C#

Wydanie II

Krzysztof Rychlicki-Kicior

TABLICE INFORMATYCZNE

Helion

SKŁADNIA JĘZYKA

Podstawowe typy danych (wybrane)

Poniższe typy są typami C#; w większości ich nazwy pokrywają się z nazwami typów .NET (z tą różnicą, iż typy w .NET zaczynają się od wielkiej litery, np. `Byte`).

`byte`	(1 bajt) — typ całkowitoliczbowy, zakres: od *0* do *255*.
`short`	(2 bajty) — typ całkowitoliczbowy, zakres: od *–32 768* do *32 767*.
`int`	(4 bajty) — typ całkowitoliczbowy, zakres: od *–2 147 483 648* do *2 147 483 647*.
`long`	(8 bajtów) — typ całkowitoliczbowy, zakres: od *–9 223 372 036 854 775 808* do *9 223 372 036 854 775 807*.
`float`	(4 bajty) — nazwa w .NET: `Single`. Typ zmiennoprzecinkowy pojedynczej precyzji (do 7 cyfr znaczących). Zakres: od *±1.5e–45* do *±3.4e38*. Wymagane jest dołączenie za wartością tego typu litery *F*, np. *3.14F*.
`double`	(8 bajtów) — typ zmiennoprzecinkowy podwójnej precyzji (do 15 – 16 cyfr znaczących). Zakres: od *±5.0e–324* do *±1.7e308*. Jest to domyślny typ dla liczb zmiennoprzecinkowych, nie wymaga sufiksu.
`decimal`	(16 bajtów) — typ zmiennoprzecinkowy. Używany do operacji wymagających dużych liczb i dużego bezpieczeństwa przy modyfikacji zmiennych. Zakres: od *±1.0e–28* do *±7.9e28*. Wymaga dołączenia litery *m*, np. *9.5m*.
`char`	(2 bajty) — typ znakowy. Służy do przechowywania pojedynczych znaków Unicode. Znaki są zapisywane w apostrofach, np. `'a'`.
`bool`	(1 bajt) — typ logiczny. Zmienne tego typu przechowują wartość logiczną — prawda (`true`) lub fałsz (`false`).

Ponadto dostępne są typy `uint`, `ushort` i `ulong` (liczby bez znaków), które mają dwukrotnie większy zakres liczb dodatnich od swoich tradycyjnych odpowiedników (np. `ushort` — od *0* do *65 535*).

Komentarze

Komentarze pozwalają na dodanie uwag do kodu. Nie są one przetwarzane przez kompilator. Wyróżniamy dwa rodzaje komentarzy:

```
// komentarz w jednej linijce
/*
Komentarz
w dwóch i większej liczbie
linijek
*/
```

Instrukcje

Instrukcją jest każda czynność wykonywana w ramach programu. Zazwyczaj kończy się średnikiem:

```
instrukcja;
```

Pojedynczą instrukcję można zastąpić blokiem instrukcji, czyli ciągiem pojedynczych instrukcji ujętych w nawiasy klamrowe:

```
{
  instrukcja1;
  instrukcja2;
  instrukcja3;
}
```

Zmienne

Zmienne deklaruje się, umieszczając nazwę typu oraz nazwę zmiennej. Każdą deklarację (podobnie jak inne instrukcje języka C#) kończy się średnikiem.

Przykład:

```
int liczba;
```

Nazwa zmiennej może składać się z cyfr, liter i znaku podkreślenia, przy czym nie może zaczynać się od cyfry.

