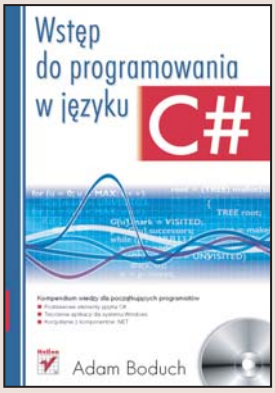
Wstęp do programowania w języku C#



Autor: Adam Boduch
ISBN: 83-246-0523-1
Format: B5, stron: 392
Przykłady na ftp: 1421 kB

Zostań profesjonalnym programistą C#

C# jest uważany przez wielu programistów za najlepszy język stosowany do tworzenia aplikacji dla platformy .NET. Język ten, opracowany w firmie Microsoft, łączy w sobie najbardziej wartościowe cechy C++ i Javy, ścisłą integrację z komponentami .NET, wysoki poziom bezpieczeństwa i ogromne możliwości. świetnie nadaje się do tworzenia aplikacji sieciowych i bazodanowych przeznaczonych zarówno dla systemu Windows, jak i dla urządzeń przenośnych, takich jak PocketPC. Popularność C# stale rośnie, a nieodpłatne udostępnienie przez firmę Microsoft środowiska programistycznego Visual C# Express Edition sprawiło, że coraz więcej twórców oprogramowania wykorzystuje je w swojej pracy.

Książka „Wstęp do programowania w języku C#" to podręcznik przeznaczony dla tych, którzy chcą poznać ten język od podstaw. Czytając ją, poznasz możliwości C# i platformy .NET. Nauczysz się tworzyć własne aplikacje, korzystając ze środowiska Visual C# Express Edition. W książce znajdziesz omówienie składni i elementów języka C#, wyjaśnienie zasad programowania obiektowego i sposobów realizacji różnych zadań programistycznych. Napiszesz aplikacje dla Windows wykorzystujące biblioteki klas .NET i obsługujące pliki w formacie XML.

- Konstrukcja platformy .NET
- Składnia C#
- Elementy języka
- Programowanie obiektowe
- Obsługa zdarzeń
- Korzystanie z tablic i kolekcji
- Obsługa wyjątków
- Biblioteka Windows Forms
- Stosowanie komponentów .NET
- Operacje na plikach i strumieniach
- Obsługa formatu XML

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 7.

Tablice i kolekcje

Omówiliśmy już sporą część tego, co oferuje język C#. Powiedzieliśmy sobie o najważniejszym — programowaniu obiektowym, które może przysporzyć najwięcej kłopotów początkującemu programiście. Nie zaprezentowałem do tej pory bardzo ważnego elementu wielu języków programowania, a mianowicie obsługi tablic.

Jest to bardzo wygodna funkcja języka programowania; przekonasz się o tym podczas pisania przykładowej aplikacji podsumowującej dotychczasową wiedzę o języku C#. Będzie to znana i lubiana gra — kółko i krzyżyk. O tym jednak pod koniec tego rozdziału. Nie przedłużając, spieszę z wyjaśnieniem, czym są tablice…

Czym są tablice

Wyobraź sobie, że w swojej aplikacji musisz przechować wiele zmiennych tego samego typu. Dla przykładu, niech będą to dni tygodnia typu `string`. Proste? Owszem, wystarczy zadeklarować siedem zmiennych typu `string`:

```
string pon, wt, śr, czw, pt, so, nd;
```

Teraz do tych zmiennych należy przypisać wartość:

```
pon = "Poniedziałek";
wt = "Wtorek";
// itd
```

Teraz wyobraź sobie sytuację, w której musisz zadeklarować 12 zmiennych oznaczających nazwy miesięcy. Nieco uciążliwe? Owszem. Do tego celu najlepiej użyć tablic, które służą do grupowania wielu elementów tego samego typu. Osobiście z tablic korzystam bardzo często, jest to znakomity, czytelny sposób na przechowywanie dużej ilości danych.

Przejdźmy jednak do rzeczy. W C# istnieje możliwość deklaracji zmiennej, która przechowywać będzie wiele danych. Tak w skrócie i uproszczeniu możemy powiedzieć o tablicach.

