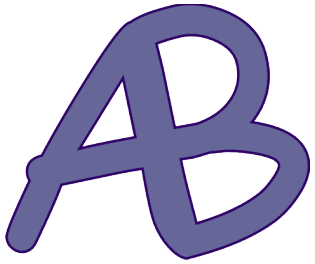
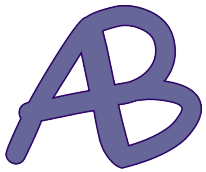


Wykład 1
Składnia języka C#



Budowa projektu

Struktura programu

Najprostszym przypadkiem jest aplikacja konsolowa

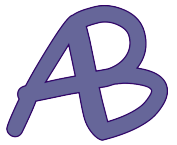
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Deklaracje użytych w projekcie przestrzeni nazw (*ang. namespace*)

Nasza przestrzeń nazw. W niej znajduje się klasa Program zawierająca jedną metodę o nazwie Main.

Kod utworzony przez MS Visual C#

Pierwszy program, czyli tradycyjne „Witaj świecie”



Najprostszym przypadkiem jest aplikacja konsolowa

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

namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Out.WriteLine("Hello World!");
            Console.ReadKey();
        }
    }
}
```

Wypisanie tekstu do strumienia wyjściowego (Out) konsoli

Wczytanie znaku (oczekiwanie na reakcję użytkownika)

Operacje we/wyjścia konsoli

Wyprowadzanie danych przy użyciu konsoli do strumienia wyjściowego Out Console.Out.

```
Console.Out.Write("Tekst");
String s = "Ala ma kota";
Console.Out.Write(s);
```

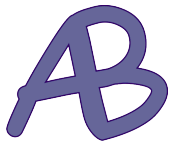
Wyprowadzanie łańcuchów

```
int liczba = 100;
double pi = 3.14;
Console.Out.Write(liczba);
Console.Out.Write(pi);
Console.Out.Write(liczba+" "+pi);
Console.WriteLine("");
```

Metoda **Write** jest przeciążona i pozwala także na wyprowadzanie zmiennych liczbowych

Zwróćmy uwagę, że jeżeli w parametrze metody Write możemy łączyć zmienne i łańcuchy – operatorem +

Operacje we/wyjścia konsoli

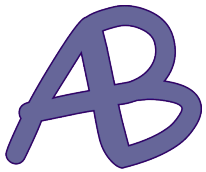


Wczytywanie danych z konsoli:

Matoda Console.WriteLine( ) wczytuje linie tekstu jako łańcuch.

Wczytanie danych liczbowych wymaga zamiany łańcucha na jego wartość (parsowania)

```
int a;
a = int.Parse(Console.ReadLine());
Console.Out.WriteLine(a);

double p=3.6;
p = Double.Parse(Console.ReadLine());
Console.Out.WriteLine(p);
```



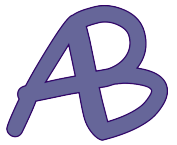
Zmienne

Typy zmiennych

Nazwa typu (słowo kluczowe, alias klasy)	Klasa z obszaru nazw System	Liczba zajmowanych bitów	Możliwe wartości (zakres)	Domyślna wartość
bool	Boolean	1 bajt = 8 bitów	true, false	false
sbyte	SByte	8 bitów ze znakiem	od –128 do 127	0
byte	Byte	8 bitów bez znaku	od 0 do 255	0
short	Int16	2 bajty = 16 bitów ze znakiem	od –32 768 do 32 767	0
int	Int32	4 bajty = 32 bity ze znakiem	od –2 147 483 648 do 2 147 483 647	0
long	Int64	8 bajtów = 64 bity ze znakiem	od –9 223 372 036 854 775 808 do 9 223 372 036 854 775 807	0L
ushort	UInt16	2 bajty = 16 bitów bez znaku	maks. 65 535	0
uint	UInt32	4 bajty = 32 bity bez znaku	maks. 4 294 967 295	0
ulong	UInt64	8 bajtów = 64 bity bez znaku	maks. 18 446 744 073 709 551 615	0L

Użyte w tej prezentacji tabelki pochodzą z książki: Visual Studio 2013. Podręcznik programowania w C# z zadaniami Autor: Matulewski Jacek

Typy zmiennych

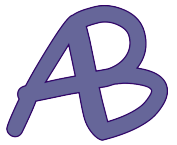


`char`	`Char`	2 bajty = 16 bitów (zob. komentarz)	Znaki Unicode od U+0000 do U+FFFF (od 0 do 65 535)	`'\0'`
`float`	`Single`	4 bajty (zapis zmiennoprzecinkowy)	Wartość może być dodatnia lub ujemna; najmniejsza bezwzględna wartość to $1{,}4 \cdot 10^{-45}$, największa to ok. $3{,}403 \cdot 10^{38}$	`0.0F`
`double`	`Double`	8 bajtów (zapis zmiennoprzecinkowy)	Najmniejsza wartość to $4{,}9 \cdot 10^{-324}$, największa to ok. $1{,}798 \cdot 10^{308}$	`0.0D`
`decimal`	`Decimal`	16 bajtów (większa precyzja, ale mniejszy zakres niż `double`)	Najmniejsza wartość to 10^{-28}, największa to $7{,}9 \cdot 10^{28}$	`0.0M`

Zakres większości typów „prostych” można sprawdzić za pomocą statycznych pól MinValue i MaxValue, np. double.MaxValue, natomiast ich rozmiar liczony w bajtach zwraca operator sizeof.

Wartość domyślną typu można odczytać za pomocą słowa kluczowego default, np. default(int). W przypadku typów liczbowych jest nią zero

Typy zmiennych



W języku C# wszystkie zmienne (w tym także tak zwane typy proste) są strukturami, a słowa kluczowe znane z C, C++, takie jak int, double, czy string są aliasami nazw tych struktur.

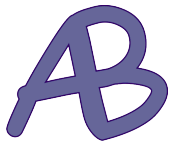
Stała typu int jest instancją struktury System.Int32 i pozwala na dostęp do wszystkich metod tej klasy.