Zmiennej przypisuje się wartość za pomocą znaku `=` (operator przypisania), np.:

```
liczba = 5;
```

Zmienne można zadeklarować i przypisać im wartość w jednej instrukcji:

```
int liczba = 5;
```

Za pomocą jednej instrukcji można przypisać dwóm zmiennym jedną wartość, np.:

```
int liczba, inna_liczba;
liczba = inna_liczba = 3;
```

W obrębie metod (więcej na ich temat w dalszych sekcjach) istnieje możliwość deklarowania zmiennych typowanych niejawnie:

```
var liczba = 3;
```

Trzeba pamiętać, że w przeciwieństwie do języków takich jak JavaScript powyższy zapis deklaruje zmienną silnie typowaną, przy czym ów typ jest określany przez kompilator na podstawie przypisanej wartości. Co za tym idzie, poniższy zapis spowodowałby błąd kompilacji:

```
var liczba = 3;
liczba = "test";
```

Stałe

Deklarowanie stałych odbywa się w ten sam sposób, co deklarowanie i inicjalizowanie zmiennych, z wyjątkiem dodania słowa kluczowego `const` na początku instrukcji:

```
const int liczba = 3;
```

Wartości stałej nie można zmieniać w trakcie działania programu.

Operatory

Za pomocą operatora można wykonywać operacje na wyrażeniach (zmiennych bądź wartościach stałych). Operatory dzielimy na:

- arytmetyczne,
- logiczne,
- bitowe,
- relacji,
- przypisania,
- inne.

Operatory arytmetyczne znamy z życia codziennego. Są to:

- `+` (dodawanie);
- `-` (odejmowanie);
- `*` (mnożenie);
- `/` (dzielenie);
- `%` (modulo, reszta z dzielenia);
- `++` (zwiększa wartość zmiennej o jeden);
- `--` (zmniejsza wartość zmiennej o jeden).

Dzielenie dwóch liczb całkowitych (np. typu `int`) zwraca wynik całkowity (np. 5/3 = 1). Jeśli chcemy uzyskać wynik realny, trzeba np. dodać do licznika lub mianownika wartość typu zmiennoprzecinkowego (najszybciej 0.0), np.: wyrażenie `(5+0.0)/3` zwróci wynik *1,66666666666667* (w przybliżeniu).

Operatory logiczne zwracają wartość typu `boolean`, uzależniając ją od podanych operandów.

`(w1 && w2)` — zwróci `true`, jeśli `w1` i `w2` mają wartość `true` (suma logiczna).

`(w1 || w2)` — zwróci `true`, jeśli `w1` lub `w2` ma wartość `true` (alternatywa logiczna).

`(!w1)` — zwróci `true`, jeśli `w1` ma wartość `false` (negacja logiczna).

`(w1 ^ w2)` — zwróci `true`, jeśli tylko jedna z wartości ma wartość `true` (rozłączna alternatywa logiczna).

Operatory bitowe wykonują operacje na liczbach w postaci bitowej (system dwójkowy). Można je też wykorzystywać zamiast analogicznych operatorów logicznych.

`(w1 & w2)` — suma bitowa.

`(w1 | w2)` — alternatywa bitowa.

`(w1 ^ w2)` — rozłączna alternatywa bitowa.

`(~w1)` — negacja bitowa (nie może być zastosowana do wartości logicznych).

`(w1 >> w2)` — przesunięcie bitowe w prawo — przesuwa lewy operand w prawo o liczbę bitów podaną w operandzie `w2` (usuwa ostatnie bity liczby).

`(w1 << w2)` — przesunięcie bitowe w lewo — przesuwa lewy operand o liczbę bitów podaną w operandzie `w2` (uzupełnia zerami postać bitową).

Operatory relacji umożliwiają porównanie dwóch wartości. Wynik takiego porównania jest typu `bool`.

`(w1 == w2)` — zwraca `true`, jeśli obydwie wartości są równe.

`(w1 != w2)` — zwraca `true`, jeśli obydwie wartości są różne.

`(w1 > w2)` — zwraca `true`, jeśli `w1` jest większe od `w2`.

`(w1 >= w2)` — zwraca `true`, jeśli `w1` jest większe lub równe `w2`.

Dwa ostatnie operatory można stosować w odwrotny sposób, tj. `<` i `<=`.

