font` i `FontFamily`.

Rysunek 10.2.

*Lista czcionek
uzyskana przy
użyciu programu
z ćwiczenia 10.2*



Skoro wiadomo już tak dużo na temat fontów, spróbujmy wyświetlić tekst na etykiecie wybraną czcionką, np. typu Courier. W większości systemów jest ona zainstalowana standardowo. Skorzystamy z przed-

stawionego wcześniej konstruktora klasy `Font`, nie będziemy jednak tworzyć bezpośrednio obiektu klasy `FontFamily`, lecz pozwolimy, aby system zrobił to za nas. Zamiast więc pisać:

```
new Font(new FontFamily("Courier"), 20, FontStyle.Bold);
```

skorzystamy z konstrukcji:

```
new Font("Courier", 20, FontStyle.Bold);
```

Ć W I C Z E N I E

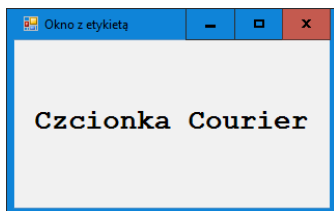
10.3 Etykieta wykorzystująca wybrany krój czcionki

Dodaj do okna aplikacji komponent `Label` z dowolnym tekstem napisanym czcionką `Courier` o wielkości 20 punktów (rysunek 10.3).

```
using System.Windows.Forms;
using System.Drawing;

public class MainForm : Form
{
    Label label = new Label();
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;
        this.Text = "Okno z etykietą";
        label.Font = new Font("Courier", 20, FontStyle.Bold);
        label.Text = "Czcionka Courier";
        label.AutoSize = true;
        this.Controls.Add(label);
        label.Left = (this.ClientSize.Width - label.Width) / 2;
        label.Top = (this.ClientSize.Height - label.Height) / 2;
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Rysunek 10.3.
*Etykieta, na
której użyto
czcionki Courier
o wielkości
20 punktów*



Przyciski (Button)

Klasa `Button` pozwala na tworzenie standardowych przycisków z dowolnymi napisami. Aby z niej skorzystać, należy utworzyć nowy obiekt tej klasy, ustalić jego położenie oraz napisać procedurę obsługi zdarzeń (wykonywaną po kliknięciu przycisku). Przycisk umieszczamy w oknie, wykorzystując właściwości `Top` oraz `Left`, podobnie jak w przypadku innych komponentów (na przykład etykiet). Przydatne właściwości klasy `Button` przedstawione są w tabeli 10.3.

Tabela 10.3. Wybrane właściwości klasy `Button`

Typ	Nazwa właściwości	Znaczenie
Color	<code>BackColor</code>	Określa kolor tła przycisku.
Bounds	<code>Bounds</code>	Określa rozmiar oraz położenie przycisku.
Cursor	<code>Cursor</code>	Określa rodzaj kursora wyświetlanego, kiedy wskaźnik myszy znajdzie się nad przyciskiem.
FlatStyle	<code>FlatStyle</code>	Modyfikuje styl przycisku.
Font	<code>Font</code>	Określa rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się na przycisku.
Color	<code>ForeColor</code>	Określa kolor tekstu przycisku.
int	<code>Height</code>	Określa wysokość przycisku.
Image	<code>Image</code>	Obraz wyświetlany na przycisku.
int	<code>Left</code>	Określa położenie lewego górnego rogu w poziomie, w pikselach.
Point	<code>Location</code>	Określa współrzędne lewego górnego rogu przycisku.
string	<code>Name</code>	Nazwa przycisku.
Control	<code>Parent</code>	Referencja do obiektu zawierającego przycisk (obektu nadrzędnego).
Size	<code>Size</code>	Określa wysokość i szerokość przycisku.
string	<code>Text</code>	Tekst wyświetlany na przycisku.
Content ↪Alignment	<code>TextAlign</code>	Określa położenie tekstu na przycisku.
int	<code>Top</code>	Określa położenie lewego górnego rogu w pionie, w pikselach.
bool	<code>Visible</code>	Określa, czy przycisk ma być widoczny.
int	<code>Width</code>	Określa rozmiar przycisku w poziomie, w pikselach.

Ć W I C Z E N I E

10.4 Umieszczenie przycisku w oknie aplikacji

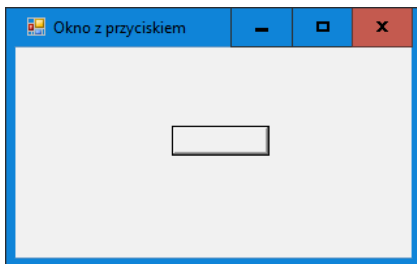
Utwórz okno, w którym będzie znajdował się przycisk (rysunek 10.4).

```
using System;
using System.Windows.Forms;

public class MainForm : Form
{
    Button button = new Button();
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;
        this.Text = "Okno z przyciskiem";
        button.Top = 60;
        button.Left = 120;
        this.Controls.Add(button);
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Rysunek 10.4.

*Przycisk dodany
w głównym oknie
aplikacji*



Przycisk utworzony w ćwiczeniu 10.4 nie reaguje na kliknięcia, nie ma też na nim żadnego napisu. Konieczne zatem będą kolejne modyfikacje kodu programu. Na przypisanie tekstu do przycisku pozwala właściwość `Text`. Z kolei reakcje na kliknięcia zapewni dodanie procedury obsługi zdarzeń do właściwości `Click`:

```
button.Click += eh;
```

gdzie `eh` to obiekt klasy `EventHandler` (rozdział 9).

Ć W I C Z E N I E

10.5 Obsługa kliknięć przycisku

Zmodyfikuj kod z ćwiczenia 10.4 w taki sposób, aby po kliknięciu przycisku następowało zamknięcie aplikacji. Na przycisku powinien też pojawić się napis.

```
using System;
using System.Windows.Forms;

public class MainForm : Form
{
    Button button = new Button();
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;
        this.Text = "Okno z przyciskiem";
        button.Top = 60;
        button.Left = 120;
        button.Text = "Kliknij mnie!";

        EventHandler eh = new EventHandler(this.CloseClicked);
        button.Click += eh;
        this.Controls.Add(button);
    }
    public void CloseClicked(Object sender, EventArgs e)
    {
        this.Close();
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Tekst na przycisku został zdefiniowany za pomocą właściwości `Text`. Powstał także obiekt typu `EventHandler` powiązany z metodą `CloseClicked`. Metoda ta ma dwa argumenty. Pierwszy (`sender`) wskazuje obiekt będący źródłem zdarzenia (w tym przypadku jest to przycisk `button`), drugi (`e`) zawiera informacje powiązane ze zdarzeniem (np. współrzędne kliknięcia). Te dane nie są jednak używane. Jedynym działaniem metody `CloseClicked` jest zamknięcie okna aplikacji (wywołanie metody `Close`). Procedura obsługi zdarzenia jest wiązana ze zdarzeniem (właściwość `Click` przycisku) w instrukcji:

```
button.Click += eh;
```

Pola tekstowe (TextBox)

Klasa `TextBox` służy do tworzenia pól tekstowych umożliwiających wprowadzanie przez użytkownika ciągów znaków. W zależności od ustawień pole tekstowe może być jedno- lub wielowierszowe. Zmiany można wykonać poprzez odpowiednie ustawienie właściwości `Multiline`. Wybrane właściwości klasy przedstawione są w tabeli 10.4.

Tabela 10.4. Wybrane właściwości klasy `TextBox`

Typ	Nazwa właściwości	Znaczenie
bool	<code>AutoSize</code>	Określa, czy pole tekstowe ma automatycznie dopasowywać swój rozmiar do zawartego w nim tekstu.
Color	<code>BackColor</code>	Określa kolor tła pola tekstowego.
Image	<code>BackgroundImage</code>	Obraz znajdujący się w tle okna tekstowego.
BorderStyle	<code>BorderStyle</code>	Określa styl ramki otaczającej pole tekstowe.
Bounds	<code>Bounds</code>	Określa rozmiar oraz położenie pola tekstowego.
Cursor	<code>Cursor</code>	Rodzaj kursora wyświetlanego, kiedy wskaźnik myszy znajdzie się nad polem tekstowym.
Font	<code>Font</code>	Określa rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się w polu.
Color	<code>ForeColor</code>	Określa kolor tekstu pola tekstowego.
int	<code>Height</code>	Określa wysokość pola tekstowego.
int	<code>Left</code>	Określa położenie lewego górnego rogu w poziomie, w pikselach.
string[]	<code>Lines</code>	Tablica zawierająca poszczególne linie tekstu zawarte w polu tekstowym.
Point	<code>Location</code>	Określa współrzędne lewego górnego rogu pola tekstowego.
int	<code>MaxLength</code>	Maksymalna liczba znaków, które można wprowadzić w polu tekstowym.
bool	<code>Modified</code>	Określa, czy zawartość pola tekstowego była modyfikowana.

Tabela 10.4. Wybrane właściwości klasy `TextBox` — ciąg dalszy

Typ	Nazwa właściwości	Znaczenie
Bool	Multiline	Określa, czy pole tekstowe ma zawierać jedną, czy wiele linii tekstu.
String	Name	Nazwa pola tekstowego.
Control	Parent	Referencja do obiektu zawierającego pole tekstowe (obiektu nadrzędnego).
char	PasswordChar	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> .
bool	ReadOnly	Określa, czy pole tekstowe ma być ustawione w trybie tylko do odczytu.
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	Określa rozmiar pola tekstowego.
string	Text	Tekst wyświetlany w polu tekstowym.
Content ➔Alignment	TextAlign	Określa położenie tekstu w polu tekstowym.
int	Top	Określa położenie lewego górnego rogu w pionie, w pikselach.
bool	Visible	Określa, czy pole tekstowe ma być widoczne.
int	Width	Określa rozmiar pola tekstowego w poziomie.
bool	WordWrap	Określa, czy słowa mają być automatycznie przenoszone do nowej linii, kiedy nie mieszczą się w bieżącej; aby można było zastosować tę funkcję, właściwość <code>Multiline</code> musi być ustawiona na <code>true</code> .

Ć W I C Z E N I E

10.6 Obsługa pola tekstowego

Umieść w oknie aplikacji pole tekstowe i przycisk. Po kliknięciu przycisku wyświetl wprowadzony w polu tekst w oknie dialogowym.