```
int i=10;
string s=i.ToString();
```

Dotyczy to także stałych dosłownych (liczbowych):

```
string s = 5.ToString();
```

Typy zmiennych

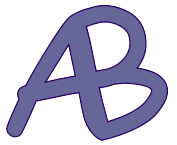


Oprócz stałych dosłownych (liczbowych) o typowej postaci, jak 1, 1.0 lub 1E0, można wykorzystywać dodatkowe stałe uzupełnione o litery określające typ stałej

Stałe liczbowe

Zapis w kodzie C#	Typ	Opis
`1`	`int`	Liczba całkowita ze znakiem
`1U`	`uint`	Liczba całkowita bez znaku
`1L`	`long`	Liczba całkowita 64-bitowa
`1F`	`float`	Liczba
`1M, 1.0M`	`decimal`	Liczba całkowita ze znakiem o zwiększonej precyzji (dzięki zapisowi podobnemu do liczb zmiennoprzecinkowych)
`1E0, 1.0E0, 1.0`	`double`	Liczba zmiennoprzecinkowa
`01`		Liczba ósemkowa
`0x01`		Liczba szesnastkowa

Typy zmiennych – „niejawny” typ var



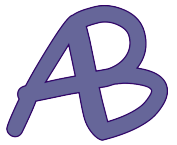
Typ zmiennej określić można przy inicjalizacji przy użyciu pseudotypu var

```
var i = 100;
var l = 100L;
var s = "Wyklad";
var f = 100.0f;
var d = 100.0;
```

Uwaga - typ ustalany jest tylko raz i nie podlega zmianom przez cały czas istnienia zmiennej. Więc do zmiennej typu var zainicjalizowanej stałą typu int nie można pużniej przypisać łańcucha.

Zmienne tego typu mogą być jedynie zmiennymi lokalnymi – mogą być deklarowane i inicjalizowane tylko wewnątrz metody, nie mogą być polami klas.

Konwersja typów i rzutowanie

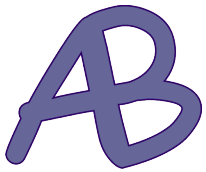


Typ zwracanej przez operator wartości zależy od typów użytych argumentów. Wybierany jest typ o większej precyzji lub większym zakresie.

```
1.0+1;    // double
1.0/2     // double, wartość 0.5
1/2       // int, wartość 0
```

Rzutowanie typów:

```
(double)1/2    //double wartość 0.5
```



Operatory

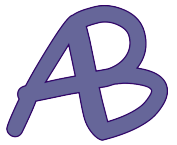
Operatory

Operatory podstawowe

Operator	Opis
`x.y`	Dostęp do pola, metody, właściwości i zdarzenia
`f()`, `f(x)`	Wywołanie metody
`a[n]`	Odwołanie do elementu tablicy lub indeksatora
`x++`, `x--`	Zwiększenie lub zmniejszenie wartości o 1 po wykonaniu innej instrukcji (np. po wykonaniu `int x=2; int y=x++;` otrzymamy `y=2, x=3`)
`new`	Tworzenie obiektu, składnia: *`Klasa referencja`*`=new` *`Klasa(arg_konstr);`*
`typeof`	Zwraca zmienną typu `System.Type` opisującą typ argumentu: `typeof(int).ToString()` równy jest `System.Int32`
`checked`, `unchecked`	Operatory kontrolujące zgłaszanie wyjątku przekroczenia zakresu dla operacji na liczbach całkowitych `int i = unchecked(int.MaxValue + 1);`

Użyte w tej prezentacji tabelki pochodzą z książki: Visual Studio 2013. Podręcznik programowania w C# z zadaniami Autor: Matulewski Jacek, Helion

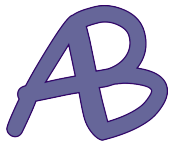
Operatory



Operatory jednoargumentowe

`x.y`	Dostęp do pola, metody, właściwości i zdarzenia
`f()`, `f(x)`	Wywołanie metody
`a[n]`	Odwołanie do elementu tablicy lub indeksatora
`x++`, `x--`	Zwiększenie lub zmniejszenie wartości o 1 po wykonaniu innej instrukcji (np. po wykonaniu `int x=2; int y=x++;` otrzymamy y=2, x=3)
`new`	Tworzenie obiektu, składnia: *`Klasa referencja`*`=new` *`Klasa(arg_konstr);`*
`typeof`	Zwraca zmienną typu `System.Type` opisującą typ argumentu: `typeof(int).ToString()` równy jest `System.Int32`
`checked`, `unchecked`	Operatory kontrolujące zgłaszanie wyjątku przekroczenia zakresu dla operacji na liczbach całkowitych `int i = unchecked(int.MaxValue + 1);`

Operatory



Operatory równości

`==, !=`	Wyrażenie `0.5==1/2f` jest prawdziwe (wartość logiczna). `0.5!=1/2`, czyli `0.5!=0` jest również prawdziwe

Operatory porównania

`<, >, <=, >=`	Porównanie wartości: `1>2` ma wartość `false`
`is`	Porównanie typów: `1 is int` ma wartość `true`
`as`	Rzutowanie równoznaczne z: *`wyrażenie`* `is` *`typ`* `?` `(`*`typ`*`)`*`wyrażenie`* `:` `(`*`typ`*`)null`

Operatory

Operatory dodawania i odejmowania

+, -	Dodawanie i odejmowanie. Typ wyniku zależy od typu argumentów, więc: `uint i=1; uint j=3; uint k=i-j;` spowoduje, że `k=4294967294`

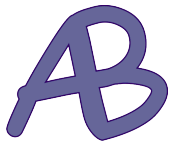
Operatory mnożenia

*, /, %	Operator dzielenia zwraca wynik zgodny z typem argumentów: `5/2=2, 5/2.0=2.5, 5f/2=2.5` Operator reszty z dzielenia: `5%2=1`

Operacje na wartościach logicznych

&&	Logiczne `AND` (`true` jedynie wtedy, gdy oba argumenty są `true`)
||	Logiczne `OR` (`true`, gdy przynajmniej jeden argument jest `true`)

Operatory



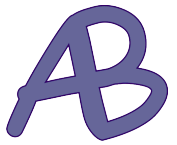
Operacje na bitach

&	Bitowe AND (i), tzn. bit jest zapalany tylko wtedy, jeżeli oba odpowiednie bity argumentów są zapalone 74&15=10 (01001010 & 00001111 = 00001010)
^	Bitowe XOR (rozłączne lub) — bit jest równy 1 tylko wtedy, gdy odpowiednie bity argumentów są różne 74^15=69 (01001010 ^ 00001111 = 01000101)
|	Bitowe OR (lub) — bit jest zapalony, jeżeli przynajmniej jeden odpowiedni bit argumentów jest zapalony 74|15=79 (01001010 | 00001111 = 01001111)

Operatory przesunięcia bitowego

<<, >>	1<<1 = 2, bo jedynka została przesunięta z ostatniej pozycji na przedostatnią i zostało dostawione zero

Operatory



Operacje na wartościach logicznych

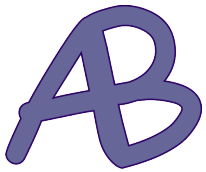
`&&`	Logiczne `AND` (`true` jedynie wtedy, gdy oba argumenty są `true`)
`||`	Logiczne `OR` (`true`, gdy przynajmniej jeden argument jest `true`)

Operatory przesunięcia bitowego

`?:`	*`warunek`*`?`*`wartość`*`_jeżeli_true:`*`wartość`*`_jeżeli_false`, gdzie *`warunek`* musi mieć wartość typu `bool`, np. `x>y?x:y` zwraca większą wartość spośród `x` i `y`

Operatory przypisania

`=` oraz `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`	Inicjacja i zmiana wartości zmiennych, np. polecenia:
	`int x=1;`
	`x+=2;`
	nadadzą zmiennej `x` wartość 3



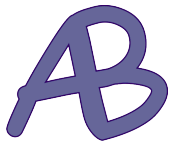
Łańcuchy (string)

String

Łańcuchy są implementowane w klasie System.String (używamy częściej aliasu string), tam zdefiniowane zostały metody i właściwości. Pozwalają one m.in. na porównywanie łańcuchów, analizę ich zawartości oraz modyfikacje poszczególnych znaków lub fragmentów.. Dostęp do metod klasy String możliwy jest zarówno wtedy, gdy dysponujemy zmienną typu string, jak i na rzecz stałych łańcuchowych.

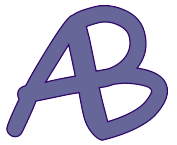
```
string s = "Wyklad";

int dlugosc = s.Length;