Operatory przypisania umożliwiają zastąpienie przypisania zmieniającego wartość zmiennej, np.:

```
x = x + 5;
```

na postać:

```
x += 5;
```

W analogiczny sposób można używać operatorów: `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`.

Pozostałe wybrane operatory:

`.` — operator dostępu do elementów przestrzeni nazw lub klasy, np. `obiekt.metoda();`

`[]` — operator dostępu do elementów indeksowanych, np. tablic lub atrybutów, np. `tablica[0]` ;

`?:` — operator trójargumentowy — np. `int x = (i > 1) ? 3 : 0`.

Zmienna `x` będzie miała wartość *3*, jeśli `i` będzie większe od *1*, w przeciwnym razie `x` będzie równe *0*.

Operatory posiadają swoje priorytety; niektóre z nich są zgodne z zasadami matematyki (dzielenie jest wykonywane przed dodawaniem itd.). Poznane operatory można uszeregować w następujący sposób według priorytetu operatora:

`++, --`
`!, ~`
`*, /, %`
`+, -`
`<<, >>`
`<, <=, >, >=`
`==, !=`
`&`
`^`
`|`
`&&`
`||`
`?:`
`=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=`

Im wyżej operator znajduje się na powyższej liście, tym większy jest jego priorytet. Aby zmienić priorytet, można zastosować operatory, np. wyrażenie `4+2/2` jest równe *5*, a wyrażenie `(4+2)/2` jest równe *3*.

Instrukcje złożone

Język C# udostępnia kilka rodzajów instrukcji złożonych. Instrukcja warunkowa pozwala uzależnić wykonanie pewnego ciągu instrukcji od danego warunku, np.:

```
if (warunek)
{
} else
{
}
```

gdzie *warunek* jest tylko i wyłącznie typu `bool`! Przeciwnie niż np. w C++ nie jest możliwe wykonanie następującej operacji:

```
int liczba = 3;
if (liczba = 5) // BŁĄD
```

W takiej sytuacji w nawiasie zostaje dokonane przypisanie i wyrażenie przyjmuje wartość *5*. W języku C# nie można jednak domyślnie skonwertować tej wartości na typ `bool`, stąd takie wyrażenie spowoduje błąd kompilatora.

Instrukcja `switch` (wyboru) zastępuje szereg instrukcji `if`, jeśli wszystkie odnoszą się do tej samej zmiennej.

Przykład:

```
switch (w)
{
  case 0:
  // instrukcje
  break;
  case 1:
  // kolejne instrukcje
  break;
  default:
  // inne instrukcje
  break;
}
```

Zmienna `w` może być typu całkowitoliczbowego, znakowego lub być łańcuchem znaków. Instrukcje wewnątrz danej klauzuli `case` zostaną wykonane, jeśli zmienna `w` będzie miała wartość określoną w klauzuli. Każda instrukcja `case` musi kończyć się instrukcją `break;`. Przeciwnie niż w języku C++ nie jest możliwe „spadanie" między poszczególnymi instrukcjami `case`. Jeśli żaden z określonych warunków nie zostanie spełniony, wykonywane są instrukcje z sekcji `default`.

Kolejną często wykorzystywaną konstrukcją są pętle. Pętla to blok instrukcji kodu, który jest wykonywany cyklicznie. C# oferuje trzy rodzaje pętli: `for`, `while`, `do..while`.

Pętla `for` jest używana, kiedy znamy liczbę przebiegów pętli.

Przykład:

```
for (int licznik=0; licznik < 10;
licznik++) // instrukcje
```

Powyższa pętla wykona się 10 razy. Dodatkowo wewnątrz bloku instrukcji można wykorzystywać wartość zmiennej `licznik`, np. do wyświetlania jego wartości na ekranie. W pierwszej części pętli inicjujemy zmienną iterującą, drugą stanowi warunek; musi on zwracać wartość `true`, aby pętla kontynuowała działanie. W ostatniej części pętli określamy, jak ma się zwiększać (lub zmniejszać) zmienna iteracyjna (w powyższym przykładzie — o jeden).