```
using System;
using System.Windows.Forms;

public class MainForm : Form
{
    Button button = new Button();
    TextBox textBox = new TextBox();
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;
        this.Text = "Aplikacja";

        button.Top = 120;
        button.Left = (this.ClientSize.Width - button.Width) / 2;
        button.Text = "Kliknij mnie!";

        textBox.Top = 60;
        textBox.Left = (this.ClientSize.Width - textBox.Width) / 2;

        EventHandler eh = new EventHandler(this.ButtonClicked);
        button.Click += eh;
        this.Controls.Add(button);
        this.Controls.Add(textBox);
    }
    public void ButtonClicked(Object sender, EventArgs e)
    {
        MessageBox.Show(textBox.Text);
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

W oknie aplikacji zostały umieszczone przycisk (button) oraz pole tekstowe (textBox). Pozycja tych elementów w pionie ustalana jest arbitralnie, natomiast pozycja w poziomie jest wyliczana dynamicznie, tak aby oba elementy były wyśrodkowane. Przycisk otrzymał procedurę obsługi zdarzenia w postaci metody ButtonClicked (odbywa się to na takich samych zasadach jak w ćwiczeniu 10.5). W metodzie ButtonClicked jest używana metoda Show klasy MessageBox wyświetlająca komunikat (zawartość pola Text pola tekstowego textBox).



Ć W I C Z E N I E

10.7 Inne użycie pola tekstowego

Umieść w oknie aplikacji pole tekstowe, etykietę tekstową i przycisk tak jak na rysunku 10.5. Po kliknięciu przycisku wyświetl na etykiecie ciąg znaków wprowadzony przez użytkownika w polu tekstowym.

```
using System;
using System.Drawing;
using System.Windows.Forms;

public class MainForm : Form
{
    Button button = new Button();
    Label label = new Label();
    TextBox textBox = new TextBox();
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;
        this.Text = "Aplikacja";

        button.Top = 60;
        button.Left = (this.ClientSize.Width - button.Width) / 2;
        button.Text = "Kliknij mnie!";

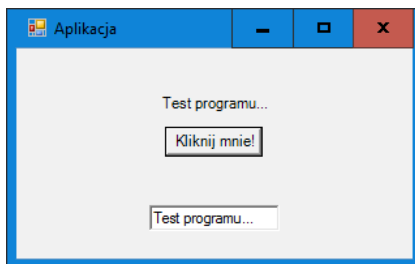
        label.Top = 30;
        label.Text = "Etykieta";
        label.TextAlign = ContentAlignment.MiddleCenter;
        label.Left = (this.ClientSize.Width - label.Width) / 2;

        textBox.Top = 120;
        textBox.Left = (this.ClientSize.Width - textBox.Width) / 2;

        EventHandler eh = new EventHandler(this.ButtonClicked);
        button.Click += eh;
        this.Controls.Add(button);
        this.Controls.Add(label);
        this.Controls.Add(textBox);
    }
    public void ButtonClicked(Object sender, EventArgs e)
    {
        label.Text = textBox.Text;
        label.Left = (this.ClientSize.Width - label.Width) / 2;
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Rysunek 10.5.

Wynik działania kodu
z ćwiczenia 10.7



Ogólna budowa aplikacji jest podobna do prezentowanej w ćwiczeniu 10.6. Dodatkowo pojawiła się jedynie etykieta reprezentowana przez pole `label` klasy `Label`. Tekst etykiety jest pozycjonowany (w tym przypadku centrowany) dzięki zmianie właściwości `TextAlign`. Przypisywaną wartością jest `MiddleCenter` (oznacza wyśrodkowanie w pionie i poziomie) pochodząca z klasy `ContentAlignment`. Procedurą obsługi zdarzenia `Click` przycisku `button` jest metoda `ButtonClicked`. W tej metodzie tekst pobrany z pola tekstowego (właściwość `Text` obiektu `textBox`) jest przypisywany etykiecie (właściwość `Text` obiektu `label`). Dodatkowo zmieniana jest pozycja etykiety, tak aby zawsze znajdowała się w środku okna (ze względu na użycie właściwości `TextAlign` można by to rozwiązać inaczej — etykieta mogłaby mieć szerokość okna; jest to jednak ćwiczenie do samodzielnego wykonania).

Pola wyboru (CheckBox, RadioButton)

Pola wyboru to kolejne znane elementy interfejsu Windows, które można z łatwością stosować, korzystając z dostępnych w .NET klas `CheckBox` i `RadioButton`. Wybrane właściwości tych klas przedstawione są w tabeli 10.5.

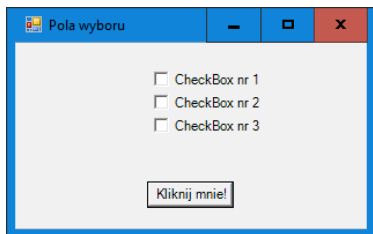
Aby sprawdzić, czy pole wyboru jest zaznaczone, należy odwołać się do właściwości `Checked`. Ustawiona na `true` sygnalizuje, że pole jest zaznaczone, na `false`, że zaznaczone nie jest. Właściwości tej można również przypisywać wartości i samemu decydować o stanie pola.

Tabela 10.5. Wybrane właściwości klas *CheckBox* i *RadioButton*

Typ	Nazwa właściwości	Znaczenie
bool	AutoCheck	Określa, czy pole ma być automatycznie zaznaczane lub zaznaczenie ma być usuwane, kiedy użytkownik kliknie je myszą.
Color	BackColor	Określa kolor tła pola.
Bounds	Bounds	Określa rozmiar oraz położenie pola.
bool	Checked	Pozwala stwierdzić, czy pole jest zaznaczone.
CheckState	CheckState	Określa sposób zaznaczania pola.
Cursor	Cursor	Rodzaj kursora wyświetlanego, kiedy wskaźnik myszy znajdzie się nad polem.
FlatStyle	FlatStyle	Określa styl, w jakim pole będzie wyświetlane.
Font	Font	Określa rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się przy polu.
Color	ForeColor	Określa kolor tekstu pola.
int	Height	Określa wysokość pola.
int	Left	Określa położenie lewego górnego rogu w poziomie, w pikselach.
Point	Location	Określa współrzędne lewego górnego rogu pola.
string	Name	Nazwa pola.
Control	Parent	Referencja do obiektu zawierającego pole.
Size	Size	Określa wysokość i szerokość pola.
string	Text	Tekst wyświetlany przy polu.
Content ↩Alignment	TextAlign	Określa położenie tekstu znajdującego się przy polu.
bool	ThreeState	Określa, czy pole ma być dwu-, czy trójstanowe.
int	Top	Określa położenie pola w pionie, w pikselach.
bool	Visible	Określa, czy pole ma być widoczne.
int	Width	Określa rozmiar pola w poziomie.

Wykonajmy proste ćwiczenie, w którym wyświetlimy na ekranie okno zawierające trzy elementy typu CheckBox oraz przycisk (rysunek 10.6). Po kliknięciu przycisku pojawi się informacja, które pola zostały zaznaczone.

Rysunek 10.6.
*Okno aplikacji
zawierającej
trzy pola wyboru
typu CheckBox*



Ć W I C Z E N I E

10.8 Obsługa pól wyboru typu CheckBox

Umieść w oknie trzy obiekty typu CheckBox oraz przycisk. Po kliknięciu przycisku wyświetl informację, które opcje zostały zaznaczone.