Deklarowanie tablic

Tablice deklaruje się podobnie jak zwykłe zmienne. Jedyną różnicą jest zastosowanie nawiasów kwadratowych:

```
typ[] Nazwa;
```

W miejsce *typ* należy podać typ danych elementów tablicowych (np. `int`, `string`), a w miejsce *nazwa* — nazwę zmiennej tablicowej. Przykładowo:

```
int[] Foo;
```

W tym miejscu zadeklarowaliśmy tablicę `Foo`, która może przechowywać elementy typu `int`. Przed użyciem takiej tablicy należy zadeklarować, z ilu elementów ma się ona składać. W tym celu korzystamy ze znanego nam operatora `new`:

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

Taka konstrukcja oznacza zadeklarowanie w pamięci komputera miejsca dla pięciu elementów tablicy `Foo`. Przypisywanie danych do poszczególnych elementów odbywa się również przy pomocy symboli nawiasów kwadratowych:

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

Foo[0] = 100;
Foo[1] = 1000;
Foo[2] = 10000;
Foo[3] = 100000;
Foo[4] = 1000000;

Console.WriteLine(Foo[4]);
```

Możliwy jest również skrótowy zapis deklaracji tablic, podobny do tego znanego z tworzenia obiektów:

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

Indeks

Tablica składa się z *elementów*. Każdemu z nich przypisany jest tzw. *indeks*, dzięki któremu odwołujemy się do konkretnego elementu tablicy. Ów indeks ma postać liczby i wpisujemy go w nawiasach kwadratowych, tak jak to zaprezentowano w poprzednim przykładzie. Spójrz na kolejny przykład:

```
char[] Foo = new char[5];

Foo[0] = 'H';
Foo[1] = 'e';
Foo[2] = 'l';
Foo[3] = 'l';
Foo[4] = 'o';
```

Indeksy numerowane są od zera do $N-1$, gdzie N to ilość elementów tablicy. Aby lepiej to zrozumieć, spójrz na tabelę 7.1.

Tabela 7.1. *Prezentacja zależności indeksów elementów*

Indeks	0	1	2	3	4
Wartość	H	e	l	l	o

Należy uważać, aby nie odwołać się do elementu, który nie istnieje! Jeżeli zadeklarowaliśmy tablicę 5-elementową i odwołujemy się do szóstego elementu (poprzez indeks nr 5), kompilator C# nie zareaguje! Błąd zostanie wyświetlony dopiero po uruchomieniu programu.

Inicjalizacja danych

Po utworzeniu tablicy każdemu elementowi przypisywana jest domyślna wartość. Np. w przypadku typu `int` jest to cyfra 0. Programista po zadeklarowaniu takiej tablicy ma możliwość przypisania wartości dla konkretnego elementu.

Istnieje możliwość przypisania wartości dla konkretnego elementu już przy deklarowaniu tablicy. Należy wówczas wypisać wartości w klamrach:

```
char[] Foo = new char[5] {'H', 'e', 'l', 'l', 'o'};

Console.WriteLine(Foo[4]);
```

Język C# dopuszcza uproszczony zapis takiego kodu — wystarczy pominąć ilość elementów tablicy:

```
char[] Foo = {'H', 'e', 'l', 'l', 'o'};
```

Kompilator oblicza rozmiar takiej tablicy po ilości elementów uporządkowanych pomiędzy klamrami.

Tablice wielowymiarowe

C# umożliwia także deklarowanie tzw. tablic *wielowymiarowych*. Przykładowo, poniższy kod tworzy tablicę 7×2 (7 kolumn i 2 wiersze):

```
string[,] Foo = new string[7, 2];
```

Zasada deklarowania tablic wielowymiarowych jest prosta. W nawiasie kwadratowym wpisujemy znak przecinka (`,`), natomiast podczas inicjalizacji musimy podać wymiar tablicy (ilość elementów należy również rozdzielić znakiem średnika). Podczas przypisywania danych do elementów należy podać dokładny indeks:

```
Foo[0, 0] = "Pn";
Foo[1, 0] = "Wt";
Foo[2, 0] = "Śr";
Foo[3, 0] = "Czw";
Foo[4, 0] = "Pt";
Foo[5, 0] = "So";
Foo[6, 0] = "Nd";