int dlugosc2 = "Wyklad".Length;
```

String



- ✓ C# używa znaków Unicode (każdy znak kodowany jest dwoma bajtami) - nie istnieje zatem problem ze znakami narodowymi.
- ✓ Przeciążony operator + służący do łączenia łańcuchów.
- ✓ Ciągi definiujące łańcuchy mogą zawierać sekwencje specjalne rozpoczynające się od lewego ukośnika (znaku backslash) \ (często wykorzystywane: znak końca linii \n, znak cudzysłowu \„).
- ✓ Sekwencje pozwalające definiować znaki Unicode (także spoza dostępnego na klawiaturze zestawu ASCII) zaczynają się od \u i numer znaku, np. \u0048.

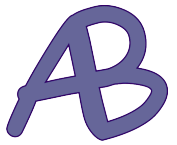
Metody klasy String



Funkcja wartość nazwa(argumenty)	Opis	Przykład użycia
indeksator []	Zwraca znak na wskazanej pozycji; pozycja pierwszego znaku to 0	`"Helion"[0]`
bool Equals(*string*)	Porównuje łańcuch z podanym w argumencie	`"Helion".Equals("HELP")`
int IndexOf(*char*), *int* IndexOf(*String*)	Pierwsze położenie litery lub łańcucha; −1, gdy nie zostanie znaleziony	`"Lalalalala".IndexOf('a')`
int LastIndexOf(*char*), *int* LastIndexOf(*String*)	Jw., ale ostatnie wystąpienie litery lub łańcucha	`"Lalalalala".LastIndexOf('a')`
int Length	Zwraca długość łańcucha, właściwość	`"Helion".Length`

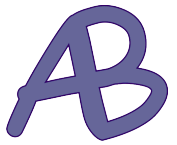
Użyte w tej prezentacji tabelki pochodzą z książki: Visual Studio 2013. Podręcznik programowania w C# z zadaniami Autor: Matulewski Jacek, Helion

Metody klasy String



string Replace(*string*,*string*), *string* Replace(*char*,*char*)	Zamiana wskazanego fragmentu na inny	"HELP".Replace("P","ION")
string Substring(*int*,*int*)	Fragment łańcucha od pozycji w pierwszym argumencie o długości podanej w drugim	"Wydawnictwo".Substring(5,3)
string Remove(*int*,*int*)	Usuwa wskazany fragment	"Helion".Remove(1,2)
string Insert(*int*,*string*)	Wstawia łańcuch przed literą o podanej pozycji	"Helion".Insert(2, "123")
string ToLower() *string* ToUpper()	Zmienia wielkość wszystkich liter	"Helion". ToLower()
string Trim() oraz TrimStart, TrimEnd	Usuwa spacje z przodu i z tyłu łańcucha	" Helion ".Trim()

Metody klasy String



`string PadLeft(int,char)` `string PadRight(int,char)`	Uzupełnia łańcuch znakiem podanym w drugim argumencie, aż do osiągnięcia długości zadanej w pierwszym argumencie	`"Helion".PadLeft(10,' ')`
`bool EndsWith(string),` `bool StartsWith(string)`	Sprawdza, czy łańcuch rozpoczyna się lub kończy podanym fragmentem	`"csc.exe".EndsWith("exe")`

StringBuilder

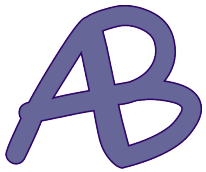
string jest typem referencyjnym, lecz jego operator przypisania = zdefiniowany jest tak, że powoduje klonowanie obiektu.
Dotyczy to również jego metod służących do manipulacji zawartością łańcucha (nie modyfikują bieżącej instancji łańcucha, a tworzą i zwracają nowy łańcuch)

```
string s = "uzytkownikX";
s += "domena.pl";
s = s.Replace("X", "@");
Console.WriteLine(s);
```

string

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("uzytkownikX");
sb.Append("domena.pl");
sb.Replace("X", "@");
Console.WriteLine(sb.ToString());
```

StringBuilder



Instrukcje sterujące

Instrukcja warunkowa if … else

```
if(warunek) instrukcja;
if(warunek) instrukcja; else alternatywna_instrukcja;
```

Przykład:

```
int x;
Console.Write("x= ");
string s = Console.ReadLine();
x = int.Parse(s);

if (x >= 0) Console.WriteLine("Liczba dodatnia");
else Console.WriteLine("Liczba ujemna");
```

Instrukcja wyboru switch

Przykład:

```
Console.Write("Podaj numer dnia tygodnia: ");
string s = Console.ReadLine();
int n = int.Parse(s);
string opis;
switch (n)
{
    case 1: opis = "niedziela"; break;
    case 2: opis = "poniedziałek"; break;
    case 3: opis = "wtorek"; break;
    case 4: opis = "środa"; break;
    case 5: opis = "czwartek"; break;
    case 6: opis = "piątek"; break;
    case 7: opis = "sobota"; break;
    default: opis = "Tydzień ma tylko 7 dni"; break;
}
Console.WriteLine("Dzień tygodnia: " + n + ", " + opis);
```

Pętla for

```
for (inicjalizacja_indeksu ; warunek ; inkrementacja)
  instrukcja;
```

Przykład:

```
int n = 100;
string tekst = "Nauczę się programować w C#";
for (int i=0; i<n; i++) Console.Write(i.ToString()+ " " + tekst + " ");
```

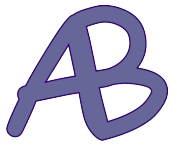
Pętla while

```
while (warunek) instrukcja;
```

Przykład:

```
int n = 100;
string tekst = "Nauczę się programować w C#";
int i=0;
while (i<n)
{
    Console.Write(i.ToString()+ " " + tekst + " ");
    i++;
}
```

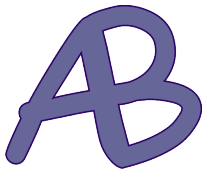
Pętla do … while



```
do instrukcja while (warunek);
```

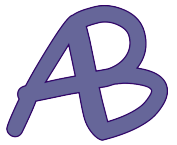
Przykład:

```
int n = 100;
string tekst = "Nauczę się programować w C#";
int i=0;
do
{
    Console.Write(i.ToString() + " " + tekst + " ");
    i++;
} while (i < n);
Console.ReadKey();
```



Metody

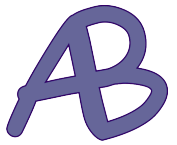
Metody



- ✓ W C# nie jest możliwe definiowanie funkcji niebędących metodami jakiejś klasy.
- ✓ Funkcja może być statyczną składową klasy, ale zawsze jest metodą (tj. właśnie funkcją składową zdefiniowaną w obrębie klasy).