```
using System;
using System.Drawing;
using System.Windows.Forms;

public class MainForm : Form
{
    Button button = new Button();
    CheckBox chb1, chb2, chb3;
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;
        this.Text = "Pola wyboru";

        button.Top = 120;
        button.Left = (this.ClientSize.Width - button.Width) / 2;
        button.Text = "Kliknij mnie!";

        chb1 = new CheckBox();
        chb1.Left = 120;
        chb1.Top = 20;
        chb1.Text = "CheckBox nr 1";

        chb2 = new CheckBox();
        chb2.Left = 120;
        chb2.Top = 40;
        chb2.Text = "CheckBox nr 2";
```

```
chb3 = new CheckBox();
chb3.Left = 120;
chb3.Top = 60;
chb3.Text = "CheckBox nr 3";

EventHandler eh = new EventHandler(this.ButtonClicked);
button.Click += eh;
this.Controls.Add(button);
this.Controls.Add(chb1);
this.Controls.Add(chb2);
this.Controls.Add(chb3);
}
public void ButtonClicked(Object sender, EventArgs e)
{
    string s1 = "", s2 = "", s3 = "";
    if(chb1.Checked)
    {
        s1 = " 1 ";
    }
    if(chb2.Checked)
    {
        s2 = " 2 ";
    }
    if(chb3.Checked)
    {
        s3 = " 3 ";
    }
    MessageBox.Show("Zostały zaznaczone opcje: " + s1 + s2 + s3);
}
public static void Main()
{
    Application.Run(new MainForm());
}
}
```

Zasada działania kodu jest podobna do zasad działania wcześniejszych przykładowych kodów. Pola wyboru (chb1, chb2, chb3) tworzone są za pomocą konstruktora klasy `CheckBox`. Każde z nich otrzymuje przypisany mu tekst, a także jest pozycjonowane przy użyciu właściwości `Left` oraz `Top`. Kliknięcie przycisku będzie powodowało wywołanie metody `ButtonClicked`. W jej wnętrzu badany jest stan właściwości `Checked` obiektów `chb1`, `chb2` i `chb3`. Jeśli któraś z tych właściwości ma wartość `true`, zmieniany jest stan odpowiedniej zmiennej pomocniczej: `s1`, `s2` lub `s3`. Na koniec za pomocą metody `Show` wyświetlany jest komunikat informacyjny złożony ze stałego tekstu połączonego z ciągami zapisanymi w wymienionych zmiennych.



Druga z omawianych klas, `RadioButton`, ma działanie podobne do klasy `CheckBox`, jednak wyświetlane pola nie są prostokątne, ale okrągłe. Dodatkową różnicą jest to, że omawiane pola są polami wzajemnie wykluczającymi się, czyli w jednej grupie może być zaznaczone tylko jedno z nich.

Ć W I C Z E N I E**10.9 Obsługa pól wyboru typu radio**

Umieść w oknie trzy obiekty klasy `RadioButton` oraz przycisk. Po kliknięciu przycisku wyświetl informację, która z opcji została zaznaczona.

```
using System;
using System.Drawing;
using System.Windows.Forms;

public class MainForm : Form
{
    Button button = new Button();
    RadioButton rb1, rb2, rb3;
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;
        this.Text = "Pola wyboru";

        button.Top = 120;
        button.Left = (this.ClientSize.Width - button.Width) / 2;
        button.Text = "Kliknij mnie!";

        rb1 = new RadioButton();
        rb1.Left = 120;
        rb1.Top = 20;
        rb1.Text = "Opcja nr 1";

        rb2 = new RadioButton();
        rb2.Left = 120;
        rb2.Top = 40;
        rb2.Text = "Opcja nr 2";

        rb3 = new RadioButton();
        rb3.Left = 120;
        rb3.Top = 60;
        rb3.Text = "Opcja nr 3";

        EventHandler eh = new EventHandler(this.ButtonClicked);
        button.Click += eh;
        this.Controls.Add(button);
    }
}
```

```
this.Controls.Add(rb1);
this.Controls.Add(rb2);
this.Controls.Add(rb3);
}
public void ButtonClicked(Object sender, EventArgs e)
{
    string s = "Zaznaczona została opcja ";
    if(rb1.Checked)
    {
        s += " 1.";
    }
    else if(rb2.Checked)
    {
        s += " 2.";
    }
    else if(rb3.Checked)
    {
        s += " 3.";
    }
    else
    {
        s = "Nie została zaznaczona żadna z opcji.";
    }
    MessageBox.Show(s);
}
public static void Main()
{
    Application.Run(new MainForm());
}
}
```

Zasada działania programu jest identyczna z zasadą z ćwiczenia 10.8. Różnica sprowadza się do zastosowania klasy `RadioButton` zamiast `CheckBox`. Dzięki temu powstaną pola typu `radio`, z których w jednym czasie aktywne (zaznaczone) będzie mogło być tylko jedno. Występująca w ćwiczeniu 10.8 seria instrukcji `if` została także zmieniona na jedną złożoną instrukcję `if..else`, zmniejszona została też liczba zmiennych pomocniczych — obecnie wystarcza tylko jedna (`s`). Komunikat otrzymuje też inne brzmienie, gdy żadna z opcji nie została zaznaczona (ostatni blok `else`).



Listy rozwijane (ComboBox)

Listy rozwijane (rozwijalne) można tworzyć przy użyciu klasy `ComboBox`. Udostępniane przez nią właściwości przedstawiono w tabeli 10.6. Najważniejsza z nich jest właściwość `Items`, jako że zawiera wszystkie elementy znajdujące się na liście. Właściwość ta jest w rzeczywistości kolekcją elementów typu `object`. Dodawanie elementów można zrealizować, stosując konstrukcję:

```
ComboBox.Items.Add("element");
```

Natomiast ich usunięcie spowodujemy, wykorzystując:

```
ComboBox.Items.Remove("element");
```

Tabela 10.6. Wybrane właściwości klasy `ComboBox`

Typ	Nazwa właściwości	Znaczenie
Color	<code>BackColor</code>	Określa kolor tła listy.
Bounds	<code>Bounds</code>	Określa rozmiar oraz położenie listy.
Cursor	<code>Cursor</code>	Określa rodzaj kursora wyświetlanego, kiedy wskaźnik myszy znajdzie się nad listą.
Font	<code>Font</code>	Określa rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się na liście.
Color	<code>ForeColor</code>	Określa kolor tekstu.
int	<code>Height</code>	Określa wysokość listy.
int	<code>ItemHeight</code>	Określa wysokość pojedynczego elementu listy.
Object ↳Collection	<code>Items</code>	Zbiór elementów listy.
int	<code>Left</code>	Określa położenie lewego górnego rogu w poziomie, w pikselach.
Point	<code>Location</code>	Określa współrzędne lewego górnego rogu listy.
int	<code>MaxDropDownItems</code>	Maksymalna liczba elementów, które będą wyświetlane po rozwinięciu listy.
int	<code>MaxLength</code>	Maksymalna liczba znaków wyświetlanych w polu edycyjnym listy.
string	<code>Name</code>	Nazwa listy.
Control	<code>Parent</code>	Referencja do obiektu zawierającego listę.

Tabela 10.6. Wybrane właściwości klasy *ComboBox* — ciąg dalszy

Typ	Nazwa właściwości	Znaczenie
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.

Jeżeli chcemy dodać większą liczbę elementów jednocześnie, najwygodniej zastosować metodę `AddRange` w postaci:

```
ComboBox.Items.AddRange  
(  
    new object [ ]  
    {  
        "Element 1"  
        "Element 2"  
        // ...  
        "Element N"  
    }  
);
```

Wybranie przez użytkownika elementu z listy można wykryć poprzez oprogramowanie zdarzenia o nazwie `SelectedIndexChanged`. Odniesienie do wybranego elementu znajdziemy natomiast we właściwości `SelectedItem`.

Ć W I C Z E N I E

10.10 Sposób użycia listy rozwijanej

Umieść w oknie aplikacji element `ComboBox` (rysunek 10.7). Po wybraniu pozycji z listy wyświetl jej nazwę w oknie dialogowym.

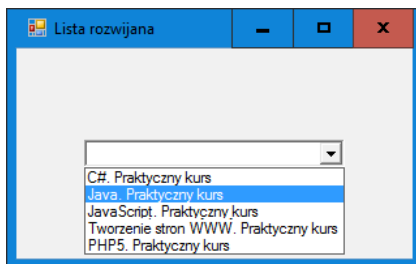
```
using System;  
using System.Drawing;  
using System.Windows.Forms;
```

