

Marcin Lis

C# Ćwiczenia



darmowe ebooki
aktualne czasopisma



ebookgigs.com

Spis treści

Wstęp	5
Część I Język programowania.....	7
Rozdział 1. Pierwsza aplikacja	9
Język C#	9
Środowisko uruchomieniowe	10
Narzędzia.....	11
Najprostszy program.....	11
Kompilacja i uruchamianie.....	12
Visual Studio	13
Dyrektiva using.....	16
Rozdział 2. Zmienne i typy danych.....	17
Typy danych.....	17
Typy arytmetyczne	17
Typ boolean.....	19
Deklarowanie zmiennych	19
Typy referencyjne.....	22
Typ string	23
Typ object.....	23
Wartość null	23
Operatory.....	24
Operatory Arytmetyczne	24
Operatory bitowe	29
Operatory logiczne	30
Operatory przypisania	30
Operatory porównania	31
Operator warunkowy (?)	31
Priorytety operatorów.....	32
Komentarze.....	32
Rozdział 3. Instrukcje	35
Instrukcje warunkowe.....	35
Instrukcja if...else	35
Instrukcja if...else if.....	38
Instrukcja switch.....	39
Instrukcja goto	41
Pętle.....	43
Pętla for	43
Pętla while	48
Pętla do while	49

Wprowadzanie danych.....	50
Argumenty wiersza poleceń	51
Instrukcja ReadLine	54
Rozdział 4. Programowanie obiektowe.....	61
Klasy.....	61
Metody.....	63
Konstruktory.....	69
Specyfikatory dostępu.....	71
Dziedziczenie	75
Rozdział 5. Tablice	71
Deklarowanie tablic.....	77
Inicjalizacja.....	80
Pętla foreach	81
Tablice wielowymiarowe.....	83
Rozdział 6. Wyjątki	89
Obsługa błędów	89
Blok try...catch	93
Hierarchia wyjątków.....	97
Własne wyjątki	99
Rozdział 7. Interfejsy	101
Prosty interfejs.....	101
Interfejsy w klasach potomnych	104
Czy to interfejs?.....	110
Rzutowanie.....	113
Słowo kluczowe as	115
Słowo kluczowe is.....	116
Część II Programowanie w Windows.....	117
Rozdział 8. Pierwsze okno	119
Utworzenie okna.....	119
Wyświetlanie komunikatu	122
Zdarzenie ApplicationExit.....	123
Rozdział 9. Delegacje i zdarzenia.....	125
Delegacje	125
Zdarzenia	128
Rozdział 10. Komponenty.....	133
Etykiety (Label).....	133
Przyciski (klasa Button).....	137
Pola tekstowe (TextBox)	140
Pola wyboru (CheckBox, RadioButton)	143
Listy rozwijalne (ComboBox)	146
Listy zwykłe (ListBox)	149
Menu.....	151
Menu główne.....	151
Menu kontekstowe	157
Właściwości Menu	159
Skróty klawiaturowe.....	162
	162

Rozdział 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 odzyskiwanie pamięci. Programiści korzystający na co dzień z wymienionych języków programowania, będą się czuli doskonale w tym środowisku. Z kolei dla osób nie znających 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 przez firmę Borland pakietu Delphi, a także Turbo Pascala! W Microsoftie Hejlsberg rozwijał m.in. środowisko Visual J++. To wszystko nie pozostało bez wpływu najnowszy produkt, w którym można dojrzeć wyraźne związki zarówno z C i C++, jak Javą i Delphi, czyli Object Pascalem.

C# jest w pełni obiektowy, zawiera wspomniane już mechanizmy odzyskiwania pamięci i obsługę wyjątków. Jest też ściśle powiązany ze środowiskiem uruchomieniowym .NET, co oczywiście nie oznacza, że nie mogą powstawać jego implementacje przeznaczone dla innych platform. Oznacza to jednak, że doskonale sprawdza się w najnowszym środowisku Windows oraz w sposób bezpośredni może korzystać z klas .NET, co pozwala na szybkie i efektywne pisanie aplikacji.

Środowisko uruchomieniowe

Programy pisane w technologii .NET, niezależnie od zastosowanego języka, wymagają specjalnego środowiska uruchomieniowego, tak zwanego CLR — *Common Language Runtime*. Próba uruchomienia takiego programu w zwykłym Windows skończy się niepowodzeniem i komunikatami o braku odpowiednich bibliotek. Niezbędne jest zatem zainstalowanie pakietu *.NET Framework* dostępnego na stronach Microsoftu (<http://www.microsoft.com>). Dotyczy to wszystkich systemów wcześniejszych niż Windows XP.

Skąd takie wymogi? Otóż program pisany w technologii .NET, czy to w C#, Visual Basicu czy innym języku, nie jest komplikowany do kodu natywnego danego procesora, ale do kodu pośredniego¹ (przypomina to w pewnym stopniu *byte-code* znany z Javy). Tenże 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 owym 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)
}
```

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

Najważniejsze dla nas w tej chwili 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.

¹ Komplikacja kodu pośredniego do kodu natywnego procesora jest wykonywana przez kompilator just-in-time w momencie uruchomienia aplikacji.

Narzędzia

Środowiskiem programistycznym służącym do tworzenia aplikacji C# jest oczywiście produkowany przez firmę Microsoft pakiet *Visual C# .NET*. Jest on dostępny jako oddzielny produkt, jak również jako część pakietu *Visual Studio .NET*.

Oczywiście wszystkie prezentowane w niniejszej książce przykłady mogą być tworzone przy użyciu tego właśnie produktu.

Zintegrowane środowiska programistyczne są niewątpliwie bardzo wygodne w użyciu, jednak również bardzo drogie. Na szczęście istnieje także darmowy kompilator C# (*csc.exe*). Jest on częścią pakietu *.NET Framework SDK* (pakietu tego należy szukać pod adresem <http://msdn.microsoft.com>). Jest to kompilator uruchamiany w wierszu poleceń, nie oferuje więc żadnego wsparcia przy budowaniu aplikacji, jednak do naszych celów jest w zupełności wystarczający.

Co więcej, właśnie ten kompilator polecam do wykonywania prezentowanych ćwiczeń. Z dwóch powodów. Po pierwsze, nie trzeba za niego płacić, a po drugie, 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.

Najprostszy program

Na początku napiszmy najprostszy chyba program, którego zadaniem będzie wyświetlenie na ekranie dowolnego napisu. Nie jest to skomplikowane zadanie, pewne rzeczy będziemy jednak musieli przyjąć „na słowo”, dopóki nie poznamy definicji klas, każdy bowiem program napisany w C# składa się ze zbioru klas. Struktura programu wyświetlającego napis na ekranie powinna wyglądać następująco:

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

Takiej też 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 funkcji *Main()*. Taką instrukcją jest:

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

Ćwiczenie 1.1.

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

```
using System;

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

Plik stworzony poprzez wykonanie ćwiczenia 1.1 można zapisać na dysku pod nazwą *program.cs*. Oczywiście możliwe jest również nadanie innej, praktycznie dowolnej nazwy.



Wielkość liter w kodzie źródłowym ma znaczenie! Jeśli się pomylimy, komilacija się nie uda!

Kompilacja i uruchamianie

Kod z ćwiczenia 1.1 należy teraz skompilować i uruchomić. Korzystać będziemy z komplatora uruchamianego w wierszu poleceń — *csc.exe*. W najprostszym przypadku, jako parametr, należy podać nazwę pliku z kodem źródłowym, pisząc, na przykład:

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

nie należy 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 komplator przyjmie bez problemów również dowolny inny plik zawierający poprawny tekst programu.

Po komplikacji powstanie plik wynikowy *program.exe*, który można uruchomić z poziomu systemu, czyli tak jak każdą inną aplikację, o ile oczywiście został zainstalowany wcześniej pakiet *.NET Framework*.

Ćwiczenie 1.2.

Skompiluj i uruchom program napisany w ćwiczeniu 1.1.

Etap komplikacji oraz wynik działania programu widoczny jest na rysunku 1.1. Jak widać, o ile w kodzie źródłowym nie występują błędy, komplikator nie wyświetla jakichkolwiek informacji oprócz noty copyright, generuje natomiast plik wykonywalny *exe*. Komplikator *csc* umożliwia stosowanie różnych opcji umożliwiających ingerencję w proces komplikacji, są one pokazane w tabeli 1.1.

Rysunek 1.1.
Kompilacja i uruchomienie pierwszego programu w C#.

```

C:\>WINNT\System32\cmd.exe
D:\redakcja\helion\csharp\>csc program.cs
D:\redakcja\helion\csharp\>h:\winnt\Microsoft.NET\Framework\v1.0.3705\csc.exe p
rogram.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

D:\redakcja\helion\csharp\>program.exe
Mój pierwszy program!
D:\redakcja\helion\csharp\>

```

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.
/recurse:	-	maska	Komplikuje wszystkie pliki (z katalogu bieżącego oraz katalogów podrzędnych), których nazwa jest zgodna 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 (-) komplikację przyrostową.
/warnaserror	-	-	Włącza tryb traktowania ostrzeżeń jako błędów.
/warn:	/w:	0 – 4	Ustawia poziom ostrzeżeń.
/nowarn:	-	lista ostrzeżeń	Wyłącza generowanie podanych ostrzeżeń.
/help	/?	-	Wyświetla listę opcji.
/nologo	-	-	Nie wyświetla noty copyright.

Visual Studio

W naszych ćwiczeniach korzystać będziemy z darmowego kompilatora uruchamianego w wierszu poleceń CSC, dostępnego w pakiecie .NET SDK. Dla osób, które wolałby wykorzystać do nauki pakiet *Visual Studio* lub *Visual C#*, wykonamy kliknięcia ćwiczeń pokazujących, w jaki sposób stworzyć projekt i dokonać komplikacji.

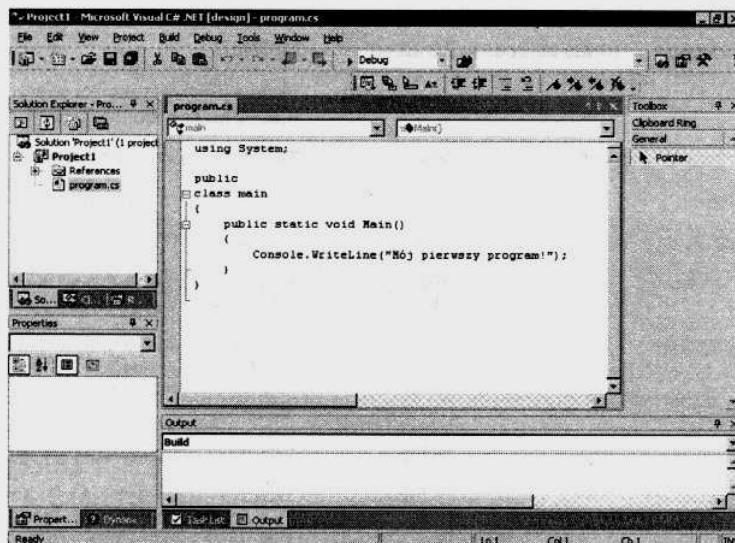
Ćwiczenie 1.3.

Uruchom Visual Studio (Visual C#), utwórz pusty projekt, a następnie dodaj do niego przygotowany plik program.cs. Dokonaj komplikacji kodu.

Po uruchomieniu pakietu musimy utworzyć nowy projekt (polecenia *File/New/Project*). Następnie wybieramy opcję *Empty Project* oraz, korzystając z menu *Project* i opcji *Add Existing Item* dodajemy do niego plik *program.cs*. Jeżeli zawartość dodanego pliku nie pojawi się na ekranie, można ją wyświetlić, klikając dwukrotnie jego nazwę w oknie *Solution Explorer* (rysunek 1.2). Plik *exe* tworzymy, wybierając z menu *Build* pozycję o tej samej nazwie (*Build*).

Rysunek 1.2.

Plik program.cs dodany do projektu w pakiecie Visual Studio .NET



Gdybyśmy chcieli pisać kod bezpośrednio w edytorze *Visual C#*, musimy postąpić inaczej niż w ćwiczeniu 1.2. Mamy do wyboru dwa sposoby — możemy napisać cały kod od zera lub też kazać wygenerować pakietowi szkielet aplikacji i dopiero ten szkielet wypełnić instrukcjami. Do ćwiczeń z niniejszej książki polecam raczej pierwszy sposób, jako że kod generowany automatycznie będzie się nieco różnił od prezentowanych dalej przykładów. Niemniej kolejne dwa ćwiczenia pokażą, jak zastosować oba sposoby.

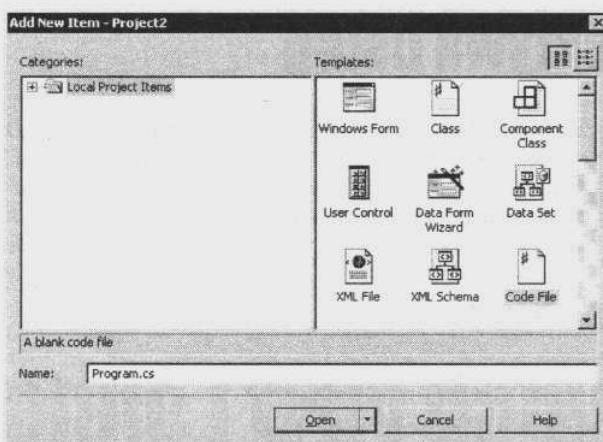
Ćwiczenie 1.4.

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

Po utworzeniu projektu, analogicznie jak w ćwiczeniu 1.3, należy z menu *Project* wybrać pozycję *Add New Item*. W kolejnym oknie zaznaczamy ikonę *Code File*, w polu *Name* wpisujemy nazwę pliku z kodem programu i klikamy *Open* (rysunek 1.3). Następnie w edytorze wprowadzamy instrukcje z ćwiczenia 1.1. Utworzony w ten sposób projekt komplujemy w sposób opisany w ćwiczeniu 1.3.

Rysunek 1.3.

Tworzenie nowego pliku z kodem źródłowym w Visual Studio

**Ćwiczenie 1.5.**

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 *File*, również wybieramy pozycje *New i Project*. Zamiast ikony *Empty Project* odszukujemy jednak *Console Application* i klikamy przycisk *OK*. Utworzony zostanie szkielet aplikacji (rysunek 1.4). Kod należy wpisać w miejscu oznaczonym *TODO: Add code application here*. Wpisujemy tu samą instrukcję:

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

Rysunek 1.4.

Wygenerowany przez Visual Studio szkielet aplikacji konsolowej

```
Class1.cs
ConsoleApplication1.Class1
Main(string[] args)

using System;
namespace ConsoleApplication1
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            // 
            // TODO: Add code to start application here
            //
        }
    }
}
```

Kompilacji i uruchomienia programu dokonujemy tutaj 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 nasz program będzie korzystał z klas zdefiniowanych w przestrzeni nazw *System*. Przypomina to nieco dyrektywę *import* znaną z Javy. Dzięki takiemu zapisowi kompilator wie, gdzie ma szukać klasy *Console* i metody *WriteLine*, które wykorzystaliśmy do wyświetlenie napisu na ekranie.

Rozdział 2.

Zmienne i typy danych

Typy danych

Zmienna jest to miejsce w programie, w którym możemy przechowywać jakieś dane, np. liczby czy ciągi znaków. Każda zmienna ma swoją nazwę, która ją jednoznacznie identyfikuje, oraz typ, który określa, jakiego rodzaju dane może ona przechowywać. Na przykład zmienna typu `integer` może przechowywać liczby całkowite, a zmienna typu `float` liczby rzeczywiste.

Typy danych możemy podzielić na następujące rodzaje:

- ❖ typy arytmetyczne,
- ❖ typ `boolean`,
- ❖ typ `string`,
- ❖ typ `object`,
- ❖ typy referencyjne.

Typy arytmetyczne

Rodzinę typów arytmetycznych możemy podzielić na typy całkowitoliczbowe (ang. *integral type*) oraz typy zmiennoprzecinkowe (ang. *floating-point type*). Pierwsze służą oczywiście do reprezentacji liczb całkowitych, drugie do reprezentacji 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 są w tabeli 2.1.

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 to U+FFFF	16-bitowy znak Unicode
short	od -32 768 do 32 767	16-bitowa liczba ze znakiem
ushort	od 0 do 65 535	16-bitowa liczba bez znaku
int	od -2 147 483 648 do 2 147 483 647	32-bitowa liczba ze znakiem
uint	od 0 do 4 294 967 295	32-bitowa liczba bez znaku
long	od -9 223 372 036 854 775 808 do 9 223 372 036 854 775 807	64-bitowa liczba ze znakiem
ulong	od 0 do 18 446 744 073 709 551 615	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).

Typy zmiennoprzecinkowe występują tylko w dwóch odmianach:

- ❖ float (pojedynczej precyzji),
- ❖ double (podwójnej precyzji).

Zakres oraz precyzja liczb, jakie możemy za ich pomocą przedstawić, przedstawione są w tabeli 2.2.

Tabela 2.2. Typy zmiennoprzecinkowe w C#

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

Typ boolean

Zmienne tego typu mogą przyjmować tylko dwie wartości: true i false (prawda i fałsz). Będą one używane przy konstruowaniu wyrażeń logicznych, do porównywania danych oraz wskazywania, czy dana operacja zakończyła się sukcesem.

Uwaga dla osób znających C albo C++: wartości true i false nie mają przełożenia na wartości liczbowe, jak w przypadku wymienionych języków. Oznacza to, że poniższy fragment kodu jest niepoprawny.

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

W takim wypadku błąd zostanie zgłoszony już na etapie komplikacji, nie istnieje bowiem domyślna konwersja z typu int do typu bool wymaganego przez instrukcję if.

Deklarowanie zmiennych

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

```
typ_zmiennej nazwa_zmiennej;
```

Po takim zadeklarowaniu zmieniona jest już gotowa do użycia, tzn. możemy jej przypisywać różne wartości bądź też wykonywać na niej różne operacje, np. dodawanie.

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

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

Jeżeli jednak chcemy równocześnie wyświetlić jakiś łańcuch znaków, 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 kolejną metodę:

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

W takiej sytuacji w miejsce ciągu znaków {0} zostanie wstawiona wartość zmiennej zm1, natomiast w miejsce ciągu znaków {1} wartość zmiennej zm2.

Ćwiczenie 2.1.

Zadeklaruj dwie zmienne całkowite i przypisz im dowolne wartości. Wyświetl wyniki na ekranie, tak jak widoczne jest to 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);
    }
}

```

Rysunek 2.1.

Wynik działania programu z ćwiczenia 2.1



Instrukcja `Console.WriteLine()` pozwala na wyprowadzenie ciągu znaków na ekran. Wartość zmiennej można przypisać już w trakcie deklaracji, 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 w dowolnym miejscu funkcji czy metody.

Ćwiczenie 2.2.

Zadeklaruj i jednocześnie zainicjalizuj dwie zmienne typu całkowitego. Wynik wyświetl 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);
    }
}

```

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

Rysunek 2.2.

Próba wykorzystania niezainicjowanej zmiennej powoduje błąd kompilacji

The screenshot shows a Windows Command Prompt window titled 'cmd.exe'. The command entered is 'csc prog.cs'. The output shows the following errors:

```
D:\> csc prog.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

prog.cs(11,36): error CS0165: Use of unassigned local variable 'i'
prog.cs(12,37): error CS0165: Use of unassigned local variable 'j'
prog.cs(13,37): error CS0165: Use of unassigned local variable 'k'

D:\>
```

Ćwiczenie 2.3.

Zadeklaruj kilka zmiennych typu całkowitego w jednym wierszu. Kilka 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("pierwsza liczba: " + pierwszaLiczba);
        Console.WriteLine("druga liczba: " + drugaLiczba);
        Console.WriteLine("zmienna i: " + i);
        Console.WriteLine("zmienna j: " + j);
        Console.WriteLine("zmienna k: " + k);
    }
}
```

Ćwiczenie 2.4.

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

```
using System;

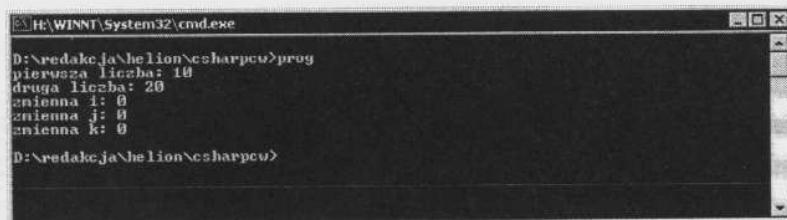
public class main
{
    public static void Main()
    {
        int pierwszaLiczba = 10, drugaLiczba = 20, i, j, k;
        i = 0;
        j = 0;
        k = 0;
        Console.WriteLine("pierwsza liczba: " + pierwszaLiczba);
        Console.WriteLine("druga liczba: " + drugaLiczba);
```

```

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

```

Rysunek 2.3.
Prawidłowo
zainicjowane
zmienne mogą zastać
użyte w programie



Przy nazywaniu zmiennych obowiązują pewne zasady. Otóż nazwa taka może składać się z dużych i małych liter oraz cyfr, ale nie może się zaczynać od cyfry. Nie należy również stosować polskich znaków diakrytycznych. Nazwa zmiennej powinna także odzwierciedlać funkcję pełnioną w programie. Na przykład, jeżeli określa ona liczbę punktów w jakimś zbiorze, to najlepiej ją nazwać *liczbaPunktow* lub nawet *liczbaPunktowWZbiorze*. Mimo że tak duga nazwa może wydawać się dziwna, poprawia jednak bardzo czytelność programu oraz ułatwia jego analizę. Naprawdę warto ten sposób stosować. Często przyjmuje się też, co również jest bardzo wygodne, że nazwę zmiennej rozpoczętym małą literą, a poszczególne człony tej nazwy (wyrazy, które się na nią składają) piszemy literą wielką. Dokładnie tak jak w powyższych przykładach.

Typy referencyjne

Typy referencyjne nazywane również odnośnikowymi lub obiektowymi służą do deklarowania zmiennych, które są odwołaniami do obiektów. Samymi obiektami zajmiemy się dopiero w rozdziale czwartym, w tej chwili przyjrzymy się tylko, w jaki sposób deklarujemy tego rodzaju zmienne. Zmienne te deklarujemy podobnie jak w przypadku zmiennych typów podstawowych, tzn. pisząc:

```

typ_zmiennej nazwa_zmiennej;
lub
typ_zmiennej nazwa_zmiennej_1, nazwa_zmiennej_2, nazwa_zmiennej_3;

```

W ten sposób zdeklarowaliśmy jednak jedynie tzw. odniesienie (ang. *reference*) do zmiennej obiektowej, a nie samą zmienną! Takiemu odniesieniu przypisana jest domyślnie wartość pusta (`null`), czyli praktycznie nie możemy wykonywać na niej żadnej operacji. Dopiero po utworzeniu odpowiedniego obiektu w pamięci możemy powiązać go z tak zadeklarowana zmienną. Jeśli zatem napiszemy, np.:

```

int a;
mamy gotową do użycia zmienną typu całkowitego. Możemy jej przypisać, na przykład,
wartość 10. Żeby jednak móc 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];
```

Zajmiemy się tym tematem bliżej w rozdziale piątym.

Typ string

Typ `string` służy do reprezentacji łańcuchów znakowych, inaczej napisów. Z pewnością docenią go wszyscy programujący w C i C++, gdzie takiego typu po prostu nie ma. Nie jest to jednak typ bezpośrednio wbudowany w język, tak jak ma to miejsce w Pascalu, ale, podobnie jak w Javie, jest typ referencyjny. Co prawda nie istnieje konieczność jawnego wołania konstruktora klasy `String`, czyli zmienne możemy deklarować, tak jak zmienne typów prostych:

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

Niemniej po takiej deklaracji utworzony zostanie obiekt klasy `String`, zatem na zmiennej `zmienna` możemy dokonywać dowolnych operacji możliwych do wykonania na klasie `String`.

Typ object

Typ `object` jest typem nadrzędnym, z którego wyprowadzone są wszystkie inne typy danych. Zawarty jest on w przestrzeni nazw `System`, co oznacza, że słowo `object` w rzeczywistości tłumaczone jest na `System.Object`. Nie będziemy zagłębiać się w niuanse we wnętrznej realizacji typów w C#, należy jedynie pamiętać, 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, na przykład, że można w stosunku do takiej zmiennej wywołać metody danej klasy np.:

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

Co więcej możliwe jest również zastosowanie konstrukcji:

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

Wartość null

Jest to pewien specjalny typ danych, który oznacza po prostu nic (`null`). Wartości te używamy, gdy chcemy wskazać, że dana zmienna referencyjna jest pusta, czyli nie został do niej przypisany żaden obiekt. Typ `null` występuje w większości obiektowych języków programowania, w przypadku Object Pascal (język, 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 wykonujemy za pomocą różnych operatorów, np. odejmowania, dodawania, przypisania itd. Operatory te możemy podzielić na następujące grupy:

- ❖ arytmetyczne,
- ❖ bitowe,
- ❖ logiczne,
- ❖ przypisania,
- ❖ porównania.

Operatory Arytmetyczne

Wśród tych operatorów znajdziemy działające standardowo:

- ❖ + — dodawanie,
- ❖ - — odejmowanie,
- ❖ * — mnożenie,
- ❖ / — dzielenie.

Ćwiczenie 2.5.

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 przypisujemy wynik odejmowania b - a, czyli zawierać ona będzie liczbę 15.

Kolejny krok to wyświetlenie wyników dotychczasowych działań i przypisań na ekranie. Następnie przypisujemy do zmiennej c wynik mnożenia a * b (czyli liczbę 250) i wartość tego działania również wypisujemy 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.

Ćwiczenie 2.6.

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

Kolejne operatory typu arytmetycznego to operator inkrementacji i dekrementacji. Operator inkrementacji, czyli zwiększenia, powoduje przyrost wartości zmiennej o jeden. Operator ten, zapisywany jako ++, może występować w formie przyrostkowej bądź przedrostkowej. Oznacza to, że jeśli mamy zmienną, która nazywa się x, formą przedrostkową będzie $++x$, natomiast przyrostkową — $x++$.

Oba te wyrażenia zwiększą wartość zmiennej x o jeden, jednak wcale nie są one sobie równoważne. Otóż operator $x++$ zwiększa wartość zmiennej po jej wykorzystaniu, natomiast $++x$ przed jej wykorzystaniem. Czasem takie rozróżnienie jest bardzo pomocne przy pisaniu programu.

Ćwiczenie 2.7.

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 programu 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 wartość zmiennej *x* jest wyświetlona (*x* = 2), a dopiero potem zwiększana o 1 (*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*. W wierszu ostatnim, ósmym, zwiększamy *y* o jeden (*y* = 6) i wyświetlamy na ekranie.

Operator dekrementacji `--` działa analogicznie z tym, że zamiast zwiększać wartości zmiennych zmniejsza je, oczywiście zawsze o jeden.

Ćwiczenie 2.8.

Zmień kod z ćwiczenia 2.7 tak, aby operator `++` został zastąpiony operatorem `--`. Następnie przeanalizuj jego działanie 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 zmienienna 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 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 zwiększamy wartość x o jeden ($x = -3$), a następnie przypisujemy tę wartość zmiennej y . W wierszu ostatnim, ósmym, zwiększamy y o jeden ($y = -4$) i wyświetlamy na ekranie.

Ostatnim operatorem arytmetycznym jest jednoargumentowy operator zmiany znaku, który zapisujemy znakiem minus (-). Działa on we wszystkim znany sposób, czyli zmienia znak danej liczby na przeciwny.

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

Ćwiczenie 2.9.

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);
    }
}
```

Drugim problemem jest to, co się stanie, jeżeli przekroczymy zakres jakiejś zmiennej. Pamiętamy na przykład, że zmienienna typu byte jest zapisywana na 8 bitach i przyjmuje wartości od 0 do 255 (patrz tabela 2.1). Spróbujmy zatem przypisać zmiennej tego typu wartość 256.