  Metody statyczne, to takie, które można wywołać bez tworzenia instancji klasy, w której są zdefiniowane. Do ich definicji dodamy modyfikator **static**
- ✓ Metody definiować możemy w obrębie istniejącej klasy (klasy Program). Będą to metody statyczne, bo metoda main( ) z której będą one wywoływane jest statyczna.

Metody



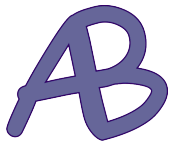
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication4
{
    class Program
    {
        static void Metoda()
        {
            Console.WriteLine("Hello World!"); // ciało metody
        }

        static void Main(string[] args)
        {
            Metoda();
        }
    }
}
```

Przeciążanie metod

Język C# umożliwia definiowanie wielu metod o tych samych nazwach, pod warunkiem że różnią się parametrami (dzięki temu mają również inne sygnatury). Nazywa się to przeciążaniem metody (ang. overload). Niemożliwe jest natomiast definiowanie dwóch metod różniących się jedynie zwracanymi wartościami.

```
static void Metoda()
{
    Console.WriteLine("Hello World!");
}
static void Metoda(string tekst)
{
    Console.WriteLine(tekst);
}
static void Main(string[] args)
{
    Metoda();
    Metoda("Witaj, świecie!");
}
```

Domyślne wartości metod



Możliwe jest ustalanie domyślnych wartości parametrów metod. Dzięki temu przy wywołaniu metody argument jest opcjonalny — jeżeli nie wystąpi w liście argumentów w instrukcji wywołania metody przyjmie wartość domyślną

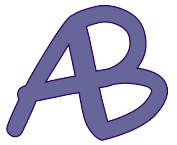
```
static void Main(string[] args)
{
    Przywitaj_sie();          //wypisze Witaj
    Przywitaj_sie("Cześć");   //wypisze Cześć
    Console.ReadKey();
}

private static void Przywitaj_sie(string s = "Witaj")
{
    Console.WriteLine(s);
}
```

Domyślne wartości metod

PRZYKŁAD

```
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication4
{
    class Program
    {
        static void Metoda(string tekst, ConsoleColor kolor = ConsoleColor.White)
        {
            ConsoleColor bieżącyKolor = Console.ForegroundColor;
            Console.ForegroundColor = kolor;
            Console.WriteLine(tekst);
            Console.ForegroundColor = bieżącyKolor;
        }
        static void Main(string[] args)
        {
            Metoda("Witaj Świecie", ConsoleColor.Cyan);
            Metoda("Witaj Świecie");
            Console.ReadKey();
        }
    }
}
```

Argumenty nazwane



Możliwa jest identyfikacja parametrów nie za pomocą ich kolejności, a przy użyciu ich nazw, np.:

Dla metody:

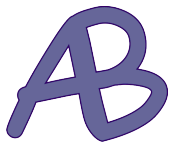
```
static void Metoda(string tekst, ConsoleColor kolor = ConsoleColor.White)
{…..}
```

Poprawne są oba wywołania

```
Metoda(kolor: ConsoleColor.Green, tekst: "Witaj, świecie!");
Metoda(tekst: "Witaj, świecie!", kolor: ConsoleColor.Green);
```

Zaletą tego rozwiązania jest czytelność kodu.

Przekazywanie argumentów do metody przez wartości i referencje

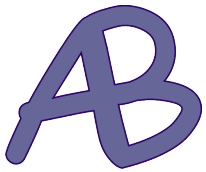


```
static private void zwiększ(int liczba)
{
    liczba = liczba + 100;
    Console.WriteLine("Po zwiekszeniu: " + liczba);
}
static void Main(string[] args)
{
    int x = 0;
    zwiększ(x);
    Console.WriteLine("Po wyjsciu z metody: " + x);
    Console.ReadKey();
}
```

```
Po zwiekszeniu: 100
Po wyjsciu z metody: 0
```

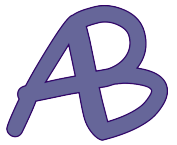
```
Po zwiekszeniu: 100
Po wyjsciu z metody: 100
```

```
static private void zwiększ(ref int liczba)
{
    liczba = liczba + 100;
    Console.WriteLine("Po zwiekszeniu: " + liczba);
}
static void Main(string[] args)
{
    int x = 0;
    zwiększ(ref x);
    Console.WriteLine("Po wyjsciu z metody: " + x);
    Console.ReadKey();
}
```



Zmienne dynamiczne

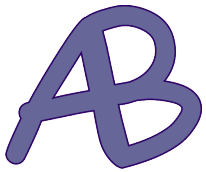
Zmienne dynamiczne



Zmienne proste możemy deklarować jako statyczne lub dynamiczne (na stosie lub stercie) – w C# nie ma to jednak tak dużego znaczenia jak w C++

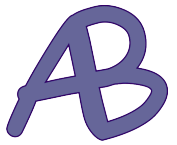
```
int liczba1;
int liczba2 = new int( );

char znak1;
char znak2 = new char( );
```

```
int32 liczba = new int32();
int liczba = new int();
int liczba = 1;
```



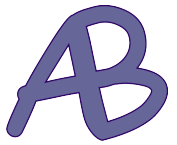
Kolekcje

Pojęcie kolekcji



- ✓ Struktury danych: tablice, listy, kolejki, drzewa itp. zostały w C# nazwane kolekcjami.
- ✓ Typowe kolekcje zostały zaimplementowane w platformie .NET i są gotowe do użycia.
- ✓ Kolekcje zawarte są przestrzeni nazw System.Collections.Generic
- ✓ oraz System.Collections.Specialized zawierającej wyspecjalizowane wersje kolekcji.

Tablice



- Tablica jest w instancją klasy System.Array
- Składnia deklaracji referencji (wskaźnika) do tablicy elementów typu int:

```
int[ ] tab;
```

- Deklaracji referencji wraz z utworzeniem obiektu tablicy (rezerwowana jest pamięć na stercie):

```
int[ ] tab = new int[100];
```

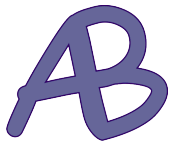
Po utworzeniu obiektu tablicy jest ona automatycznie inicjowana wartościami domyślnymi dla danego typu czyli w przypadku typu int — zerami.

- Inicjalizacja elementów tablicy:

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

```
Int[ ] tab = new int[ ] {0,1,2,3,4,5,6,7,8,9};  //tablica 10-cio elementowa
```

Tablice wielowymiarowe



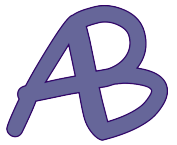
- Deklaracji referencji wraz z utworzeniem obiektu tablicy dwuwymiarowej

```
int[ , ] tab2D = new int[2, 3] { { 0, 1, 2 }, { 3, 4, 5 } };
```

Przykład:

```
int[,] tab2D = new int[2, 3] { { 0, 1, 2 }, { 3, 4, 5 } };
for (int i = 0; i < 2; i++)
{
    for (int j = 0; j < 3; j++)
        Console.Write(tab2D[i, j]);
    Console.WriteLine();
}
```

Pętla foreach



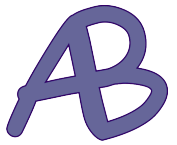
Instrukcja **foreach** wykonuje instrukcję lub blok instrukcji dla każdego elementu w określonym wystąpieniu typu, który implementuje (np. tablicy)

```
int[] tab = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
foreach (int i in tab)
{
    System.Console.Write(i+" ");
}
```

Instrukcja **foreach** działa także na tablicach wielowymiarowych

```
int[,] tab2D = new int[3, 2] { { 100, 200 }, { 101, 201 }, { 102, 202 } };
// int[,] tab2D = { { 100, 200 }, { 101, 201 }, { 102, 202 } };
foreach (int i in tab2D)
{
    System.Console.Write(i+" ");
}
```

Tablice – metody klasy System.Array



Niezwykle przydatną operacją na tablicach jest sortowania

```
int[] tab = new int[50];
Random r = new Random();
for (int indeks = 0; indeks < tab.Length; indeks++)
    tab[indeks] = r.Next(100);

string s = "Wylosowana tablica: ";
foreach (int los in tab) s += los.ToString() + " ";
Console.WriteLine(s);

Array.Sort(tab);  //całe sortowanie w jednej linijce ;)

s = "Tablica po posortowaniu: ";
foreach (int los in tab) s += los.ToString() + " ";
Console.WriteLine(s);
```

Tablice – metody klasy System.Array

Inne przydatne matody klasy System.Array

Właściwość Length – zwraca długość tablicy.

```
tab.Length
```

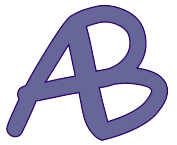
Właściwość Rank – zwraca liczbę wymiarów tablicy

```
tab.Rank
```

Metoda Initialize( ) – inicjuje table wartościociami 0, null, false (zależnie od typu elementów)

```
tab.Initialize()
```

Tablice – metody klasy System.Array



Metoda Sort( ) – sortuje tablicę

```
Array.Sort(tablica);
```

Metoda Resize( ) – zmienia rozmiar tablicy

```
Array.Resize( ref tablica, tablica.Lenght + 5)
```

Metoda Clear( ) – zeruje określoną liczbę elementów tablicy (parametry: tablica, indeks początkowy, indeks końcowy)

```
Array.Clear (tablica, 0, tabliaca.Length -1)
```

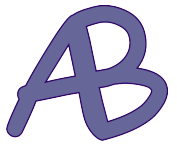
Metoda Clone( ) – tworzy kopię tablicy

```
Int[ ] kopia = (int[ ]) tablica.Clone( );
```

Metoda Copy( ) – Kopiuje elementy do wskazanej tablicy (parametry: tablica źródłowa, tablica docelowa, liczba elementów)

```
Array.Copy (tablica, tablica_docelowa, tabliaca.Length)
```

Tablice – metody klasy System.Array



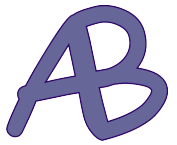
Metoda IndexOf( ) – zwraca index elementu spełniającego podane kryteria

```
int pozycja = Array.IndexOf(tablica, szukan_wartosc);
```

Metoda BinarySearch() – wyszukiwanie binarne w tablicy – zwraca numer indeksu na którym występuje szukana wartość, lub -1 w przypadku braku dopasowań (Uwaga – stosujemy wyłacznie do tablic posortowanych)