```
public class MainForm : Form
{
    ComboBox cb = new ComboBox();
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;
        this.Text = "Lista rozwijana";

        cb.Items.AddRange
        (
            new object[]
            {
                "C#. Praktyczny kurs",
                "Java. Praktyczny kurs",
                "JavaScript. Praktyczny kurs",
                "Tworzenie stron WWW. Praktyczny kurs",
                "PHP5. Praktyczny kurs"
            }
        );
        cb.Width = 200;
        cb.Left = (this.ClientSize.Width - cb.Width) / 2;
        cb.Top = (this.ClientSize.Height - cb.Height) / 2;

        EventHandler eh = new EventHandler(this.OnSelection);
        cb.SelectedIndexChanged += eh;
        this.Controls.Add(cb);
    }
    public void OnSelection(Object sender, EventArgs e)
    {
        string s = ((ComboBox)sender).SelectedItem.ToString();
        MessageBox.Show("Wybrano element: " + s);
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Rysunek 10.7.
*Element ComboBox
utworzony
w ćwiczeniu 10.10*



Lista rozwijana jest reprezentowana przez pole `cb` typu `ComboBox`. Pozycje listy dodawane są za pomocą metody `AddRange` właściwości `Items`. Argumentem metody jest nowo utworzona tablica zawierająca poszczególne ciągi znaków składające się na tytuły kilku książek z serii *Praktyczny kurs*. Aby możliwa była reakcja na wybór elementu listy, zdarzenie `SelectedIndexChanged` wiązane jest z procedurą obsługi, którą jest metoda `OnSelection` klasy `MainForm`. Zastosowana została nieużywana do tej pory metoda dotarcia do obiektu wywołującego zdarzenie — stosowany jest argument `sender` metody. Konieczne było przy tym rzutowanie na typ `ComboBox` (`((ComboBox)sender)`), bo typem formalnym argumentu jest `object`. Wybrana przez użytkownika pozycja listy jest uzyskiwana przez odwołanie się do właściwości `SelectedItem`, a zawarty w niej tekst — przez wywołanie metody `ToString`.

Listy zwykłe (ListBox)

Obsługa zwykłych list jest bardzo podobna do obsługi elementów `ComboBox`. Do dyspozycji jest jednak kilka dodatkowych właściwości, które pozwalają na obsługę sytuacji, kiedy na liście znajdzie się wiele zaznaczonych elementów. Te dodatkowe właściwości przedstawione są w tabeli 10.7.

Tabela 10.7. Wybrane właściwości klasy `ListBox`

Typ	Nazwa właściwości	Znaczenie
<code>bool</code>	<code>MultiColumn</code>	Określa, czy elementy listy mogą być wyświetlane w wielu kolumnach.
<code>bool</code>	<code>ScrollAlwaysVisible</code>	Określa, czy pasek przewijania ma być stale widoczny.
<code>SelectedIndex</code> ↳ <code>Collection</code>	<code>SelectedIndices</code>	Lista zawierająca indeksy wszystkich zaznaczonych elementów.
<code>SelectedObject</code> ↳ <code>Collection</code>	<code>SelectedItems</code>	Lista zawierająca wszystkie zaznaczone elementy.
<code>SelectionMode</code>	<code>SelectionMode</code>	Określa sposób, w jaki będą zaznaczane elementy listy.

Podstawową różnicą w stosunku do listy `ComboBox`, oprócz metody wyświetlania, jest oczywiście możliwość zaznaczania więcej niż jednego elementu. Sposób, w jaki elementy będą zaznaczane, można kontrolować za pomocą właściwości `SelectionMode`, której przypisuje się następujące wartości:

- ❑ `MultiSimple` — może być zaznaczonych wiele elementów;
- ❑ `MultiExtended` — może być zaznaczonych wiele elementów, do zaznaczania można używać klawiszy *Shift*, *Ctrl* i klawiszy kursora;
- ❑ `One` — tylko jeden element może być zaznaczony;
- ❑ `None` — elementy nie mogą być zaznaczane.

Dostęp do zaznaczonych elementów uzyskuje się dzięki właściwości `SelectedIndices`, która zawiera indeksy wszystkich zaznaczonych elementów, oraz właściwości `SelectedItems`, zawierającej listę zaznaczonych elementów. Należy zwrócić uwagę, że jeżeli lista pracuje w trybie `MultiSimple` lub `MultiExtended`, właściwości `SelectedIndex` (indeks wybranego elementu) i `SelectedItem` (wybrany element) będą wskazywały dowolny z zaznaczonych elementów.

Ć W I C Z E N I E

10.11 Lista wielokrotnego wyboru

Umieść w oknie aplikacji listę `ListBox` zawierającą kilka elementów oraz przycisk (rysunek 10.8). Po kliknięciu przycisku wyświetl nazwy elementów, które zostały zaznaczone.

```
using System;
using System.Drawing;
using System.Windows.Forms;

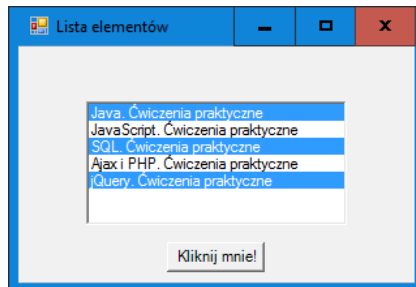
public class MainForm : Form
{
    ListBox lb = new ListBox();
    Button button = new Button();
    public MainForm()
    {
        this.Width = 320;
        this.Height = 220;
        this.Text = "Lista elementów";

        button.Top = 150;
        button.Left = (this.ClientSize.Width - button.Width) / 2;
        button.Text = "Kliknij mnie!";
    }
}
```

```
lb.Items.AddRange
(
    new object[]
    {
        "Java. Ćwiczenia praktyczne",
        "JavaScript. Ćwiczenia praktyczne",
        "SQL. Ćwiczenia praktyczne",
        "Ajax i PHP. Ćwiczenia praktyczne",
        "jQuery. Ćwiczenia praktyczne"
    }
);
lb.Width = 200;
lb.Left = (this.ClientSize.Width - lb.Width) / 2;
lb.Top = (this.ClientSize.Height - lb.Height) / 2;
lb.SelectionMode = SelectionMode.MultiExtended;

EventHandler eh = new EventHandler(this.OnButtonClick);
button.Click += eh;
this.Controls.Add(lb);
this.Controls.Add(button);
}
public void OnButtonClick(Object sender, EventArgs e)
{
    string str = "";
    foreach(string name in lb.SelectedItems)
    {
        str += " '" + name + "'\n";
    }
    MessageBox.Show("Zaznaczone elementy:\n" + str);
}
public static void Main()
{
    Application.Run(new MainForm());
}
}
```

Rysunek 10.8.
*Lista pracująca
w trybie
MultiExtended
z ćwiczenia 10.11*



Lista tworzona jest na takiej samej zasadzie jak w ćwiczeniu 10.10 — jedynie zamiast klasy `CheckBox` została użyta `ListBox`. Dodatkowo w oknie został również umieszczony przycisk typu `Button` (wygląd aplikacji można zobaczyć na rysunku 10.8). Lista jest reprezentowana przez pole `lb`. Aby możliwe było zaznaczanie wielu elementów, właściwość `SelectionMode` otrzymała wartość `SelectionMode.MultiExtended`. Kliknięcie przycisku `button` będzie powodowało wywołanie metody `OnButtonClick`. W jej wnętrzu znajduje się pętla `foreach` odczytująca wszystkie elementy właściwości `SelectedItems` listy `lb`. Każdy taki element odpowiada pojedynczej zaznaczonej pozycji. Wewnątrz pętli konstruowany jest ciąg znaków `str` składający się z tekstów zawartych w zaznaczonych pozycjach (elementów kolekcji `SelectedItems`). Po zakończeniu pętli ciąg zapisany w `str` jest wyświetlany na ekranie w osobnym oknie dialogowym.

Menu

Czym są menu, z pewnością nie trzeba nikomu przypominać. Czas zatem nauczyć się, w jaki sposób skorzystać z tych pożytecznych elementów interfejsu w C# na platformie .NET. Do dyspozycji są zarówno menu główne, jak i menu kontekstowe. Do ich tworzenia służą klasy `MainMenu`, `ContextMenu` oraz `MenuItem`. Wszystkie znajdują się w przestrzeni nazw `Windows.Forms`.

Menu główne

Aby dodać do aplikacji menu główne, trzeba wykonać kilka czynności. Przede wszystkim należy utworzyć obiekt klasy `MainMenu` i przypisać go właściwości `Menu` obiektu klasy `Form`. W ten sposób do aplikacji zostanie dodane menu główne, które jednak nie będzie zawierało żadnej pozycji. W celu utworzenia pozycji budujemy obiekt klasy `MenuItem` i dodajemy go do menu głównego za pomocą metody `Add`. Schemat postępowania jest zatem następujący:

```
MainMenu obiekt_menu = new MainMenu();  
MenuItem obiekt_pozycji = new MenuItem();  
objekt_menu.MenuItems.Add(objekt_pozycji);
```

Po wykonaniu tych instrukcji wystarczy tylko przypisać utworzonej pozycji tekst, jaki ma się na niej pojawiać, np. *Plik*. Sposób wykonania tego zadania jest wyjątkowo prosty. Wystarczy właściwości `Text` obiektu klasy `MenuItem` przypisać odpowiedni ciąg znaków. W omawianym przykładzie należałoby użyć instrukcji:

```
obiekt_pozycji.Text = "Plik";
```

ĆWICZENIE

10.12 Dodanie do aplikacji menu głównego

Napisz aplikację wyświetlającą okno zawierające menu główne z pozycją *Plik*, co jest widoczne na rysunku 10.9.