Foo[0, 1] = "Mon";
Foo[1, 1] = "Tue";
Foo[2, 1] = "Wed";
Foo[3, 1] = "Thu";
Foo[4, 1] = "Fri";
Foo[5, 1] = "Sat";
Foo[6, 1] = "Sun";
```

Język C# nie ogranicza nas w ilości wymiarów. Możemy więc wprowadzić do naszej tablicy kolejny wymiar. Poniższy fragment prezentuje deklarację tablicy 2×4×2:

```
string[, ,] Foo = new string[2, 4, 2];
```

Inicjalizacja danych tablicy wielowymiarowej jest analogiczna do standardowej tablicy:

```
int[,] Foo = new int[3, 3] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

Zauważ jednak, że poszczególne elementy zawarte w klamrach są rozdzielone znakiem przecinka. Oczywiście istnieje możliwość skrótowego zapisu:

```
int[,] Foo = new int[,] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

lub:

```
int[,] Foo = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

Pętla foreach

Podczas omawiania zagadnienia pętli nie wspomniałem o jednej ważnej pętli służącej do operowania na tablicach. Ponieważ tematyka tablic w rozdziale 3. nie była omawiania, pragnę wspomnieć o tej pętli właśnie tutaj.

Pętla ta, znana programistom PHP, Perl czy też Delphi, dla .NET jako parametru wymaga tablicy. Spójrz na poniższy fragment kodu:

```
string[] Foo = new string[7];

Foo[0] = "Pn";
Foo[1] = "Wt";
Foo[2] = "Śr";
Foo[3] = "Czw";
Foo[4] = "Pt";
Foo[5] = "So";
Foo[6] = "Nd";
```

```
foreach (string Bar in Foo)
{
    Console.WriteLine(Bar);
}
```

Uruchomienie takiej aplikacji spowoduje wyświetlenie, jeden pod drugim, kolejnych elementów tablicy. Po każdej iteracji kolejny element tablicy przypisywany jest do zmiennej `Bar`. Tutaj ważna uwaga. Zmienna `Bar` nie może być zadeklarowana lub użyta we wcześniejszych fragmentach kodu. Np. poniższa konstrukcja jest błędna:

```
string Bar = "Test";
foreach (string Bar in Foo)
{
    Console.WriteLine(Bar);
}
```

Przy próbie kompilacji wyświetlony zostanie błąd: `A local variable named 'Bar' cannot be declared in this scope because it would give a different meaning to 'Bar', which is already used in a 'parent or current' scope to denote something else.`

Identyczny rezultat jak ten pokazany przed chwilą można osiągnąć, stosując pętlę `for`:

```
for (int i = 0; i < Foo.Length; i++)
{
    Console.WriteLine(Foo[i]);
}
```

Konstrukcja `Foo.Length` zwraca rozmiar tablicy.

Zasadniczo wygodniejszym i czytelniejszym sposobem jest użycie pętli `foreach`, która w końcu została stworzona po to, by operować na tablicach. Jednakże użycie pętli `for` ma jedną przewagę nad `foreach` — można w niej modyfikować wartości elementów. Spójrz na poniższy przykład:

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

Identycznego efektu nie uzyskamy, stosując pętlę `foreach`:

```
foreach (string Bar in Foo)
{
    Bar = "Foo";
}
```

W tym momencie kompilator zasygnalizuje błąd: `Cannot assign to 'Bar' because it is a 'foreach iteration variable'.`

Pętla foreach a tablice wielowymiarowe

Pętla `foreach` z powodzeniem działa na tablicach wielowymiarowych. W takim wypadku kolejność iteracji jest następująca: najpierw przekazana zostanie wartość elementu [0, 1], następnie [0, 2] itd. Krótki kod prezentujący takie działanie:

```
string[,] Foo = new string[7, 2];

Foo[0, 0] = "Pn";
Foo[1, 0] = "Wt";
Foo[2, 0] = "Śr";
Foo[3, 0] = "Czw";
Foo[4, 0] = "Pt";
Foo[5, 0] = "So";
Foo[6, 0] = "Nd";

Foo[0, 1] = "Mon";
Foo[1, 1] = "Tue";
Foo[2, 1] = "Wed";
Foo[3, 1] = "Thu";
Foo[4, 1] = "Fri";
Foo[5, 1] = "Sat";
Foo[6, 1] = "Sun";

foreach (string Bar in Foo)
{
    Console.WriteLine(Bar);
}
```

Kolejność wyświetlania danych na konsoli będzie następująca:

```
Pon
Mon
Wt
Tue
...
```

W przypadku tablic wielowymiarowych konstrukcja `Tablica.Length` zwraca liczbę wszystkich elementów w tablicy. W prezentowanym przykładzie będzie to 7×2, czyli 14.