Ćwiczenie 2.10.

Zadeklaruj zmienną typu byte. Przypisz jej wartość 256. Spróbuj dokonać komplikacji otrzymanego kodu.

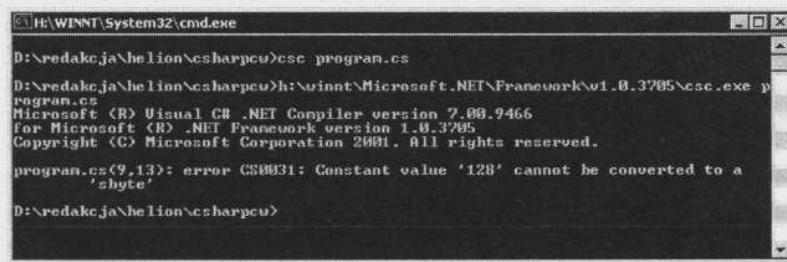
```
using System;

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

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

Rysunek 2.4.

Próba
przekroczenia
dopuszczalnego
zakresu zmiennej
powoduje błąd
komplikacji



Niestety, kompilator nie zawsze będzie w stanie wykryć tego typu błąd. Może się bowiem zdarzyć, że zakres przekroczymy nie w trakcie komplikacji, ale w trakcie wykonywania programu. Co się wtedy stanie? Przekonamy się o tym, wykonując kolejne ćwiczenie.

Ćwiczenie 2.11.

Zadeklaruj dwie zmienne typu long. Wykonaj operacje arytmetyczne przekraczające dopuszczalną wartość takiej zmiennej. Wyświetl wynik na ekranie.

```
using System;

public
class main
{
    public static void Main()
    {
        long b = (long) Math.Pow(2, 63), a;
        a = b + b;
        Console.WriteLine("a = " + a);
    }
}
```

Operacja (`long`) `Math.pow(2, 63)` oznacza podniesienie liczby dwa do potęgi 63, a następnie skonwertowanie wyniku (który jest liczba typu `double`) do typu `long`. Wynikiem działania `b + b` jest... *zero*. Okazuje się, że jeżeli wynik działania przekracza dopuszczalny zakres dla swojego typu, jest wykonywane (z matematycznego punktu widzenia) dodatkowe dzielenie modulo. W powyższym przykładzie jest to $a = (b + b) \% 2^{64}$. Oznacza to, że nie następuje przepełnienie i błąd aplikacji, tylko całość *zawija* się do początku.

Operatory bitowe

Operatory bitowe, jak sama nazwa wskazuje, służą do wykonywania operacji na bitach. Reprezentacja bitowa (dwójkowa) liczb jest bardzo wygodna dla komputera, niemniej dla nas, ludzi, przyzwyczajonych do systemu dziesiętnego, nie jest najwygodniejsza. Musimy jednak przypomnieć sobie podstawowe wiadomości o systemie dwójkowym. W systemie dziesiętnym do reprezentacji liczb używamy 10 cyfr od 0 do 9, w systemie szesnastkowym dodatkowo liter od A do F, a w systemie ósemkowym cyfr od 0 do 7. Nietrudno się zatem domyślić, że w systemie dwójkowym będziemy używać tylko dwóch cyfr: 0 i 1. Kolejnych 15 liczb w tym systemie wraz z odpowiednikami w systemie dziesiętnym jest przedstawionych w tabeli 2.3.

Tabela 2.3. Reprezentacja liczb w systemie dwójkowym i dziesiętnym

System dwójkowy	System dziesiętny
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Na tak zdefiniowanych liczbach możemy dokonywać znanych ze szkoły operacji bitowych AND, OR oraz XOR. Symbolem operatora AND jest znak ampersand (`&`), operatora OR znak pionowa kreska (`|`), natomiast operatora XOR znak strzałki w góre (`^`). Zestawienie tych operacji przedstawione jest w tabeli 2.4.

Tabela 2.4. Operatory bitowe

Rodzaj działania	Symbol w C#
bitowe AND	&
bitowe OR	
XOR	^
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 (dziesięć jest mniejsze od dwudziestu), zatem jego wartość logiczna jest równa true. W grupie tej wyróżniamy trzy: operatory logiczne AND (`&&`), logiczne OR (`||`) i logiczna negacja (`!`).

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

Operatory przypisania

Operacje przypisania są dwuargumentowe i powodują przypisanie wartości argumentu prawostronnego do argumentu lewostronnego. Najprostszym operatorem tego typu jest oczywiście klasyczny znak równości. Oprócz niego mamy jeszcze do dyspozycji operatory łączące klasyczne przypisanie z innym operatorem arytmetycznym bądź bitowym. Operatory przypisania występujące w C# przedstawione są w tabeli 2.5.

Tabela 2.5. 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

Operatory porównania służą oczywiście do porównywania argumentów. Wynikiem takiego porównania jest wartość logiczna `true` (jeśli jest ono prawdziwe) lub `false` (jeśli jest fałszywe). Do dyspozycji mamy operatory porównania zawarte w tabeli 2.6.

Tabela 2.6. Operatory porównania w C#

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

Operator warunkowy (?)

Operator warunkowy ma składnię następującą:

warunek ? wartość1 : wartość2;

Zapis ten należy rozumieć: jeżeli warunek jest prawdziwy, wyrażenie przybiera wartość1, w przeciwnym przypadku wyrażenie przybiera wartość2. Aby lepiej sobie to uzmysłowić, wykonamy proste ćwiczenie.

Ćwiczenie 2.12. 

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 oczywiście wiersz

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

który oznacza: jeżeli `x` jest równe 1, przypisz zmiennej `y` wartość 10, w przeciwnym przypadku przypisz zmiennej `y` wartość 20. Ponieważ zmienną `x` zainicjalizowaliśmy wartością 1, na ekranie zostanie wyświetlony ciąg znaków `y = 10`.

Priorytety operatorów

Skoro znamy już operatory, musimy jeszcze wiedzieć, w jakiej kolejności są one wykonywane. Ilustruje to tabela 2.7.

Tabela 2.7. Priorytety operatorów w C#

Grupa operatorów	Symbol
inkrementacja przyrostkowa	<code>++, --</code>
inkrementacja przedrostkowa, negacja	<code>++, --, ~, !</code>
mnożenie, dzielenie, dzielenie modulo	<code>*, /, %</code>
przesunięcia bitowe	<code><<, >></code>
relacyjne	<code><, >, <=, >=</code>
porównania	<code>==, !=</code>
bitowe AND	<code>&</code>
bitowe XOR	<code>^</code>
bitowe OR	<code> </code>
logiczne AND	<code>&&</code>
logiczne OR	<code> </code>
warunkowe	<code>?</code>
przypisania	<code>=, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, =</code>

Komentarze

Komentowanie tekstu źródłowego programu jest ważne ze względu na zachowanie jego przejrzystości. Zaniechanie tej czynności powoduje zwykle, że sam programista po pewnym czasie musi poświęcić dużo czasu na analizowanie swojego własnego kodu. Oczywiście w przypadku krótkich programów, takich, jakie prezentowane są w niniejszej książce, zazwyczaj nie ma potrzeby stosowania komentarzy, należy jednak widzieć, w jaki sposób można je stosować.

W C# mamy dwie nowe możliwości zastosowania komentarza. Obie są zapozyczone z języków programowania takich 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 tak zwany komentarz blokowy, jako że cały blok tekstu znajdujący się pomiędzy wymienionymi sekwencjami znaków jest ignorowany przez kompilator.

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

```
/*Komentarz blokowy  
/*Komentarz zagnieżdzony*/  
*/
```

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

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

Ćwiczenie 2.13.

Użyj komentarza składającego się z dwóch ukośników do opisania 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);  
    }  
}
```

Ćwiczenie 2.14.

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);  
    }  
}
```

Rozdział 3.

Instrukcje

Instrukcje warunkowe

Instrukcja if...else

Bardzo często w programie zachodzi potrzeba sprawdzenia jakiegoś warunku i, w zależności od tego, czy jest on prawdziwy czy fałszywy, dalsze wykonywanie różnych instrukcji. Do takiego sprawdzania służy właśnie instrukcja warunkowa if...else. Ma ona ogólną postać:

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

Spróbujmy zatem wykorzystać taką instrukcję do sprawdzenia, czy zmienna całkowita jest mniejsza od zera.

Ćwiczenie 3.1. ——————

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

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

```

{
    int zmienna = -5;
    if (zmienna < 0){
        Console.WriteLine("Zmienna jest mniejsza od zera.");
    }
    else{
        Console.WriteLine("Zmienna nie jest mniejsza od zera.");
    }
}

```

Spróbujmy teraz czegoś nieco bardziej skomplikowanego. Zajmijmy się klasycznym przykładem liczenia pierwiastków równania kwadratowego. Przypomnijmy, że jeśli mamy równanie w postaci:

$$A * x^2 + B * x + C = 0,$$

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

$$B^2 - 4 * A * C.$$

Jeżeli delta jest większa od zera, mamy dwa pierwiastki:

$$x_1 = (-B + \sqrt{\Delta}) / 2 * A$$

$$x_2 = (-B - \sqrt{\Delta}) / 2 * A.$$

Jeżeli delta jest równa zero, istnieje tylko jedno rozwiązanie, a mianowicie:

$$x = -B / 2 * A.$$

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

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

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

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

```

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

```

Ćwiczenie 3.2.

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

```
using System;

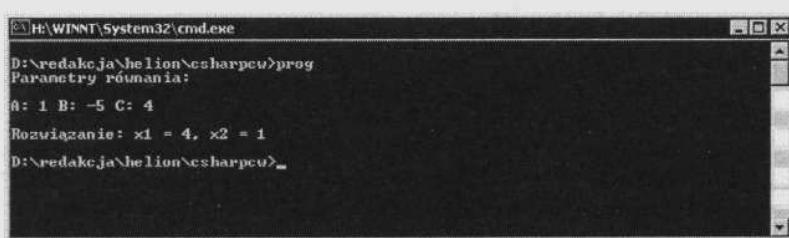
class Pierwiastek
{
    public static void Main(string[] args)
    {
        int parametrA = 1, parametrB = -5, parametrC = 4;

        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("To równanie nie ma 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.WriteLine("Rozwiązanie: x1 = " + wynik);
                    wynik = (- parametrB - Math.Sqrt(delta)) /
                            -2 * parametrA;
                    Console.WriteLine("x2 = " + wynik);
                }
            }
        }
    }
}
```

Rysunek 3.1.

Wynik działania programu obliczającego pierwiastki równania kwadratowego



Instrukcja if...else if

Jak pokazało nam ćwiczenie 3.2, instrukcję warunkową można zagnieźdzać, to znaczy po jednym `if` może występować kolejne, po nim następne itd. Jednakże taka budowa kodu powoduje, że przy wielu zagnieżdżeniach staje się on bardzo nieczytelny. Aby tego uniknąć, możemy wykorzystać instrukcję `if..else if`. Zamiast tworzyć niewygodną konstrukcję, taką jak przedstawiona poniżej:

```
if (warunek1){
    Instrukcje1
}
else{
    if (warunek2){
        instrukcje2
    }
    else{
        if (warunek3){
            instrukcje3
        }
        else{
            instrukcje4
        }
    }
}
```

Całość możemy 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
}
```

Ćwiczenie 3.3.

Zmodyfikuj program napisany w ćwiczeniu 3.2 tak, aby wykorzystywał on instrukcję `if...else if`.

```
using System;
class Pierwiastek
{
    public static void Main(string[] args)
    {
        int 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("To równanie nie ma rozwiązania");
            // 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

Jeżeli mamy do sprawdzenia wiele warunków, instrukcja switch pozwoli nam wygodnie zastąpić ciągi instrukcji if..else if. Jeśli mamy następujący fragment kodu:

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

możemy zastąpić poniższym:

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

Sprawdzamy tu po kolej, czy a nie jest przypadkiem równe jeden, potem dwa i w końcu trzy. Jeżeli tak, wykonywane są instrukcje po odpowiedniej klauzuli case. Jeżeli a nie jest równe żadnej z wymienionych liczb, wykonywane są instrukcje po słowie default. Instrukcja break powoduje wyjście z bloku switch. Czyli, jeśli a będzie równe jeden, zostaną wykonane instrukcje1, jeśli a będzie równe dwa, zostaną wykonane instrukcje instrukcje2, jeśli a będzie równe trzy, zostaną wykonane instrukcje instrukcje3. W przypadku gdyby a nie było równe ani jeden, ani dwa, ani trzy, zostaną wykonane instrukcje instrukcje4.

Ćwiczenie 3.4.

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.

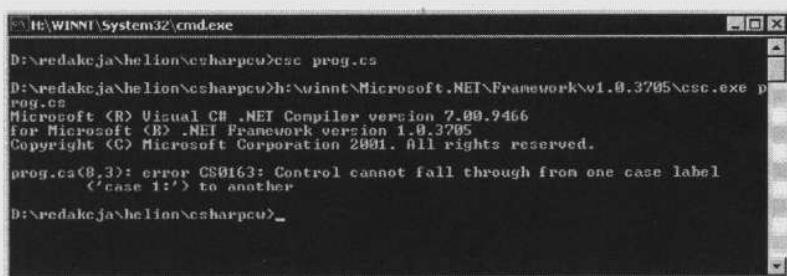
```
using System;

class Pierwiastek
{
    public static void Main(string[] args)
    {
        int a = 1;
        switch (a){
            case 1:
                Console.WriteLine("a = 1");
                break;
            case 2:
                Console.WriteLine("a = 2");
                break;
            default:
                Console.WriteLine("Zmienna a nie jest równa ani 1, ani 2.");
                break;
        }
    }
}
```

Warto w tym miejscu zauważyc, że w przypadku C#, odmiennie niż w językach takich jak C, C++ czy Java, nie możemy pominać instrukcji break, powodującej wyjście z instrukcji switch. Próba taka skończy się błędem komilacji (rysunek 3.2). Ścisłe rzecz biorąc, 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 komilacji



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 ona obecna, choć zaleca się jej stosowanie jedynie w przypadku bloków switch..case oraz wychodzenia z zagnieżdżonych pętli, w których ta sytuacji jest faktycznie użyteczna. Używamy jej w postaci:

goto etykieta;

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

```
for(int j = 0; j < 500; j++){
    if(j == 100) goto labell;
}
for(int i = 0; i < 1000; i++){
    labell:;
}
```

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 labell;
}
labell:
for(int i = 0; i < 1000; i++){
}
```

Taki kod jest już jak najbardziej poprawny.

Ćwiczenie 3.5.

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

```
using System;
public
class main
{
    public static void Main()
    {
        for(int j = 0; j < 500; j++){
            for(int i = 0; i < 1000; i++){
                if((j == 100) && (i > 200)){
                    goto labell;
                }
            }
        }
        return;
labell:
        Console.WriteLine("Pętla została przerwana!");
    }
}
```

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

Zauważmy, że przed etykietą znajduje się instrukcja `return` powodująca zakończenie działania funkcji `Main`. 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, także `return` nie jest niezbędne, ale już w *prawdziwej* aplikacji zapomnienie o tym drobnym z pozoru fakcie mogłoby przysporzyć nam wielu problemów.

Spróbujmy teraz wykorzystać instrukcję `goto` w drugim z zalecanych przypadków jej użycia, to znaczy w bloku `switch...case`.

Ćwiczenie 3.6.

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

```
using System;

public class main
{
    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 3.3. 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 klasy Random. Klasa ta służy właśnie do generowania losowych, a dokładniej pseudolosowych, liczb.

Rysunek 3.3.

Przykładowe wyniki działania programu ćwiczenia 3.6

```
D:\>program
Uylosowano wartosc i: 2
i jest mniejsze od 3.

D:\>program
Uylosowano wartosc i: 3
i jest mniejsze od 6.
Nie został wykonany ani blok case 1, ani blok case 2

D:\>program
Uylosowano wartosc i: 0
Nie został wykonany ani blok case 1, ani blok case 2
D:\>
```

Następnie w bloku switch..case rozpatrujemy 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

Pętle

Pętle w językach programowania pozwalają na wykonywanie powtarzających się czynności. Dokładnie w ten sam sposób działają one również w C#. Jeśli chcemy, na przykład, wypisać na ekranie 10 razy napis C#, to możemy zrobić to, pisząc 10 razy `Console.WriteLine("C#");`. Jeżeli jednak chcielibyśmy mieć, na przykład, 100 takich napisów, to, pomijając oczywiście sensowność takiej czynności, byłby to już problem. Na szczęście z pomocą przychodzą nam właśnie 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 ilości wykonania pętli. Wyrażenie warunkowe określa warunek, jaki musi być spełniony, aby dokonać kolejnego przejścia w pętli, wyrażenie modyfikujące używane jest zwykle do modyfikacji zmiennej będącej licznikiem.

Ćwiczenie 3.7.

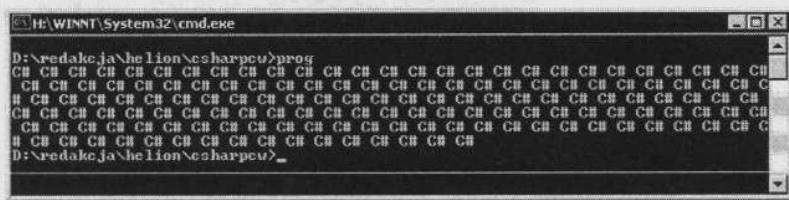
Wykorzystując pętlę typu *for*, napisz program wyświetlający na ekranie 100 razy napis C#.

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

Wynik działania tego prostego programu widoczny jest na rysunku 3.4.

Rysunek 3.4.

*Wynik działania pętli
for z ćwiczenia 3.7*



Zmienna *i* to tzw. *zmienna iteracyjna*, której na początku przypisujemy wartość 1 (*int i = 1*). Następnie, w każdym przebiegu pętli jest ona zwiększana o jeden (*i++*) oraz wykonywana jest instrukcja *Console.WriteLine("C# ")*; Wszystko trwa tak długo, aż *i* osiągnie wartość 100 (*i <= 100*).

Wyrażenie modyfikujące jest zwykle używane do modyfikacji zmiennej iteracyjnej. Takiej modyfikacji możemy jednak dokonać również wewnątrz pętli. Struktura tego typu wygląda następująco:

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

Ćwiczenie 3.8.

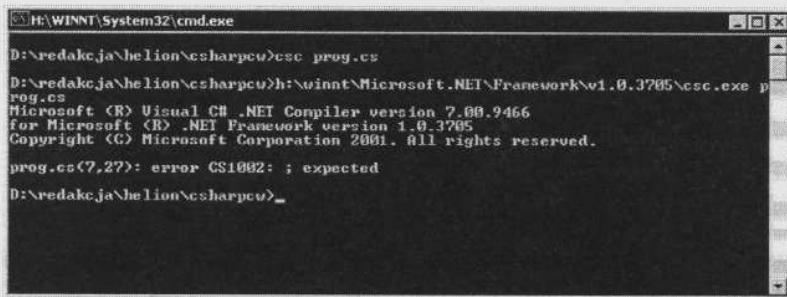
Zmodyfikuj pętle typu *for* z ćwiczenia 3.7 tak, aby wyrażenie modyfikujące znalazło się w bloku instrukcji *for*.

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

Zwracam uwagę, że mimo iż wyrażenie modyfikujące jest teraz wewnątrz pętli, średnik znajdujący się po `i <= 100` jest niezbędny! Jeśli o nim zapomnimy, kompilator zgłosi błąd (rysunek 3.5).

Rysunek 3.5.

Pominiecie średnika w pętli for powoduje błąd komplikacji



Kolejna ciekawą możliwością jest połączenie wyrażenia warunkowego i modyfikującego. Konkretnie należy spowodować, aby wyrażenie warunkowe było jednocześnie wyrażeniem modyfikującym. Tworzenie takich konstrukcji umożliwiają nam, na przykład, operatory inkrementacji i dekrementacji.

Ćwiczenie 3.9.

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

```
using System;

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

W podobny sposób jak w poprzednich przykładach możemy się w pozbyć również wyrażenia początkowego. Należy je przenieść przed pętlę. Schematyczna konstrukcja wygląda następująco:

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

Ćwiczenie 3.10.

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

```
using System;
class main
{
    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. Jest to jak najbardziej możliwe!

Ćwiczenie 3.11.

Zmodyfikuj pętle typu for w taki sposób, aby wyrażenie początkowe znalazło się przed pętlą, natomiast wyrażenie modyfikujące i warunkowe wewnętrz pętli.

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

Jak można zauważyć, taka pętla też jest możliwa. Przypominam raz jeszcze, że średniki (tym razem już dwa) w nawisach po for są niezbędne! Warto też zwrócić uwagę na zmianę kierunku nierówności. Wcześniej sprawdzaliśmy, czy *i* jest mniejsze bądź równe 100, a teraz, czy jest większe bądź równe. Dzieje się tak dlatego, że we wcześniejszych ćwiczeniach sprawdzaliśmy, czy pętla ma się dalej wykonywać, natomiast w obecnym, czy ma się zakończyć.

W zasadzie, zamiast nierówności moglibyśmy napisać również `if (i++ == 100) break;`. Przy okazji nauczyliśmy się, w jaki sposób wykorzystuje się instrukcję `break` w przypadku pętli. Służy ona oczywiście do natychmiastowego przerwania wykonywania pętli.

Kolejna przydatna instrukcja, `continue`, powoduje rozpoczęcie kolejnej iteracji — w miejscu jej wystąpienia wykonywanie bieżącej iteracji jest przerywane i rozpoczyna się kolejny obieg. Najlepiej zobaczyć to na konkretnym przykładzie.

Ćwiczenie 3.12.

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

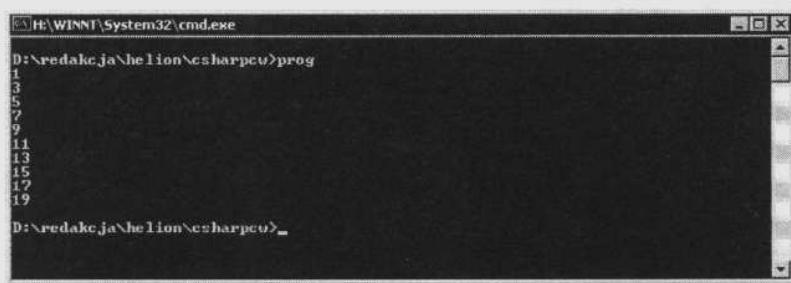
```
using System;

class main
{
    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 3.6. Przypominam, że $\%$ to operator dzielenia modulo — dostarcza on resztę z dzielenia. W każdym zatem przebiegu sprawdzamy, czy i modulo 2 nie jest przypadkiem równe 0. Jeśli jest (zatem i jest podzielne przez 2), przerwamy bieżącą iterację instrukcją `continue`. W przeciwnym przypadku (i modulo 2 jest różne od zera) wykonujemy instrukcję `Console.WriteLine(i)`. Ostatecznie na ekranie otrzymamy wszystkie liczby od jeden do dwudziestu, które nie są podzielne przez dwa.

Rysunek 3.6.

Program wyświetlający
liczby od 1 do 20
niepodzielne przez 2



Ten sam program można by oczywiście napisać bez użycia instrukcji `continue`. Jak tego dokonać, pokaże nam kolejne ćwiczenie.

Ćwiczenie 3.13.

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

```
using System;

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

Pętla while

O ile pętla typu for służyła raczej do wykonywania z góry znanej ilości operacji, w przypadku pętli while zwykle nie jest ona znana. Nie jest to oczywiście obligatoryjne. Tak naprawdę pętlę while można napisać tak, by była dokładnym funkcjonalnym odpowiednikiem pętli for, a pętle for tak, by była odpowiednikiem pętli while. Ogólna konstrukcja pętli typu while jest następująca:

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

Instrukcje są wykonywane tak długo, dopóki wyrażenie warunkowe jest prawdziwe. Oznacza to, że gdzieś w pętli musi nastąpić modyfikacja warunku, bądź też instrukcja break. Inaczej pętla będzie się wykonywała w nieskończoność!

Ćwiczenie 3.14.

Używając pętli typu while, napisz program wyświetlający na ekranie 100 razy napis C#.

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

Ćwiczenie 3.15.

Zmodyfikuj kod z ćwiczenia 3.14 tak, aby wyrażenie warunkowe zmieniało jednocześnie wartość zmiennej i.

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

Ćwiczenie 3.16.

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

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

Zauważmy, że w tym przypadku nie możemy skorzystać z konstrukcji takiej jak w poprzednim ćwiczeniu. Tym razem zmienna iteracyjna *i* musi być modyfikowana we wnętrzu pętli. Dzieje się tak dlatego, że wewnątrz korzystamy z jej wartości. Gdybyśmy zatem napisali pętlę tą w postaci *while (i++ <= 20)*, otrzymalibyśmy na ekranie liczby od 2 do 21 niepodzielne przez 2, co byłoby niezgodne z założeniami programu.

Pętla do while

Oprócz przedstawionych dotychczas istnieje jeszcze jedna odmiana pętli. Mianowicie *do...while*. Jej konstrukcja jest następująca:

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

Zapis ten należy rozumieć jako: wykonuj instrukcje, dopóki warunek jest prawdziwy. Spróbujmy zatem wykonać zadanie przedstawione ćwiczeniu 3.14, ale korzystając z pętli typu *do...while*.

Ćwiczenie 3.17.

Korzystając z pętli do...while, napisz program wyświetlający na ekranie 100 razy napis C#.

```
using System;
class main
{
    public static void Main(string[] args)
    {
        int i = 1;
        do{
            Console.Write("C# ");
        }
    }
}
```

```
        }
    while (i++ != 10);
}
```

Wydawać by się mogło, 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. Otóż w przypadku pętli `do...while` instrukcje wykonane będą co najmniej jeden raz, nawet jeśli warunek jest na pewno fałszywy. Dzieje się tak oczywiście dlatego, że sprawdzenie warunku zakończenia pętli odbywa się dopiero po jej pierwszym przebiegu.

Ćwiczenie 3.18.

Zmodyfikuj kod z ćwiczenia 3.17 w taki sposób, aby wyrażenie warunkowe na pewno było fałszywe. Zaobserwuj wyniki działania programu.

```
using System;

class main
{
    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. Prawda?

Wprowadzanie danych

Wiemy, 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 dokonywania konwersji typów danych.

Argumenty wiersza poleceń

Przekazywanie danych do aplikacji jako argumentów w wierszu poleceń przy wywoływaniu programu jest dobrze znanym większości programistów sposobem. Taka możliwość występuje prawdopodobnie w większości 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 string. Ponieważ w C#, podobnie jak w Javie, tablice są obiektami, nie ma potrzeby dodatkowego przekazywania parametru określającego liczbę argumentów, jak ma to miejsce w C i C++. Jej rozmiar określany jest parametrem Length (dalejsze informacje o tablicach znajdują się w rozdziale piątym).

Ćwiczenie 3.19.