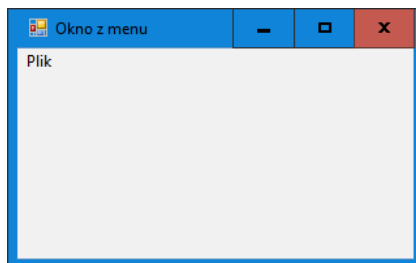
```
using System;
using System.Windows.Forms;

public class MainForm : Form
{
    MainMenu mainMenu = new MainMenu();
    MenuItem menuItem1 = new MenuItem();
    public MainForm()
    {
        this.Text = "Okno z menu";
        this.Width = 320;
        this.Height = 200;

        menuItem1.Text = "Plik";
        mainMenu.MenuItems.Add(menuItem1);
        this.Menu = mainMenu;
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Rysunek 10.9.

*Aplikacja
zawierająca
menu główne*



Sposób postępowania jest zgodny z podanym wyżej opisem. Menu główne reprezentuje obiekt wskazywany przez pole `mainMenu` (utworzony za pomocą konstruktora klasy `MainMenu`), natomiast pozycję menu — obiekt wskazywany przez `menuItem1` (który powstał przy użyciu konstruktora klasy `MenuItem`). Pozycja menu jest dodawana do menu przez wywołanie metody `Add`. Menu jest z kolei umieszczane w oknie aplikacji przez przypisanie referencji `mainMenu` właściwości `Menu` (tę właściwość zawiera obiekt reprezentujący okno aplikacji — obiekt klasy `MainForm` reprezentowany przez wskazanie `this`).

Do menu *Plik* powstałego w ćwiczeniu 10.12 należałoby dodać kolejną pozycję — niech to będzie *Zamknij* — będzie można wtedy wykorzystać ją do końca pracy z aplikacją. Sposób postępowania jest tu taki sam jak w przypadku menu głównego. Trzeba utworzyć nowy obiekt klasy `MenuItem` oraz skorzystać z metody `Add` klasy `MenuItems`. Jak to wygląda w praktyce, pokazano w ćwiczeniu 10.13.

Ć W I C Z E N I E

10.13 Dodawanie pozycji do menu

Do menu otrzymanego w ćwiczeniu 10.12 dodaj pozycję o nazwie *Zamknij*, co widoczne jest na rysunku 10.10.

```
using System;
using System.Windows.Forms;

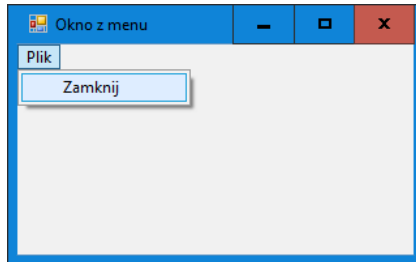
public class MainForm : Form
{
    MainMenu mainMenu = new MainMenu();
    MenuItem menuItem1 = new MenuItem();
    MenuItem menuItem2 = new MenuItem();
    public MainForm()
    {
        this.Text = "Okno z menu";
        this.Width = 320;
        this.Height = 200;

        menuItem1.Text = "Plik";
        menuItem2.Text = "Zamknij";
        mainMenu.MenuItems.Add(menuItem1);
        menuItem1.MenuItems.Add(menuItem2);
        this.Menu = mainMenu;
    }
}
```



```
public static void Main()
{
    Application.Run(new MainForm());
}
```

Rysunek 10.10.
*Aplikacja ma
teraz dodatkowe
podmenu*



Struktura programu pozostała taka sama jak w ćwiczeniu 10.12. Dodana została jedynie nowa pozycja menu odzwierciedlana przez menuItem2. Obiekt ten został dodany do pozycji *Zamknij* (reprezentowanej przez obiekt menuItem1) za pomocą wywołania metody Add (udostępnianej przez właściwość MenuItems).

Nazwy poszczególnym menu można nadawać również bezpośrednio w konstruktorze klasy MenuItem. Pozwala to na zaoszczędzenie pewnej ilości miejsca w kodzie programu. Żądany ciąg znaków trzeba podać jako argument konstruktora; schematycznie:

```
MenuItem obiekt_pozycji = new MenuItem("nazwa_pozycji");
```

Ć W I C Z E N I E

10.14 Bezpośrednie nadawanie nazw menu

Utwórz menu tak jak w ćwiczeniu 10.13. Skorzystaj z metody bezpośredniego nadawania nazw pozycjom.

```
using System;
using System.Windows.Forms;

public class MainForm : Form
{
    MainMenu mainMenu = new MainMenu();
    MenuItem menuItem1 = new MenuItem("Plik");
    MenuItem menuItem2 = new MenuItem("Zamknij");
    public MainForm()
    {
```

```
this.Text = "Okno z menu";
this.Width = 320;
this.Height = 200;

mainMenu.MenuItems.Add(menuItem1);
menuItem1.MenuItems.Add(menuItem2);
this.Menu = mainMenu;
}
public static void Main()
{
    Application.Run(new MainForm());
}
}
```

Oprócz opisanych powyżej istnieje jeszcze jedna możliwość tworzenia podmenu — nie trzeba bezpośrednio tworzyć obiektów `MenuItem`. Zrobi to za nas równie dobrze metoda `Add`, zastosowana do właściwości `MenuItems`¹. Zamiast pisać:

```
MenuItem obiekt_pozycji = new MenuItem("nazwa");
objekt_menu.MenuItems.Add(objekt_pozycji);
```

można zastosować konstrukcję:

```
MenuItem menuItem1 = obiekt_menu.MenuItems.Add("nazwa");
```

Co prawda tak utworzonego obiektu (referencji do obiektu) nie trzeba koniecznie przypisywać zmiennej (w tym przypadku `menuItem1`), jednak lepiej to zrobić. Dzięki temu będzie można odwoływać się do niego w dalszej części kodu.

Ć W I C Z E N I E

10.15 Bezpośrednie tworzenie pozycji menu

Utwórz pozycję menu tak jak w ćwiczeniu 10.13. Skorzystaj z nowej metody dodawania podmenu.

```
using System;
using System.Windows.Forms;

public class MainForm:Form
```

¹ W rzeczywistości `MenuItems` to wskazanie na obiekt klasy zagnieżdżonej `Menu.MenuItemCollection`. Dopiero ta klasa zawiera zestaw przeciążonych metod `Add`. Bliższe informacje można znaleźć w dokumentacji .NET oraz Visual Studio (Visual C#).

```
{
    MainMenu mainMenu = new MainMenu();
    MenuItem menuItem1;
    MenuItem menuItem2;
    public MainForm()
    {
        this.Text = "Okno z menu";
        this.Width = 320;
        this.Height = 200;

        menuItem1 = mainMenu.MenuItems.Add("Plik");
        menuItem2 = menuItem1.MenuItems.Add("Zamknij");
        this.Menu = mainMenu;
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```


Utworzone w poprzednich ćwiczeniach menu są niestety nieaktywne, tzn. po wybraniu danej pozycji nic się nie dzieje. Z pewnością nie jest to wielkim zaskoczeniem, nie powstały przecież jeszcze odpowiednie procedury obsługi. Po lekturze rozdziału 9. łatwo się domyślić, że trzeba będzie skorzystać ze zdarzeń oraz delegacji EventHandler.