Istnieje możliwość pominięcia dowolnej części pętli; jeśli zachodzi konieczność, aby pętla wykonywała się nieskończoną ilość razy, można użyć zapisu:

```
for (;;)
  // instrukcje
```

Pozostałe dwa rodzaje pętli są używane w sytuacjach, gdy nie jest z góry określone, ile razy pętle mają być wykonywane.

Pętla `while` ma następującą składnię:

```
while (warunek)
  // instrukcje
```

Instrukcje w pętli są wykonywane dopóty, dopóki *warunek* ma wartość `true`. Należy zwrócić uwagę, że pętla może nie zostać wykonana ani razu!

Kolejnym wariantem tej pętli jest pętla `do..while`. Jedyną różnicę między nimi stanowi umiejscowienie warunku — w tym wypadku znajduje się on po bloku instrukcji:

```
do
  // instrukcje
while (warunek);
```

Tablice

Do przechowywania wielu elementów tego samego typu używa się zmiennych tablicowych. Tablicę tworzy się z użyciem operatora `new`, np.:

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

Powyższa instrukcja utworzy 5-elementową tablicę liczb całkowitych. Aby korzystać z tablicy, należy używać indeksu tablicy ujętego w nawiasy kwadratowe:

```
tablica[0] = 3;
int liczba = tablica[0];
```

Tablice w języku C# są indeksowane od *0*, tj. w tablicy *n*-elementowej możemy odwoływać się do elementów o indeksach od *0* do *n–1*. Podobnie jak zwykłe zmienne tablice również mogą mieć wartości przypisywane przy ich utworzeniu, np.:

```
int[] tablica = new int[] { 3, 5, 7 };
```

W takiej sytuacji można pominąć rozmiar tablicy; w nawiasach klamrowych wystarczy podać kolejne wartości, jakie mają znaleźć się w tablicy (oddzielone przecinkami).

C# podobnie jak inne języki programowania daje możliwość tworzenia tablic wielowymiarowych. W tym celu w momencie deklarowania tablicy należy w nawiasach klamrowych wpisać *n–1* przecinków, gdzie *n* jest liczbą określającą, ile wymiarów ma mieć tablica:

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

W ten sposób powstanie tablica trójwymiarowa. Podczas tworzenia takiej tablicy musimy podać długości tablicy w poszczególnych wymiarach w sposób następujący:

```
tablica = new int[2,3,4];
```

Indeks tablicy jest podawany w analogiczny sposób, jak jej wymiary w konstruktorze, np.:

```
tablica[2,2,1] = 5;
```

Istnieje możliwość przypisania tablicy wielowymiarowej przy utworzeniu, np.:

```
int[,] tablica = new int[,] { { 1, 3
}, { 4, 6 }, { 7, 9 }, { 10, 12 } };
```

KLASY

Klasa jest podstawową formą reprezentowania danych i dostarczania funkcjonalności do programów. Klasy reprezentują różnego rodzaju obiekty, np. komponenty graficzne takie jak przycisk czy pole tekstowe lub złożone mechanizmy obsługi baz danych. Również podstawowe typy danych, opisane powyżej, są klasami.

Klasa jest też rodzajem definicji. Konkretny egzemplarz (nazywany też instancją) klasy, o określonej pozycji i opisie tekstowym, np. przycisk znajdujący się w programie, nazywany jest obiektem.

Programista może deklarować własne klasy. W tym celu należy użyć słowa kluczowego `class`:

```
class Czlowiek
{
}
```

Po słowie kluczowym `class` umieszczamy nazwę klasy, a następnie blok instrukcji zawierających deklarację klasy.