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

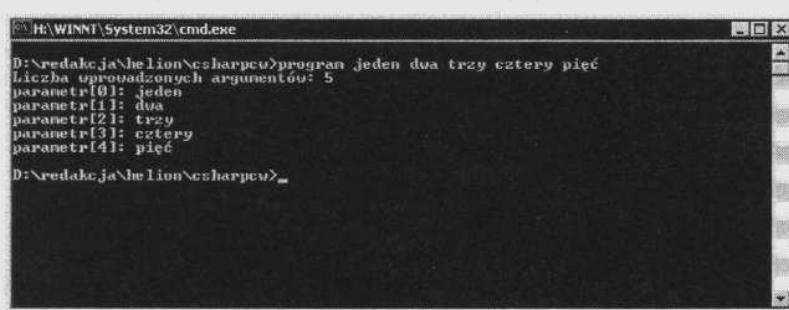
```
using System;

public class main
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Liczba wprowadzonych argumentów: {0}", args.Length);
        for(int i = 0; i < args.Length; i++){
            Console.WriteLine("parametr[{0}]: {1}", i, args[i]);
        }
    }
}
```

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

Rysunek 3.7.

Program wyświetlający argumenty podane w wierszu poleceń



Przy pracy z argumentami przekazywanymi z wiersza poleceń napotkamy na pewien problem. Założymy bowiem, że chcielibyśmy 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 polecen!");
            return;
        }
        Console.WriteLine("Wynikiem działania jest: {0}", args[0] + args[1]);
    }
}
```

Oczywiście nie jest ono prawidłowe, gdyż dokonujemy tutaj operacji na dwóch ciągach znaków, a nie dwóch liczbach! Zatem wynikiem dodawania, na przykład, 12 i 8 będzie w powyższym programie 128, zamiast spodziewanego 20. Aby aplikacja działała poprawnie, należy najpierw dokonać konwersji argumentów z typu string do typu int, a dopiero potem wykonywać działanie. Konwersji takiej dokonamy przy wykorzystaniu statycznej metody Parse w postaci:

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

Ćwiczenie 3.20.

Napisz program dokonujący dodawania dwóch liczb podanych jako parametry w wierszu polecen.

```
using System;
public class main
{
    public static void Main(string[] args)
    {
        int a, b;
        if(args.Length < 2){
            Console.WriteLine("Należy podać dwa argumenty w wierszu polecen!");
            return;
        }
        try{
            a = Int32.Parse(args[0]);
            b = Int32.Parse(args[1]);
        }
        catch(Exception){
            Console.WriteLine("Jeden z argumentów nie jest poprawną liczbą!");
            return;
        }
        Console.WriteLine("Wynikiem działania jest: {0}", a + b);
    }
}
```

Konwersji z typu `string` do typu `int` dokonuje wspomniana już instrukcja `Int32.Parse`. Dodatkowo sprawdzamy również, czy konwersja ta zakończyła się sukcesem poprzez zastosowanie bloku `try...catch`. Dokładniejsze wytlumaczenie tej konstrukcji znajduje się w rozdziale szóstym, w którym opisane jest stosowanie wyjątków.

Przypomnijmy sobie teraz aplikację napisaną 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, musielibyśmy ponownie komplilować program. Na pewno nie było to zbyt wygodne. Teraz, kiedy wiemy już, w jaki sposób stosować argumenty, podając je w wierszu poleceń, i wiemy, w jaki sposób dokonać konwersji danych, możemy pokusić się o spore usprawnienie tamtych aplikacji.

Ćwiczenie 3.21.

Napisz program obliczający pierwiastki równania kwadratowego, w którym parametry równania są wprowadzane w wierszu poleceń (rysunek 3.8).

```
using System;

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

        if(args.Length < 3){
            Console.WriteLine("Wywołanie programu: program parametr1
                ->parametr2 parametr3");
            return;
        }

        try{
            parametrA = Int32.Parse(args[0]);
            parametrB = Int32.Parse(args[1]);
            parametrC = Int32.Parse(args[2]);
        }
        catch(Exception){
            Console.WriteLine("Jeden z parametrów równania nie jest poprawna
                ->liczbą całkowitą!");
            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.");
            }
            else{
                double pierwiastek = Math.Sqrt(delta);
                double x1 = (-parametrB - pierwiastek) / (2 * parametrA);
                double x2 = (-parametrB + pierwiastek) / (2 * parametrA);

                Console.WriteLine("Pierwiastki równania:");
                Console.WriteLine("x1 = " + x1);
                Console.WriteLine("x2 = " + x2);
            }
        }
    }
}
```

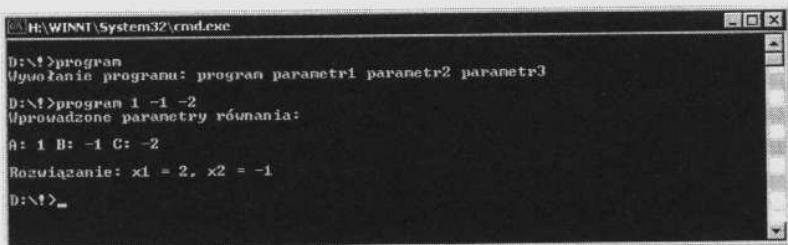
```

        Console.WriteLine("To równanie nie ma 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);
    }
}
}

```

Rysunek 3.8.

Równania kwadratowe o parametrach podanych w wierszu poleceń



Instrukcja ReadLine

Wprowadzanie danych do programu w wierszu poleceń nie zawsze jest wygodne. Często chcielibyśmy wykonywać tę czynność już w trakcie działania programu. Pomoże nam w tym instrukcja `Console.ReadLine()`, która zwraca wprowadzoną przez użytkownika jedną linię tekstu (ciąg znaków zakończony znakiem końca linii).

Ćwiczenie 3.22. Wózki

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;

class Pierwiastek
{
    public static void Main()
    {
        string s;
        while(!(s = Console.ReadLine()).Equals("quit"))
            Console.WriteLine(s);
    }
}
```

Warunek w pętli `while` zapisaliśmy w bardzo skondensowanej formie. Kolejność wykonywania instrukcji jest tu następująca:

1. Odczytanie linii znaków z konsoli (`Console.ReadLine()`).
2. Przypisanie odczytanej wartości do zmiennej `s` (`s=`).
3. Wywołanie metody `Equals` na rzecz obiektu wskazywanego przez `s` (`Equals("quit")`).

Należy w tym miejscu zwrócić również uwagę, że taki zapis może w pewnych sytuacjach spowodować błędów w działaniu aplikacji, konkretnie wygenerowanie wyjątku `NullReferenceException`. Dlaczego? Otóż może się zdarzyć, że metoda `ReadLine()` zamiast ciągu znaków zwróci wartość `null`. W takim wypadku wartością przypisaną do zmiennej `s` również będzie `null`. Skoro tak, nie będzie możliwe wykonanie metody `Equals`, stąd też wygenerowanie wyjątku.

Zapobiec takiej sytuacji można na dwa sposoby. Albo rozbijając warunek pętli `while` na kilka oddzielnych instrukcji i sprawdzając, czy `s` nie jest równe `null`, albo przez odwrócenie kolejności wykonywania instrukcji, czyli zamiast pisać

```
(s = Console.ReadLine()).Equals("quit")
```

można zastosować konstrukcję

```
"quit".Equals(s = Console.ReadLine())
```

Ćwiczenie 3.23. ——————

Popraw kod z ćwiczenia 3.22, usuwając warunek z pętli while, i przenieś go do wnętrza pętli.

```
using System;

class Pierwiastek
{
    public static void Main()
    {
        string s = "";
        while(true){
            s = Console.ReadLine();
            if(s != null && !s.Equals("quit")){
                Console.WriteLine(s);
            }
            else{
                break;
            }
        }
    }
}
```

Skoro potrafimy już wykorzystywać instrukcję `ReadLine()`, możemy poprawić program do obliczania pierwiastków równania kwadratowego tak, aby parametry były wprowadzane w trakcie działania programu. Będzie on teraz prosił użytkownika o podanie kolejnych liczb i podstawał je pod zmienne `parametrA`, `parametrB` i `parametrC`.

Oczywiście przed przypisaniem danych do wspomnianych zmiennych musimy dokonać konwersji z typu `string` na typ `int`. Co jednak powinniśmy zrobić w sytuacji, kiedy użytkownik nie poda prawidłowej liczby, ale, na przykład, wpisze dowolną kombinację znaków? Najlepiej byłoby poprosić o ponowne wprowadzenie parametru. Jak tego dokonać? Najwygodniej będzie skorzystać z pętli `while` lub `do...while` i prosić użytkownika o wprowadzanie liczby tak długo, dopóki nie poda poprawnej.

Skoro jednak mamy trzy zmienne, a tym samym trzy parametry do wprowadzenia, nie ma sensu pisać trzech pętli wyglądających praktycznie tak samo. Lepiej utworzyć dodatkową funkcję, której zadaniem będzie właśnie dostarczenie prawidłowej liczby całkowitej wprowadzonej przez użytkownika.

Ćwiczenie 3.24. ——————

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

```
using System;

class Pierwiastek
{
    public static int pobierzLiczbe(string param)
    {
        int liczba = 0;
        bool sukces;
        do{
            Console.WriteLine("Proszę podać {0} parametr równania:", param);
            try{
                liczba = Int32.Parse(Console.ReadLine());
                sukces = true;
            }
            catch(Exception){
                Console.WriteLine("Podany parametr nie jest prawidłową
→ liczbą całkowitą!");
                sukces = false;
            }
        }
        while(!sukces);
        return liczba;
    }
    public static void Main(string[] args)
    {
        int parametrA, parametrB, parametrC;

        parametrA = pobierzLiczbe("pierwszy");
        parametrB = pobierzLiczbe("drugi");
        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("To równanie nie ma rozwiązania");
                // 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);
            }
        }
    }
}
```

Na tym w zasadzie moglibyśmy zakończyć ćwiczenia z wprowadzania danych i rozwiązywania równań kwadratowych, ale spróbujmy wykonać jeszcze jeden przykład. Zauważmy bowiem, że wygodnie byłoby, aby nasz program umożliwiał zarówno podawanie parametrów w wierszu poleceń, jak i w trakcie swojego działania. Dzięki temu użytkownik mógłby wybrać bardziej wygodny dla niego sposób. Co więcej, jeśli przy podawaniu danych w wierszu poleceń pomyli się, aplikacja pozwoli na ponowne ich wprowadzenie.

Musimy zatem połączyć kod z ćwiczenia 3.21 z kodem z ćwiczenia 3.24. Dodatkowo powinniśmy wprowadzić jeszcze jedno usprawnienie. Do tej pory zakładaliśmy bowiem, że parametry równania muszą być liczbami rzeczywistymi. Nie ma jednak żadnego powodu, aby dalej utrzymywać takie ograniczenie. Pozwólmy, aby nasza aplikacja potrafiła również rozwiązywać równania, w których parametrami są liczby rzeczywiste.

Dodatkowymi zmianami będzie więc zmiana typu zmiennych parametrA, parametrB i parametrC z int na double oraz skorzystanie z metody Parse klasy Double.

Ćwiczenie 3.25.

Napisz program rozwiązywający 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)
    {
        double liczba = 0;
        bool sukces;
        do{
```

```
Console.WriteLine("Proszę podać {0} parametr równania:", param);
try{
    liczba = Double.Parse(Console.ReadLine());
    sukces = true;
}
catch(Exception){
    Console.WriteLine("Podany parametr nie jest prawidłową liczbą!");
    sukces = false;
}
}
while(!sukces);
return liczba;
}
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");

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

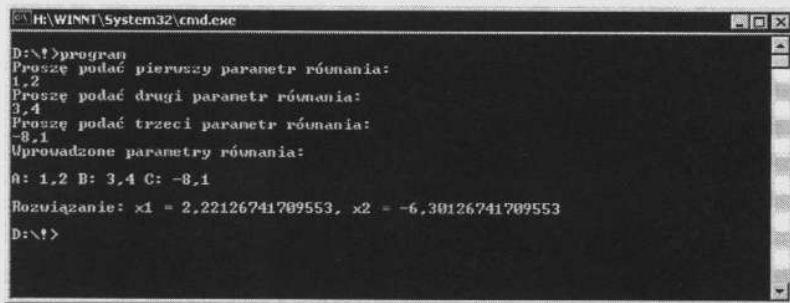
        if (delta < 0){
            Console.WriteLine("Delta < 0.");
            Console.WriteLine("To równanie nie ma rozwiązań
                ↪w zbiorze liczb rzeczywistych");
        }
        else if (delta == 0){
            wynik = - parametrB / 2 * parametrA;
            Console.WriteLine("Rozwiązanie: 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);
    }
}
```

Nasz program potrafi już skorzystać z liczb rzeczywistych (rysunek 3.9). Jeżeli w wierszu poleceń podane zostały trzy parametry i każdy z nich jest prawidłową liczbą całkowitą, zostaną one użyte do rozwiązyania równania. Jeśli jednak parametrów jest mniej lub przy wprowadzaniu któregośkolwiek z nich został popełniony błąd, aplikacja poprosi o ponowne ich wprowadzenie.

Rysunek 3.9.

Parametry
równania mogą być
podawane w postaci
liczb rzeczywistych



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

Rozdział 4.

Programowanie obiektowe

Klasy

Każdy program z C# składa się z klas. Dotychczas używaliśmy tylko jednej klasy, nazywała się ona `main`. Przypomnijmy sobie nasz pierwszy program, wyświetlający na ekranie napis. Jego kod wyglądał następująco:

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

Założyliśmy wtedy, że szkielet kolejnych aplikacji, na których będziemy się uczyć struktur języka programowania, ma właśnie tak wyglądać. Teraz nadszedł czas, aby wyjaśnić, dla czego. Wszystko stanie się jasne po przeczytaniu tego właśnie rozdziału.

W klasach zawarty jest kod wykonywalny, który realizuje przypisane mu zadania. Klassy 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 nierzozumiałe, nic należy się tym jednak przejmować, w rzeczywistości nie ma w tym nic skomplikowanego. Skoro klasa definiuje typ obiektu, przypomnijmy sobie, czym jest ten typ. Otóż, typ określa rodzaj wartości, jakie może przyjmować dany byt programistyczny, na przykład zmienna. Jeśli zatem typem zmiennej jest `int`, to może ona przyjmować wartości typu `int`, czyli liczb całkowitych z danego przedziału.

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

Ćwiczenie 4.1.

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

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

Składowymi klas są pola i metody. Pola to zmienne, w których możemy przechowywać dane dotyczące klasy. Metody to kod wykonywalny, który może wykonywać różne operacje. W naszym przypadku klasa zawiera tylko dwa pola `x` i `y`, które opisują położenie naszego punktu na ekranie.

Zadeklarujmy teraz zmienną typu `Punkt`. Jest to bardzo proste:

```
Punkt nowyPunkt;
```

Jak widać, podajemy najpierw nazwę klasy, potem nazwę zmiennej, podobnie jak dla zmiennych poznanych już wcześniej typów podstawowych. Tak zadeklarowana zmienna jest jednak pusta (dokładniej zadeklarowaliśmy w ten sposób jedynie referencję, inaczej odniesienie, do obiektu klasy `Punkt`), musimy do niej przypisać obiekt klasy `Punkt`. Aby to zrobić, musimy go najpierw utworzyć. Wykorzystujemy w tym celu operator `new` w postaci:

```
new NazwaKlasy();
```

Czyli w naszym 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 w C++ jest inaczej! To znaczy, już napisanie:

```
Punkt nowyPunkt;
```

spowodowałoby utworzenie samego obiektu typu `Punkt`, a nie tylko referencji do niego. Jeżeli ktoś jest przyzwyczajony do C++, powinien o tym pamiętać, gdyż jest to częstą przyczyną popełniania błędów.

Metody

Metody, jak już wiemy, zawierają kod operujący na polach danej klasy bądź też na dostarczonych z zewnątrz danych. Metody wywołujemy za pomocą operatora kropka (.), poprzedzając je nazwą zmiennej odnośnikowej (referencyjnej). Wygląda to następująco:

```
nazwa_zmiennej.nazwa_metody(parametry metody);
```

W nawiasach okrągłych po nazwie metody podajemy jej parametry. W ten sposób możemy przekazać metodzie jakieś dane. W przypadku naszej klasy Punkt moglibyśmy stworzyć dwie metody, które zwracaliby odpowiednio współrzędną *x* i współrzędną *y* punktu.

Ćwiczenie 4.2.

*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;  
    }  
}
```

W tej chwili, aby uzyskać informację o wartości współrzędnej *x*, możemy napisać:

```
punkt.getX();
```

zakładając oczywiście, że został wcześniej utworzony obiekt o nazwie punkt.

Ćwiczenie 4.3.

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

```
public  
class Punkt  
{  
    public int x;  
    public int y;  
    public void ustawWspolredne(int wspX, int wspY)  
    {  
        x = wspX;  
        y = wspY;  
    }  
    Punkt pobierzWspolredne()
```

```

{
    Punkt punkt = new Punkt();
    punkt.x = x;
    punkt.y = y;
    return punkt;
}

```

Klasa ta zawiera dwa pola typu int o nazwie x i y, które będą nam służyły do przechowywania współrzędnych danego punktu. Mamy również dwie metody. Jedna zajmuje się ustawieniem pól danego obiektu, druga pobiera wartości. Metoda ustawWspolrzedne() jest typu void, co oznacza, że nie zwraca ona żadnej wartości. Ma natomiast dwa parametry, wspX i wspY, oba typu int. W ciele 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. Do pól danego obiektu dostajemy się tak samo jak w przypadku metod, tzn. za pomocą operatora kropka (.).

Metoda druga — pobierzPunkt — jest typu Punkt. Oznacza to, że zwraca ona referencję do 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 polem tego obiektu przypisywane są pola x i y obiektu bieżącego. Obiekt punkt (dokładniej referencja do tego obiektu) jest zwracany instrukcją return, która powoduje przerwanie wykonywania danej metody i ewentualnie zwrócenie przez nią jakiejś wartości.

Najwyższy czas wykorzystać tak stworzony kod w konkretnym przykładzie.

Ćwiczenie 4.4.

Napisz program wykorzystujący obiekty klasy Punkt.

```

using System;
public class main
{
    public static void Main()
    {
        Punkt punkt = new Punkt();
        Punkt pomocniczyPunkt;
        pomocniczyPunkt = punkt.pobierzWspolrzedne();

        Console.WriteLine("Przed ustawieniem wartości:");
        Console.WriteLine("WspółrzędnaX = " + pomocniczyPunkt.x);
        Console.WriteLine("WspółrzędnaY = " + pomocniczyPunkt.y);

        punkt.ustawWspolrzedne(1, 2);
        pomocniczyPunkt = punkt.pobierzWspolrzedne();
    }
}
```

```
Console.WriteLine("\nPo ustawieniu wartości:");
Console.WriteLine("WspółrzędnaX = " + pomocniczyPunkt.x);
Console.WriteLine("WspółrzędnaY = " + pomocniczyPunkt.y);
}
}

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;
    }
}
```

Jak zatem działa nasz najnowszy program (efekt widoczny jest na rysunku 4.1)? Na początku tworzymy referencję o nazwie punkt i przypisujemy jej nowy obiekt klasy Punkt oraz drugą referencję pomocniczyPunkt. Ponieważ metoda pobierzWspolrzedne() zwraca referencję do obiektu typu Punkt, możemy dokonać przypisania pomocniczyPunkt = punkt.pobierzWspolrzedne();.

Rysunek 4.1.
Wynik działania
programu
z ćwiczenia 4.4

```
D:\>program
Przed ustawieniem wartości:
WspółrzędnaX = 0
WspółrzędnaY = 0

Po ustawieniu wartości:
WspółrzędnaX = 1
WspółrzędnaY = 2

D:\>
```

Zauważmy jednak, że oznacza to również, iż tak naprawdę zmienna pomocniczyPunkt wcale nie jest nam potrzebna. Moglibyśmy równie dobrze dokonać odwołania bezpośredniego. Modyfikacji należy oczywiście poddać metodę Main.

Ćwiczenie 4.5.

Zmodyfikuj kod metody Main tak, aby nie było konieczności użycia zmiennej pomocniczyPunkt.

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

```

Punkt punkt = new Punkt();

Console.WriteLine("Przed ustawieniem wartości:");
Console.WriteLine("WspółrzędnaX = " + punkt.pobierzWspolrzedne().x);
Console.WriteLine("WspółrzędnaY= " + punkt.pobierzWspolrzedne().y);

punkt.ustawWspolrzedne (1, 2);

Console.WriteLine("\nPo ustawieniu wartości:");
Console.WriteLine("WspółrzędnaX = " + punkt.pobierzWspolrzedne().x);
Console.WriteLine("WspółrzędnaY = " + punkt.pobierzWspolrzedne().y);
}
}

```

Dlaczego jednak dokonujemy tej, na pozór karkołomnej, konstrukcji, tworząc najpierw w metodzie `pobierzWspolrzedne()` nowy obiekt typu `Punkt`, przypisując mu odpowiednie wartości `x` i `y` i zwracając dopiero nowy byt? Odpowiedź jest prosta. Nie możemy na raz zwrócić współrzędnej `x` i współrzędnej `y`. Metoda może bowiem zwracać tylko jedną zmienianą typu podstawowego lub referencyjnego (odnośnikowego).

Możemy co najwyżej napisać dwie dodatkowe metody, które będą osobno zwracały wartość `x`, a osobno wartość `y`, tak jak zrobiliśmy to w ćwiczeniu 3.2. Podobnie można stworzyć dwie metody ustawiające osobno wartości `x` i `y`.

Może to okazać się bardzo przydatne, gdy gdzieś w programie będziemy chcieli zmodyfikować tylko jedną ze współrzędnych. Założmy, że chcemy zmodyfikować tylko `x`, ustawiając jego wartość na 5. W obecnej sytuacji musielibyśmy zastosować konstrukcję:

```
punkt.ustawWspolrzedne (5, punkt.pobierzWspolrzedne.y);
```

Dopiszmy więc brakujące metody, niech będą to: `ustawX`, `ustawY`, `pobierzX`, `pobierzY`.

Ćwiczenie 4.6.

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

```

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

Oprócz przedstawionych w ostatnim ćwiczeniu, istnieje jeszcze jedna możliwość równoczesnego uzyskania wartości x i y — można metodzie `pobierzPunkt()` przekazać parametr. Wyglądałoby to następująco:

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

Zatrzymajmy się tu na chwilę. Przecież jeżeli dopiszemy do klasy `Punkt` taka metodę, to będzie ona zawierała dwie metody o takiej samej nazwie — `pobierzWspolrzedne()`. Czy jest to dopuszczalne? Otóż tak, pod jednym jednak warunkiem — metody te muszą się od siebie różnić parametrami wywołania. Sytuację taką nazywamy przeciążaniem metod lub funkcji. W naszym przypadku jedna metoda nie ma żadnych parametrów wywołania, druga jako parametr przyjmuje referencję na obiekt typu `Punkt`. Wszystko zatem jest w porządku.

Ćwiczenie 4.7.

Dolacz do klasy `Punkt` przeciążoną metodę `pobierzWspolrzedne`.

```
public
class Punkt
{
    public int x, 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;
    }
    public void ustawX (int wspX)
    {
        x = wspX;
    }
    public void ustawY (int wspY)
    {
        y = wspY;
    }
    public int pobierzX ()
    {
        return x;
    }
    public int pobierzY ()
    {
        return y;
    }
}
```

Ćwiczenie 4.8. 

Napisz klasę main korzystającą z przeciążonych metod klasy Punkt.

```
public
class main
{
    public static void Main()
    {
        Punkt punkt = new Punkt();
        Punkt pomocniczyPunkt = new Punkt();
        punkt.pobierzWspolrzedne(pomocniczyPunkt);

        Console.WriteLine("Przed ustawieniem wartości:");
        Console.WriteLine("WspółrzędnaX = " + pomocniczyPunkt.pobierzX());
        Console.WriteLine("WspółrzędnaY = " + pomocniczyPunkt.pobierzY());

        punkt.ustawWspolrzedne (1, 2);
        punkt.pobierzWspolrzedne(pomocniczyPunkt);

        Console.WriteLine("\nPo ustawieniu wartości:");
        Console.WriteLine("WspółrzędnaX = " + pomocniczyPunkt.pobierzX());
        Console.WriteLine("WspółrzędnaY = " + pomocniczyPunkt.pobierzY());
    }
}
```

Konstruktory

Zwykle polom danego obiektu chcemy przypisać jakieś wartości początkowe. Jeżeli pola te zawierają zmienne referencyjne, trzeba wręcz stworzyć przypisane im obiekty. Czasami też przed użyciem danego obiektu, chcemy wykonać bardziej złożony kod. Można oczywiście napisać do tego celu dodatkową zwykłą metodę i używać jej, na przykład, następująco:

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

Zakładając, że nowo tworzonym obiektom znanej nam już klasy Punkt, będziemy chcieli przypisywać początkowe wartości $x = 320$, $y = 200$, metoda ta wyglądałaby jak poniżej:

```
void inicjujZmienne()
{
    x = 320;
    y = 200;
}
```

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 następująco:

```
Punkt punkt = new Punkt();
punkt.ustawWspolrzedne(320, 200);
```

Czynności te można jednak wykonywać automatycznie. Służą do tego specjalne metody zwane *konstruktorami*. Metody te są wykonywane zawsze przy tworzeniu nowego obiektu. Konstruktory są publiczne i bezrezyultatowe (to znaczy, że nie mogą zwracać wyniku) oraz mają nazwę identyczną z nazwą klasy, której dotyczą. Mogą mieć natomiast różne parametry (może też istnieć kilka konstruktorów dla danej klasy). Napiszmy więc dwa konstruktory dla klasy Punkt. Jeden będzie bezparametrowy i będzie ustawał wartości x i y odpowiednio na 800 i 600. Drugi ustawi współrzędne na wartości podane przez użytkownika jako parametry.

Ćwiczenie 4.9. ——————

Stwórz konstruktory dla klasy Punkt.

```
public
class Punkt
{
    public int x, y;
    public Punkt ()
    {
        ustawWspolrzedne(800, 600);
    }
    public Punkt (int x, int y)
    {
        ustawWspolrzedne(x, y);
    }
}
```

```

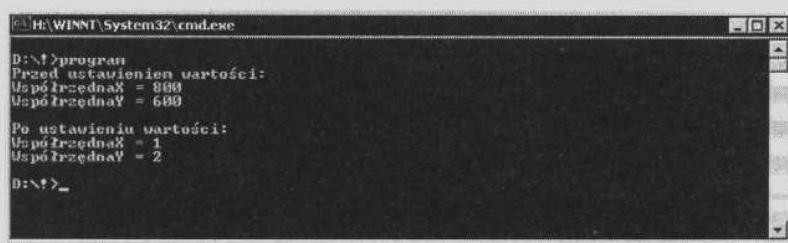
public void ustawWspolrzedne(int wspX, int wspY)
{
    x = wspX;
    y = wspY;
}
public Punkt pobierzWspolrzedne()
{
    Punkt punkt = new Punkt();
    punkt.x = x;
    punkt.y = y;
    return punkt;
}
public void pobierzWspolrzedne(Punkt punkt)
{
    punkt.x = x;
    punkt.y = y;
}
public void ustawX (int wspX)
{
    x = wspX;
}
public void ustawY (int wspY)
{
    y = wspY;
}
public int pobierzX ()
{
    return x;
}
public int pobierzY ()
{
    return y;
}
}

```

Zauważmy, że jeśli przetestujemy tak przygotowaną klasę Punkt, korzystając z klasy Main powstałej w ćwiczeniu 4.8, dwoma pierwszymi wynikami będą liczby 800 i 600 (rysunek 4.2), a nie dwa zera, jak miało to miejsce, kiedy korzystaliśmy z klasy Punkt z ćwiczenia 4.7. Widać więc wyraźnie, że konstruktor faktycznie został wykonany. Aby to sobie lepiej uzmysolić, wykonajmy jeszcze jedno ćwiczenie.

Rysunek 4.2.

Wykorzystanie klasy main z ćwiczenia 4.8 w połączeniu z klasą Punkt z ćwiczenia 4.9



Ćwiczenie 4.10.

Napisz klasę korzystającą z obu konstruktorów klasy Punkt powstalej w ćwiczeniu 4.9 (rysunek 4.3).