Należy zatem zdefiniować metodę obsługującą zdarzenie. Deklaracja tej metody musi być zgodna z delegacją EventHandler, która ma postać:

```
public delegate void EventHandler(object sender, EventArgs args);
```

Parametry sender i args nie muszą być używane. Na przykład metoda, której zadaniem byłoby reagowanie na wybranie z menu pozycji *Zamknij*, mogłaby wyglądać następująco:

```
public void ZamknijClicked(Object sender, EventArgs e)
{
    this.Close();
}
```

Pozostaje zatem powiązanie takiej metody ze zdarzeniem polegającym na wybraniu pozycji *Zamknij*. Należy utworzyć delegację typu Event Handler i przekazać jej w postaci argumentu metodę ZamknijClicked:

```
EventHandler eh = new EventHandler(ZamknijClicked);
```

Po takiej deklaracji wiąże się delegację z konkretną pozycją w menu, pisząc:

```
MenuItems.Add("Zamknij", eh);
```

Ć W I C Z E N I E

10.16 Aktywacja menu

Uzupełnij kod z ćwiczenia 10.15 w taki sposób, aby po wybraniu z menu Plik pozycji Zamknij następowało wyjście z aplikacji.

```
using System;
using System.Windows.Forms;

public class MainForm:Form
{
    MainMenu mainMenu = new MainMenu();
    MenuItem menuItem1;
    MenuItem menuItem2;
    public MainForm()
    {
        this.Text = "Okno z menu";
        this.Width = 320;
        this.Height = 200;

        menuItem1 = mainMenu.MenuItems.Add("Plik");
        EventHandler eh = new EventHandler(this.ZamknijClicked);
        menuItem2 = menuItem1.MenuItems.Add("Zamknij", eh);
        this.Menu = mainMenu;
    }
    public void ZamknijClicked(Object sender, EventArgs e)
    {
        this.Close();
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Ć W I C Z E N I E

10.17 Obsługa zdarzeń powiązanych z menu

Napisz aplikację zawierającą kilka pozycji w menu głównym. Po wybraniu dowolnej pozycji z menu wyświetl jej nazwę, korzystając z metody Show klasy MessageBox.

```
using System;
using System.Windows.Forms;

public class MainForm:Form
{
    MainMenu mainMenu = new MainMenu();
```

```
MenuItem menuItem1;
MenuItem menuItem2;
MenuItem menuItem3;
MenuItem menuItem1SubItem1;
MenuItem menuItem1SubItem2;
MenuItem menuItem2SubItem1;
MenuItem menuItem3SubItem1;
MenuItem menuItem3SubItem2;
MenuItem menuItem3SubItem3;
public MainForm()
{
    this.Text = "Okno z menu";
    this.Width = 320;
    this.Height = 200;

    menuItem1 = mainMenu.MenuItems.Add("Pozycja 1");
    menuItem2 = mainMenu.MenuItems.Add("Pozycja 2");
    menuItem3 = mainMenu.MenuItems.Add("Pozycja 3");

    EventHandler eh = new EventHandler(this.MenuItemClicked);

    menuItem1SubItem1 = menuItem1.MenuItems.Add("Podpozycja 1", eh);
    menuItem1SubItem2 = menuItem1.MenuItems.Add("Podpozycja 2", eh);

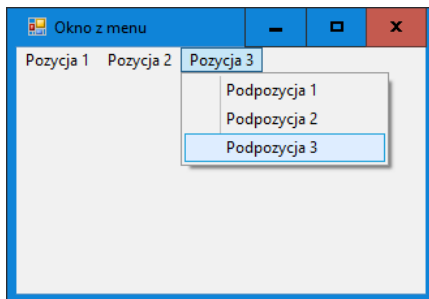
    menuItem2SubItem1 = menuItem2.MenuItems.Add("Podpozycja 1", eh);

    menuItem3SubItem1 = menuItem3.MenuItems.Add("Podpozycja 1", eh);
    menuItem3SubItem2 = menuItem3.MenuItems.Add("Podpozycja 2", eh);
    menuItem3SubItem3 = menuItem3.MenuItems.Add("Podpozycja 3", eh);

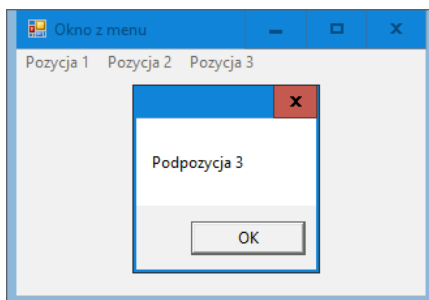
    this.Menu = mainMenu;
}
public void MenuItemClicked(object sender, EventArgs e)
{
    MessageBox.Show(((MenuItem)sender).Text);
}
public static void Main()
{
    Application.Run(new MainForm());
}
}
```

Okno tak skonstruowanej aplikacji będzie miało wygląd zaprezentowany na rysunku 10.11. Wybranie dowolnej pozycji podmenu spowoduje wyświetlenie jej nazwy w dodatkowym oknie dialogowym (rysunek 10.12). Sposób dodawania obsługi zdarzeń do poszczególnych pozycji jest taki sam jak w ćwiczeniu 10.16 i wcześniejszych (w każdym przypadku jest to delegacja eh typu `EventHandler` powiązana z metodą `MenuItemClicked`). Sama procedura obsługi została oczywiście

Rysunek 10.11.
Struktura menu
utworzona
w ćwiczeniu 10.17



Rysunek 10.12.
Wybranie dowolnej
pozycji podmenu
powoduje
wyświetlenie
jej nazwy



zmieniona, choć używane są w niej konstrukcje prezentowane już w poprzednich przykładach. Zastosowana została metoda `Show` klasy `MessageBox`, która pozwala na wyświetlenie okna dialogowego z dowolnie podanym tekstem.

Chwili uwagi wymaga zastosowany sposób uzyskania nazwy wybranego podmenu (jest podobny do przykładu z listą rozwijaną z ćwiczenia 10.10). Korzystamy tutaj z faktu, że do procedury obsługi zdarzenia, czyli metody `MenuItemClicked`, jest przekazywana referencja do obiektu, który ją wywołuje. Jest to argument `sender`. Rzecz jasna, skoro parametr ten jest typu `object`, należy dokonać rzutowania na typ `MenuItem`. Po dokonaniu rzutowania można bezpośrednio odwołać się do właściwości `Text`.

_____■

Menu kontekstowe

Tworzenie menu kontekstowego jest bardzo podobne do budowania menu głównego. Strukturę definiuje się dokładnie w taki sam sposób. Różnica jest taka, że zamiast przypisania:

```
Form.Menu = nasze_menu;
```

jest stosowane przypisanie:

```
Form.ContextMenu = nasze_menu;
```

oraz zamiast klasy MainMenu używa się klasy ContextMenu.

Ć W I C Z E N I E

10.18 Dodanie do aplikacji menu kontekstowego

Napisz aplikację posiadającą menu kontekstowe (rysunek 10.13).

```
using System;
using System.Windows.Forms;

public class MainForm : Form
{
    ContextMenu contextMenu = new ContextMenu();
    MenuItem menuItem1;
    MenuItem menuItem2;
    MenuItem menuItem3;
    MenuItem menuItem4;

    public MainForm()
    {
        this.Text = "Okno z menu kontekstowym";
        this.Width = 320;
        this.Height = 200;

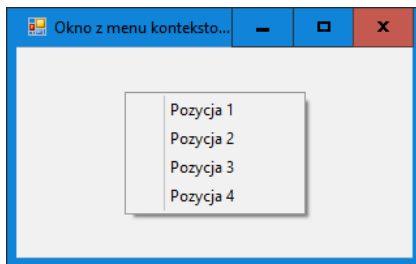
        menuItem1 = contextMenu.MenuItems.Add("Pozycja 1");
        menuItem2 = contextMenu.MenuItems.Add("Pozycja 2");
        menuItem3 = contextMenu.MenuItems.Add("Pozycja 3");
        menuItem4 = contextMenu.MenuItems.Add("Pozycja 4");

        this.ContextMenu = contextMenu;
    }

    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Rysunek 10.13.

*Aplikacja
posiadająca
menu kontekstowe*



Obiekt reprezentujący menu kontekstowe został utworzony za pomocą konstruktora klasy `ContextMenu`, a referencja do niego została przypisana zmiennej (polu) `contextMenu`. Następnie dzięki użyciu metody `Add` kolekcji `MenuItem`s powstały cztery pozycje menu. Odwołania do nich znalazły się w polach typu `MenuItem`. Zbudowane w ten sposób menu kontekstowe zostało dodane do okna aplikacji przez przypisanie referencji `contextMenu` właściwości `ContextMenu`. Wygląd tego menu pokazano na rysunku 10.13.

Menu kontekstowe może mieć wiele poziomów, podobnie jak i menu zwykłe. Identyczna jest też zasada tworzenia takiej konstrukcji. Do istniejących pozycji należy dodawać kolejne, aż do zbudowania pożądanego hierarchii. Przykład aplikacji zawierającej wielopoziomowe menu kontekstowe został przedstawiony w ćwiczeniu 10.19.

