

Podstawy programowania

Wykład 1 : C# podstawy

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace ConsoleTesting  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
        }  
    }  
}
```

Pro C# 8 with .NET 3. Foundational Principles and Practices in Programming, 9th Edition, Andrew Troelsen, Phil Japikse, wyd. **Apress**, 2020

Wstęp do programowania w C#

Słowa kluczowe

- Słowa kluczowe – we wszystkich językach programowania wysokiego poziomu – zbiór słów zastrzeżonych, których nie można "przedefiniować", m.in.:
 - abstract, as, base, bool, break, byte, case, catch, char, class, const, continue, decimal, default, do, double, else, enum, event, false, float, for, foreach, if, in, int, interface, internal, long, namespace, new, null, object, operator, out, override, private, protected, public, readonly, ref, return, sbyte, short, sizeof, static, string, struct, switch, this, throw, true, try, typeof, uint, ulong, ushort, using, virtual, void, volatile, while

Słowa kluczowe

- Słowa kluczowe – we wszystkich językach programowania wysokiego poziomu – zbiór słów zastrzeżonych, których nie można "przedefiniować", m.in.:

- ...

- Łatwo rozpoznawalne:

```
using System;
```

```
namespace PodProLab1
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            Console.ReadKey(true);
```

```
        }
```

```
    }
```

```
}
```

Słowa kluczowe kontekstowe

- Słowa, które są lub nie są kluczowe, zależnie od kontekstu; Nie są zastrzeżone, można użyć np. do nazwania zmiennych:
 - add, alias, async, await, by, descending, dynamic, equals, from, get, global, group, into, join, let, nameof, on, orderby, partial, remove, select, set, value, var, when, where, yield

Przykładowy program

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

namespace SimpleCSharpApp
{
    class Program
    {
        static void Main(string[] args)
        {
            // Wyświetl natchniony napis
            Console.WriteLine("***** My First C# App *****");
            Console.WriteLine("Hello World!");
            Console.WriteLine();
            // Czeka na Enter:
            Console.ReadLine();
        }
    }
}
```

Składnia

- C# jest językiem czułym na wielkość liter (*case sensitive*).
- O ile nazewnictwo własnych zmiennych, klas, obiektów, metod, itd. może być w zasadzie dowolne (byle nie zaczynało się od cyfry czy znaku specjalnego), należy jednak zwracać uwagę na poprawne (co do wielkości znaku) wpisywanie rozkazów języka a przede wszystkim nazw typów wbudowanych.
- Jak tylko zobaczymy błąd z tekstem '*undefined symbols*' należy w pierwszej kolejności szukać literówek w nazwach C# i .NET

Nazwy

- Zasady formalne:
 - Może zawierać litery, cyfry oraz znak podkreślenia "_" (tzw. podłoga), można używać znaków diakrytycznych,
 - Małe i wielkie litery SĄ rozróżniane
 - Nie może zaczynać się od cyfry
 - Nie może być słowem kluczowym
- `Int32 gałąź_numer_3;` // ok
 - `Int32 licznik, Licznik, LICZNIK;` // ok, 3 różne nazwy!
 - `Int32 2pi;` // błąd!
 - `Int32 event;` // błąd! Słowo kluczowe

Nazwy

- Zasady nieformalne:
 - Klasy, pola, metody, stałe – konwencja Pacal, nazwy klas i pól – rzeczowniki, metod – czasowniki:
 - DaneWykresu (klasa – należy unikać nazw na "I")
KolorLinii (pole)
ZapiszDane (metoda)
 - Interfejsy – Pascal poprzedzone literą "I"
 - ISortable
 - Zmienne lokalne i parametry funkcji – camelCase:
 - promień
poleKoła

Nazwy

- Zasady nieformalne:

- Nazwy powinny być znaczące:

- p, k,
poleKoła, kolorLinii

Kod powinien być czytelny, kiedy do niego zajrzeć po kilku miesiącach albo dla innego programisty (tzw. samodokumentujący się)

- Język angielski czy polski?