```

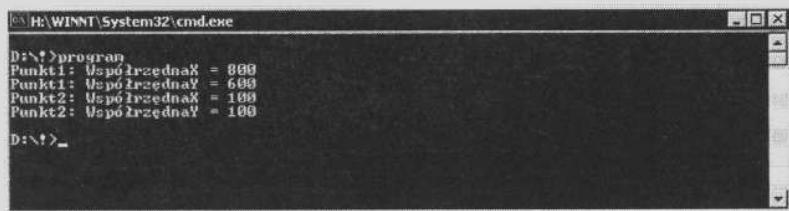
public
class main
{
    public static void Main()
    {
        Punkt punkt1 = new Punkt();
        Punkt punkt2 = new Punkt(100, 100);

        Console.WriteLine("Punkt1: WspółrzędnaX = " + punkt1.pobierzX());
        Console.WriteLine("Punkt1: WspółrzędnaY = " + punkt1.pobierzY());
        Console.WriteLine("Punkt2: WspółrzędnaX = " + punkt2.pobierzX());
        Console.WriteLine("Punkt2: WspółrzędnaY = " + punkt2.pobierzY());
    }
}

```

Rysunek 4.3.

*Wykorzystanie konstruktorów klasy
Punkt w ćwiczeniu 4.10*



Specyfikatory dostępu

W dotychczasowych naszych programach przed słowem `class` pojawiało się zwykle słowo `public`. Oznacza ono, że dana klasa jest publiczna czyli, że mogą z niej korzystać wszystkie inne klasy. Specyfikatory dostępu pojawiają się jednak nie tylko przy nazwach klas, ale także przy nazwach pól i metod. Każde pole oraz metoda 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 oraz że są dziedziczone przez podklasy (o dziedziczeniu napiszemy już w następnym 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. Znaczenie tych specyfikatorów dostępu jest praktycznie takie samo jak w innych językach obiektowych, na przykład w Javie.

Do dyspozycji mamy jednak dodatkowo specyfikatory `internal` i `protected internal`. Słowo `internal` oznacza, że dana składowa klasy będzie dostępna dla wszystkich klas z danego pakietu. 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 pakietu. Niemniej tymi dwoma specyfikatorami nie będziemy się bliżej zajmować, jako że nie będą przydatne przy dalszych ćwiczeniach. Spotkamy się natomiast ze słowami `public`, `private` i `protected`.

Zobaczmy jednak, jak to wygląda w praktyce. Wróćmy do przykładowej klasy `Punkt`. Pola oznaczające współrzędne `x` i `y` zostały wyspecyfikowane jako prywatne. Oznacza to, że nie możemy się do nich odwoływać bezpośrednio.

Ćwiczenie 4.11.

Zmodyfikuj kod klasy `Punkt` z ćwiczenia 4.2 tak, aby składowe `x` i `y` były zadeklarowane jako prywatne. Napisz klasę `main`, w której nastąpi bezpośrednie odwołanie do składowych klasy `Punkt`. Spróbuj skompilować otrzymany kod.

```
using System;

public
class Punkt
{
    private int x;
    private int y;
    public int getX()
    {
        return x;
    }
    public int getY()
    {
        return y;
    }
}

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

Jak widać, do pól typu `private` nie możemy się odwoływać ani przy odczycie, ani przy zapisie (rysunek 4.4). Pamiętamy jednak, że w takim celu zdefiniowaliśmy odpowiednie metody `pobierzWspolrzedne()` i `ustawWspolrzedne()`. Metody te zostały zadeklarowane jako publiczne, zatem mamy do nich dostęp z innych klas. Oczywiście 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.

Dlaczego jednak nie stosować wyłącznie składowych publicznych? Otóż po to, aby programista nie miał bezpośredniego dostępu do wnętrza danej klasy. Przydaje się to zwykle w przypadku bardziej skomplikowanych projektów, jednak nawet na przykładzie naszej prostej klasy `Punkt` jesteśmy w stanie pokazać, dlaczego może być to potrzebne.

Rysunek 4.4.

Próba bezpośredniego odwołania się do prywatnych składowych klasy kończy się niepowodzeniem

```
D:\>WINNT\System32\cmd.exe
D:\>redakcja\helion\csharp\c#\>h:\winnt\Microsoft.NET\Framework\v1.0.3705\csc.exe p
rogram.cs
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

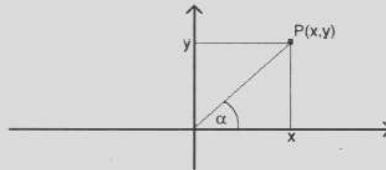
program.cs(25,3): error CS0122: 'Punkt.x' is inaccessible due to its protection
level
program.cs(26,3): error CS0122: 'Punkt.y' is inaccessible due to its protection
level
program.cs(27,43): error CS0122: 'Punkt.x' is inaccessible due to its protection
level

D:\>redakcja\helion\csharp\c#\>
```

Załóżmy, że mamy napisany program, ale z jakichś powodów zmieniliśmy reprezentację współrzędnych i teraz punkt identyfikujemy za pomocą kąta alfa oraz odległości punku od początku układu współrzędnych (tzw. współrzędne biegunowe, rysunek 4.5). 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 by zmienić odwołania, to także dokonywać ich przeliczeń. Spowodowałoby to naprawdę duże komplikacje.

Rysunek 4.5.

Położenie punktu reprezentowane za pomocą współrzędnych biegunowych



Jeżeli jednak zmienne te będą prywatne, trzeba będzie tylko i wyłącznie przeddefiniować metody klasy Punkt. Cała reszta programu nawet nie „zauważa”, że coś się zmieniło! Przekonajmy się o tym!

Ćwiczenie 4.12.

Zmień definicję klasy Punkt w taki sposób, aby położenie punktu było reprezentowane w układzie biegunowym.

```
public
class Punkt
{
    private double modul, sinalfa;
    public Punkt()
    {
        modul = Math.Sqrt (Math.Pow(800, 2) + Math.Pow(600, 2));
        sinalfa = (double) 600 / modul;
    }
    public Punkt(int x, int y)
    {
        ustawWspolrzedne(x, y);
    }
    public void ustawWspolrzedne(int wspX, int wspY)
    {
        modul = Math.Sqrt (Math.Pow(wspX, 2) + Math.Pow(wspY, 2));
        sinalfa = (double) wspY / modul;
    }
    public Punkt pobierzWspolrzedne()
    {
```

```

        Punkt punkt = new Punkt();
        punkt.sinalpha = sinalpha;
        punkt.modul = modul;
        return punkt;
    }
    public void pobierzWspolrzedne(Punkt punkt)
    {
        punkt.sinalpha = sinalpha;
        punkt.modul = modul;
    }
    public void ustawX(int wspX)
    {
        ustawWspolrzedne(wspX, pobierzY());
    }
    public void ustawY(int wspY)
    {
        ustawWspolrzedne(pobierzX(), wspY);
    }
    public int pobierzX()
    {
        double x;
        x = modul * Math.Sqrt(1 - Math.Pow(sinalpha, 2));
        return (int)Math.Round(x);
    }
    public int pobierzY()
    {
        double y;
        y = modul * sinalpha;
        return (int)Math.Round(y);
    }
}

```

Przeliczenie współrzędnych kartezjańskich (tzn. w postaci x,y) na układ biegunowy (czyli kąt i moduł) nie jest skomplikowane. Najwygodniejsza jest tu funkcja sinus, dlatego też użyliśmy jej w powyższym przykładzie. Zatem $\sin(\alpha)$ jest równy $y/\text{moduł}$. Natomiast sam moduł to

$$\sqrt{x^2 + y^2}$$

W druga stronę jest niestety nieco trudniej. Co prawda y to po prostu $\text{moduł} * \sin(\alpha)$, za to $x = \text{moduł} * \sqrt{1 - \sin^2(\alpha)}$. Metoda `Math.pow(arg1, arg2)` zwraca wartość `arg1` podniesioną do potęgi `arg2`. Metoda `Math.Round(arg)` powoduje zaokrąglenie argumentu do liczby całkowitej.

Zapis (*nazwaTypu*) *zmienna* np. `(double) x` oznacza konwersję zmiennej do podanego typu. W naszym przykładzie konwersję zmiennej `x` typu `int` do typu `double`. Tych konwersji musimy dokonywać, gdyż, jak pamiętamy, inne są wyniki działań arytmetycznych (w szczególności dzielenia) na liczbach całkowitych, a inne na liczbach zmienno-przecinkowych.

Ćwiczenie 4.13.

Wykorzystaj klasę main z ćwiczenia 4.10 do przetestowania klasy punkt z ćwiczenia 4.12.

Po wykonaniu tego ćwiczenia okaze się, że wynik działania jest identyczny jak w przypadku ćwiczenia 4.10, mimo że całkowicie zmieniliśmy reprezentację klasy. Co więcej, nie musieliśmy dokonywać żadnych modyfikacji metody Main!

Dziedziczenie

Znamy już dosyć dobrze klasę Punkt, założymy jednak, że chcielibyśmy mieć klasę opisującą nie tylko współrzędne punktu, ale również jego kolor. Chcielibyśmy jednak mieć również możliwość jednoczesnego korzystania z obu klas. Można oczywiście stworzyć nową klasę typu KolorowyPunkt, która mogłaby wyglądać następująco:

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

Musielibyśmy teraz dopisać wszystkie zdefiniowane wcześniej metody klasy Punkt oraz, zapewne, dodatkowo ustawKolor() i pobierzKolor(). W ten sposób jednak dwukrotnie piszemy ten sam kod. Przecież klasy *Punkt* i *KolorowyPunkt* robią w dużej części to samo. Dokładniej mówiąc, to klasa *KolorowyPunkt* jest swoistego rodzaju rozszerzeniem klasy *Punkt*. Zatem niech *KolorowyPunkt* przejmie własności klasy *Punkt*, a dodatkowo dodajmy do niej pole określające kolor. Jest to znane m.in. z Javy i C++ dziedziczenie. Zobaczmy, w jaki sposób wygląda to w C#.

Ćwiczenie 4.14.

Utwórz klasę *KolorowyPunkt* rozszerzającą klasę *Punkt* o możliwość przechowywania informacji o kolorze.

```
public  
class KolorowyPunkt:Punkt  
{  
    protected int kolor;  
    public KolorowyPunkt()  
    {  
        kolor = 0;  
    }  
    public KolorowyPunkt(int wspX, int wspY, int nowyKolor)  
    {  
        x = wspX;  
        y = wspY;  
        kolor = nowyKolor;  
    }  
    public void ustawKolor (int nowyKolor)  
    {  
        kolor = nowyKolor;  
    }  
    public int pobierzKolor()  
    {  
        return kolor;  
    }  
}
```

Ćwiczenie 4.15.

Napisz klasę Main umożliwiającą przetestowanie działania klasy *KolorowyPunkt*.

```

using System;

public
class main
{
    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());
    }
}

```

Oczywiście z klasy KolorowyPunkt możemy wyprowadzić kolejną klasę, na przykład DwukolorowyPunkt, dla punktów, które przyjmują dwa różne kolory, w zależności od znaku wartości współrzędnej x. Z klasy DwukolorowyPunkt może dziedziczyć kolejna klasa itd.

Zwróćmy w tym miejscu uwagę na trójparametry konstruktor klasy KolorowyPunkt. Przyjmuje on parametry dotyczące współrzędnej x oraz współrzędnej y i przypisuje je odpowiednim polom. Spełnia on swoje zadanie, jednak wygodniej byłoby po prostu wywołać konstruktor klasy bazowej (czyli klasy Punkt). W C# robimy to w sposób następujący:

```
void KonstruktorKlasyPotomnej(param1, param2):base(param1);
```

param1 to parametry konstruktora klasy bazowej, a param2 parametry konstruktora klasy potomnej. W konkretnym przypadku klasy KolorowyPunkt wywołanie to powinno wyglądać następująco:

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

Ćwiczenie 4.16.

Utwórz klasę KolorowyPunkt rozszerzającą klasę Punkt o możliwość przechowywania informacji o kolorze. Pamiętaj o wywołaniu konstruktora klasy bazowej, stosując składnię base.

```

public
class KolorowyPunkt:Punkt
{
    protected int kolor;
    public KolorowyPunkt()
    {
        kolor = 0;
    }
    public KolorowyPunkt(int wspX, int wspY, int nowyKolor):base(wspX, wspY)
    {
        kolor = nowyKolor;
    }
    public void ustawKolor (int nowyKolor)
    {
        kolor = nowyKolor;
    }
    public int pobierzKolor()
    {
        return kolor;
    }
}

```

Rozdział 5.

Tablice

Tablice to jedne z podstawowych struktur danych i znane są z pewnością nawet początkującym programistom. Przypomnijmy jednak na wstępnie podstawowe wiadomości i pojęcia z nimi związane. Tablica to stosunkowo prosta struktura danych pozwalająca na przechowanie uporządkowanego zbioru elementów danego typu. Składa się z ponumerowanych kolejno komórek, a każda taka komórka może przechowywać pewną porcję danych.

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.

Deklarowanie tablic

Przed skorzystaniem z tablicy musimy zadeklarować zmienną tablicową, a ponieważ w C# tablice są obiektami, należy również utworzyć odpowiedni obiekt. Schematycznie robimy to w sposób następujący:

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

Oczywiście deklaracji zmiennej tablicowej oraz przypisania jej nowo utworzonego elementu można dokonać w osobnych instrukcjach, np. pisząc:

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

Pisząc zatem:

```
int tablica[];
```

zadeklarujemy odniesienie do tablicy, która będzie zawierała elementy typu `int`, czyli 32-bitowe liczby całkowite. Samej tablicy jednak jeszcze wcale nie ma. Przekonamy się o tym, wykonując kolejne ćwiczenie.

Ćwiczenie 5.1.

Zadeklaruj tablicę elementów typu całkowitego. Przypisz zerowemu elementowi tablicy dowolną wartość. Spróbuj wyświetlić zawartość tego elementu na ekranie.

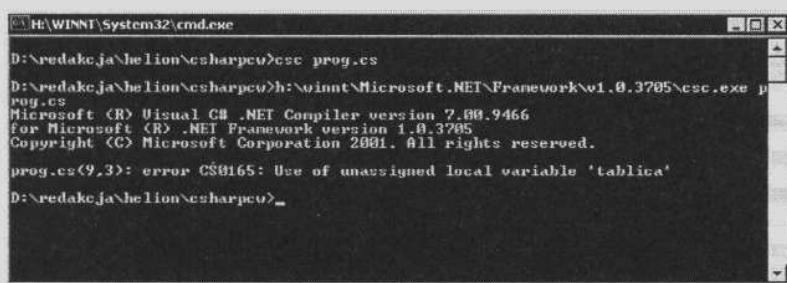
```
using System;

public
class main
{
    public static void Main()
    {
        int[] tablica;
        tablica[0] = 11;
        Console.WriteLine("Zerowy element tablicy to: " + tablica[0]);
    }
}
```

Już przy próbie komplikacji kompilator uprzejmie poinformuje nas, że chcemy odwołać się do zmiennej, która prawdopodobnie nie została zainicjalizowana, wyświetlając na ekranie Use of unassigned local variable 'tablica' (rysunek 5.1).

Rysunek 5.1.

Próba użycia niezainicjalowanej zmiennej tablicowej



Skoro więc wystąpił błąd, należy go natychmiast naprawić.

Ćwiczenie 5.2.

Zadeklaruj i zainicjalizuj tablicę elementów typu całkowitego. Przypisz zerowemu elementowi tablicy dowolną wartość. Spróbuj wyświetlić zawartość tego elementu na ekranie.

```
using System;

public
class main
{
    public static void Main()
    {
        int[] tablica = new int[1];
        tablica[0] = 10;
        Console.WriteLine("Zerowy element tablicy to: " + tablica[0]);
    }
}
```

Wyrażenie `new int[1]` oznacza stworzenie nowej, jednowymiarowej, jednoelementowej tablicy liczb typu `int`. Ta nowa tablica została przypisana zmiennej odnośnikowej o nazwie `tablica`. W tej chwili możemy odwoływać się do kolejnych elementów tej tablicy, pisząc:

```
tablica[index]
```

Warto przy tym zauważyć, że elementy tablicy numerowane są od zera, 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?

Ćwiczenie 5.3.

Zadeklaruj i zainicjuj tablicę dziesięcioelementową. Spróbuj przypisać elementowi o indeksie 10 dowolną liczbę całkowitą.

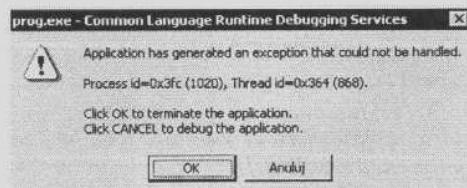
```
using System;

public
class main
{
    public static void Main()
    {
        int[] tablica = new int[10];
        tablica[10] = 1;
        Console.WriteLine("Zerowy element tablicy to: " + tablica[10]);
    }
}
```

Powyższy kod daje się bez problemu skompilować, jednak przy próbie uruchomienia takiego programu na ekranie zobaczymy okno widoczne na rysunku 5.2 informujące o wystąpieniu błędu. Chwilę później ujrzymy na konsoli komunikat podający konkretne informacje o typie błędu oraz funkcji, w której on wystąpił (rysunek 5.3).

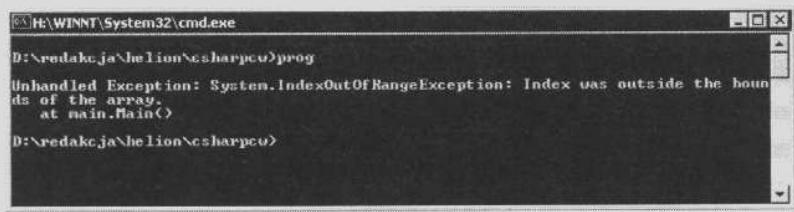
Rysunek 5.2.

Próba odwołania się do nieistniejącego elementu tablicy powoduje błąd aplikacji



Rysunek 5.3.

Systemowa informacja o błędzie



Wbrew pozorom nie stało się jednak nic strasznego. Program co prawda nie działa, ale błąd został wychwycony przez środowisko uruchomieniowe.

Konkretnie mówiąc, został wygenerowany tak zwany wyjątek i aplikacja zakończyła działanie. Taki wyjątek możemy jednak przechwycić i tym samym zapobiec niekontrolowanemu zakończeniu wykonywania kodu. To jednak odrębny temat, którym, zajmiemy się w rozdziale szóstym. Ważne jest to, że próba odwołania się do nieistniejącego elementu została wykryta i to odwołanie tak naprawdę nie wystąpiło! Program nie naruszył więc obszaru pamięci nie zarezerwowanego dla niego.

Taki sam program w C/C++ spowoduje już konieczność interwencji systemu operacyjnego. W przypadku Windows otrzymamy znany chyba wszystkim komunikat Program wykonał nieprawidłową operację.... Można się o tym naocznie przekonać, kompilując i uruchamiając poniższy kod.

```
#include <iostream.h>
int main (void)
{
    int tablica[10];
    tablica[10] = 11;
    cout << "Dziesiąty element tablicy to: " << tablica[10];
    return 0;
}
```

Inicjalizacja

Tablice można zainicjalizować już w momencie ich tworzenia. Dane, które mają się znaleźć w poszczególnych komórkach, podajemy w nawiasach klamrowych po deklaracji tablicy. Schematycznie wygląda to następująco:

```
typ[] nazwa = new typ [liczba_elementów]{dane1, dane2,...,danan}
```

Jeśli zatem chcemy utworzyć pięcioelementową tablicę liczb całkowitych i od razu zainicjować ją liczbami od 1 do 5, możemy zrobić to w sposób następujący:

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

Ćwiczenie 5.4.

Zadeklaruj tablicę pięcioelementową typu int i zainicjuj ją liczbami od 1 do 5. Zawartość tablicy wyświetl na ekranie.

```
using System;

public
class main
{
    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 wyświetlane zostały kolejne liczby od 1 do 5.

Rysunek 5.4.

Zawartość kolejnych komórek tablicy utworzonej w ćwiczeniu 5.4

```
D:\>redakcja>helion\csharp\c>prog
tablica[0] = 1
tablica[1] = 2
tablica[2] = 3
tablica[3] = 4
tablica[4] = 5
D:\>redakcja>helion\csharp\c>_
```

Kiedy inicjujemy tworzoną tablicę z góry znaną liczbą elementów, możemy pominąć fragment kodu związany z tworzeniem obiektu, kompilator doda go za nas. Zamiast pisać:

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

możemy również dobrze użyć konstrukcji:

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

Oba sposoby są sobie równoważne i możemy używać tego, który jest dla nas wygodniejszy.

Ćwiczenie 5.5.

- Zadeklaruj tablicę pięcioelementową typu `int` i zainicjuj ją liczbami od 1 do 5. Użyj drugiego z poznanych sposobów inicjalizacji. Zawartość tablicy wyświetl na ekranie.

`using System;`

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

Pętla `foreach`

Dotychczas poznaliśmy trzy rodzaje pętli: `for`, `while` i `do...while`. W przypadku tablic (jak również kolekcji, którymi się w niniejszej publikacji nie zajmujemy) możemy również skorzystać z pętli typu `foreach`. Jest ona bardzo wygodna, gdyż umożliwia prostą iterację po wszystkich elementach tablicy; nie musimy też wprowadzać dodatkowej zmiennej iterującej. Pętla `foreach` ma postać następującą:

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

Konkretnie, jeżeli mamy tablicę o nazwie `tab` zawierającą liczby typu `int`, powinniśmy zastosować konstrukcję:

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

W tym wypadku w kolejnych przebiegach pętli pod `i` będą podstawiane kolejne elementy tablicy.

Ćwiczenie 5.6. ——————

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

```
using System;

public class main
{
    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);
        }
    }
}
```

Ćwiczenie 5.7. ——————

Wykorzystaj pętlę foreach w celu sprawdzenia, ile jest liczb parzystych, a ile nieparzystych w tablicy z elementami typu int.

```
using System;

public class main
{
    public static void Main()
    {
        int[] tab = new int[100];
        int parzyste = 0, nieparzyste = 0;
        Random rand = new Random(1000);
        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("Tablica zawiera {0} liczb parzystych i {1}
    ➔liczb nieparzystych", parzyste, nieparzyste);
    }
}
```

Tym razem przy wypełnianiu tablicy danymi korzystamy z obiektu klasy Random, która udostępnia wartości pseudolosowe. Dokładniej mówiąc, kolejną pseudolosową liczbę całkowitą uzyskujemy, wywołując metodę Next tejże klasy. Do stwierdzenia, czy kolejna komórka tablicy zawiera liczbę parzystą czy nieparzystą wykorzystujemy operator dzielenia modulo. Oczywiście liczba parzysta modulo dwa daje wynik zero i taki też warunek sprawdzamy w instrukcji if.

Tablice wielowymiarowe

Tablice nie muszą być jednowymiarowe, jak w dotychczas prezentowanych przykładach. Tych wymiarów może być więcej, na przykład dwa, otrzymujemy wtedy strukturę widoczną na rysunku 5.5, czyli rodzaj tabeli o zadanej ilości wierszy i kolumn. W tym przypadku mamy dwa wiersze oraz cztery kolumny. Oczywiście w takiej sytuacji, aby jednoznacznie wyznać komórkę, trzeba podać dwie liczby.

Rysunek 5.5.

Przykład tablicy dwuwymiarowej

	0	1	2	3	4
0					
1					

Musimy się teraz dowiedzieć, w jaki sposób zadeklarować tego typu tablicę. Zaczniemy od deklaracji samej zmiennej tablicowej. W przypadku tablicy dwuwymiarowej ma ona postać:

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

Samą tablicę tworzymy natomiast za pomocą instrukcji:

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

Przykładowo, dwuwymiarowa tablicę widoczną na rysunku 5.5 utworzymy następująco (zakładając, że ma ona przechowywać liczby całkowite):

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

Inicjalizacja samych komórek może odbywać się, podobnie jak w przypadku tablic jednowymiarowych, już w trakcie deklaracji:

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

Zobaczmy jak wygląda to na konkretnym przykładzie.

Ćwiczenie 5.8.

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

```
using System;

public class main
{
    public static void Main()
    {
        int[,] tablica = new int[2, 5];
        int counter = 0;
        for(int i = 0; i < 2; i++){
            for(int j = 0; j < 5; j++){
                tablica[i, j] = counter++;
            }
        }
        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 używamy dwóch zagnieżdzonych pętli for. Pierwsza, zewnętrzna, odpowiada za iterację po indeksach wierszy tablicy, druga za iterację po indeksach kolumn. Zmienna counter służy nam jako licznik i jest w każdym przebiegu zwiększana o jeden, dzięki czemu w kolejnych komórkach uzyskujemy kolejne liczby całkowite. Po wypełnieniu danymi nasza tablica ma postać widoczną na rysunku 5.6.

Rysunek 5.6.

Tablica z ćwiczenia 5.8 po wypełnianiu danymi

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

Do wyświetlenia danych używamy analogicznej konstrukcji z dwoma zagnieżdzonymi pętlami. Po uruchomieniu kodu na ekranie zobaczymy widok przedstawiony na rysunku 5.7. Jak widać, dane te zgodne są ze strukturą przedstawioną na rysunku 5.6.

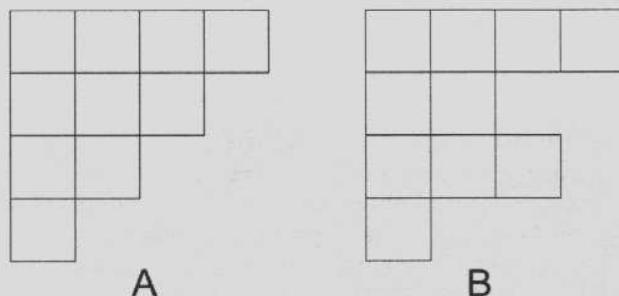
Tablica dwuwymiarowa nie musi mieć wcale, tak jak w poprzednich przykładach, kształtu prostokątnego, tzn. takiego, gdzie ilość komórek w każdym wierszu i każdej kolumnie jest stała.

Rysunek 5.7.
Wynik działania programu z ćwiczenia 5.6

```
D:\redakcja\helion\csharp>prog
tablica[0, 0] = 0
tablica[0, 1] = 1
tablica[0, 2] = 2
tablica[0, 3] = 3
tablica[1, 0] = 4
tablica[1, 1] = 5
tablica[1, 2] = 6
tablica[1, 3] = 7
tablica[2, 0] = 8
tablica[2, 1] = 9
tablica[2, 2] = 0
tablica[2, 3] = 1
tablica[3, 0] = 2
tablica[3, 1] = 3
tablica[3, 2] = 4
tablica[3, 3] = 5
D:\redakcja\helion\csharp>
```

Równie dobrze możemy stworzyć tablicę o kształcie trójkąta (rysunek 5.8.A) lub zupełnie nieregularną (rysunek 5.8.B). Przy tworzeniu tego typu struktur musimy się jednak nieco więcej napracować, gdyż każdy wiersz zazwyczaj trzeba tworzyć ręcznie, pisząc odpowiednią linię kodu.

Rysunek 5.8.
Przykłady bardziej skomplikowanych tablic dwuwymiarowych



Zacznijmy od utworzenia struktury przedstawionej na rysunku 5.8.B. Zauważmy, że każdy wiersz można traktować jako oddzielną tablicę jednowymiarową. Zatem tak naprawdę jest to jednowymiarowa tablica, której poszczególne komórki zawierają inne jednowymiarowe tablice. Inaczej mówiąc, jest to tablica tablic. Wystarczy zatem zadeklarować zmienną tablicową o odpowiednim typie, a następnie poszczególnym jej elementom przypisać nowo utworzone tablice jednowymiarowe o zadanej długości. To całe rozwiązanie problemu.