Pola i metody

Najważniejszymi elementami klasy są zmienne klasy (nazywane też polami) i metody klasy. Metoda to blok instrukcji mogący przyjąć określone parametry, zwracający określoną wartość. W językach zorientowanych proceduralnie (jak np. C) metody noszą nazwę funkcji. Deklaracja metody składa się z nagłówka oraz ciała:

```
int Dodaj(int liczba1, int liczba2)
{
  int wynik = liczba1 + liczba2;
  return wynik;
}
```

Nagłówek metody zawiera typ zwracanej wartości oraz nazwę metody. Następnie znajduje się lista przyjmowanych argumentów. Jeśli metoda nie pobiera argumentów, należy zapisać tylko puste nawiasy: `()`. Każdy argument składa się z nazwy typu oraz nazwy argumentu. Po nagłówku występuje blok instrukcji, w którym musi wystąpić instrukcja `return`. Powoduje ona zakończenie wykonywania metody (wszelkie instrukcje znajdujące się za nią nie zostaną wykonane) i zwrócenie podanej wartości. Metoda może nie zwracać żadnej wartości; zamiast typu danych jest używane słowo kluczowe `void`. W tym przypadku użycie słowa kluczowego `return` nie jest wymagane; można go jednak użyć (bez żadnego argumentu), aby przerwać wykonywanie metody:

```
void Metoda(int argument)
{
  int liczba = 2 * argument;
  return;
  int liczba2 = 2 * liczba; // ostrzeżenie –
  instrukcja nie zostanie wykonana
}
```

Parametry metod

Zwykłe parametry metod są wymieniane w nawiasach okrągłych po przecinku w sygnaturze (nagłówku) metody. Parametry metod mogą podlegać działaniu różnych modyfikatorów. Dwa pierwsze modyfikatory działają w bardzo podobny sposób. Modyfikatory `ref` i `out` pozwalają na przekazywanie parametrów metod przez referencję. Dzięki temu jeśli wewnątrz metody wartość parametru ulegnie zmianie, argument przekazany do metody również zmieni wartość (normalnie do metody przekazywana jest kopia argumentu):

```
public void Metoda(out int waga)
{
  waga = 3;
}
public void InnaMetoda(ref int waga)
{
  waga = 5;
}
public void Test()
{
  int argument, argument2 = 3;
  Metoda(out argument);
  InnaMetoda(ref argument2);
}
```

Modyfikatory różnią się tym, że zmienna z modyfikatorem `out` nie musi być wcześniej inicjalizowana, a zmienna z modyfikatorem `ref` — musi. Co więcej, w przypadku parametru z modyfikatorem `out` istnieje założenie, że po zakończeniu działania metody przekazany argument będzie miał określoną wartość. W związku z tym wewnątrz metody musi istnieć instrukcja, która przypisze jakąś wartość do parametru z modyfikatorem `out`. Dodatkowo parametry przekazywane przez referencję muszą zawierać modyfikatory `ref` i `out` w momencie wywołania metod.

Tablice parametrów

Tablice parametrów pozwalają na przekazywanie do metody dowolnej liczby parametrów (po przecinku). Wewnątrz metody są one widoczne w postaci tablicy parametrów:

```
public void MetodaParametry(params
object[] obiekty)
{
  for (int i = 0; i < obiekty.Length;
  i++)
    Console.WriteLine(obiekty[i]);
}
```

Z punktu widzenia metody mamy do czynienia z najzwyklejszą tablicą. Dopuszczalne wywołania metody nie mają jednak z tablicą nic wspólnego:

```
MetodaParametry("3", "4", 5);
MetodaParametry("3", "4");
MetodaParametry("3");
```

Argumenty nazwane i opcjonalne

W wywołaniach metod można jawnie podawać nazwy parametrów, dla których chcemy określić wartość, dzięki czemu nie trzeba pamiętać kolejności argumentów:

```
public static double Potega(double x,
double y)
{
  return Math.Pow(x, y);
}
...
Console.WriteLine(Metody.Potega(y: 2,
x: 3));
Console.WriteLine(Metody.Potega(x: 2,
y: 3));
```

Warto zauważyć, że użycie argumentów nazwanych nie wymaga żadnych ingerencji w deklaracjach metod.

Ciąg dalszy na str. 2