Działanie pętli `for` na tablicach wielowymiarowych jest nieco inne. Przykładowo, poniższa pętla spowoduje wyświetlenie jedynie polskich dni tygodnia:

```
for (int i = 0; i < Foo.Length / 2; i++)
{
   Console.WriteLine(Foo[i, 0]);
}
```

Tablice tablic

Mechanizm tablic jest w języku C# bardzo rozbudowany. Umożliwia nawet tworzenie tablic, które zawierają kolejne tablice. Poniższy kod prezentuje deklarację takiej tablicy:

```
int[][] Foo = new int[2][];
```

Ten zapis oznacza, iż tablica `Foo` zawierać będzie kolejne dwie tablice o nieokreślonej jeszcze liczbie elementów. Te dwie kolejne tablice również muszą zostać utworzone:

```
Foo[0] = new int[50];
Foo[1] = new int[1000];
```

Przypisywanie danych do takich tablic wygląda podobnie jak w przypadku tablic wielowymiarowych:

```
// przypisanie wartości do elementu 26. tablicy nr 1
Foo[0][25] = 100;
// przypisanie wartości do elementu 1000. tablicy drugiej
Foo[1][999] = 1;
```

Sprawa inicjalizacji tablic prezentuje się podobnie, jak to zostało zaprezentowane w trakcie omawiania bardziej podstawowych elementów. Przypisanie wartości do elementów w tablicach tego typu charakteryzuje się dość specyficzną składnią:

```
int[][] Foo = new int[][]
{
    new int[] {1, 2},     // zwróć uwagę na brak średnika!
    new int[] {1, 2, 3}
};                        // zwróć uwagę na obecność średnika!

Console.WriteLine(Foo[0][1]);
```

Moim zdaniem przejrzystszy jest skrótowy zapis powyższego kodu, również akceptowany przez kompilator C#:

```
int[][] Foo =
{
    new int[] {1, 2},
    new int[] {1, 2, 3}
};
```

Tablice struktur

O strukturach i wyliczeniach powiedzieliśmy sobie w rozdziale 5. W niektórych przypadkach przydatna okazuje się możliwość deklarowania tablic struktur lub typów wyliczeniowych. Jest to sprawa dość prosta, jeżeli znasz już podstawy użycia tablic, bowiem zamiast typu dotychczas używanego (czyli `int`, `string`, `char` itp.) należy użyć wcześniej zadeklarowanej struktury:

```
public struct Bar
{
    public string Name;
    public byte Age;
}

class Program
{
    static void Main(string[] args)
    {
        Bar[] BarArr = new Bar[2];

        BarArr[0].Name = "Janusz Kowalski";
        BarArr[0].Age = 52;

        BarArr[1].Name = "Piotr Nowak";
        BarArr[1].Age = 18;
    }
}
```

Parametr args w metodzie Main()

Gdy w rozdziale trzecim omawiałem podstawowe elementy programu C#, wspomniałem oczywiście o metodzie `Main()`, lecz pominąłem znaczenie parametru `args`. Zrobiłem to celowo, aby nie wprowadzać zamętu, gdyż tematyka tablic czy nawet typów danych nie była wówczas poruszana. Parametr `args` typu tablicowego zawiera ewentualne parametry przekazane do naszej aplikacji z linii poleceń. Czyli uruchamiając program z poziomu linii komend, mogę napisać:

```
MojaAplikacja.exe Parametr1 Parametr2
```

Zarówno *Parametr1*, jak i *Parametr2* zostaną przekazane do aplikacji, każdy zostanie przypisany do odrębnego elementu tablicy. Napiszmy dla treningu prosty program. Jego zadanie będzie banalne: sortowanie argumentów przekazanych do programu.

Właściwie najtrudniejszą rzeczą w programie jest sama konwersja danych z łańcucha `string` na wartość całkowitą `int`. Samo sortowanie tablicy realizuje metoda `Sort()` klasy `Array`. Całość programu prezentuje listing 7.1.