Pytanie brzmi: co to znaczy „odpowiedni typ tablicy”? Pomyślmy — jeśli w tablicy (jednowymiarowej) chcieliśmy przechowywać liczby całkowite typu `int`, to typem tej tablicy było `int`. Pisaliśmy wtedy:

```
int[] tablica;
```

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

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

Z kolei utworzenie czteroelementowej tablicy zawierającej tablice z liczbami całkowitymi wymaga wpisu:

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

Te wiadomości powinny nam wystarczyć do wykonania kolejnego ćwiczenia.

Ćwiczenie 5.9.

Napisz kod tworzący strukturę tablicy widocznej na rysunku 5.8.B przechowującej liczby całkowite. W kolejnych komórkach powinny znaleźć się kolejne liczby całkowite, zaczynając od 1.

```
using System;

public
class main
{
    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};
    }
}
```

W tej chwili nasza struktura została wypełniona danymi, tak jak widoczne jest to na rysunku 5.9. Jak sobie jednak poradzić z jej wyświetleniem na ekranie. Oczywiście możemy zrobić to ręcznie, pisząc kod oddzielnie dla każdego wiersza. W przypadku tak małej tablicy nie będzie to problemem. Czy jednak tej czynności nie da się zautomatyzować? Najwygodniej byłoby przecież wyprowadzać dane na ekran w zagnieżdżonej pętli, tak jak w przypadku ćwiczenia 5.8.

Brysunek 5.9.

Tablica z ćwiczenia
5.9 wypełniona
przykładowymi danymi

1	2	3	4	
5	6			
7	8	9		
10				

Okazuje się, że jest to jak najbardziej możliwe, a z nieregularnością naszej tablicy poradzimy sobie w bardzo prosty sposób. Okazuje się, że każda tablica, ponieważ jest obiektem, posiada właściwość *Length*, dzięki której możemy sprawdzić jej długość. To rozwiązuje całkowicie problem wyświetlenia danych nawet z tak nieregularnej struktury jak obecnie opisywana.

Ćwiczenie 5.10.

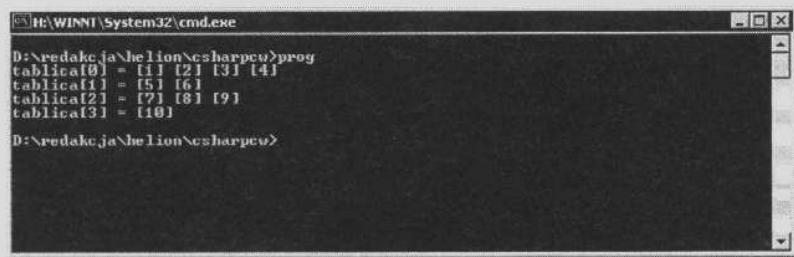
Zmodyfikuj kod z ćwiczenia 5.9 w taki sposób, aby dane zawarte w tablicy zostały wyświetlane na ekranie (rysunek 5.10). W tym celu użyj zagnieżdżonej pętli *for*.

```
using System;

public
class main
{
    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 5.10.

Wyświetlenie danych z nieregularnej tablicy w ćwiczeniu 5.10



Rozdział 6.

Wyjątki

Obsługa błędów

W każdym większym programie występują jakieś błędy. Oczywiście staramy się przed nimi strzec, nigdy jednak nie uda nam się ich całkowicie wyeliminować. Co więcej, aby program wychwytywał przynajmniej część błędów, których przyczyną jest, na przykład, wprowadzenie złych wartości przez użytkownika, musimy napisać wiele wierszy dodatkowego kodu, który zaciemnia główny kod programu. Przykładowo, jeżeli zadeklarowaliśmy tablicę pięcioelementową, należałoby sprawdzić, czy nie odwołujemy się do nieistniejącego elementu.

Ćwiczenie 6.1.

W klasie Main zadeklaruj tablicę pięcioelementową. Spróbuj odczytać wartość nieistniejącego, dwudziestego elementu tej tablicy.

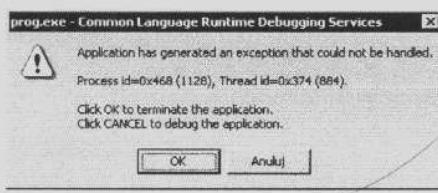
```
using System;

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

Próba wykonania powyższego kodu spowoduje oczywiście błąd, ponieważ nie ma w tablicy table elementu o indeksie 20. Najpierw zobaczymy wygenerowane przez środowisko uruchomieniowe (CLR) okno z informacją, że aplikacja wygenerowała wyjątek (rysunek 6.1), a następnie na konsoli zobaczymy stosowny komunikat podający dokładny typ błędu (rysunek 6.2).

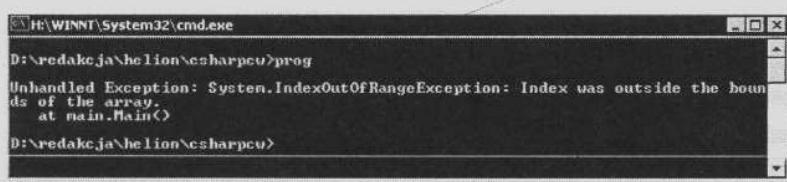
Rysunek 6.1.

Środowisko uruchomieniowe zgłasza błąd aplikacji



Rysunek 6.2.

Informacje o błędzie wygenerowanym w ćwiczeniu 6.1



W powyższym przykładzie jednak popełniony przez nas błąd jest evidentnie widoczny. Gdybyśmy jednak deklarowali tablicę w jednej klasie, a odwoływali się do niej w drugiej, nie byłoby to już tak oczywiste.

Ćwiczenie 6.2.

Napisz dwie takie klasy, aby w jednej zadeklarować tablicę pięcioelementową, a w drugiej następowalo odwołanie do tej tablicy.

```
using System;
public class main
{
    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 tej chwili w klasie Main wywołujemy metodę obiektu typu Tablica, nie wiedząc (bez sprawdzenia kodu klasy *Tablica*), jakiej wielkości jest sama tablica. Bardzo łatwo przekroczyć więc maksymalny indeks i spowodować błąd. Tak też dzieje się w powyższym przykładzie.

Możemy tego uniknąć, sprawdzając, czy podawana jako argument metody `getElement()` wartość nie przekracza zakresu 0 – 4. Takiego sprawdzenia można dokonać, stosując znaną już instrukcję `if`.

Ćwiczenie 6.3.

Popraw kod z ćwiczenia 6.2 tak, aby po przekroczeniu dopuszczalnego indeksu tablicy, nie występował błąd w programie.

```
using System;

public class main
{
    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);
        }
    }
}

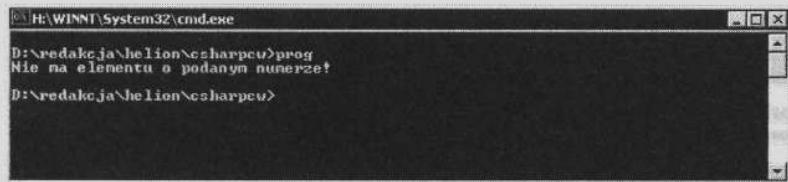
public class Tablica
{
    int[] tab;
    public Tablica()
    {
        tab = new int[5];
    }
    public int getElement(int index)
    {
        if ((index >=0) && (index < 5)){
            return tab[index];
        }
        else{
            return -1;
        }
    }
}
```

Tym razem program reaguje już poprawnie (rysunek 6.3). Po odwołaniu do tablicy sprawdzamy, czy metoda `getElement()` nie zwróciła przypadkiem wartości -1. Jeśli tak, oznacza to, że odwołanie nie powiodło się. Moglibyśmy oczywiście zakończyć wykonywanie

programu już w samej metodzie `getElement()`, zwykle jednak istnieje potrzeba poinformowania funkcji wywołującej o wystąpieniu błędu. W powyższym rozwiązaniu o sytuacji błędnej informuje nas właśnie zwrócenie wartości `-1`.

Rysunek 6.3.

*Odwzorowanie
do nieistniejącego
elementu nie powoduje
już błędu aplikacji*



Taki sposób ma jednak bardzo poważną wadę! Otóż żaden z elementów tablicy nie może mieć wartości równej `-1`. To poważne ograniczenie funkcjonalności naszego programu. Moglibyśmy poradzić sobie jednak z tym problemem, dodając do klasy `Main` pole typu `boolean`, np. o nazwie `isError`.

Ćwiczenie 6.4.

Zmodyfikuj kod z ćwiczenia 6.3 tak, aby o wystąpieniu błędu informowała dodatkowa zmienna.

```
using System;

public
class main
{
    public static void Main()
    {
        Tablica tab = new Tablica();
        int value = tab.getElement(20);
        if (tab.isError){
            Console.WriteLine("Nie ma elementu 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;
        }
    }
}
```

```
        }
    else{
        isError = true;
        return -1;
    }
}
```

To jest już całkiem dobre rozwiązanie, jednak, postępując w ten sposób, dodajemy coraz więcej nowych zmiennych i warunków. Warto by skorzystać z jakiejś innej metody obsługi błędów. Z pomocą przychodzi nam tutaj tak zwane wyjątki. Wyjątek jest to typ programistyczny, który powstaje w chwili wystąpienia błędu. Możemy go jednak przechwycić i wykonać nasz własny kod obsługi błędu. Jeżeli tego nie zrobimy, zostanie on obsłużony przez system, a na konsoli zobaczymy wtedy, gdzie i jakiego typu błąd wystąpił. W naszym przypadku występował wyjątek o nazwie `IndexOutOfRangeException` (widać to wyraźnie na rysunku 6.2). Oznacza, że został przekroczyony 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 (TypWyjątku1 identyfikatorWyjątku1){
    obsługa wyjątku 1
}
catch (TypWyjątku2 identyfikatorWyjątku2){
    obsługa wyjątku 2
}
catch (TypWyjątkuN identyfikatorWyjątkuN){
    obsługa wyjątku n
}
```

Po `try` następuje blok instrukcji mogących spowodować wyjątek. Jeżeli podczas ich wykonywania zostanie on wygenerowany, wykonywanie zostanie przerwane, a sterowanie przekazane do bloku instrukcji `catch`. Tu z kolei sprawdzane jest, czy któraś z instrukcji odpowiada wygenerowanemu wyjątkowi. Jeżeli tak, wykonany zostanie kod po niej następujący.

Ćwiczenie 6.5. ——————

Zastosuj instrukcję `try...catch` do przechwycenia wyjątku generowanego przez system w ćwiczeniu 6.2.

```
using System;

public
class main
{
```

```
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];
    }
}
```

Wykonanie powyższego kodu spowoduje pojawianie się na ekranie napisu Nie ma elementu o numerze 20. Widać więc wyraźnie, że wyjątek faktycznie został przechwycony i prawidłowo obsłużony. Zauważmy jednak, że nasza realizacja nieco odbiega od przedstawionego wcześniej schematu, choć oczywiście jest jak najbardziej poprawna. Otóż instrukcja `catch` w tym przykładzie ma postać:

```
catch(Typ_wyjątku)
zamiast:
catch(Typ_wyjątku indentyfikator_wyjątku)
```

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

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

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

Wypróbujmy tę metodę na konkretnym przykładzie.

Ćwiczenie 6.6.

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

Spróbujmy teraz złapać dosyć często występujący błąd, mianowicie dzielenie przez zero. Gdy nie kontrolujemy stanu zmiennych, taka sytuacja może doprowadzić do poważnej awarii aplikacji. Sprawdźmy najpierw jednak, czy kompilator pozwoli nam dokonać jawnego dzielenia przez zero.

Ćwiczenie 6.7.

Napisz program, w którym występuje jasne dzielenie przez zero. Spróbuj dokonać kompilacji kodu.

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

```

    {
        int x = 2;
        int y = 6;
        int z = y / 0;
        Console.WriteLine("x = " + x);
        Console.WriteLine("y = " + y);
        Console.WriteLine("z = " + z);
    }
}

```

Szybko przekonamy się, że komplikacja kodu z powyższego ćwiczenie się nie uda. Komplikator wykrywa, że chcemy dzielić przez zero i, jako że jest to operacja niedozwolona, wyświetla komunikat o błędzie. Widoczne jest to na rysunku 6.4.

Rysunek 6.4.

Przy próbie jawnego dzielenia przez zero komplikator generuje błąd

```

D:\redakcja\helion\csharp\c>csc prog.cs
D:\redakcja\helion\csharp\c>h:\winnt\Microsoft.NET\Framework\v1.0.3705\csc.exe p
prog.cs
Microsoft (R) Visual C# .NET Compiler version 2.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

prog.cs(10,11): warning CS0020: Division by constant zero
D:\redakcja\helion\csharp\c>_

```

Niestety, mądrość komplikatora kończy się w sytuacji, kiedy najpierw zero przypiszemy do jakiejś zmiennej, a dopiero tej zmiennej użyjemy jako dzielnika. W takim przypadku komplikacja uda się bez problemów, natomiast program oczywiście nie będzie działał. Zabezpieczmy się przed taką sytuacją, stosując odpowiedni blok try...catch.

Ćwiczenie 6.8.

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

```

using System;

public class main
{
    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("Nie można dzielić przez 0!");
        }
    }
}

```

Po uruchomieniu powyższego kodu na ekranie zobaczymy komunikat o treści: Nie można dzielić przez zero.

Hierarchia wyjątków

Wyjątki w C# są strukturą hierarchiczną, na czele której znajduje się klasa `Exception`. Z klasy tej wyprowadzone są kolejne klasy potomne obsługujące różne typy błędów. Na przykład, dla znanego nam już wyjątku o nazwie `IndexOutOfRangeException` hierarchia wygląda jak na rysunku 6.5.

Rysunek 6.5.

Hierarchia klas dla wyjątku
`IndexOutOfRangeException`



Skoro wyjątki są strukturą hierarchiczną, nie jest wcale obojętne, w jakiej kolejności będziemy je przechwytywać. Pamiętamy przecież, że w jednym bloku `try...catch` możemy obsłużyć wiele różnych błędów. Jaka zatem obowiązuje kolejność? Zasada jest prosta — najpierw przechwytyujemy wyjątki bardziej szczegółowe, potem bardziej ogólne.

W przykładzie widocznym na rysunku 6.5 `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.

Zmodyfikuj kod z ćwiczenia 6.5 w taki sposób, aby najpierw przechwytywany był wyjątek ogólny `Exception`, a następnie wyjątek `IndexOutOfRangeException`. Spróbuj dokonać komilacji otrzymanego kodu.

```
using System;

public class main
{
    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
{
    int[] tab;
```

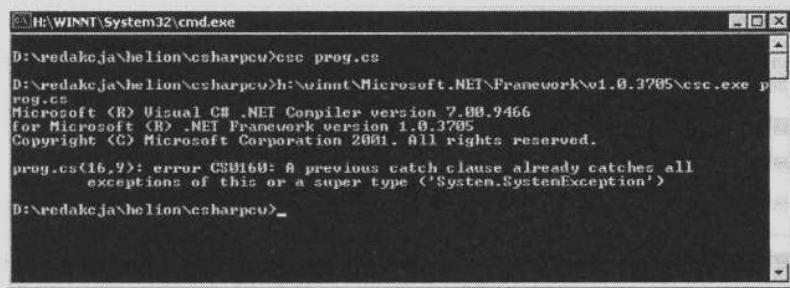
```

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

```

Kompilacja oczywiście się nie uda. Na ekranie zobaczymy komunikat pokazany na rysunku 6.6. Nie powinniśmy się temu dziwić, skoro najpierw przechwyciliśmy wyjątek ogólny SystemException kod występujący w bloku catch(IndexOutOfRangeException) nigdy nie zostałby osiągnięty, kompilator zatem protestuje. Skoro tak, poprawmy szybko nasz przykład.

Rysunek 6.6.
Próba komplikacji kodu z błędą hierarchią wyjątków



Ćwiczenie 6.10.

Popraw klasę main z poprzedniego ćwiczenia tak, aby wyjątki były przechwytywane w odpowiedniej kolejności.

```

using System;

public
class main
{
    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

Czasami chcielibyśmy sami wygenerować wyjątek. Powróćmy zatem do ćwiczenia 6.5. Konstrukcja klasy Tablica mogłaby być zupełnie inna. Może dobrze byłoby sprawdzić, czy parametr przekazywany metodzie getElement nie przekracza dopuszczalnych wartości, a jeśli tak, samemu wygenerować wyjątek. Jest to jak najbardziej możliwe, służy do tego instrukcja throw w postaci:

```
throw obiekt_wyjątku;
```

A zatem, jeśli chcemy wygenerować nowy wyjątek IndexOutOfRangeException, należy zastosować konstrukcję:

```
throw new IndexOutOfRangeException();
```

Ćwiczenie 6.11.

Zmodyfikuj kod ćwiczenia 6.5 w taki sposób, aby metoda getElement sprawdzała, czy nie następuje przekroczenie zakresu tablicy i w takiej sytuacji generowała wyjątek.

```
using System;

public class main
{
    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];
        }
    }
}
```

Co jednak zrobić w sytuacji, kiedy chcielibyśmy utworzyć nasz własny wyjątek, na przykład o nazwie NieMaTakiegoElementu i tenże wyjątek przechwytywać w bloku instrukcji catch? Nie stanowi to żadnego problemu, trzeba po prostu utworzyć nową klasę pochodną klasy Exception, nazywając ją NieMaTakiegoElementu. Mogłaby ona wyglądać następująco:

```
public  
class NieMaTakiegoElementu:Exception  
{}
```

To wszystko. Od tej chwili NieMaTakiegoElementu będzie pełnoprawnym wyjątkiem, który możemy wygenerować, stosując instrukcję throw, czyli pisząc:

```
throw new NieMaTakiegoElementu();
```

Ćwiczenie 6.12.

Zdefiniuj klasę wyjątku o nazwie NieMaTakiegoElementu i zmodyfikuj kod ćwiczenia 6.11 tak, aby z niej korzystał.

```
using System;  
  
public  
class NieMaTakiegoElementu:Exception  
{  
}  
  
public  
class main  
{  
    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];  
        }  
    }  
}
```

Rozdział 7.

Interfejsy

Prosty interfejs

Interfejsy są strukturami takimi jak klasy, z tą różnicą, że zawierają jedynie deklaracje metod, a nie ich definicję. Inaczej mówiąc, są one klasami abstrakcyjnymi. Każda *zwykła* klasa może implementować dany interfejs, co oznacza, że musi zawierać definicje wszystkich metod zawartych w tym interfejsie.

Załóżmy, że mamy zdefiniowaną klasę Point w najprostszej postaci, zawierającą jedynie dwa pola, jedno dla współrzędnej x, drugie dla współrzędnej y:

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

Naszym pierwszym zadaniem będzie napisanie interfejsu o nazwie IShow, w którym zadeklarujemy tylko jedną metodę o nazwie Show.

Ćwiczenie 7.1.

Napisz interfejs o nazwie IShow, w którym zadeklarowana będzie metoda o nazwie Show.

```
interface IShow  
{  
    void Show();  
}
```

Jak widać, zadanie to nie było wcale skomplikowane. Przypomina definicję klasy, z tym, że zamiast słowa kluczowego `class` używamy słowa kluczowego `interface`. Przekonajmy się teraz, że faktycznie w interfejsie nie wolno zdefiniować ciała metody. Posłuży nam do tego kolejne ćwiczenie.

Ćwiczenie 7.2.

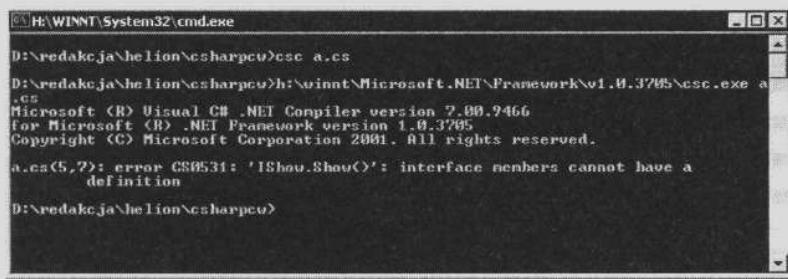
Dodaj do metody Show z interfejsu z ćwiczenia 7.1 jej implementację. Spróbuj dokonać komilacji kodu.

```
interface IShow
{
    void Show()
    {
        int a;
    }
}
```

Próba komilacji powyższego fragmentu kodu spowoduje wygenerowanie błędu komilacji CS0531 z komunikatem `interface member cannot have a definition` (składowa interfejsu nie może posiadać definicji). Widoczne jest to na rysunku 7.1.

Rysunek 7.1.

Próba komilacji interfejsu zawierającego implementację metody Show



Powróćmy jednak do klasy `Point` i interfejsu `IShow` i zobaczymy, jak wykorzystać je w praktyce. Przede wszystkim musimy dowiedzieć się, w jaki sposób spowodować, aby klasa implementowała interfejs. Okazuje się, że stosujemy tu konstrukcję analogiczną do dziedziczenia, czyli schemat definicji wygląda następująco:

```
public
class nazwa_klasy : nazwa_interfejsu
{
    //definicja klasy
}
```

Ćwiczenie 7.3.

Napisz kod, w którym klasa Point implementuje interfejs IShow.

```
using System;

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);  
    }  
}
```

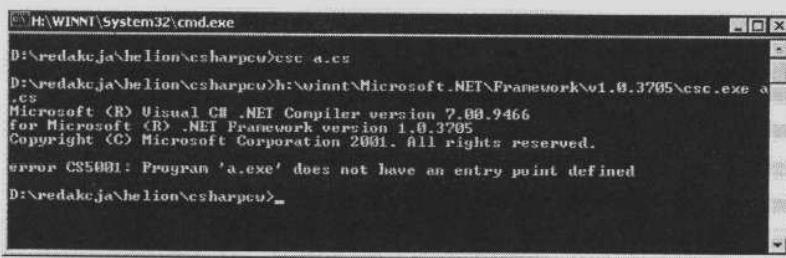
Jak widzimy, w naszym przypadku implementacja interfejsu polega na zdeklarowaniu oraz zdefiniowaniu w klasie Point metody Show w postaci:

```
public void Show()  
{  
    Console.WriteLine("Współrzędna x={0}, współrzędna y={1}", x, y);  
}
```

Metoda ta powoduje wyświetlenie na ekranie bieżących wartości x oraz y. Oczywiście próba komplikacji powyższego kodu nie powiedzie się, na ekranie zobaczymy komunikat widoczny na rysunku 7.2. Powód jest chyba jasny — nie zdefiniowaliśmy metody main, od której mogłoby zacząć się wykonywanie programu. Napiszmy zatem program, który skorzysta z naszej klasy Point.

Rysunek 7.2.

Niezdefiniowanie metody main powoduje błąd komilacji



Ćwiczenie 7.4.

Napisz program, który będzie wykorzystał metodę Show klasy Point.

```
using System;  
  
interface IShow  
{  
    void Show();  
}  
  
public  
class Point:IShow  
{  
    public int x;  
    public int y;  
    public void Show()
```

```

    {
        Console.WriteLine("Dane dotyczące punktu:\n");
        Console.WriteLine("współrzędna x = {0}", x);
        Console.WriteLine("współrzędna y = {0}", y);
    }
}

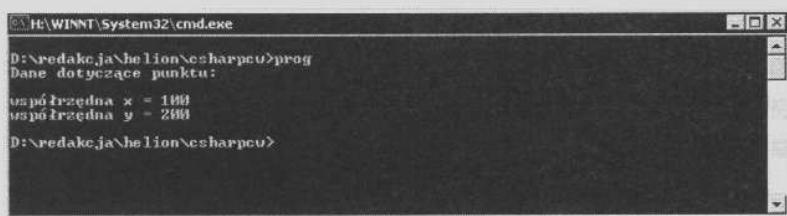
public
class main
{
    public static void Main()
    {
        Point punkt = new Point();
        punkt.x = 100;
        punkt.y = 200;
        punkt.Show();
    }
}

```

Utworzyliśmy tu dodatkową klasę `main` zawierającą publiczną i statyczną metodę `Main`, od której zaczyna się wykonywanie programu. Ten schemat jest nam dobrze znany, gdyż wykorzystywaliśmy go już wiele razy. W metodzie `Main` deklarujemy zmienną referencyjną `punkt` i przypisujemy do niej nowo utworzony obiekt klasy `Point`. Następnie inicjujemy pola `x` i `y` tego obiektu odpowiednio wartościami 100 i 200. Na zakończenie wywołujemy metodę `Show`.

Wynikiem działania tego kodu, co nie powinno być żadnym zaskoczeniem, jest pojawianie się na ekranie tekstu zaprezentowanego na rysunku 7.3.

Rysunek 7.3.
Wynik działania kodu z ćwiczenia 7.4



Interfejsy w klasach potomnych

Dotychczas implementowaliśmy interfejs jedynie w klasie bazowej, to znaczy takiej, która nie dziedziczy z innej, zdefiniowanej przez nas klasy¹. Jak zatem będzie wyglądała struktura implementująca interfejs w klasie potomnej? Na szczęście bardzo prosto:

```

public
class klasa_potomna : klasa_bazowa, interfejs
{
    //definicja klasy
}

```

¹ Dziedziczy jednak domyślnie z klasy `Object`.

Zbudujmy zatem jedną klasę podstawową o nazwie Shape i wyprowadźmy z niej dwie klasy potomne Rectangle i Triangle. Shape będzie klasą ogólną służącą do opisu figur geometrycznych, Rectangle klasą opisującą prostokąty, Triangle klasą opisującą trójkąty. Odświeżymy przy okazji wiadomości o dziedziczeniu.

Ćwiczenie 7.5. ——————

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 Square: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;  
}  
  
public  
class Point:Shape  
{  
    public int x;  
    public int y;  
}
```

Klasa Shape zawiera jedynie jedną składową oznaczającą kolor figury. W klasie Rectangle zdefiniowaliśmy pola x i y wyznaczające położenie górnego lewego wierzchołka oraz width i height specyfikujące odpowiednio jego długość i wysokość. Klasa Triangle zawiera z kolei trzy pola, wszystkie typu Point, które wyznaczają położenie opisywanego trójkąta.

Spróbujmy teraz zaimplementować znany nam już interfejs IShow do klas Rectangle i Triangle. Pamiętamy, że dla każdej z nich musimy napisać odpowiednią metodę Show. Metoda ta musi uwzględniać specyfikę danej figury. Co innego będziemy wyświetlać w przypadku prostokąta, co innego w przypadku trójkąta.

Ćwiczenie 7.6. 

Zaimplementuj interfejs `IShow` dla klas `Rectangle` i `Triangle` zdefiniowanych w ćwiczeniu 7.5.

```
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 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 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;
}
```

Ćwiczenie 7.7.

Napisz klasę main, która będzie wykorzystywała obiekty Rectangle i Triangle.