```
int pozycja = Array.BinarySearch(tablica, szukan_wartosc);
```

Tablice – metody klasy System.Array



Metoda Find( ) – wyszukuje pierwszy element spełniający podane kryteria

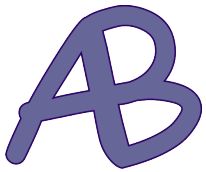
```
int pozycja = Array.Find(tablica, funkcja_testująca);
```

Metoda FindAll( ) – wyszukuje wszystkie elementy spełniający podane kryteria. Wynik zwraca w postaci tablicy

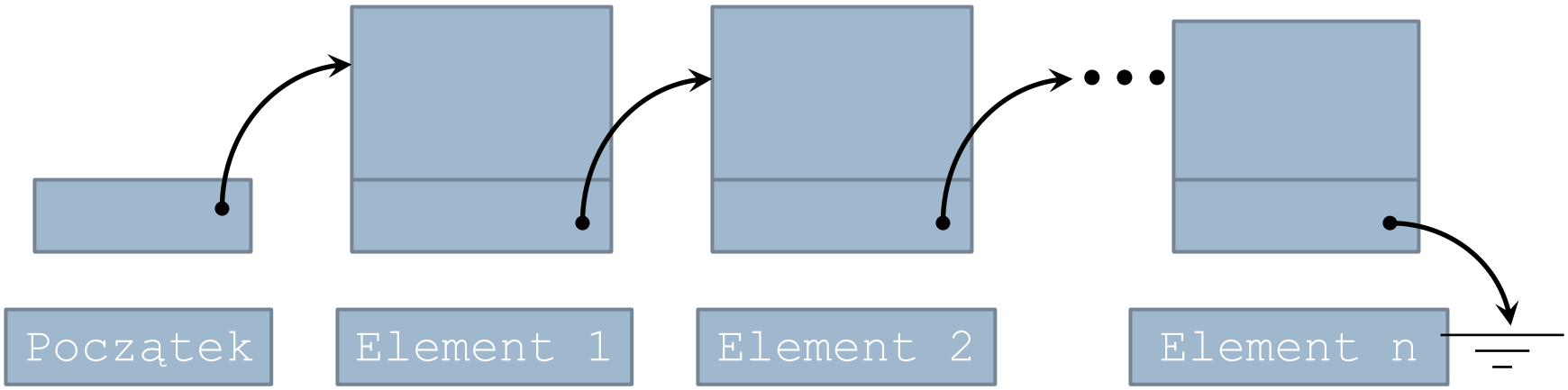
```
int[ ] pozycje = Array.FindAll(tablica, funkcja_testująca);
```

Obie powyższe metody wymagają zdefiniowania **funkcji testującej**, która otrzymuje w parametrze wartość z tablicy i zwraca wartość prawda/fałsz

```
static bool funkcja_testujaca(int obj)
{
    if (obj % 3 == 0) return true; else return false;
}
```



Kolekcje „List” i „SortedList”



Kolekcja „Listy”

Lista - należy do grupy typów ogólnych (ang.generic types).

- ✓ W porównaniu z tablicą (Array) ma tą zaletę, że liczba elementów może być zmieniana już po utworzeniu listy.
- ✓ Można dodawać elementy na koniec, na początek i w środek listy.
- ✓ Można usuwać dowolny element listy.
- ✓ Dostęp do dowolnego elementu listy możliwy jest, tak samo jak w przypadku tablicy - za pomocą operatora [ ]

Kolekcja „Listy”

Tworzenie listy:

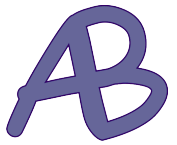
```
List<typ> l = new List<typ>(tab. wart. inicjalizujących);
```

```
List<int> l = new List<int>();

List<String> s = new List<String>();
```

W parametrze konstruktora listy możemy podać tablicę wartości inicjalizujących.

```
List<int> l = new List<int>(new int[] {1,2,3,4,5});

List<String> s = new List<String>(new String[] {"aa","bb","cc"});
```

Kolekcja „Listy”



Podstawowe operacje na listach (na przykładzie listy zawierającej elementy typu String):

```
List<String> nazwa = new List<String>();

nazwa.Add("element");
        - Dodawanie elementu
nazwa.AddRange(new String[] {"aa","bb"});
        - Daodanie tablcy elementów (na koniec listy)
nazwa.Insert(poz, "aa");
        - wstawianie elementu na wskazaną pozycję UWAGA: nie
        zastępujemy tylko wstawiamy
nazwa.InsertRange(poz, new String[]{"aa", "bb"});
        - wstawianie listy elementów na wskazaną pozycję
nazwa.RemoveAt(poz);
        - usunięcie wskazanego elementu (o wsk. indeksie)
nazwa.Remove("bb");
        - usunięcie elementu o wskazanej wartości,
```

Kolekcja „Listy”

Podstawowe operacje na listach c.d.:

nazwa.Clear();
- wyczyszczenie listy

nazwa.Sort();
- sortowanie listy

nazwa.Reverse();
- odwrócenie listy

nazwa.Count();
- podaje liczbę elementów

nazwa.ToArray(TablicaDocelowa);
- eksportuje listę do tablicy.

Operacje na liście - przykład