Ć W I C Z E N I E**10.19 Wielopoziomowe menu kontekstowe**

Napisz aplikację posiadającą wielopoziomowe menu kontekstowe widoczne na rysunku 10.14.

```
using System.Windows.Forms;

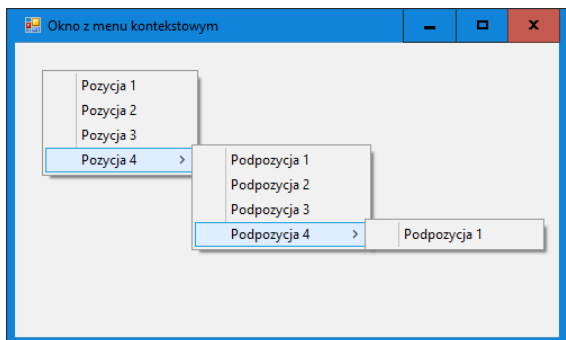
public class MainForm : Form
{
    ContextMenu contextMenu = new ContextMenu();
    MenuItem menuItem1;
    MenuItem menuItem2;
    MenuItem menuItem3;
    MenuItem menuItem4;

    MenuItem menu4SubItem1;
    MenuItem menu4SubItem2;
```



```
MenuItem menu4SubItem3;  
MenuItem menu4SubItem4;  
  
MenuItem menu4SubItem4SubItem1;  
  
public MainForm()  
{  
    this.Text = "Okno z menu kontekstowym";  
    this.Width = 500;  
    this.Height = 300;  
  
    menuItem1 = contextMenu.MenuItems.Add("Pozycja 1");  
    menuItem2 = contextMenu.MenuItems.Add("Pozycja 2");  
    menuItem3 = contextMenu.MenuItems.Add("Pozycja 3");  
    menuItem4 = contextMenu.MenuItems.Add("Pozycja 4");  
  
    menu4SubItem1 = menuItem4.MenuItems.Add("Podpozycja 1");  
    menu4SubItem2 = menuItem4.MenuItems.Add("Podpozycja 2");  
    menu4SubItem3 = menuItem4.MenuItems.Add("Podpozycja 3");  
    menu4SubItem4 = menuItem4.MenuItems.Add("Podpozycja 4");  
  
    menu4SubItem4SubItem1 = menu4SubItem4.MenuItems.Add("Podpozycja 1");  
  
    this.ContextMenu = contextMenu;  
}  
public static void Main()  
{  
    Application.Run(new MainForm());  
}  
}
```

Rysunek 10.14.
*Aplikacja
z wielopoziomowym
menu kontekstowym*



Właściwości menu

W menu — zarówno głównym, jak i kontekstowym — istnieje możliwość zaznaczania każdej pozycji. Odbywa się to poprzez zmianę właściwości `Checked` obiektu klasy `MenuItem` na `true` (pozycja zaznaczona) lub `false` (pozycja niezaznaczona).

Ć W I C Z E N I E

10.20 Menu z możliwością zaznaczania pozycji

Napisz aplikację posiadającą menu kontekstowe, w którym co druga pozycja jest zaznaczona (rysunek 10.15).

```
using System.Windows.Forms;

public class MainForm : Form
{
    ContextMenu contextMenu = new ContextMenu();
    MenuItem menuItem1;
    MenuItem menuItem2;
    MenuItem menuItem3;
    MenuItem menuItem4;

    public MainForm()
    {
        this.Text = "Okno z menu kontekstowym";
        this.Width = 320;
        this.Height = 200;

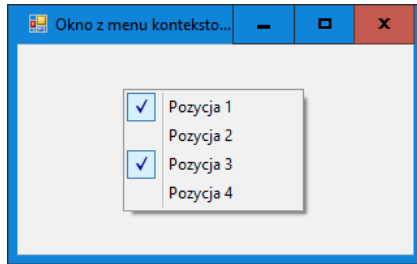
        menuItem1 = contextMenu.MenuItems.Add("Pozycja 1");
        menuItem1.Checked = true;
        menuItem2 = contextMenu.MenuItems.Add("Pozycja 2");
        menuItem3 = contextMenu.MenuItems.Add("Pozycja 3");
        menuItem3.Checked = true;
        menuItem4 = contextMenu.MenuItems.Add("Pozycja 4");

        this.ContextMenu = contextMenu;
    }

    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Rysunek 10.15.

*Menu
kontekstowe
z zaznaczonymi
niektórymi
pozycjami*



Menu wraz z pozycjami zostało utworzone w taki sam sposób jak w ćwiczeniu 10.18. Pozycje pierwsza (reprezentowana przez menuItem1) i trzecia (reprezentowana przez menuItem3) zostały zaznaczone przez przypisanie wartości true właściwości Checked odpowiednich obiektów. Dlatego po uruchomieniu aplikacji i wywołaniu menu na ekranie pojawi się widok przedstawiony na rysunku 10.15.

Korzystając z obsługi zdarzeń oraz właściwości Checked, można utworzyć menu, w którym stan pozycji będzie się zmieniał dynamicznie. Znaczy to, że kiedy użytkownik pierwszy raz wybierze daną pozycję, zostanie ona zaznaczona, kolejne wybranie tejże pozycji spowoduje usunięcie jej zaznaczenia itd. Taki sposób obsługi często spotykamy w aplikacjach, warto więc przećwiczyć jego wykonanie.

Ć W I C Z E N I E**10.21 Menu z możliwością zmiany zaznaczeń**

Utwórz menu główne, w przypadku którego stan poszczególnych pozycji będzie się zmieniał po każdym wybraniu danej pozycji.

```
using System;
using System.Windows.Forms;

public class MainForm : Form
{
    MainMenu mainMenu = new MainMenu();
    MenuItem menuItem1;
    MenuItem menuSubItem1;
    MenuItem menuSubItem2;
    MenuItem menuSubItem3;
    MenuItem menuSubItem4;

    public MainForm()
    {
```

```
this.Text = "Okno z menu";
this.Width = 320;
this.Height = 200;

EventHandler eh = new EventHandler(this.MenuItemClicked);

menuItem1 = mainMenu.MenuItems.Add("Menu 1");
menuItem1.MenuItems.Add("Pozycja 1", eh);
menuItem1.MenuItems.Add("Pozycja 2", eh);
menuItem1.MenuItems.Add("Pozycja 3", eh);
menuItem1.MenuItems.Add("Pozycja 4", eh);

this.Menu = mainMenu;
}
public void MenuItemClicked(Object sender, EventArgs e)
{
    ((MenuItem)sender).Checked = !((MenuItem)sender).Checked;
}
public static void Main()
{
    Application.Run(new MainForm());
}
}
```

Delegacji `eh` został przypisany nowy obiekt typu `EventHandler` (ściślej referencja do obiektu) powiązany z metodą `MenuItemClicked`. Następnie został on użyty w instrukcjach tworzących pozycje menu. W związku z tym wybranie dowolnej pozycji będzie powodowało wywołanie wspomnianej metody. W jej wnętrzu wartość właściwości `Checked` danej pozycji jest po prostu zmieniana na przeciwną, czyli jeśli właściwość była równa `false`, będzie równa `true`, jeśli natomiast była równa `true`, będzie równa `false`. Jest to wykonywane w instrukcji:

```
((MenuItem)sender).Checked = !((MenuItem)sender).Checked;
```

Tym samym wskazanie niezaznaczonej pozycji menu spowoduje jej zaznaczenie i odwrotnie — wskazanie zaznaczonej pozycji spowoduje usunięcie zaznaczenia.

Oprócz standardowego sposobu zaznaczenia pozycji, który został użyty w ćwiczeniu 10.20, istnieje także inny. Jeśli właściwości `RadioCheck` obiektu klasy `MenuItem` przypiszemy wartość `true`, to po nadaniu właściwości `Checked` również wartości `true` menu będą zaznaczane za pomocą symbolu kropki (kółka).

Ć W I C Z E N I E

10.22 Alternatywny sposób zaznaczania pozycji menu

Zmodyfikuj kod z ćwiczenia 10.21 w taki sposób, aby wtedy, gdy pierwsza pozycja menu (*Pozycja1*) jest aktywna (zaznaczona), pozostałe były zaznaczane przez symbol kropki (rysunek 10.16), a kiedy jest ona nieaktywna — przez symbol domyślny.

```
using System;
using System.Windows.Forms;

public class MainForm : Form
{
    MainMenu mainMenu = new MainMenu();
    MenuItem menuItem1;
    MenuItem menuSubItem1;
    MenuItem menuSubItem2;
    MenuItem menuSubItem3;
    MenuItem menuSubItem4;

    public MainForm()
    {
        this.Text = "Okno z menu";
        this.Width = 320;
        this.Height = 200;

        EventHandler eh = new EventHandler(this.MenuItemClicked);

        menuItem1 = mainMenu.MenuItems.Add("Menu 1");
        menuSubItem1 = menuItem1.MenuItems.Add("Pozycja 1", eh);
        menuSubItem2 = menuItem1.MenuItems.Add("Pozycja 2", eh);
        menuSubItem3 = menuItem1.MenuItems.Add("Pozycja 3", eh);
        menuSubItem4 = menuItem1.MenuItems.Add("Pozycja 4", eh);

        menuSubItem1.RadioCheck = false;

        this.Menu = mainMenu;
    }

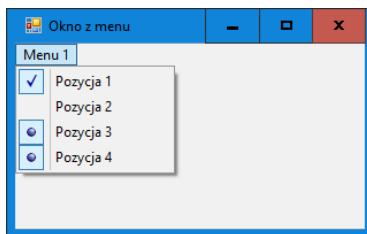
    public void MenuItemClicked(Object sender, EventArgs e)
    {
        ((MenuItem)sender).Checked = !((MenuItem)sender).Checked;
        if(sender == menuSubItem1)
        {
            if(((MenuItem)sender).Checked)
            {
                menuSubItem2.RadioCheck = true;
                menuSubItem3.RadioCheck = true;
                menuSubItem4.RadioCheck = true;
            }
        }
    }
}
```