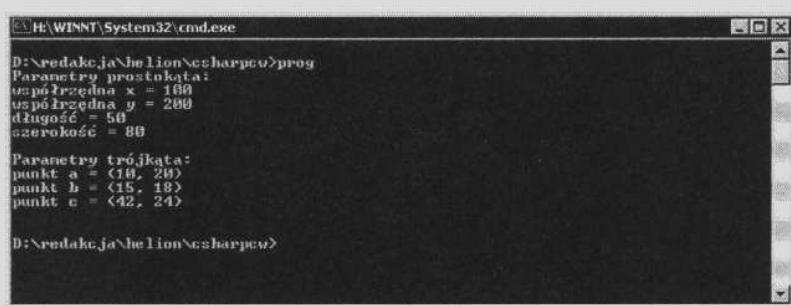
```
using System;

public
class main
{
    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();
        trojkat.Show();
    }
}
```

W metodzie Main tworzymy obiekt klasy Rectangle i przypisujemy jego polom przykładowe wartości. Podobnie postępujemy z obiektem klasy Triangle. Ponieważ pola w tej klasie są typu Point, musimy 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



Zauważmy jednak, że o ile we wcześniejszych ćwiczeniach (7.3, 7.4) implementowaliśmy interfejs IShow dla klasy Point, tym razem wyświetlanie parametrów punktu obsługujemy z poziomu klasy Triangle, np. w linii:

```
Console.WriteLine("punkt a = ({0}, {1})", a.x, a.y);
```

Jest to pewna niespójność w kodzie, jako że punkt też przecież jest figurą geometryczną i nie ma powodu, aby pozbawiać klasę Point możliwości samodzielnego wyświetlania parametrów na ekranie. Za chwilę naprawimy to małe niedopatrzenie, jednak zmodyfikujemy metodę Show tak, aby dane wyświetlane były w postaci:

(wartość_x, wartość_y)

Z pewnością zauważycie, 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 zainicjowaniu jednego z pól. Cały kod jest też dużo mniej czytelny.

Wszystkie te poprawki wprowadzimy w kolejnym ćwiczeniu.

Ćwiczenie 7.8.

Zaimplementuj interfejs IShow dla klasy Point, dodaj konstruktory dla klas Rectangle i Triangle. Napisz klasę main testującą nowy kod.

```
using System;

public class main
{
    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();
    }
}

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.Write("\n");  
    }  
}  
  
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.Write("({0}, {1})", x, y);  
    }  
}
```

Czy to interfejs?

Wiemy już sporo o interfejsach w C#, przydałyby nam się jednak jeszcze wiadomości o tym, w jaki sposób stwierdzić, czy dany obiekt implementuje dany interfejs. Oczywiście w najprostszym przypadku taką wiedzą dysponować będziemy już na etapie komplikacji. Założmy, że operujemy na klasach Rectangle i Triangle stworzonych w ćwiczeniu 7.8 i klasie Point z ćwiczenia 7.6. To znaczy klasy Rectangle i Triangle implementują interfejs IShow, natomiast klasa Point nie.

Co się zatem stanie, jeśli spróbujemy skompilować następujący fragment kodu:

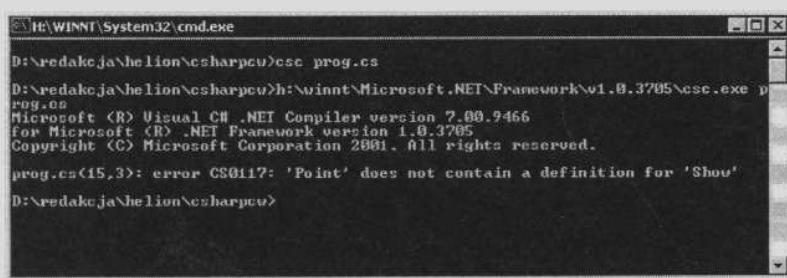
```
using System;

public class main
{
    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 komplikacja się nie uda, jako że w klasie Point nie ma metody Show, a na ekranie zobaczymy komunikat o błędzie widoczny na rysunku 7.5.

Rysunek 7.5.

Próba wywołania nieistniejącej metody powoduje błąd komplikacji



W tym przypadku nie było żadnych problemów, nie zawsze jednak mamy tak klarowną sytuację. Rozpatrzmy nieco bardziej skomplikowany przykład. W tym celu dokonamy pewnych modyfikacji utworzonych dotychczas klas Shape, Rectangle, Triangle i Point.

W klasie Shape zdefiniujemy wirtualną metodę Show(). Zmiana to wymusi przeciążenie metod Show w klasach Rectangle i Triangle, jako że wciąż będą one implementować interfejs IShow. W klasie Point metody Show nie zadeklarujemy.

Ćwiczenie 7.9.

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 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 prostokąta:");
        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;
    }
}

```

Nie trudno się domyślić, że w tej chwili wykonanie serii instrukcji:

```

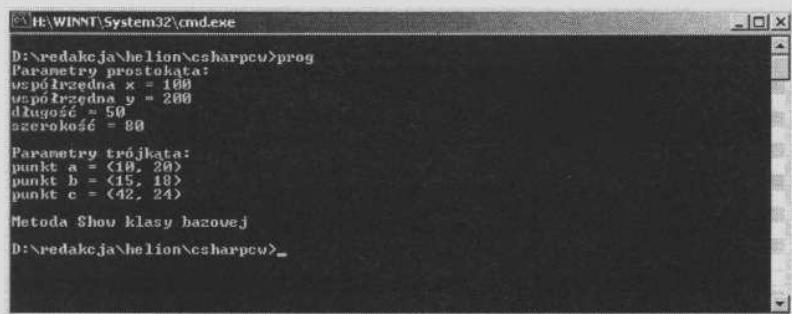
Rectangle prostokat = new Rectangle(100, 200, 50, 80);
Triangle trojkat = new Triangle(new Point(10, 20),
                                new Point(15, 18),
                                new Point(42, 24));
Point punkt = new Point(150, 300);
prostokat.Show();
trojkat.Show();
punkt.Show();

```

jest jak najbardziej możliwe i poprawne. Zakładając oczywiście, że wcześniej zostały stworzone obiekty prostokat, trojkat i punkt. Efekt działania takiego kodu jest widoczny na rysunku 7.6. Widać wyraźnie, że w przypadku obiektów prostokat i trojkat zostały wywołane metody Show klas Rectangle i Triangle, natomiast w przypadku klasy Point wywołana została metoda klasy bazowej.

Rysunek 7.6.

Dla obiektu
punkt została
wywołana metoda
Show klasy Shape



Problem pojawia się w sytuacji, kiedy interesuje nas wyłącznie wywołanie metody implementowanej poprzez interfejs, a nie wiemy dokładnie, jakiego typu jest obiekt. Inaczej mówiąc, metodę Show chcemy wywoływać wyłącznie wtedy, kiedy dany obiekt implementuje interfejs IShow. W powyższym przypadku teoretycznie to wiemy, wystarczy zająrzyć do kodu źródłowego danej klasy. Nie zawsze jednak mieli taką możliwość.

Ćwiczenie 7.10.

Zadeklaruj tablicę obiektów klasy *Shape* i przypisz jej komórkom obiekty klas *Rectangle*, *Triangle* i *Point*. Dla każdego z elementów tablicy, w pętli *for*, wywołaj metodę *Show*.

```
using System;
public
class main
{
    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();
        }
    }
}
```

Taki program oczywiście zadziała, choć nie do końca zgodnie z naszymi założeniami. Ponieważ nie sprawdzamy, jaki jest rzeczywisty typ obiektu zawartego w danej komórce tablicy, wywołamy metodę *Show* również dla komórki numer trzy, a obiekt tam znajdujący się jest klasą *Punkt*. My tymczasem chcielibyśmy wywoływać metodę *Show* jedynie dla obiektów implementujących interfejs *IShow*.

Jak zatem stwierdzić, czy dany obiekt implementuje ten interfejs, czy też nie? Na szczęście do dyspozycji mamy aż trzy metody:

- ❖ rzutowanie typu,
- ❖ konstrukcja ze słowem *is*,
- ❖ konstrukcja ze słowem *as*.

Rzutowanie

Metoda pierwsza to dokonanie rzutowania obiektu na typ interfejsu. Jeśli zatem mamy w tablicy obiekt typu *Shape*, próbujemy rzutować go na typ *IShow*, czyli wykonujemy konstrukcję:

```
(IShow) tablica[indeks];
```

Jeśli teraz obiekt znajdujący się w tablicy implementuje nasz interfejs, takie rzutowanie zakończy się sukcesem. W przeciwnym przypadku wygenerowany zostanie wyjątek *InvalidCastException*. Wyjątek ten możemy przechwycić i odpowiednio zareagować. Zastosujmy zatem tę metodę i poprawmy kod z ćwiczenia 7.10, tak aby działał on zgodnie z pierwotnymi założeniami.

Cwiczenie 7.11.

Popraw kod z ćwiczenia 7.10 w taki sposób, aby metodą Show była wywoływana tylko w stosunku do obiektów implementujących interfejs IShow. Zastosuj metodę rzutowania.

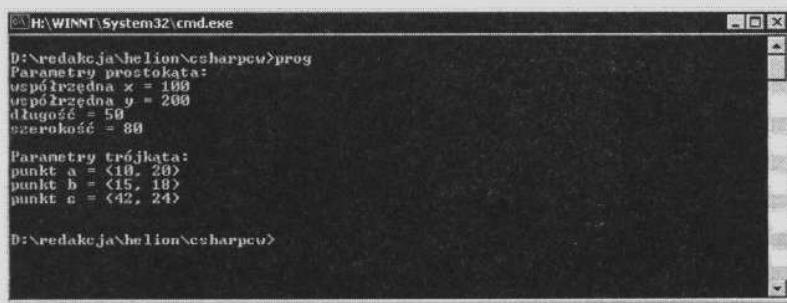
```
using System;

public class main
{
    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){
            }
        }
    }
}
```

Tym razem metoda Show zostanie wywołana jedynie dla obiektów klas Rectangle i Triangle. Widać to wyraźnie na rysunku 7.7.

Rysunek 7.7.

Metoda Show została wywołana jedynie dla obiektów klas Rectangle i Triangle



Oczywiście blok try...catch możemy zapisać w nieco bardziej czytelnej postaci, mianowicie:

```
try{
    IShow tempObj = (IShow)tab[i];
    tempObj.Show();
}
catch(InvalidCastException){
}
```

Możliwe byłoby również wyrzucenie instrukcji `tempObj.Show()` poza blok `try...catch`. Pętla for wyglądałaby w takim przypadku następująco:

```
for(int i = 0; i < tab.Length; i++){
    IShow tempObj;
    try{
        tempObj = (IShow)tab[i];
    }
    catch(InvalidCastException){
        continue;
    }
    tempObj.Show();
}
```

Słowo kluczowe as

Sposób drugi to użycie słowa kluczowego `as`. Możemy je wykorzystać w sposób następujący:
obiekt as interfejs;

Na przykład:

```
IShow obj = tablica[indeks] as IShow;
```

W przypadku, jeśli obiekt nie implementuje danego interfejsu, konstrukcja `as` zwraca wartość `null`. W powyższym przypadku, jeśli w tablicy tablica na pozycji indeks nie znajduje się obiekt implementujący interfejs `IShow`, zmiennej `obj` zostanie przypisana wartość `null`.

Tak naprawdę konstrukcja ze słowem kluczowym `as` to również rodzaj rzutowania, realizowany tylko nieco innym sposobem niż w poprzednim przykładzie.

Ćwiczenie 7.12.

Popraw kod z ćwiczenia 7.10 w taki sposób, aby metodą `Show` była wywoływana tylko w stosunku do obiektów implementujących interfejs `IShow`. Wykorzystaj konstrukcję ze słowem kluczowym `as`.

```
using System;

public class main
{
    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++){
            IShow tempObj = tab[i] as IShow;
            if(tempObj != null){
                tempObj.Show();
            }
        }
    }
}

```

Słowo kluczowe `is`

Ostatnim z prezentowanych sposobów stwierdzenia, czy obiekt implementuje dany interfejs jest użycie słowa kluczowego `is`. Najczęściej używamy go w połączeniu z klasyczną konstrukcją warunkową `if`. Schemat postępowania wygląda tu następująco:

```

if(obiect is interfejs){
    //instrukcje do wykonania, gdy obiekt implementuje interfejs
}

```

Widać więc wyraźnie, że i ten sposób doskonale nadaje się do użycia w naszym przypadku. Wykonajmy zatem ostatnie w tym rozdziale ćwiczenie.

Ćwiczenie 7.13.

Popraw kod z ćwiczenia 7.10 w taki sposób, aby metodą `Show` była wywoływana tylko w stosunku do obiektów implementujących interfejs `IShow`. Wykorzystaj konstrukcję ze słowem kluczowym `is`.

```

using System;

public
class main
{
    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++){
            if(tab[i] is IShow){
                tab[i].Show();
            }
        }
    }
}

```

Część II

Programowanie w Windows

Rozdział 8.

Pierwsze okno

Utworzenie okna

Napisaliśmy już bardzo wiele programów konsolowych, najwyższy zatem czas zając się tworzeniem aplikacji z graficznym interfejsem użytkownika. Podobnie jak w części pierwszej kod będziemy pisać „ręcznie”, nie korzystając z pomocy edytora *Visual Studio*. Dzięki temu dobrze poznamy mechanizmy rządzace aplikacjami okienkowymi.

Podstawowy szablon kodu pozostaje taki sam jak w przypadku przykładów tworzonych w części pierwszej. Dodatkowo musimy poinformować kompilator o tym, że chcemy korzystać z klas pakietu *Windows.Forms*. Dodajemy zatem dyrektywę `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 nam potrzebna klasa `Form` z platformy .NET. Należy utworzyć jej instancję oraz przekazać ją jako parametr w wywołaniu instrukcji `Application.Run()`. A zatem w metodzie `Main` powinna znaleźć się linia: `Application.Run(new Form());`

Ćwiczenie 8.1.

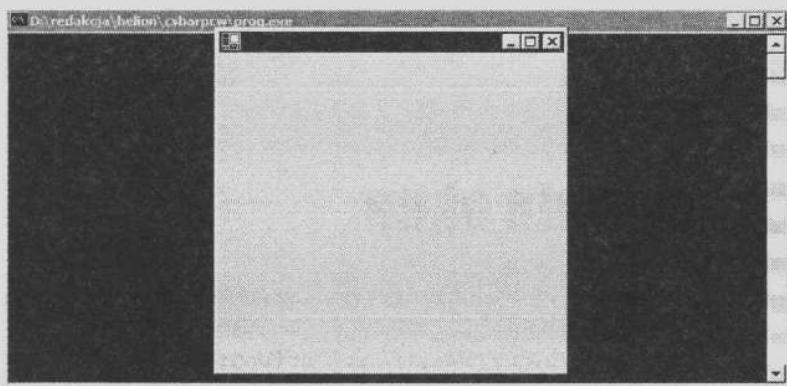
Napisz aplikację okienkową, której jedynym zadaniem będzie wyświetlenie na ekranie okna (rysunek 8.1).

```
using System;
using System.Windows.Forms;

public
class main
{
    public static void Main()
    {
        Application.Run(new Form());
    }
}
```

Rysunek 8.1.

Wynik działania kodu z ćwiczenia 8.1



Takie okienko nic robi nie pozytecznego, ale zauważmy, że mamy do dyspozycji działające przyciski służące do minimalizacji, maksymalizacji oraz zmykania, a także typowe menu systemowe. Osoby programujące w Javie powinny zwrócić też uwagę, że faktycznie taką aplikację faktycznie można zamknąć, klikając odpowiedni przycisk.

Nie będziemy jednak zapewne zadowoleni z jednej rzeczy. Otóż, niezależnie od tego, czy będziemy uruchamiali tak skompilowany program z poziomu wiersza poleceń, czy też klikając jego ikonę, zawsze w tle pojawiać się będzie okno konsoli. Jest to widoczne na rysunku 8.1. Podwód takiego zachowania jest prosty do wyjaśnienia. Domyślnie kompilator zakłada, że tworzymy aplikację konsolową. Dopiero ustawienie odpowiedniej opcji komplikacji zmieni ten stan rzeczy (por. tabela 1.1). Zamiast zatem pisać:

```
csc program.cs
```

musimy skorzystać z polecenia:

```
csc /target:winexe program.cs
```

lub z jego formy skróconej:

```
csc /t:winexe program.cs
```

Klasa Form udostępnia nam cały zbiór właściwości, które wpływają na jej zachowanie. Część z nich zaprezentowana jest w tabeli 8.1. Pozwalają one, między innymi, na zmianę rozmiarów, kolorów czy przypisanego kroju czcionki.

Tabela 8.1. Wybrane właściwości klasy Form

Nazwa właściwości	Typ	Znaczenie
<i>AutoScale</i>	bool	określa, czy okno ma automatycznie dopasowywać swój rozmiar do używanego aktualnie kroju czcionki
<i>AutoScroll</i>	bool	określa, czy w oknie mają się automatycznie pojawiać paski przewijania
<i>BackColor</i>	Color	kolor tła okna
<i>BackGround-Image</i>	Image	obraz tła okna
<i>Bounds</i>	Bounds	rozmiar oraz położenie okna
<i>ClientSize</i>	Size	rozmiar obszaru roboczego okna
<i>ContextMenu</i>	ContextMenu	menu kontekstowe przypisane do okna
<i>Cursor</i>	Cursor	rodzaj kurSORA wyświetlanego kiedy wskaźnik myszy znajdzie się nad oknem
<i>Font</i>	Font	rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się na oknie
<i>ForeColor</i>	Color	kolor używany do rysowania obiektów w oknie
<i>FormBorder-Style</i>	FormBorder-Style	ustala typ ramki okalającej okno
<i>Height</i>	int	wysokość okna
<i>Icon</i>	Icon	ustala ikonę przypisaną do okna
<i>Left</i>	int	położenie lewego górnego rogu, w poziomie, w pikselach
<i>Location</i>	Point	współrzędne lewego górnego rogu okna
<i>Menu</i>	Menu	menu główne przypisane do okna
<i>Modal</i>	bool	decyduje, czy okno ma być modalne
<i>Name</i>	string	nazwa okna
<i>Parent</i>	Control	referencja do obiektu nadrzędnego okna
<i>ShowTaskBar</i>	bool	decyduje, czy ma być wyświetlany pasek narzędzi okna
<i>Size</i>	Size	wysokość i szerokość okna
<i>Top</i>	int	położenie okna w pionie, w pikselach
<i>Visible</i>	bool	określa, czy okno ma być widoczne
<i>Width</i>	int	rozmiar okna w poziomie, w pikselach
<i>WindowState</i>	FormWindow-State	reprezentuje bieżący stan okna

Ćwiczenie 8.2.

Napisz aplikację wyświetlającą okno o rozmiarze 320×200 pikseli.

```
using System;
using System.Windows.Forms;

public
class MainForm:Form
{
    Button button = new Button();
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;
    }
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Wyświetlanie komunikatu

Aby wyświetlić okno z komunikatem (rysunek 8.2), należy skorzystać z klasy MessageBox. Udostępnia ona metodę Show wyświetlającą ciąg znaków podany jako parametr. Ponieważ jest to metoda statyczna, nie musimy wcześniej tworzyć obiektu MessageBox, wystarczy wywołanie:

```
MessageBox.Show("tekst");
```

Ćwiczenie 8.3.

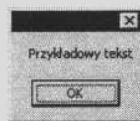
Wyświetl okno z komunikatem (rysunek 8.2).

```
using System;
using System.Windows.Forms;

public
class main
{
    public static void Main()
    {
        MessageBox.Show("Przykładowy tekst");
    }
}
```

Rysunek 8.2.

Wynik działania
metody Show
klasy MessageBox



Wyświetlając tą metodę okno z komunikatem jest niezależne od okna aplikacji. W powyższym przykładzie okna aplikacji nawet nie tworzyliśmy. Ten fakt można wykorzystać, na przykład, do wyświetlania krótkiego tekstu licencji, przed uruchomieniem głównego programu.

Ćwiczenie 8.4.

Wyświetl okno z komunikatem, które pojawi się przed pojawieniem się okna aplikacji.

```
using System;
using System.Windows.Forms;

public
class main
{
    public static void Main()
    {
        MessageBox.Show("Przykładowy tekst");
        Application.Run(new Form());
    }
}
```

Zdarzenie ApplicationExit

Co jednak zrobić w sytuacji, kiedy chcielibyśmy wyświetlić komunikat w czasie, gdy aplikacja jest zamknięta? W jaki sposób wychwycić ten moment? Z pomocą przychodzi nam zdarzenie ApplicationExit. Jeśli połączymy je odpowiednio z napisana przez nas metodą, będziemy mogli wykonać dowolny kod przed wyjściem z programu. Schemat postępowania jest tu następujący:

Piszemy metodę o dowolnej nazwie, na przykład onExit, i następującej deklaracji:

```
private void nazwa_metody(object sender, EventArgs ea)
{
    //tutaj kod metody
};
```

W dalszej kolejności w konstruktorze klasy pochodnej od klasy Form (por. ćwiczenie 8.2) wywołujemy linię:

```
Application.ApplicationExit += new EventHandler(this.nazwa_metody);
```

Ćwiczenie 8.5.

Napisz program wyświetlający okno główne. Przy próbie zamknięcia aplikacji wyświetl okno z dowolnym komunikatem.

```
using System;
using System.Windows.Forms;

public
class MainForm:Form
```

```
{  
    Button button = new Button();  
    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());  
    }  
}
```

Rozdział 9.

Delegacje i zdarzenia

Delegacje

Zanim zajmiemy się dalszym rozwijaniem aplikacji okienkowych, musimy zapoznać się z tematem delegacji i zdarzeń. Wiadomości te będą nam niezbędne, abyśmy mogli reagować na generowane przez użytkownika zdarzenia, 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 oczywiście są konstrukcjami bardziej zaawansowanymi.

Rozważmy taki przykład: mamy dowolną, stworzoną przez nas klasę. Klasa ta będzie zawierała dwa pola o nazwach `x` i `y` oraz metodę `Show` wyświetlającą dane na ekranie. Realizacja takiego zadania na pewno nie przysporzy nam żadnego problemu, wszystkie potrzebne wiadomości zostały podane w rozdziale czwartym. Napiszmy więc krótki fragment kodu.

Ćwiczenie 9.1.

Napisz kod klasy zawierającej dwa pola typu `int` o nazwach `x` i `y` oraz metodę `Show`, wyświetlającą dane na ekranie.

```
using System;

public
class example
{
    public int x;
    public int y;
    public example(int x, int y)
    {
        this.x = x;
        this.y = y;
```

```
    }
    public void Show()
    {
        Console.WriteLine("x = {0}", x);
        Console.WriteLine("y = {0}", y);
    }
}
```

Załóżmy jednak, że chcielibyśmy mieć możliwość decydowania o kształcie metody Show i tym, co ona robi w zupełnie innym miejscu programu. Mówiąc konkretnie, klasa example ma zawierać metodę Show, ale implementacja tej metody mogłaby się zmieniać w trakcie działania programu. Raz wypisywałibyśmy na ekranie tylko wartość zmiennej x, innym razem tylko zmiennej y, a czasem chcielibyśmy, aby na ekranie pojawiało się pełne zdanie Wartość x wynosi ..., a wartość y Jak to zrobić? Najprościej będzie użyć właśnie delegacji.

Ćwiczenie 9.2.

Zmodyfikuj klasę z ćwiczenia 9.1 w taki sposób, aby wywoływanie funkcji Show odbywało się przez delegację.

```
using System;

public class Example
{
    public int x;
    public int y;
    public Example(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public delegate void ShowCallBack(Example examp);
    public void Show(ShowCallBack examp)
    {
        examp(this);
    }
}
```

Co tu miało miejsce? Przede wszystkim została zadeklarowana delegacja o nazwie ShowCallBack, której parametrem jest obiekt klasy Example. 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. Zapewne dla osób nie znających delegacji brzmi to bardzo zawile, ale zobaczymy, jak w praktyce wygląda wykorzystanie tego mechanizmu, a wszystko powinno stać się bardziej zrozumiałe.

Ćwiczenie 9.3.

Wykonaj przykład wykorzystujący mechanizm delegacji utworzonej w ćwiczeniu 9.2 klasy Example.

```
public  
class main  
{  
    public static void Show(Example examp)  
    {  
        Console.WriteLine("x = {0}", examp.x);  
        Console.WriteLine("y = {0}", examp.y);  
    }  
    public static void Main()  
    {  
        Example przyklad = new Example(100, 200);  
        Example.ShowCallBack callBack = new Example.ShowCallBack(main.Show);  
        przyklad.Show(callBack);  
    }  
}
```

Utworzyliśmy w tym ćwiczeniu klasę main. Zawiera ona funkcję Show, która otrzymuje jako parametr obiekt klasy Example i odczytuje wartości x i y tego obiektu. Wykonywanie kodu zaczyna się, jak dobrze wiemy, od funkcji Main. Tworzymy w niej nowy obiekt przykład klasy Example

```
Example przyklad = new Example(100, 200);
```

Następnie tworzymy nową delegację callBack powiązaną z funkcją Show

```
Example.ShowCallBack callBack = new Example.ShowCallBack(main.Show);
```

Ostatnim krokiem jest wywołanie metody Show i przekazanie jej jako parametru delegacji callBack.

Ćwiczenie 9.4. ——————

Napisz i przetestuj kilka delegacji dla metody Show klasy Example.

```
public  
class main  
{  
    public static void Show1(Example examp)  
    {  
        Console.WriteLine("x = {0}", examp.x);  
        Console.WriteLine("y = {0}\n", examp.y);  
    }  
    public static void Show2(Example examp)  
    {  
        Console.WriteLine("Wartość x wynosi {0}", examp.x);  
        Console.WriteLine("Wartość y wynosi {0}\n", examp.y);  
    }  
    public static void Show3(Example examp)  
    {  
        Console.WriteLine("Obiekt ten posiada następujące pola:");  
        Console.WriteLine("Pole x o wartości {0}", examp.x);  
        Console.WriteLine("Pole y o wartości {0}\n", examp.y);  
    }  
    public static void Main()  
    {  
        Example przyklad = new Example(100, 200);  
        Example.ShowCallBack callBack1 = new Example.ShowCallBack(main.Show1);  
    }  
}
```

```

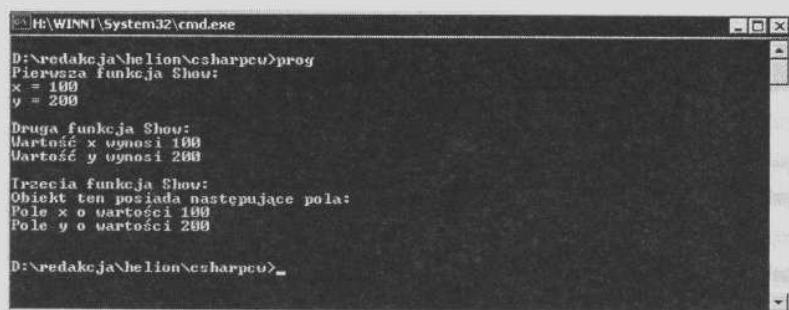
Example.ShowCallBack callBack2 = new Example.ShowCallBack(main.Show2);
Example.ShowCallBack callBack3 = new Example.ShowCallBack(main.Show3);
Console.WriteLine("Pierwsza funkcja Show:");
przyklad.Show(callBack1);
Console.WriteLine("Druga funkcja Show:");
przyklad.Show(callBack2);
Console.WriteLine("Trzecia funkcja Show:");
przyklad.Show(callBack3);
}
}

```

Efekt działania kodu z powyższego ćwiczenia widoczny jest na rysunku 9.1. Naocznie możemy się więc przekonać, że delegacje działają zgodnie z naszymi założeniami.

Rysunek 9.1.

Wynik działania trzech różnych delegacji dla funkcji Show



Tym razem zdefiniowaliśmy trzy delegacje i trzy funkcje Show, które w różny sposób wyświetlają dane z obiektu klasy Example. Sposób postępowania jest tu oczywiście dokładnie taki sam jak w ćwiczeniu 9.3. Jeszcze raz podkreślimy, że najważniejsze jest dla nas to, że wywołania metody Show pozwalają na wykonanie praktycznie dowolnego kodu, nie wymagając przy tym żadnych zmian w klasie Example.

Zdarzenia

Mechanizm zdarzeń oparty jest na delegacjach i pozwala, aby obiekt mógł informować o swoim stanie, czy też właśnie o zajściu jakiegoś zdarzenia. Najczęściej wykorzystuje się tę możliwość podczas tworzenia aplikacji z interfejsem graficznym. Właśnie dzięki zdarzeniom możemy wtedy reagować na kliknięcie myszą przycisku czy wybranie pozycji z menu. Zanim jednak przejdziemy do obsługi zdarzeń w środowisku okienkowym, zobaczymy, w jaki sposób te mechanizmy działają.

Do definiowania zdarzeń służy słowo kluczowe event. Jednak wcześniej musimy utworzyć odpowiadającą zdarzeniu delegację, a zatem, postępowanie jest dwuetapowe:

1. Definiujemy delegację w postaci:

```
public void delegate nazwa_delegacji(parametry);
```

2. Definiujemy jedno lub więcej zdarzeń w postaci:

```
public static event nazwa_delegacji nazwa_zdarzenia
```

Spróbujmy wykonać praktyczny przykład. Założymy, że mamy klasę Example w postaci:

```
public  
class Example  
{  
    private int x;  
    public Example()  
    {  
        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 chcemy przypisać liczbę ujemną, wysyła nam, poprzez zdarzenie, stosowaną informację. Czas zatem na utworzenie odpowiedniej delegacji i zdarzenia.