```
class Program
{
    static Random r = new Random();
    static List<int> l = new List<int>(new int[] { 1, 2, 3, 4, 5 });

    static void Main(string[] args)
    {
        wypisz("test 1");
        for (int i = 0; i < 10; i++)
            l.Add(i);
        wypisz("test 2");
        l.InsertRange(0, new int[] { 10, 20, 30, 40, 50 });
        wypisz("test 3");
        l.Insert(0, 100);
        wypisz("test 4");
        for (int i = 0; i < l.Count; i++)
            if (l[i] % 5 == 0) { l.RemoveAt(i); i--; }
        wypisz("test 5");
        Console.ReadKey();
    }
    static void wypisz(String opis="Zawartosc")
    {
        String s = opis+": ";
        for (int i = 0; i < l.Count; i++)
            s += l[i].ToString() + " ";
        Console.WriteLine(s);
    }
}
```

Kolekcja „SortedList”

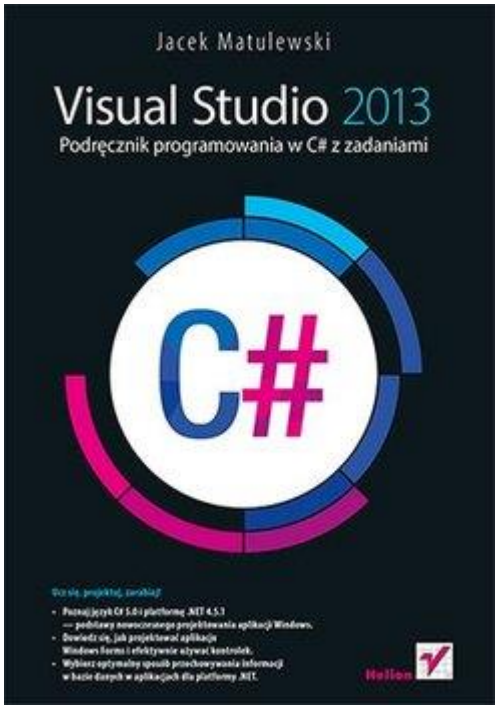
SortedList - w odróżnieniu od omówionej wcześniej jest „dwukolumnowa”.

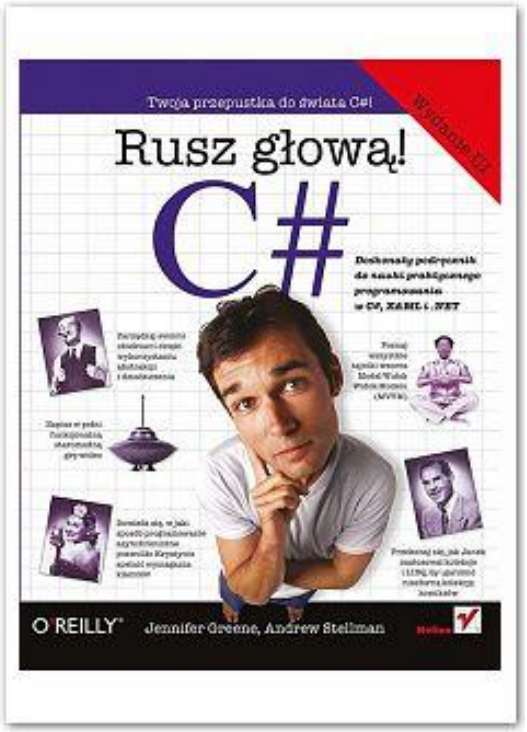
- ✓ Każdy element listy przechowuje klucz i wartość (właściwości Key i Value).
- ✓ Pozwala to sortowanie obu wartości według klucza.

```
class Program
{
    static void Main(string[] args)
    {
        SortedList<string,string> artysci = new SortedList<string,string>();
        artysci.Add("Sting","Gordon Matthew Sumner");
        artysci.Add("Bolesław Prus","Aleksander Głowacki");
        artysci.Add("Pola Negri","Barbara Apolonia Chałupiec");
        artysci.Add("John Wayne","Marion Michael Morrison");
        artysci.Add("Chico","Leonard Marx");
        artysci.Add("Harpo","Arthur Marx");
        artysci.Add("Groucho","Julius Marx");
        artysci.Add("Bono","Paul Hewson");
        artysci.Add("Ronaldo","Luiz Nazario de Lima");
        artysci.Add("Madonna","Madonna Louise Veronica Ciccone");
        artysci.Add("Gabriela Zapolska","Maria G. Śnieżko-Błocka");
        string komunikat = "Zawartość listy:\n";
        foreach (KeyValuePair<string, string> artysta in artysci)
        komunikat += artysta.Key + " - " + artysta.Value + "\n";
        Console.WriteLine(komunikat);
        Console.ReadKey();
    }
}
```

```
Zawartość listy:
Bolesław Prus - Aleksander Głowacki
Bono - Paul Hewson
Chico - Leonard Marx
Gabriela Zapolska - Maria G. Śnieżko-Błocka
Groucho - Julius Marx
Harpo - Arthur Marx
John Wayne - Marion Michael Morrison
Madonna - Madonna Louise Veronica Ciccone
Pola Negri - Barbara Apolonia Chałupiec
Ronaldo - Luiz Nazario de Lima
Sting - Gordon Matthew Sumner
```

Literatura:





Użyte w tej prezentacji tabelki pochodzą z książki:Visual Studio 2013. Podręcznik programowania w C# z zadaniami
Autor: Matulewski Jacek, Helion