Listing 7.1. *Pobieranie i sortowanie argumentów programu*

```
using System;

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

```
            if (args.Length == 0)
            {
                Console.WriteLine("Brak argumentów programu!");
                return;
            }
            int[] ArrInt = new int[args.Length];
            int count = 0;

            foreach (string element in args)
            {
                ArrInt[count] = Int32.Parse(element);
                ++count;
            }

            Array.Sort(ArrInt);

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

Argumenty przekazane do aplikacji konwertujemy, a następnie zapisujemy w nowo utworzonej tablicy `ArrInt`. Zwróć uwagę, że do konwersji danych ze `string` na `int` użyłem metody `Parse()` z klasy `Int32`. To również jest dopuszczalny sposób, równie dobrze mogłem także użyć klasy `Convert`.

Posortowaną tablicę prezentuję na konsoli, wyświetlając w pętli kolejne jej elementy.

Element `{0}` wykorzystany w łańcuchu w metodzie `WriteLine()` zostanie zastąpiony wartością zmiennej przekazanej w tej samej metodzie. Jest to czytelny i prosty sposób formatowania łańcuchów — np.:

```
string sName = "Adam";
string sLocation = "Wrocławiu";
Console.WriteLine("Mam na imię {0} i mieszkam we {1}", sName,  sLocation);
```

Klasa System.Array

Chyba oswoiłeś się już z myślą, że całe środowisko .NET oparte jest na klasach, strukturach i wyliczeniach? Nie inaczej jest w przypadku tablic. Każda tablica w języku C# dziedziczy po klasie `System.Array`, która dostarcza podstawowych mechanizmów do manipulacji na elementach tablicy. To dzięki metodom tej klasy możemy pobrać ilość elementów w tablicy, posortować ją czy przeszukać. Kilka najbliższych stron zostanie przeznaczonych na opisanie podstawowych elementów tej klasy.

Jeżeli chodzi o właściwości klasy, to najważniejszą jest `Length`, która zwraca aktualną liczbę elementów tablicy. Ta sama klasa udostępnia również właściwość `LongLength`, która zwraca 64-bitową wartość określającą rozmiar wszystkich elementów w przypadku tablic wielowymiarowych.

Warto również wspomnieć o właściwości `Rank`, która zwraca liczbę wymiarów danej tablicy:

```
int[,] Foo = new int[3, 2] { { 1, 2 }, { 1, 2 }, { 1, 2 } };

Console.WriteLine(Foo.Rank);      // tablica dwuwymiarowa (wyświetli 2)
```

Metody klasy

W trakcie omawiania klasy `System.Array` należy wspomnieć o paru metodach, które mogą Ci się przydać przy okazji operowania na tablicach.

BinarySearch()

Używając algorytmu przeszukiwania binarnego, przeglądam elementy tablicy, aby znaleźć żądaną wartość. Pierwszym parametrem tej metody musi być nazwa tablicy, na której będzie ona operować. Drugim parametrem — szukany element. Oto przykład użycia tej metody:

```
string[] Foo = new string[] { "Pn", "Wt", "Śr", "Czw", "Pt" };

Console.WriteLine(Array.BinarySearch(Foo, "Śr"));
```

Metoda zwraca numer indeksu, pod jakim znajduje się szukany element, lub –1, jeżeli nic nie zostało znalezione. W zaprezentowanym przykładzie metoda zwróci wartość 2.

Clear()

Metoda umożliwia wyczyszczenie tablicy. W rzeczywistości ustawia każdemu elementowi wartość 0 lub `null` w zależności od jego typu. Metoda przyjmuje trzy parametry. Pierwszym jest nazwa tablicy, drugim — numer indeksu, od którego ma rozpocząć czyszczenie, a trzecim — zasięg tego procesu. Dzięki metodzie `Clear()` można bowiem wyczyścić określone elementy z tablicy. Poniższy przykład prezentuje czyszczenie całej zawartości:

```
Array.Clear(Foo, 0, Foo.Length);
```

Metoda `Clear()` nie zmienia rozmiaru czyszczonej tablicy. Jeżeli czyścimy tablicę, która ma — powiedzmy — 5 elementów, to po przeprowadzeniu tej operacji nadal będzie ich miała tyle samo.

Do elementów wyczyszczonej tablicy ponownie możemy przypisywać jakieś wartości.

Słowo kluczowe `null` w języku C# oznacza wartość pustą.