```
        else
        {
            menuSubItem2.RadioCheck = false;
            menuSubItem3.RadioCheck = false;
            menuSubItem4.RadioCheck = false;
        }
    }
}

public static void Main()
{
    Application.Run(new MainForm());
}
```

Rysunek 10.16.

*Elementy menu
zaznaczane
za pomocą
symbolu kropki*



Różnica w stosunku do przykładu z ćwiczenia 10.21 sprowadza się przede wszystkim do modyfikacji metody `MenuItemClicked`. Instrukcja zmieniająca stan pozycji menu pozostała wprawdzie niezmieniona, za nią jednak pojawiły się zagnieżdżone instrukcje warunkowe. Badane jest, czy inicjatorem zdarzenia (argument `sender`) jest obiekt pierwszej pozycji menu (`menuSubItem1`), mówiąc prościej — czy wybrana została pierwsza pozycja menu. Jeśli została wybrana, dalsze postępowanie zależy od tego, czy ta pozycja jest zaznaczona (jej wartość `Checked` jest równa `true`). Gdy jest zaznaczona, właściwości `RadioCheck` pozostałych pozycji są zaznaczane (otrzymują wartość `true`), w przeciwnym przypadku ich zaznaczenia są usuwane (właściwości `RadioCheck` otrzymują wartość `false`).

Skróty klawiaturowe

Dostęp do menu może być zrealizowany przy użyciu skrótów klawiaturowych z klawiszem *Alt*. Wiele aplikacji udostępnia ten praktyczny sposób, który na szczęście jest bardzo prosty do realizacji.

Wystarczy, że podczas tworzenia pozycji menu literę, która będzie używana do aktywacji tej pozycji, poprzedzi się znakiem &. To całe zadanie.

Ć W I C Z E N I E

10.23 Dostęp do menu za pomocą skrótów klawiaturowych

Napisz aplikację zawierającą menu *Plik*, w którym będzie się znajdować pozycja *Zamknij*, a wybranie jej spowoduje zamknięcie programu. Menu powinno być dostępne poprzez skróty klawiaturowe z klawiszem *Alt*.

```
using System;
using System.Windows.Forms;

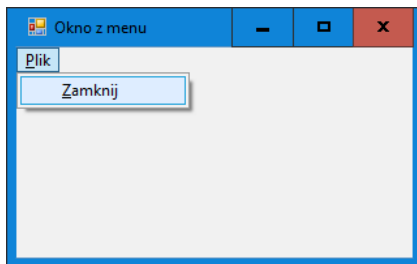
public class MainForm:Form
{
    MainMenu mainMenu = new MainMenu();
    MenuItem menuItem1;
    MenuItem menuItem2;
    public MainForm()
    {
        this.Text = "Okno z menu";
        this.Width = 320;
        this.Height = 200;

        menuItem1 = mainMenu.MenuItems.Add("&Plik");
        EventHandler eh = new EventHandler(this.ZamknijClicked);
        menuItem2 = menuItem1.MenuItems.Add("&Zamknij", eh);
        this.Menu = mainMenu;
    }
    public void ZamknijClicked(Object sender, EventArgs e)
    {
        this.Close();
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Na rysunku 10.17 widać, że dostęp do menu może być realizowany poprzez naciśnięcie klawisza *Alt* i wybranej litery. Aktywne znaki są podkreślone. Warto jednak zwrócić uwagę, że przy tym sposobie realizacji (tzn. przy aktywowaniu menu klawiszem *Alt*) nie można dostać

Rysunek 10.17.

Menu dostępne
poprzez skróty
klawiaturowe
z klawiszem Alt



się bezpośrednio do danej pozycji. W powyższym przypadku, aby wybrać pozycję *Zamknij*, należałoby najpierw wcisnąć klawisz *Alt* i, przytrzymując go, nacisnąć kolejno *P* i *Z*.

Możemy jednak skorzystać z bezpośrednich skrótów klawiaturowych z klawiszem *Control*. Skoro na co dzień używamy takich kombinacji jak *Ctrl+C*, *Ctrl+V* czy *Ctrl+X*, nie ma powodu, aby nie wyposażać aplikacji pisanych w C# w takie właśnie udogodnienia.

Powiązanie menu ze skrótem klawiaturowym będzie wyglądało zupełnie inaczej niż w poprzednim ćwiczeniu. Do dyspozycji są dwa sposoby: albo przypisanie skrótu do menu już podczas tworzenia konkretnej pozycji, albo najpierw utworzenie danej pozycji, a następnie zmodyfikowanie jej właściwości *Shortcut*. Oba sposoby zostaną sprawdzone w kolejnych dwóch ćwiczeniach.

Najpierw przypiszemy odpowiedni skrót już podczas tworzenia pozycji. Wymaga to użycia trójargumentowego konstruktora w postaci:

```
public MenuItem  
(  
    string text,  
    EventHandler eh,  
    Shortcut shortcut  
)
```

Jak widać, w tym przypadku niezbędne będzie jednocześnie przypisanie obsługi zdarzenia. Skrót definiowany jest za pomocą wyliczenia *Shortcut*, np. dla kombinacji *Ctrl+W* należy użyć wartości *Shortcut.CtrlW*.

Ć W I C Z E N I E

10.24 Skróty definiowane razem z menu

Napisz aplikację zawierającą menu *Plik*, w którym będzie znajdować się pozycja *Zamknij*. Wybranie tej pozycji powinno spowodować zamknięcie programu. Do pozycji *Zamknij* przypisz skrót klawiaturowy *Ctrl+W* (lub inny).

```
using System;
using System.Windows.Forms;

public class MainForm : Form
{
    MainMenu mainMenu = new MainMenu();
    MenuItem menuItem1;
    MenuItem menuItem2;
    public MainForm()
    {
        this.Text = "Okno z menu";
        this.Width = 320;
        this.Height = 200;

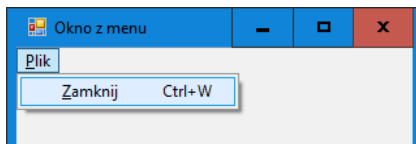
        menuItem1 = mainMenu.MenuItems.Add("&Plik");
        EventHandler eh = new EventHandler(this.ZamknijClicked);
        menuItem2 = new MenuItem("&Zamknij", eh, Shortcut.CtrlW);
        menuItem1.MenuItems.Add(menuItem2);
        this.Menu = mainMenu;
    }
    public void ZamknijClicked(Object sender, EventArgs e)
    {
        this.Close();
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Jeśli spojrzymy na rysunek 10.18, zobaczymy, że do pozycji *Zamknij* w menu *Plik* dopisany został skrót *Ctrl+W*. Wciśnięcie na klawiaturze tej kombinacji znaków (nawet gdy menu nie jest aktywne) spowoduje automatyczne wywołanie pozycji *Zamknij*, a tym samym wykonanie metody *ZamknijClicked*. Ponieważ jedynym zadaniem tej metody jest zamknięcie aplikacji, program zakończy swoje działanie.



Rysunek 10.18.

Menu z przypisanym skrótem klawiaturowym



Drugą z metod dodawania skrótów klawiaturowych jest modyfikacja właściwości `Shortcut` w obiekcie klasy `MenuItem`. Najpierw należy utworzyć pozycję menu (z wykorzystaniem jednego z przedstawionych wyżej sposobów) i dopiero do niej przypisać skrót.

Ć W I C Z E N I E**10.25 Przypisywanie skrótów do istniejących menu**

Napisz aplikację zawierającą menu *Plik*, w którym będzie znajdować się pozycja *Zamknij*. Do pozycji *Zamknij* przypisz skrót klawiaturowy *Ctrl+W* (lub inny), użyj drugiego z poznanych sposobów definiowania skrótów.

```
using System;
using System.Windows.Forms;

public class MainForm:Form
{
    MainMenu mainMenu = new MainMenu();
    MenuItem menuItem1;
    MenuItem menuItem2;
    public MainForm()
    {
        this.Text = "Okno z menu";
        this.Width = 320;
        this.Height = 200;

        menuItem1 = mainMenu.MenuItems.Add("&Plik");
        EventHandler eh = new EventHandler(this.ZamknijClicked);
        menuItem2 = menuItem1.MenuItems.Add("&Zamknij", eh);
        menuItem2.Shortcut = Shortcut.CtrlW;
        this.Menu = mainMenu;
    }
    public void ZamknijClicked(Object sender, EventArgs e)
    {
        this.Close();
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