Odpada argument z kaleczeniem języka polskiego

=> reguły zespołu lub indywidualne preferencje

Styl

- Środowisko formatuje wg reguł klasycznych:
 - Nawias "{" i "}" zawsze w nowej linii, "else" w nowej linii, wcięcia odpowiadające głębokości zagnieżdżenia
 - Automatyczne formatowanie – po zakończeniu pisania instrukcji (po wpisaniu ";" lub ostatniego "}"), o ile nie ma błędów; Nie ma re-formatowania po zmianach
 - Na żądanie (Edit > Advanced > Format Document)

Np. dla takiego kodu...

- `if (x>0) {for (i=0; i<3; i++)`
- `{z += t[i]; }`
- `} else {z = 0;}`

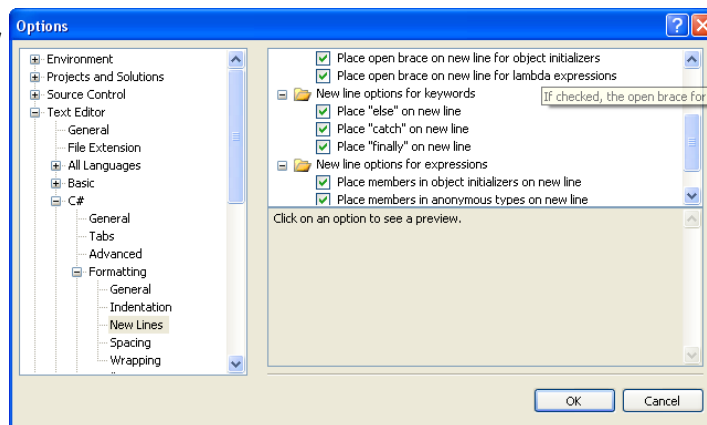
Styl

- Środowisko formatuje wg reguł klasycznych:
 - Nawias "{" i "}" zawsze w nowej linii, "else" w nowej linii, wcięcia odpowiadające głębokości zagnieżdżenia:
 - `if (x > 0)`
 - `{`
 - `for (i = 0; i < 3; i++)`
 - `{`
 - `z += t[i];`
 - `}`
 - `}`
 - `else`
 - `{`
 - `z = 0;`
 - `}`

Styl

- Alternatywne reguły formatowania, np.:
 - Nawias "{" na końcu linii, "}" w nowej linii, "else" razem z "}" (bardziej zwarty):
 - ```
if (x > 0) {
 for (i = 0; i < 3; i++) {
 z += t[i];
 }
} else {
 z = 0;
}
```

- można ustawić w



# Styl

- Komentarze służą do dokumentowania kodu
  - Komentarz liniowy – od "//" do końca linii
    - // Jedna linia wyjaśnienia  
kod...
  - Komentarz blokowy – od "/\*" do "\*/"
    - /\* Kilka linii  
wyjaśnienia \*/  
kod...

# Styl

- Komentarze służą do dokumentowania kodu
  - ...
  - Sekwencja "///" powoduje wstawienie szablonu dokumentacji XML, który jest czytany przez intellisense
    - ```
/// <summary>
///
/// </summary>
/// <param name="a"></param>
/// <param name="b"></param>
/// <returns></returns>
static Int32 NWD(Int32 a, Int32 b)
{
}
```

Inne (prawidłowe) wersje metody Main

```
// int return type, array of strings as the parameter.
static int Main(string[] args)
{
    // Must return a value before exiting!
    return 0;
}
```

```
// No return type, no parameters.
static void Main()
{
}
```

```
// int return type, no parameters.
static int Main()
{
    // Must return a value before exiting!
    return 0;
}
```

- W .NET 7.1 doszły wersje asynchroniczne funkcji `Main()`:

- `static Task Main()`
- `static Task<int> Main()`
- `static Task Main(string[])`
- `static Task<int> Main(string[])`

Main – wartości zwracane