Clone()

Metoda `Clone()` zwraca kopię tablicy, z której została wywołana — np.:

```
Bar = Foo.Clone();
```

Od tej pory `Bar` będzie posiadała takie same elementy co `Foo`. Ponieważ metoda `Clone()` zwraca dane w postaci typu `object`, należy dokonać rzutowania na właściwy typ. Tzn. jeżeli mamy tablicę typu `string`, należy na niego dokonać odpowiedniego rzutowania, co prezentuje poniższy przykład:

```
string[] Foo = new string[] { "Pn", "Wt", "Śr", "Czw", "Pt" };
// tworzenie kopii
string[] Bar = (string[])Foo.Clone();

foreach (string element in Bar)
{
    Console.WriteLine(element);
}
```

Copy()

Być może lepszym sposobem na utworzenie kopii tablicy będzie zastosowanie metody `Copy()`. Umożliwia ona dodatkowo określenie rozmiarów kopiowania, tj. ile elementów zostanie skopiowanych. Oto przykład:

```
string[] Foo = new string[] { "Pn", "Wt", "Śr", "Czw", "Pt" };
string[] Bar = new string[Foo.Length];
// tworzenie kopii
Array.Copy(Foo, Bar, Foo.Length);
```

Pierwszym parametrem tej metody musi być tablica źródłowa (kopiowana), drugim — tablica, do której skopiowane zostaną elementy. Trzeci parametr to oczywiście ilość kopiowanych elementów.

Find()

Metoda umożliwia przeszukanie całej tablicy w celu znalezienia danego elementu. Kwalifikacja danego elementu jako *znaleziony* lub też nie odbywa się przy pomocy zewnętrznej metody. Oto przykładowy program:

```
{
    Point[] points = {
        new Point(10, 20),
        new Point(100, 200),
        new Point(400, 500)
    };
```

```
        Point first = Array.Find(points, pointFind);

        Console.WriteLine("Found: {0}, {1}", first.X, first.Y);
        Console.Read();
    }

    private static bool pointFind(Point point)
    {
        if (point.X % 2 == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
```

Na samym początku zadeklarowałem tablicę struktur `Point`, które będą przeszukiwane. Program wyszukuje elementów tablicy, w której element `X` struktury `Point` jest liczbą parzystą. Po znalezieniu pierwszego metoda `pointFind()` zwraca `true` i kończy swe działanie.

Struktura `Point` zadeklarowana jest w przestrzeni nazw `System.Drawing`. Nie zapomnij zadeklarować jej użycia przy pomocy słowa `using` oraz dołączyć odpowiedniego podzespołu (*System.Drawing.dll*).

FindAll()

Jak sama nazwa wskazuje, metoda `FindAll()` wyszukuje wszystkie elementy, które spełniają dane kryteria poszukiwań. Oto jak powinien wyglądać program z poprzedniego listingu, jeśli ma wyszukiwać wszystkie elementy:

```
static void Main(string[] args)
{
    Point[] points = {
        new Point(10, 20),
        new Point(100, 200),
        new Point(400, 500)
    };

    Point[] find = Array.FindAll(points, pointFind);

    foreach (Point element in find)
    {
        Console.WriteLine("Found: {0}, {1}", element.X, element.Y);
    }
    Console.Read();
}
```

```
private static bool pointFind(Point point)
{
    if (point.X % 2 == 0)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

FindLast()

Metoda ta działa podobnie jak `Find()`. Jedyna różnica jest taka, że `FindLast()` szuka ostatniego wystąpienia danego elementu; nie kończy pracy, gdy znajdzie pierwszy element pasujący do kryteriów.

GetLength()

Zwraca ilość elementów w tablicy. Umożliwia działanie na tablicach wielowymiarowych. W parametrze tej metody należy podać numer wymiaru, którego ilość elementów ma być pobrana. Jeżeli mamy do czynienia z tablicą jednowymiarową, w parametrze wypisujemy cyfrę 0.

Klasa `System.Array` udostępnia również metodę `GetLongLength()`, która działa analogicznie do `GetLength()`, z tą różnicą, iż zwraca dane w postaci liczby typu `long`.

GetLowerBund(), GetUpperBund()

Metoda `GetLowerBund()` zwraca numer najmniejszego indeksu w tablicy. W przeważającej części przypadków będzie to po prostu cyfra 0. Metoda `GetUpperBund()` zwraca natomiast największy indeks danej tablicy. Obie metody mogą działać na tablicach wielowymiarowych; wówczas należy w parametrze podać indeks wymiaru. Przykładowe użycie:

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

Console.WriteLine("Najmniejszy indeks: {0}, największy: {1}",
        Foo.GetLowerBound(0), Foo.GetUpperBound(0));
```