Ćwiczenie 9.5.

Zmodyfikuj klasę Example w taki sposób, aby po przypisaniu do pola x wartości ujemnej generowała ona odpowiednie zdarzenie.

```
public  
class Example  
{  
    private int x;  
    public Example()  
    {  
        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 zdefiniowaliśmy tu zarówno delegację:

```
public delegate void EventHandler();
```

jak i powiązane z nią zdarzenie:

```
public static event EventHandler XJestUjemne;
```

W metodzie setX sprawdzamy natomiast, czy przekazany parametr nie jest przypadkiem mniejszy od zera. Jeśli tak, wywołujemy zdarzenie:

```
XJestUjemne();
```

Niezbędne jest jednak wcześniejsze sprawdzenie, czy do tego zdarzenia zastała przypisana jakaś procedura obsługi! Służy temu instrukcja:

```
if(XJestUjemne != null)
```

Zobaczmy teraz, w jaki sposób przypisać tę procedurę obsługi do obiektu. Należy skorzystać z operatora += w postaci:

```
NazwaKlasy.NazwaZdarzenia += new NazwaKlasy.NazwaDelegacji(ProceduraObsługi);
```

Ćwiczenie 9.6.

Napisz procedurę obsługi zdarzenia dla klasy Example. Spróbuj przypisać polu x wartość ujemną i sprawdź, czy program działa zgodnie z założeniami.

```
using System;
public class main
{
    public static void onUjemne()
    {
        Console.WriteLine("X jest ujemne!!!");
    }
    public static void Main()
    {
        Example examp = new Example();
        Example.XJestUjemne += new Example.EventHandler(onUjemne);
        examp.setX(-1);
    }
}
```

Procedurą obsługi zdarzenia jest tu metoda onUjemne, wiążemy ją ze zdarzeniem w wierszu:

```
Example.XJestUjemne += new Example.EventHandler(onUjemne);
```

Co jednak zrobić w sytuacji, kiedy chcielibyśmy w procedurze obsługi mieć dostęp do obiektu, który zdarzenie wygenerował? Nie ma z tym oczywiście najmniejszego problemu. Wystarczy odpowiednio skonstruować delegację. A gdybyśmy chcieli jednemu zdarzeniu przypisać kilka procedur obsługi? To też jest jak najbardziej możliwe. Wystarczy kilka razy skorzystać z operatora +=.

Ćwiczenie 9.7.

Zmodyfikuj utworzone w poprzednich ćwiczeniach klasy `main` oraz `Example` w taki sposób, aby procedura obsługi zdarzenia otrzymywała jako parametr obiekt zdarzenia generujący. Napisz dwie różne procedury obsługi zdarzenia.

```
using System;

public
class Example
{
    private int x;
    public Example()
    {
        setX(0);
    }
    public delegate void EventHandler(Example param);
    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 main
{
    public static void onUjemne1(Example param)
    {
        Console.WriteLine("X jest równe: {0}", param.getX());
    }
    public static void onUjemne2(Example param)
    {
        Console.WriteLine("Druga procedura obsługi zdarzenia!");
    }
    public static void Main()
    {
        Example examp = new Example();
        Example.XJestUjemne += new Example.EventHandler(onUjemne1);
        Example.XJestUjemne += new Example.EventHandler(onUjemne2);
        examp.setX(-1);
    }
}
```

Rozdział 10.

Komponenty

Etykiety (Label)

Klasa `Label` służy do utworzenia obiektu wyświetlającego linie tekstu. Tekst ten może być zmieniany przez aplikację, użytkownik nie ma natomiast możliwości bezpośredniej jego edycji. Aby skorzystać z etykiety, należy utworzyć obiekt klasy `Label`, dodać go kolekcji `Controls` okna, na którym ta etykieta ma się znajdująć, oraz wykorzystać instrukcję `Controls.Add(nazwa_etykiety)`. Wygląd oraz zachowanie etykiety możemy modyfikować, korzystając z jej właściwości, które zebrane są w tabeli 10.1.

Tabela 10.1. Wybrane właściwości klasy `Label`

Nazwa właściwości	Typ	Znaczenie
<code>Autosize</code>	<code>bool</code>	ustala, czy etykieta ma automatycznie dopasowywać swój rozmiar do zawartego na niej tekstu
<code>BackColor</code>	<code>Color</code>	kolor tła etykiety
<code>BorderStyle</code>	<code>BorderStyle</code>	styl ramki otaczającej etykię
<code>Bounds</code>	<code>Bounds</code>	rozmiar oraz położenie etykiety
<code>Cursor</code>	<code>Cursor</code>	rodzaj kurSORA wyświetlany, kiedy wskaźnik myszy znajdzie się nad etykietą
<code>Font</code>	<code>Font</code>	rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się na etykiecie
<code>ForeColor</code>	<code>Color</code>	kolor tekstu etykiety
<code>Height</code>	<code>int</code>	wysokość etykiety
<code>Image</code>	<code>Image</code>	obraz wyświetlany na etykiecie
<code>Left</code>	<code>Int</code>	położenie lewego górnego rogu, w poziomie, w pikselach

Tabela 10.1. Wybrane właściwości klasy Label – ciąg dalszy

Nazwa właściwości	Typ	Znaczenie
<i>Location</i>	Point	współrzędne lewego górnego rogu etykiety
<i>Name</i>	string	nazwa etykiety
<i>Parent</i>	Control	referencja do obiektu zawierającego etykię
<i>Size</i>	Size	wysokość i szerokość etykiety
<i>Text</i>	string	tekst wyświetlany na etykiecie
<i>TextAlign</i>	Content-Alignment	położenie tekstu na etykiecie
<i>Top</i>	int	położenie etykiety w pionie, w pikselach
<i>Visible</i>	bool	określa, czy etykieta ma być widoczna
<i>Width</i>	int	rozmiar etykiety w poziomie

Ćwiczenie 10.1. ——————

Dodaj do okna aplikacji komponent Label z dowolnym tekstem. Etykieta powinna znaleźć się w centrum formy.

```
using System;
using System.Windows.Forms;

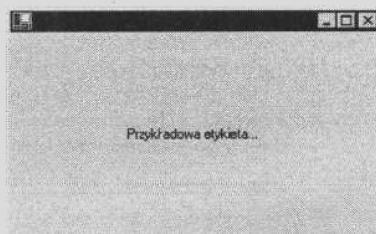
public class MainForm : Form
{
    Label label = new Label();
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;
        label.Text = "Przykładowa etykieta... ";
        label.AutoSize = true;
        label.Left = (this.ClientSize.Width - label.Width) / 2;
        label.Top = (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 formy. Takie umiejscowienie etykiety uzyskujemy poprzez modyfikację właściwości Top oraz Left. Aby uzyskać odpowiednie wartości wykonujemy tu proste działania matematyczne:

Współrzędna X = (długość okna - długość etykiety) / 2
 Współrzędna Y = (wysokość okna - wysokość etykiety) / 2

Rysunek 10.1.

Etykieta tekstowa
umiejscowiona
w centrum okna



Oczywiście działania te należy wykonać po przypisaniu etykiecie tekstu, inaczej obliczenia nie będą uwzględniały jej prawidłowej wielkości. Do uzyskania szerokości, a szczególnie wysokości formy, należy wykorzystać właściwości *ClientWidth* oraz *ClientHeight*. Podają one bowiem rozmiar okna, po odliczeniu okalającej ramki oraz paska tytułowego, i ewentualnego menu, czyli po prostu wielkość okna, którą mamy do naszej dyspozycji i na której możemy umieszczać inne obiekty.

Tekst wyświetlany na etykiecie może być wyświetlany dowolną czcionką zainstalowaną w systemie. Wystarczy zmodyfikować właściwość *Font* obiektu *Label*. Aby jednak takiej modyfikacji dokonać, trzeba najpierw utworzyć obiekt klasy *Font*. Klasa ta posiada kilka konstruktorów, my skorzystamy z konstruktora w 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*

Wartość członku	Opis
<i>Bold</i>	tekst pogrubiony
<i>Italic</i>	tekst pochylony
<i>Regular</i>	tekst zwyczajny
<i>Strikeout</i>	tekst przekreślony
<i>Underline</i>	tekst podkreślonny

FontFamily jest to klasa reprezentująca rodzinę czcionek. Udostępnia on również stałą właściwość *Families*, która zwraca tablicę z listą wszystkich dostępnych fontów w systemie. Kolejne ćwiczenie pokaże nam, jak taką listę uzyskać.

Ćwiczenie 10.2. 

Wypisz na ekranie listę wszystkich dostępnych czcionek.

```
using System;
using System.Drawing;
using System.Windows.Forms;

public
class MainForm:Form
{
```

```
public static void Main()
{
    FontFamily[] listaCzcionek;
    listaCzcionek = FontFamily.Families;
    foreach(FontFamily font in listaCzcionek){
        Console.WriteLine(font.Name);
    }
}
```

Przykładowa lista czcionek uzyskana za pomocą tego programu widoczna jest na rysunku 10.2. Sam kod nie jest zbyt skomplikowany, wykorzystujemy tu zwyczajną pętlę typu `foreach` (por. rozdział „Tablice”), która przebiega przez wszystkie elementy tablicy `listaCzcionek`. W celu uzyskania nazwy danego fontu odwołujemy się do właściwości `Name` klasy `FontFamily`.

Rysunek 10.2.

Lista czcionek uzyskana dzięki programowi z ćwiczenia 10.2



The screenshot shows a Windows Command Prompt window titled 'cmd.exe' with the path 'D:\redakcja\helion\csharp\program'. The window displays a list of font names, starting with 'Allegro BT' and ending with 'Impact'. The list includes many standard Windows fonts such as Arial, Helvetica, and Times New Roman, along with some less common ones like 'Goudy Old Style' and 'Haettenschweiler'.

```
D:\redakcja\helion\csharp\program
Allegro BT
Arial
Arial Black
AvantGarde Bk BT
AvantGarde Md BT
BookGothic Md BT
Benguiat Bk BT
BernhardFashion BT
BernhardMod BT
Book Antiqua
Bremen Bd BT
Century Gothic
Charlesworth
Comic Sans MS
CommonBullets
CopperplateGoth Bd BT
Courier New
Dauphin
Englishhilli Vivace BT
Future Lt BT
Future Md BT
Future XBold BT
FutureBlack BT
Georgia
GoudyOldStyle BT
Haettenschweiler
Humanist521 BT
Impact
```

Skoro mamy już tyle wiadomości na temat fontów, spróbujmy wyświetlić tekst na etykiecie wybrana czcionką, na przykład typu Courier. W większości systemów jest ona standardowo zainstalowana. Skorzystamy z przedstawionego wcześniej konstruktora klasy `Font`, nie będziemy tworzyć jednak bezpośrednio obiektu klasy `FontFamily`, 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);
```

Ćwiczenie 10.3.

Dodaj do okna aplikacji komponent `Label` z dowolnym tekstem napisanym czcionką `Courier` o wielkości 10 punktów (rysunek 10.3).

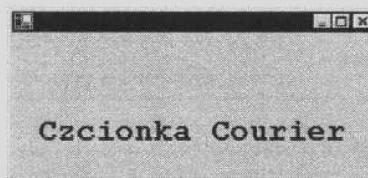
```
using System;
using System.Windows.Forms;
using System.Drawing;
```

```

public
class MainForm:Form
{
    Label label = new Label();
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;
        Font font = new Font("Courier", 20, FontStyle.Bold);
        label.Font = font;
        label.Text = "Czcionka Courier";
        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());
    }
}

```

Rysunek 10.3.
Etykietka używająca
czcionki Courier
o wielkości 20 punktów



Przyciski (klasa Button)

Klasa Button umożliwia utworzenie w oknie standardowego przycisku z dowolnym napisem. Aby z niej skorzystać, należy utworzyć nowy obiekt tej klasy, ustalić jego położenie oraz napisać procedurę obsługi zdarzeń. Przycisk umiejscawiamy na formie, wykorzystując właściwości *Top* oraz *Left*, podobnie jak przypadku innych komponentów (na przykład etykiet). Przydatne właściwości klasy Button przedstawione są w tabeli 10.3.

Tabela 10.3. Wybrane właściwości klasy Button

Nazwa właściwości	Typ	Znaczenie
<i>BackColor</i>	Color	kolor tła przycisku
<i>Bounds</i>	Bounds	rozmiar oraz położenie przycisku
<i>Cursor</i>	Cursor	rodzaj kurSORA wyświetlanego, kiedy wskaźnik myszy znajdzie się nad przyciskiem
<i>FlatStyle</i>	FlatStyle	modyfikuje styl przycisku
<i>Font</i>	Font	rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się na przycisku
<i>ForeColor</i>	Color	kolor tekstu przycisku

Tabela 10.3. Wybrane właściwości klasy Button — ciąg dalszy

Nazwa właściwości	Typ	Znaczenie
<i>Height</i>	int	wysokość przycisku
<i>Image</i>	Image	obraz wyświetlany na przycisku
<i>Left</i>	int	położenie lewego górnego rogu, w poziomie, w pikselach
<i>Location</i>	Point	współrzędne lewego górnego rogu przycisku
<i>Name</i>	string	nazwa przycisku
<i>Parent</i>	Control	referencja do obiektu zawierającego przycisk
<i>Size</i>	Size	wysokość i szerokość przycisku
<i>Text</i>	string	tekst wyświetlany na przycisku
<i> TextAlign</i>	Content-Alignment	położenie tekstu na przycisku
<i>Top</i>	int	położenie etykiety w pionie, w pikselach
<i>Visible</i>	bool	określa, czy przycisk ma być widoczny
<i>Width</i>	int	rozmiar przycisku w poziomie w pikselach

Ćwiczenie 10.4. 

Utwórz okno, w którym będzie znajdował się przycisk (rysunek 10.4).

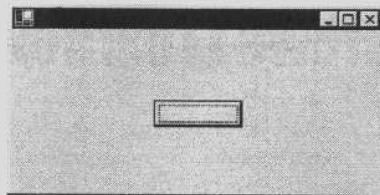
```
using System;
using System.Windows.Forms;

public class MainForm:Form
{
    Button button = new Button();
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;
        button.Top = 60;
        button.Left = 120;
        this.Controls.Add(button);
    }
}

public class main
{
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Rysunek 10.4.

Przycisk dodany
w głównym
oknie aplikacji



Przycisk taki nie reaguje jednak na kliknięcie, nie ma też na nim żadnego napisu. Konieczne zatem będą kolejne modyfikacje. Na przypisanie tekstu do przycisku pozwala właściwość *Text*. Z kolei reakcje na kliknięcie zapewni nam dodanie procedury obsługi zdarzeń do właściwości *Click*:

```
button.Click += eh;
```

gdzie eh to obiekt klasy *EventHandler*.

Ćwiczenie 10.5.

Zmodyfikuj kod z ćwiczenia 10.4 w taki sposób, aby po kliknięciu przycisku następowała zamknięcie aplikacji.

```
using System;
using System.Windows.Forms;

public
class MainForm:Form
{
    Button button = new Button();
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;
        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
class main
{
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Pola tekstowe (TextBox)

Klasa `TextBox` służy do utworzenia pola tekstowego, umożliwiającego wprowadzenie przez użytkownika ciągu znaków. W zależności od ustawień pole tekstowe może być jedno lub wielowierszowe. Zmiany można dokonać poprzez odpowiednie ustawienie właściwości `Multiline`. Wybrane właściwości klasy przedstawione są w tabeli 10.4.

Tabela 10.4. Wybrane właściwości klasy `TextBox`

Nazwa właściwości	Typ	Znaczenie
<code>Autosize</code>	<code>bool</code>	ustala, czy pole tekstowe ma automatycznie dopasowywać swój rozmiar do zawartego w nim tekstu
<code>BackColor</code>	<code>Color</code>	kolor tła pola tekstowego
<code>BackgroundImage</code>	<code>Image</code>	obraz znajdujący się w tle okna tekstowego
<code>BorderStyle</code>	<code>BorderStyle</code>	styl ramki otaczającej pole tekstowe
<code>Bounds</code>	<code>Bounds</code>	rozmiar oraz położenie pole tekstowe
<code>Cursor</code>	<code>Cursor</code>	rodzaj kursora wyświetlonego, kiedy wskaźnik myszy znajdzie się nad polem tekstowym
<code>Font</code>	<code>Font</code>	rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się na polu
<code>ForeColor</code>	<code>Color</code>	kolor tekstu pola tekstowego
<code>Height</code>	<code>int</code>	wysokość pola tekstowego
<code>Left</code>	<code>int</code>	położenie lewego górnego rogu, w poziomie, w pikselach
<code>Lines</code>	<code>string[]</code>	tablica zawierająca poszczególne linie tekstu zawarte w polu tekstowym
<code>Location</code>	<code>Point</code>	współrzędne lewego górnego rogu pola tekstowego
<code>MaxLength</code>	<code>int</code>	maksymalna ilość znaków, które można wprowadzić do pole tekstowego
<code>Modified</code>	<code>bool</code>	określa, czy zawartość pola tekstowego była modyfikowana
<code>Multiline</code>	<code>bool</code>	określa, czy pole tekstowe ma zawierać jedną, czy wiele linii tekstu
<code>Name</code>	<code>string</code>	nazwa pola tekstowego
<code>Parent</code>	<code>Control</code>	referencja do obiektu zawierającego pole tekstowe
<code>PasswordChar</code>	<code>char</code>	określa, jaki znak będzie wyświetlany w polu tekstowym w celu zamaskowania wprowadzanego tekstu; aby skorzystać z tej opcji, właściwość <code>Multiline</code> musi być ustawiona na <code>false</code>
<code>ReadOnly</code>	<code>bool</code>	określa, czy pole tekstowe ma być ustawione w trybie tylko do odczytu
<code>SelectedText</code>	<code>string</code>	zaznaczony fragment tekstu w polu tekstowym
<code>SelectionLength</code>	<code>int</code>	ilość znaków w zaznaczonym fragmencie tekstu
<code>SelectionStart</code>	<code>int</code>	indeks pierwszego znaku zaznaczonego fragmentu tekstu

Tabela 10.4. Wybrane właściwości klasy *TextBox* — ciąg dalszy

Nazwa właściwości	Typ	Znaczenie
<i>Size</i>	<i>Size</i>	rozmiar pola tekstowego
<i>Text</i>	<i>string</i>	tekst wyświetlany w polu tekstowym
<i> TextAlign</i>	<i>Content-Alignment</i>	położenie tekstu w polu tekstowym
<i>Top</i>	<i>int</i>	położenie pola tekstowego w pionie, w pikselach
<i>Visible</i>	<i>bool</i>	określa czy pole tekstowe ma być widoczne
<i>Width</i>	<i>int</i>	rozmiar pola tekstowego w poziomie
<i>WordWrap</i>	<i>bool</i>	określa, czy słowa mają być automatycznie przenoszone do nowej linii, kiedy nie mieścią się w bieżącej; aby zastosować tą funkcję właściwość <i>Multiline</i> musi być ustawiona na true

Ćwiczenie 10.6. 

Umieść w oknie aplikacji pole tekstowe i przycisk. Po kliknięciu przycisku wyświetl wprowadzony tekst w oknie dialogowym.

```
using System;
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;

        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.CloseClicked);
        button.Click += eh;
        this.Controls.Add(button);
        this.Controls.Add(textBox);
    }
    public void CloseClicked(Object sender, EventArgs e)
    {
        MessageBox.Show(textBox.Text);
    }
}

public class main
```

```
{  
    public static void Main()  
    {  
        Application.Run(new MainForm());  
    }  
}
```

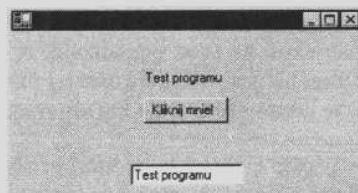
Ćwiczenie 10.7. 

Umieść w oknie aplikacji pole tekstowe, etykietę tekstową i przycisk, tak jak widoczne jest to na rysunku 10.5. Po kliknięciu przycisku przypisz 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;  
  
        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.CloseClicked);  
        button.Click += eh;  
        this.Controls.Add(button);  
        this.Controls.Add(label);  
        this.Controls.Add(textBox);  
    }  
    public void CloseClicked(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



Pola wyboru (CheckBox, RadioButton)

Pola wyboru to kolejne znane elementy interfejsu Windows, które możemy z łatwością stosować, korzystając z dostępnych w .NET klas CheckBox i RadioButton. Wybrane właściwości tych klas przedstawione są w tabeli 10.5.

Tabela 10.5. Wybrane właściwości klas CheckBox i RadioButton

Nazwa właściwości	Typ	Znaczenie
<i>AutoCheck</i>	bool	ustala, czy pole ma być automatycznie zaznaczane lub odznaczane, kiedy użytkownik kliknie na je myszą
<i>BackColor</i>	Color	kolor tła pola
<i>Bounds</i>	Bounds	rozmiar oraz położenie pola
<i>Checked</i>	bool	pozwala stwierdzić, czy pole jest zaznaczone
<i>CheckState</i>	CheckState	ustala sposób zaznaczania pola
<i>Cursor</i>	Cursor	rodzaj kurSORA wyświetlanego, kiedy wskaźnik myszy znajdzie się nad polem
<i>FlatStyle</i>	FlatStyle	ustala styl, w jakim pole będzie wyświetlane
<i>Font</i>	Font	rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się przy polu
<i>ForeColor</i>	Color	kolor tekstu pola
<i>Height</i>	int	wysokość pola
<i>Left</i>	int	położenie lewego górnego rogu, w poziomie, w pikselach
<i>Location</i>	Point	współrzędne lewego górnego rogu pola
<i>Name</i>	string	nazwa pola
<i>Parent</i>	Control	referencja do obiektu zawierającego pole
<i>Size</i>	Size	wysokość i szerokość pola
<i>Text</i>	string	tekst wyświetlany przy polu
<i>ContentAlignment</i>	Content-Alignment	położenie tekstu znajdującego się przy polu
<i>ThreeState</i>	bool	ustala, czy pole ma być dwu czy trójstanowe
<i>Top</i>	int	położenie etykiety w pionie, w pikselach
<i>Visible</i>	bool	określa, czy pole ma być widoczne
<i>Width</i>	int	rozmiar pola w poziomie

Aby sprawdzić, czy pole wyboru jest zaznaczone, należy odwołać się do właściwości *Checked*. Ustawiona na true sygnalizuje, że pole jest zaznaczone, ustawiona na false, że pole zaznaczone nie jest. Właściwości tej można również przypisywać wartości i samemu decydować, w jakim stanie pole ma się znajdować.

Wykonajmy proste ćwiczenie, w którym wyświetlimy na ekranie okno zawierające trzy elementy klasy *CheckBox* oraz przycisk (rysunek 10.6). Po kliknięciu przycisku wyświetlimy informację, które pola zostały zaznaczone.

Ćwiczenie 10.8.

Dodaj do formy trzy obiekty klasy 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;

        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";

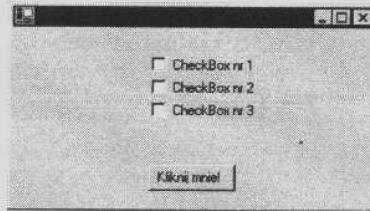
        chb3 = new CheckBox();
        chb3.Left = 120;
        chb3.Top = 60;
        chb3.Text = "CheckBox nr 3";

        EventHandler eh = new EventHandler(this.CloseClicked);
        button.Click += eh;
        this.Controls.Add(button);
        this.Controls.Add(chb1);
        this.Controls.Add(chb2);
        this.Controls.Add(chb3);
    }
    public void CloseClicked(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("Zaznaczone zostały opcje: " + s1 + s2 + s3);  
}  
public static void Main()  
{  
    Application.Run(new MainForm());  
}
```

Rysunek 10.6.

Ćwiczenie obrazujące wykorzystanie pól wyboru typu CheckBox



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 wykluczającymi, czyli w jednej grupie może być zaznaczone tylko jedno z nich.

Ćwiczenie 10.9.

Dodaj do formy trzy obiekty klasy RadioButton oraz przycisk. Po kliknięciu przycisku wyświetli 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;  
  
        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.CloseClicked);
button.Click += eh;
this.Controls.Add(button);
this.Controls.Add(rb1);
this.Controls.Add(rb2);
this.Controls.Add(rb3);

}

public void CloseClicked(object sender, EventArgs e)
{
    string s1 = "", s2 = "", s3 = "";
    if(rb1.Checked)
        s1 = " 1 ";
    if(rb2.Checked)
        s2 = " 2 ";
    if(rb3.Checked)
        s3 = " 3 ";
    MessageBox.Show("Zaznaczona została opcja: " + s1 + s2 + s3);
}

public static void Main()
{
    Application.Run(new MainForm());
}
```

Listy rozwijalne (ComboBox)

Listy rozwijalne możemy tworzyć dzięki klasie ComboBox. Listę jej wybranych właściwości przedstawia tabela 10.6. Dla nas najważniejsza będzie w tej chwili właściwość Items, jako że zawiera ona wszystkie elementy znajdujące się na liście. Właściwość ta jest w rzeczywistości kolekcją elementów typu object. Dodawanie elementów możemy zatem zrealizować, stosując konstrukcję:

```
ComboBox.Items.Add("element");  
natomiast ich usunięcie, wykorzystując:
```

```
ComboBox.Items.Remove("element");
```

Jeżeli chcielibyśmy jednak dodać większą liczbę elementów na raz najwygodniej jest zastosować metodę AddRange w postaci:

```
ComboBox.Items.AddRange(new[] object{
    "Element 1"
    "Element 2"
    ...
    "Element n"
});
```

Wybranie przez użytkownika elementu z listy możemy wykryć poprzez oprogramowanie zdarzenia o nazwie SelectedIndexChanged. Odniesienie do wybranego elementu znajdziemy natomiast we właściwości SelectedItem.

Tabela 10.6. Wybrane właściwości klasy ComboBox

Nazwa właściwości	Typ	Znaczenie
BackColor	Color	kolor tła listy
Bounds	Bounds	rozmiar oraz położenie listy
Cursor	Cursor	rodzaj kurSORA wyświetlonego, kiedy wskaźnik myszy znajdzie się nad listą
Font	Font	rodzaj czcionki, którą będzie wyświetlany tekst znajdujący się przy polu
ForeColor	Color	kolor tekstu
Height	int	wysokość listy
ItemHeight	int	wysokość pojedynczego elementu listy
Items	Object-Collection	lista elementów znajdujących się na liście
Left	int	położenie lewego górnego rogu, w poziomie, w pikselach
Location	Point	współrzędne lewego górnego rogu listy
MaxDrop-DownItems	int	maksymalna liczba elementów, które będą wyświetlane po rozwinięciu listy
MaxLength	int	maksymalna liczba znaków wyświetlanych w polu edycyjnym listy
Name	string	nazwa listy
Parent	Control	referencja do obiektu zawierającego listę
SelectedIndex	int	indeks aktualnie zaznaczonego elementu
SelectedItem	object	aktualnie zaznaczony element
Size	Size	wysokość i szerokość listy
Sorted	bool	ustala, czy elementy listy mają być posortowane
Text	string	tekst wyświetlany w polu edycyjnym listy
Top	int	położenie listy w pionie, w pikselach
Visible	bool	określa, czy lista ma być widoczna
Width	int	rozmiar listy w poziomie

Ćwiczenie 10.10.

Umieść w oknie aplikacji element ComboBox (rysunek 10.7). Po wybraniu pozycji z listy wyświetli jej nazwę przy wykorzystaniu klasy ShowMessage.