GetValue()

Prawdopodobnie nie będziesz zmuszony do częstego korzystania z tej metody, zwraca ona bowiem wartość danego elementu tablicy. W parametrze tej metody musisz podać indeks elementu, tak więc jej działanie jest równoznaczne z konstrukcją:

```
Tablica[1];     // zwraca element znajdujący się pod indeksem 1
```

Chcąc wykorzystać tę metodę, kod możemy zapisać następująco:

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

Console.WriteLine(Foo.GetValue(1));
```

Initialize()

We wcześniejszych fragmentach tego rozdziału pisałem o inicjalizacji tablicy. Metoda `Initialize()` może to ułatwić. Jej użycie powoduje przypisanie każdemu elementowi pustej wartości (czyli może to być cyfra 0 lub np. wartość `null`). Jej użycie jest bardzo proste, nie wymaga podawania żadnych argumentów:

```
Foo.Initialize();
```

IndexOf()

Przydatna metoda. Zwraca numer indeksu na podstawie podanej wartości elementu. Przykład użycia:

```
string[] Foo = { "Pn", "Wt", "Śr", "Czw", "Pt", "So", "Nd" };

Console.WriteLine(Array.IndexOf(Foo, "Pt"));
```

W powyższym przykładzie na konsoli zostanie wyświetlona cyfra 4, gdyż pod tym numerem kryje się element *Pt*.

Resize()

Metoda `Resize()` przydaje się wówczas, gdy musimy zmienić rozmiar danej tablicy. W pierwszym jej parametrze musimy podać nazwę tablicy poprzedzoną słowem kluczowym `ref` oznaczającym *referencję*. Drugim parametrem musi być nowy rozmiar tablicy:

```
string[] Foo = { "Pn", "Wt", "Śr", "Czw", "Pt", "So", "Nd" };
Array.Resize(ref Foo, Foo.Length + 5);
```

SetValue()

Metoda `SetValue()` umożliwia nadanie wartości dla danego elementu tablicy. Prawdopodobnie nie będziesz korzystał z niej zbyt często, gdyż to samo działanie można zrealizować przy pomocy operatora przypisania. Gdybyś jednak miał wątpliwości, co do jej użycia, poniżej prezentuję przykład:

```
Foo.SetValue("Weekend", 9);
```

Taki zapis oznacza przypisanie wartości `Weekend` pod indeks nr 9.

Słowo kluczowe params

Mechanizm tablic języka C# nie umożliwia tworzenia tablic dynamicznych, tj. o zmiennym rozmiarze. Zmiana rozmiaru tablic (ilości elementów) jest nieco problematyczna, podobnie jak usuwanie elementów. Warto jednak wspomnieć o słowie kluczowym `params`, używanym w połączeniu z tablicami. Konkretnie z tablicowymi parametrami metod:

```
static void Foo(params string[] args)
{
}
```

Słowo `params`, które poprzedza właściwą deklarację parametru, mówi o tym, iż liczba elementów przekazywanych do metody będzie zmienna. Oto przykład takiego programu:

```
using System;

namespace FooApp
{
    class Program
    {
        static void Foo(params string[] args)
        {
            for (int i = 0; i < args.Length; i++)
            {
                Console.Write(args[i] + " ");
            }
            Console.WriteLine();
        }
        static void Main(string[] args)
        {
            Foo("Adam", "Paulina");

            Foo("Adam", "Paulina", "Marta");

            Console.Read();
        }
    }
}
```

Jak widzisz, możliwe jest przekazanie dowolnej liczby parametrów do metody `Foo()`. Każdy parametr będzie kolejnym elementem tablicy i jest to przydatna cecha języka C#.

Możliwe jest przekazywanie parametrów różnego typu. Nagłówek metody musi wyglądać wówczas tak:

```
static void Foo(params object[] args)
```

Taką metodę można wywołać np. tak:

```
Foo("Adam", 10, 12.2);
```