```
using System;
using System.Drawing;
using System.Windows.Forms;

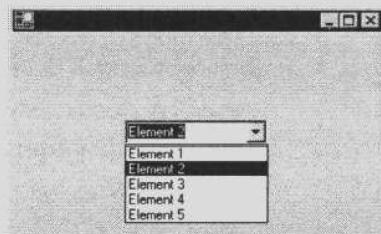
public class MainForm:Form
{
    ComboBox cb = new ComboBox();
    public MainForm()
    {
        this.Width = 320;
        this.Height = 200;

        cb.Items.AddRange(new object[]{
            "Element 1",
            "Element 2",
            "Element 3",
            "Element 4",
            "Element 5"
        });
        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



Listy zwykłe (ListBox)

Obsługa zwykłych list jest bardzo podobna do obsługi elementów ComboBox. Mamy jednak do dyspozycji 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*

Nazwa właściwości	Typ	Znaczenie
<i>MultiColumn</i>	bool	ustala, czy elementy listy mogą być wyświetlane w wielu kolumnach
<i>ScrollAlways-Visible</i>	bool	ustala, czy pasek przewijania ma być stale widoczny
<i>Selected-Indices</i>	SelectedIndex-Collection	lista zawierająca indeksy wszystkich zaznaczonych elementów
<i>Selected-Items</i>	SelectedObject-Collection	lista zawierająca wszystkie zaznaczone elementy
<i>Selection-Mode</i>	SelectionMode	określa sposób, w jaki będą zaznaczane elementy listy

Podstawową różnicą w stosunku do listy *ComboBox*, oprócz sposobu wyświetlania, jest oczywiście możliwość zaznaczania więcej niż jednego elementu. Sposób, w jakim elementy będą zaznaczane, możemy kontrolować dzięki właściwości *SelectionMode*, której można przypisać 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 uzyskujemy dzięki właściwościom *SelectedIndieces*, która zawiera indeksy wszystkich zaznaczonych elementów, oraz *SelectedItems*, która zawiera listę zaznaczonych elementów. Należy zwrócić uwagę, że jeżeli lista pracuje w trybie *MultiSimple* lub *MultiExtended*, właściwości *SelectedIndex* i *SelectedItem* będą wskazywały **dowolny** z zaznaczonych elementów.

Ćwiczenie 10.11.

Umieść w oknie aplikacji listę *ListBox* zawierającą klik elementów oraz przycisk (rysunek 10.8). Po kliknięciu przycisku wyświetl nazwy elementów, które zostały zaznaczone.

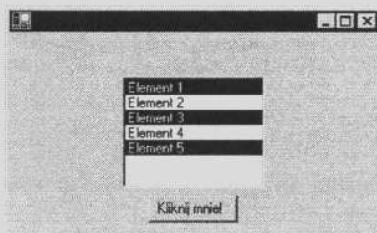
```
using System;
using System.Drawing;
using System.Windows.Forms;

public
class MainForm:Form
```

```
{  
    ListBox lb = new ListBox();  
    Button button = new Button();  
    public MainForm()  
    {  
        this.Width = 320;  
        this.Height = 200;  
  
        button.Top = 140;  
        button.Left = (this.ClientSize.Width - button.Width) / 2;  
        button.Text = "Kliknij mnie!";  
  
        lb.Items.AddRange(new object[]{  
            "Element 1",  
            "Element 2",  
            "Element 3",  
            "Element 4",  
            "Element 5"}  
    );  
    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 s = "";  
    foreach(string name in lb.SelectedItems){  
        s += " " + name + " ";  
    }  
    MessageBox.Show("Zaznaczone elementy: " + s);  
}  
public static void Main()  
{  
    Application.Run(new MainForm());  
}
```

Rysunek 10.8.

Lista pracująca
w trybie MultiExtended
z ćwiczenia 10.11



Menu

Czym są menu z pewnością nie trzeba nikomu przypominać. Czas zatem nauczyć się, w jaki sposób skorzystać z tych pozytecznych struktur w C# na platformie .NET. Do dyspozycji mamy oczywiście zarówno menu główne, jak i menu kontekstowe. Za ich realizację odpowiadają klasy `MainMenu`, `ContextMenu` oraz `MenuItem`. Wszystkie one znajdują się w przestrzeni nazw `Windows.System.Forms`.

Menu główne

Aby dodać do aplikacji menu tego typu, musimy wykonać kilka czynności. Przede wszystkim należy utworzyć obiekt klasy `MainMenu` i przypisać go właściwości `Menu` klasy `Form`. W ten sposób do aplikacji zostanie dodane menu główne, które jednak nie będzie zawierało żadnej pozycji. W celu utworzenia pozycji tworzymy obiekt klasy `MenuItem` i dodajemy go do menu głównego stosując konstrukcję:

```
MainMenu nazwa_menu = new MainMenu();
MenuItem nazwa_pozycji = new MenuItem();
nazwa_menu.MenuItems.Add(nazwa_pozycji);
```

Teraz wystarczy już tylko przypisać utworzonej pozycji tekst, jaki ma się na niej pojawiać, np. `File`. Sposób wykonania tego zadania jest wyjątkowo prosty. Wystarczy właściwości `Text` obiektu klasy `MenuItem` przypisać odpowiedni ciąg znaków. W naszym przykładzie należałoby użyć instrukcji:

```
menuItem1.Text = "File";
```

Ćwiczenie 10.12.

Napisz aplikację wyświetlającą okno zawierające menu główne z pozycją `File`, tak jak widoczne jest to na rysunku 10.9.

```
using System;
using System.Windows.Forms;

public
class MainForm:Form
{
    MainMenu mainMenu = new MainMenu();
    MenuItem menuItem1 = new MenuItem();
    public MainForm()
    {
        menuItem1.Text = "File";
        mainMenu.MenuItems.Add(menuItem1);
        this.Menu = mainMenu;
    }
}

public
class main
{
```

```
public static void Main()
{
    Application.Run(new MainForm());
}
```

Rysunek 10.9.

Aplikacja zawierająca menu główne



Do tak otrzymanego menu należałoby jednak dodać jakąś pozycję — niech to będzie Close — będziemy mogli wykorzystać je do końca pracy z aplikacją. Sposób postępowania jest tu analogiczny jak w przypadku menu głównego. Tworzymy nowy obiekt klasy MenuItem oraz korzystamy z metody MenuItems.Add. Najlepiej zobrazuje nam to kolejne ćwiczenie.

Ćwiczenie 10.13.

Do menu otrzymanego w ćwiczeniu 10.12 dodaj pozycje o nazwie Close, jak widoczne jest to 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()
    {
        menuItem1.Text = "File";
        menuItem2.Text = "Close";
        mainMenu.MenuItems.Add(menuItem1);
        menuItem1.MenuItems.Add(menuItem2);
        this.Menu = mainMenu;
    }
}

public
class main
{
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Rysunek 10.10.

Aplikacja posiada teraz dodatkowe pod-menu



Nazwy poszczególnym menu można nadawać również bezpośrednio w konstruktorze klasy MenuItem. Zaoszczędzimy tym samym nieco miejsca w kodzie programu. Oczywiście żądana ciąg znaków trzeba podać jako parametr, czyli możemy pisać:

```
MenuItem menuItem = new MenuItem("File");
```

Ćwiczenie 10.14.

Utwórz menu 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("File");
    MenuItem menuItem2 = new MenuItem("Close");
    public MainForm()
    {
        .
        mainMenu.MenuItems.Add(menuItem1);
        menuItem1.MenuItems.Add(menuItem2);
        this.Menu = mainMenu;
    }
}

public
class main
{
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Istnieje również jeszcze inna możliwość tworzenia podmenu, nie musimy bowiem bezpośrednio tworzyć obiektów MenuItem. Zrobi to za nas równe dobrze metoda Add, zastosowana do właściwości MenuItems¹. Czyli zamiast pisać

```
MenuItem menuItem1 = new MenuItem("nazwa");
MainMenuBar.MenuItems.Add(menuItem1);
```

możemy napisać:

```
MenuItem menuItem1;
menuItem1 = MainMenuBar.MenuItems.Add("nazwa");
```

Co prawda tak utworzonego obiektu nie musimy koniecznie przypisywać do zmiennej menuItem1, jednak lepiej to zrobić. Będziemy się przecież w przyszłości odwoływać do tego elementu.

¹ W rzeczywistości odwołanie do MenuItems zwraca wskazanie na obiekt klasy zagnieżdzonej Menu.MenuItemCollection. Dopiero ta klasa zawiera zestaw przeciążonych metod Add. Bliższe informacje można znaleźć w dokumentacji .NET SDK oraz Visual C# .NET.

Ćwiczenie 10.15.

Utwórz pozycje menu jak w ćwiczeniu 10.13. Skorzystaj z nowej metody dodawania podmenu.

```
using System;
using System.Windows.Forms;

public
class MainForm:Form
{
    MainMenu mainMenu = new MainMenu();
    MenuItem menuItem1;
    MenuItem menuItem2;
    public MainForm()
    {
        menuItem1 = mainMenu.MenuItems.Add("File");
        menuItem1.MenuItems.Add("Close");
        this.Menu = mainMenu;
    }
}

public
class main
{
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Stworzone w poprzednim ćwiczeniu menu na razie jest niestety nieaktywne, tzn. po jego wybraniu nic się nie dzieje. Nie jest to chyba wielkim zaskoczeniem, nie stworzyliśmy przecież jeszcze odpowiednich procedur jego obsługi. Jak to zrobić? Jak łatwo się domyślić, wykorzystamy w tym celu zdarzenia, konkretnie delegację 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
);
```

Rzecz jasna parametry sender i args nie muszą być używane. Przykładowo, metoda, której zadaniem byłoby reagowanie na wybranie z menu pozycji *Close*, mogłaby wyglądać następująco:

```
public void CloseClicked(object sender, EventArgs e)
{
    this.Close();
}
```

Pozostaje nam zatem powiązanie tejże metody ze zdarzeniem polegającym na wybraniu pozycji *Close*. Należy utworzyć delegację EventHandler i przekazać jako parametr metodę CloseClicked w postaci:

```
EventHandler eh = new EventHandler(CloseClicked);
```

Po takiej deklaracji wiążemy delegację z konkretną pozycją w menu, pisząc:

```
MenuItems.Add("Close", eh);
```

Ćwiczenie 10.16. 

Uzupełnij kod z ćwiczenia 10.15 w taki sposób, aby po wybraniu z menu File pozycji Close następował wyjście z aplikacji.

```
using System;
using System.Windows.Forms;

public class MainForm:Form
{
    MainMenu mainMenu = new MainMenu();
    MenuItem menuItem1;
    MenuItem menuItem2;
    public MainForm()
    {
        menuItem1 = mainMenu.MenuItems.Add("File");
        EventHandler eh = new EventHandler(this.CloseClicked);
        menuItem2 = menuItem1.MenuItems.Add("Close", eh);
        this.Menu = mainMenu;
    }
    public void CloseClicked(Object sender, EventArgs e)
    {
        this.Close();
    }
}

public class main
{
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Ćwiczenie 10.17. 

Napisz aplikację zawierającą kilka pozycji w menu głównym. Po wybraniu dowolnej pozycji z menu wyświetli 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 menuItemSubItem1;
```

```
MenuItem menu1SubItem2;
MenuItem menu2SubItem1;
MenuItem menu3SubItem1;
MenuItem menu3SubItem2;
MenuItem menu3SubItem3;
public MainForm()
{
    menuItem1 = mainMenu.MenuItems.Add("Item 1");
    menuItem2 = mainMenu.MenuItems.Add("Item 2");
    menuItem3 = mainMenu.MenuItems.Add("Item 3");

    EventHandler eh = new EventHandler(this.MenuItemClicked);

    menu1SubItem1 = menuItem1.MenuItems.Add("SubItem 1", eh);
    menu1SubItem2 = menuItem1.MenuItems.Add("SubItem 2", eh);

    menu2SubItem1 = menuItem2.MenuItems.Add("SubItem 1", eh);

    menu3SubItem1 = menuItem3.MenuItems.Add("SubItem 1", eh);
    menu3SubItem2 = menuItem3.MenuItems.Add("SubItem 2", eh);
    menu3SubItem3 = menuItem3.MenuItems.Add("SubItem 3", eh);

    this.Menu = mainMenu;
}
public void MenuItemClicked(Object sender, EventArgs e)
{
    MessageBox.Show(((MenuItem)sender).Text);
}
}

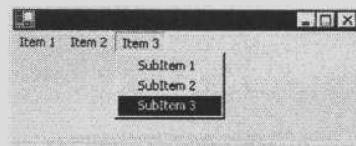
public
class main
{
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

W tej chwili nasza aplikacja ma już całą strukturę menu widoczną 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. Sama procedura obsługi została oczywiście zmieniona. Korzystamy w niej z metody Show klasy MessageBox, która pozwala na wyświetlenie okna dialogowego z dowolnie podanym tekstem.

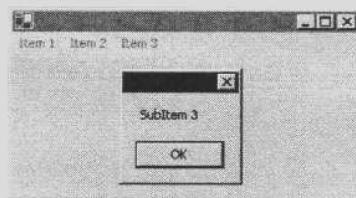
Chwili uwagi wymaga zastosowany sposób uzyskania nazwy wybranego podmenu. Korzystamy tutaj z faktu, że do procedury obsługi zdarzenia, czyli funkcji MenuItemClicked, jest przekazywana referencja do obiektu, który ją wywołuje. Jest to parametr sender. Rzecz jasna, skoro parametr ten jest typu Object, musimy dokonać rzutowania na typ MenuItem, co też czynimy. Po dokonaniu rzutowania możemy już bezpośrednio odwołać się do właściwości *Text*.

Rysunek 10.11.

Struktura
menu stworzona
w ćwiczeniu 10.17

**Rysunek 10.12.**

Wybranie
dowolnej pozycji
podmenu powoduje
wyświetlenie jej nazwy



Menu kontekstowe

Tworzenie menu kontekstowego jest bardzo podobne do tworzenia menu głównego. Strukturę definiujemy dokładnie w taki sam sposób. Różnica jest taka, że zamiast przypisania:

Form.Menu = nasze_menu;

dokonujemy przypisania:

Form.ContextMenu = nasze_menu;

oraz że zamiast klasy MainMenu stosujemy klasę ContextMenu.

Ćwiczenie 10.18.

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;
    MenuItem menuItem5;

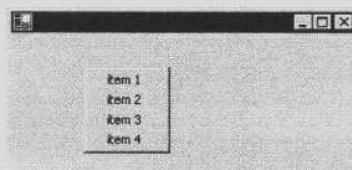
    public MainForm()
    {
        menuItem1 = contextMenu.MenuItems.Add("item 1");
        menuItem2 = contextMenu.MenuItems.Add("item 2");
        menuItem3 = contextMenu.MenuItems.Add("item 3");
        menuItem4 = contextMenu.MenuItems.Add("item 4");

        this.ContextMenu = contextMenu;
    }
}
```

```
public  
class main  
{  
    public static void Main()  
    {  
        Application.Run(new MainForm());  
    }  
}
```

Rysunek 10.13.

Aplikacja posiadająca menu kontekstowe



Ćwiczenie 10.19.

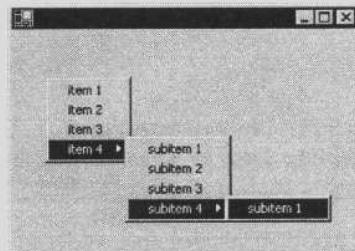
Napisz aplikację posiadającą wielopoziomowe menu kontekstowe (rysunek 10.14).

```
using System;  
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()  
    {  
        menuItem1 = contextMenu.MenuItems.Add("item 1");  
        menuItem2 = contextMenu.MenuItems.Add("item 2");  
        menuItem3 = contextMenu.MenuItems.Add("item 3");  
        menuItem4 = contextMenu.MenuItems.Add("item 4");  
  
        menu4SubItem1 = menuItem4.MenuItems.Add("subitem 1");  
        menu4SubItem2 = menuItem4.MenuItems.Add("subitem 2");  
        menu4SubItem3 = menuItem4.MenuItems.Add("subitem 3");  
        menu4SubItem4 = menuItem4.MenuItems.Add("subitem 4");  
  
        menu4SubItem4SubItem1 = menu4SubItem4.MenuItems.Add("subitem 1");  
  
        this.ContextMenu = contextMenu;  
    }  
}
```

```
public  
class main  
{  
    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* klasy MenuItem na true (pozycja zaznaczona) lub false (pozycja niezaznaczona).

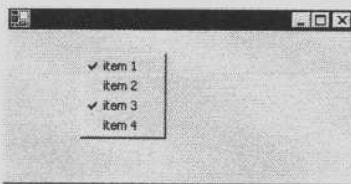
Ćwiczenie 10.20.

Napisz aplikację posiadającą menu kontekstowe, w którym co druga pozycja jest zaznaczona (rysunek 10.15).

```
using System;  
using System.Windows.Forms;  
  
public  
class MainForm:Form  
{  
    ContextMenu contextMenu = new ContextMenu();  
    MenuItem menuItem1;  
    MenuItem menuItem2;  
    MenuItem menuItem3;  
    MenuItem menuItem4;  
  
    public MainForm()  
    {  
        menuItem1 = contextMenu.MenuItems.Add("item 1");  
        menuItem1.Checked = true;  
        menuItem2 = contextMenu.MenuItems.Add("item 2");  
        menuItem3 = contextMenu.MenuItems.Add("item 3");  
        menuItem3.Checked = true;  
        menuItem4 = contextMenu.MenuItems.Add("item 4");  
  
        this.ContextMenu = contextMenu;  
    }  
}
```

```
public  
class main  
{  
    public static void Main()  
    {  
        Application.Run(new MainForm());  
    }  
}
```

Rysunek 10.15.
Menu kontekstowe
z zaznaczonymi
niektórymi pozycjami



Korzystając z obsługi zdarzeń oraz właściwości *Checked*, możemy napisać menu, w którym stan pozycji będzie się zmieniał dynamicznie. To znaczy, że kiedy użytkownik pierwszy raz wybierze daną pozycję, zostanie ona zaznaczona, kolejne wybranie tejże pozycji spowoduje jej odznaczenie itd. Taki sposób obsługi często spotykamy w aplikacjach, warto więc przećwiczyć jego wykonanie.

Ćwiczenie 10.21.

Utwórz menu główne, 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()  
    {  
        EventHandler eh = new EventHandler(this.MenuItemClicked);  
  
        menuItem1 = mainMenu.MenuItems.Add("item 1");  
        menuSubItem1 = menuItem1.MenuItems.Add("item 1", eh);  
        menuSubItem2 = menuItem1.MenuItems.Add("item 2", eh);  
        menuSubItem3 = menuItem1.MenuItems.Add("item 3", eh);  
        menuSubItem4 = menuItem1.MenuItems.Add("item 4", eh);  
  
        this.Menu = mainMenu;  
    }  
    public void MenuItemClicked(Object sender, EventArgs e)
```

```
{  
    ((MenuItem)sender).Checked = !((MenuItem)sender).Checked;  
}  
}  
public  
class main  
{  
    public static void Main()  
    {  
        Application.Run(new MainForm());  
    }  
}
```

W tej chwili wybranie dowolnej pozycji z naszego menu spowoduje jej zaznaczenie. W metodzie obsługującej zdarzenie po prostu zmieniamy wartość właściwości Checked na przeciwną, czyli jeśli była równa false, będzie równa true, jeśli natomiast była równa true, będzie równa false. Dokonuje tego instrukcja:

```
((MenuItem)sender).Checked = !((MenuItem)sender).Checked;
```

Istnieje jednak możliwość zmiany sposobu zaznaczania pozycji w menu. Jeśli bowiem właściwości RadioCheck obiektu klasy MenuItem przypiszemy wartość true, to po nadaniu właściwości Checked również wartości true menu będą zaznaczane poprzez symbol kropki.

Ćwiczenie 10.22.

Zmodyfikuj kod z ćwiczenia 10.21 w taki sposób, aby pozycje menu były zaznaczane przez symbol kropki, tak jak widoczne jest to na rysunku 10.16.

```
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()  
    {  
        EventHandler eh = new EventHandler(this.MenuItemClicked);  
  
        menuItem1 = mainMenu.MenuItems.Add("item 1");  
        menuSubItem1 = menuItem1.MenuItems.Add("item 1". eh);  
        menuSubItem2 = menuItem1.MenuItems.Add("item 2". eh);  
        menuSubItem3 = menuItem1.MenuItems.Add("item 3". eh);  
        menuSubItem4 = menuItem1.MenuItems.Add("item 4". eh);  
  
        menuSubItem1.RadioCheck = true;  
        menuSubItem2.RadioCheck = true;  
        menuSubItem3.RadioCheck = true;  
        menuSubItem4.RadioCheck = true;  
  
        this.Menu = mainMenu;
```

```

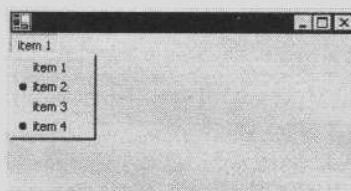
        }
        public void MenuItemClicked(Object sender, EventArgs e)
        {
            ((MenuItem)sender).Checked = !((MenuItem)sender).Checked;
        }
    }

    public
    class main
    {
        public static void Main()
        {
            Application.Run(new MainForm());
        }
    }
}

```

Rysunek 10.16.

Elementy menu zaznaczane przez symbol kropki



Skróty klawiaturowe

Dostęp do menu może być zrealizowany poprzez skróty klawiaturowe z klawiszem *Alt*. Wiele aplikacji udostępnia ten praktyczny sposób, który na szczęście jest bardzo prosty w realizacji. Wystarczy, że podczas tworzenia pozycji menu literę, która będzie używana do aktywacji tej pozycji, poprzedzimy znakiem &. To całe nasze zadanie.

Ćwiczenie 10.23.

Napisz aplikację zawierającą menu File, w którym będzie znajdować się pozycja Close, której wybranie 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()
    {
        menuItem1 = mainMenu.MenuItems.Add("&File");
        EventHandler eh = new EventHandler(this.CloseClicked);
        menuItem2 = menuItem1.MenuItems.Add("&Close", eh);
        this.Menu = mainMenu;
    }
}

```

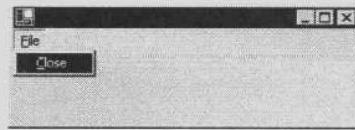
```
public void CloseClicked(Object sender, EventArgs e)
{
    this.Close();
}

public
class main
{
    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 tu podkreślone. Pamiętajmy jednak, że przy tym sposobie realizacji (tzn. aktywowania menu klawiszem *Alt*) nie możemy dostać się bezpośrednio do danej pozycji. W powyższym przypadku, aby wybrać pozycję *Close*, należałoby najpierw wcisnąć klawisz *Alt* i, trzymając go, nacisnąć kolejno *F* i *C*.

Rysunek 10.17.

Menu dostępne
poprzez skróty
klawișowe
z klawiszem Alt



Możemy przecież 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, abyśmy nie wyposażali naszych aplikacji w C# w takie właśnie udogodnienia.

Powiązanie menu ze skrótem klawiaturowym będzie jednak w tym przypadku wyglądało zupełnie inaczej niż w poprzednim ćwiczeniu. Do dyspozycji mamy dwa sposoby: albo przypisujemy skrót do menu już podczas tworzenia konkretnej pozycji, albo najpierw tworzymy daną pozycję, a następnie modyfikujemy jej właściwość *ShortCut*. Przećwiczmy oba sposoby w kolejnych dwóch ćwiczeniach.

Na początek przypiszmy odpowiedni skrót już podczas tworzenia pozycji. Wymaga to użycia trzyargumentowego konstruktora w postaci:

```
public MenuItem(
    string text,
    EventHandler eh,
    Shortcut shortcut
);
```

Jak widać, w tym przypadku niezbędne będzie jednoczesne przypisanie obsługi zdarzenia, musimy zatem mieć przygotowane odpowiednie metody.

Ćwiczenie 10.24.

Napisz aplikację zawierającą menu *File*, w którym będzie znajdować się pozycja *Exit*, której wybranie spowoduje zamknięcie programu. Do pozycji *Exit* przypisz skrót klawiaturowy *Ctrl+X*.

```
using System;
using System.Windows.Forms;

public
class MainForm:Form
{
    MainMenu mainMenu = new MainMenu();
    MenuItem menuItem1;
    MenuItem menuItem2;
    public MainForm()
    {
        menuItem1 = mainMenu.MenuItems.Add("&File");
        EventHandler eh = new EventHandler(this.CloseClicked);
        menuItem2 = new MenuItem("E&xit", eh, Shortcut.CtrlX);
        menuItem1.MenuItems.Add(menuItem2);
        this.Menu = mainMenu;
    }
    public void CloseClicked(Object sender, EventArgs e)
    {
        this.Close();
    }
}

public
class main
{
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```

Jeśli spojrzemy na rysunek 10.18, zobaczymy, że do pozycji Exit w menu File dopisany został skrót *Ctrl+X*. Wciśnięcie na klawiaturze tej kombinacji znaków spowoduje automatyczne wywołanie pozycji *Exit*, a tym samym wykonanie metody *CloseClicked*. Ponieważ jedynym zadaniem tej metody jest zamknięcie aplikacji, zakończy ona swoje działanie.

Rysunek 10.18.
Menu z przypisanym
skrótem klawiaturowym



Drugą z metod dodawania skrótów klawiaturowych jest modyfikacja właściwości *ShortCut* w obiekcie klasy *MenuItem*. Czyli najpierw tworzymy jednym ze znanych już nam sposobów pozycję menu i dopiero do niej przypisujemy skrót.

Ćwiczenie 10.25.

Napisz aplikację zawierającą menu *File*, w którym będzie znajdować się pozycja *Exit*. Do pozycji *Exit* przypisz skrót klawiaturowy *Ctrl+X*, 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()
    {
        menuItem1 = mainMenu.MenuItems.Add("&File");
        EventHandler eh = new EventHandler(this.CloseClicked);
        menuItem2 = menuItem1.MenuItems.Add("&Close", eh);
        menuItem2.Shortcut = Shortcut.CtrlX;
        this.Menu = mainMenu;
    }
    public void CloseClicked(Object sender, EventArgs e)
    {
        this.Close();
    }
}

public
class main
{
    public static void Main()
    {
        Application.Run(new MainForm());
    }
}
```