```
static int Main(string[] args)
{
    // Display a message and wait for Enter key
    // to be pressed.
    Console.WriteLine("***** My First C# App *****");
    Console.WriteLine("Hello World!");
    Console.WriteLine();
    Console.ReadLine();
    // Return an arbitrary error code.
    return -1;
}
```

WYNIK:

```
***** My First C# App *****
Hello World!
This application has failed!
return value = -1
All Done.
```

Plik run.bat (do przechwytywanie w trybie konsoli tego, co main zwraca poleceniem return)

```
@echo off
rem A batch file for SimpleCSharpApp.exe
rem which captures the app's return value.
```

```
SimpleCSharpApp
@if "%ERRORLEVEL%" == "0" goto success
```

```
:fail
echo This application has failed!
echo return value = %ERRORLEVEL%
goto end
```

```
:success
echo This application has succeeded!
echo return value = %ERRORLEVEL%
goto
```

```
:end
echo All Done.
```

Dostęp do parametrów wywołania programu z poziomu kodu

- Tablica łańcuchów znaków **args** służy do przechowywania parametrów wywołania programu:

```
static int Main(string[] args)
{
    // Process any incoming args.
    for (int i = 0; i < args.Length; i++)
        Console.WriteLine("Arg: {0}", args[i]);
    Console.ReadLine();
    return -1;
}
```

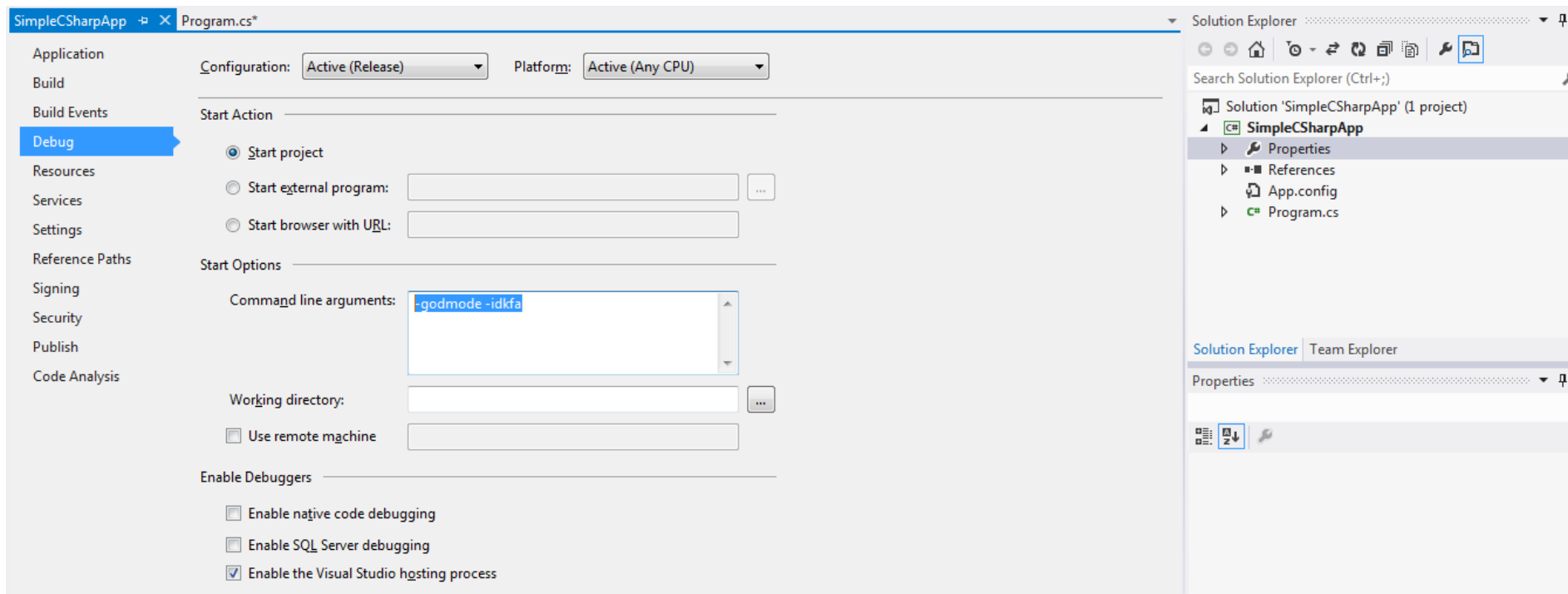
- Inna wersja:

```
static int Main(string[] args)
{
    // Get arguments using System.Environment.
    string[] theArgs = Environment.GetCommandLineArgs();
    foreach (string arg in theArgs)
        Console.WriteLine("Arg: {0}", arg);
    Console.ReadLine();
    return -1;
}
```

Testowe uruchamianie programu z parametrami

- Próba sprawdzenia czy i jak działają programy, które będą mieć parametry wywoływania jest dość uciążliwa, zwłaszcza, jeśli trzeba ją powtarzać.
- Można na szczęście podać parametry wejściowe z poziomu tworzenia aplikacji w Visual Studio:
 - Klikamy podwójnie na **Properties** w oknie Solution Explorer (domyślnie to prawe górne okno)
 - Wybieramy zakładkę **Debug**
 - Parametry wpisujemy w pole "**Command line arguments**"

Command line arguments



Klasa Environment

- Klasa ta zawiera całe mnóstwo bardzo użytecznych metod oraz właściwości, do wywoływania 'od ręki', udostępniających w formie tekstowej informacje o komputerze, np.:
 - GetLogicalDrives
 - OSVersion
 - Version
 - ProcessorCount
 - HasShutDownStarted
 - MachineName
 - Is64BitOperatingSystem
 - CurrentDirectory
 - ... i wiele innych


Klasa Console

| Nazwa | Opis |
|--|--|
| Beep() | Krótki sygnał dźwiękowy. |
| BackColor ForegroundColor | Właściwości ustawiania koloru konsoli, przyjmują wartości z typu wyliczeniowego ConsoleColor (ang. enumeration). |
| BufferHeight BufferWidth | Wysokość i szerokość BUFORA ZNAKÓW konsoli. |
| Title | Tytuł okna konsoli. Zazwyczaj "Formatting C:\ in progress..." robi odpowiednie wrażenie. |
| WindowHeight WindowWidth WindowTop WindowLeft | Kontrola wymiarów konsoli w powiązaniu z rozmiarem bufora danych konsoli. |
| Clear() | Czyści bufor danych. |

Przykład dla klasy Console

```
namespace SimpleCSharpApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Basic Console I/O *****");
            GetUserData();
            Console.ReadLine();
        }
        static void GetUserData()
        {
            // Get name and age.
            Console.Write("Please enter your name: ");
            string userName = Console.ReadLine();
            Console.Write("Please enter your age: ");
            string userAge = Console.ReadLine();
            // Change echo color, just for fun.
            ConsoleColor prevColor = Console.ForegroundColor;
            Console.ForegroundColor = ConsoleColor.Yellow;
            // Echo to the console.
            Console.WriteLine("Hello {0}! You are {1} years old.", userName, userAge);
            // Restore previous color.
            Console.ForegroundColor = prevColor;
        }
    }
}
```

Mechanizm *Code Snippets*

- Mechanizm ten służy do automatycznego tworzenia szkieletu kodu dla popularnych instrukcji / bloków.
- Wpisujemy w Visual Studio litery 'cw' i wciskamy dwukrotnie k .
- W efekcie cw zmienia się w polecenie **Console.WriteLine()**
- Można również przeglądać dostępne snippets, używając kombinacji klawiszy Ctrl+K, Ctrl+X i wybierając odpowiedni snippet z listy.
- Można też tworzyć swoje własne skróty w tym mechanizmie (patrz: manuale Microsoft do Visual Studio)

{0}, {1}, {2}, ... wewnątrz WriteLine()

- Polecenie:

```
Console.WriteLine("Hello {0}! You are {1} years old.", userName, userAge);
```

wstawia wartości dwóch zmiennych w odpowiednie miejsca w tekście napisanym ręcznie w poleceniu.

- Polecenie:

```
Console.WriteLine("{0}, Number {0}, Number {0}", 9);
```

jest jak najbardziej prawidłowe i łatwo się można domyślić jak działa –
wynikiem będzie napis:
„9, Number 9, Number 9”

Inne znaki kontrolne

.NET Numerical Format Characters

| String Format Character | Meaning in Life |
|-------------------------|--|
| C or c | Used to format currency. By default, the flag will prefix the local cultural symbol (a dollar sign [\$] for U.S. English). |
| D or d | Used to format decimal numbers. This flag may also specify the minimum number of digits used to pad the value. |
| E or e | Used for exponential notation. Casing controls whether the exponential constant is uppercase (E) or lowercase (e). |
| F or f | Used for fixed-point formatting. This flag may also specify the minimum number of digits used to pad the value. |
| G or g Stands | for <i>general</i> . This character can be used to format a number to fixed or exponential format. |
| N or n | Used for basic numerical formatting (with commas). |
| X or x | Used for hexadecimal formatting. If you use an uppercase X, your hex format will also contain uppercase characters. |

Przykład wyprowadzania danych liczbowych metodą WriteLine()

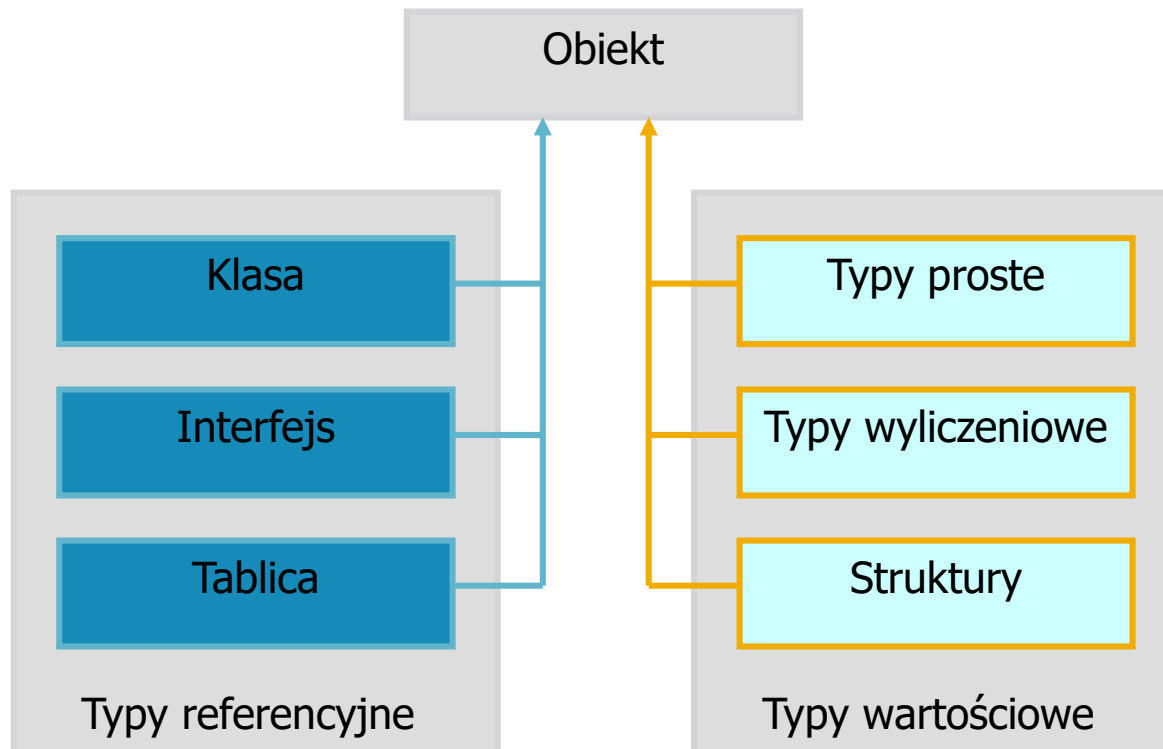
```
static void FormatNumericalData()
{
    Console.WriteLine("The value 99999 in various formats:");
    Console.WriteLine("c format: {0:c}", 99999);
    Console.WriteLine("d9 format: {0:d9}", 99999);
    Console.WriteLine("f3 format: {0:f3}", 99999);
    Console.WriteLine("n format: {0:n}", 99999);
    // Notice that upper- or lowercasing for hex
    // determines if letters are upper- or lowercase.
    Console.WriteLine("E format: {0:E}", 99999);
    Console.WriteLine("e format: {0:e}", 99999);
    Console.WriteLine("X format: {0:X}", 99999);
    Console.WriteLine("x format: {0:x}", 99999);
}
```

Wynik:

```
The value 99999 in various formats:
c format: $99,999.00
d9 format: 000099999
f3 format: 99999.000
n format: 99,999.00
E format: 9.999900E+004
e format: 9.999900e+004
X format: 1869F
x format: 1869f
```

CTS

- Wspólny system typów CTS (Common Type System)



CTS

- Typy wartościowe
 - Deklaracja
deklarowany (tworzony) jest obiekt
 - `Int32 x;`
`x = 7;`
 - Przypisanie
obiekt jest kopiowany
 - `Int32 a, b;`
`a = 13;`
`b = a;`
(są dwa obiekty `Int32`, oba zawierają wartość 13)

CTS

- Typy referencyjne

- Deklaracja

deklarowana (tworzona) jest referencja, obiekt nie istnieje dopóki nie zostanie utworzony operatorem *new*

- Button b1;
b1 = **new** Button();

- Przypisanie

kopiowany jest nie obiekt, tylko referencja do niego

- Button b2;
b2 = b1;

(jest jeden obiekt Button, obie referencje wskazują na niego)

Typy danych w C#

| C# Shorthand | CLS Compliant? | System Type | Range | Meaning in Life |
|--------------|----------------|----------------|--|--|
| bool | Yes | System.Boolean | true or false | Represents truth or falsity |
| sbyte | No | System.SByte | -128 to 127 | Signed 8-bit number |
| byte | Yes | System.Byte | 0 to 255 | Unsigned 8-bit number |
| short | Yes | System.Int16 | -32,768 to 32,767 | Signed 16-bit number |
| ushort | No | System.UInt16 | 0 to 65,535 | Unsigned 16-bit number |
| int | Yes | System.Int32 | -2,147,483,648 to 2,147,483,647 | Signed 32-bit number |
| uint | No | System.UInt32 | 0 to 4,294,967,295 | Unsigned 32-bit number |
| long | Yes | System.Int64 | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 | Signed 64-bit number |
| ulong | No | System.UInt64 | 0 to 18,446,744,073,709,551,615 | Unsigned 64-bit number |
| char | Yes | System.Char | U+0000 to U+ffff | Single 16-bit Unicode character |
| float | Yes | System.Single | -3.4 10^{38} to +3.4 10^{38} | 32-bit floating-point number |
| double | Yes | System.Double | $\pm 5.0 \cdot 10^{-324}$ to $\pm 1.7 \cdot 10^{308}$ | 64-bit floating-point number |
| decimal | Yes | System.Decimal | $(-7.9 \times 10^{28} \text{ to } 7.9 \times 10^{28}) / (10^{0 \text{ to } 28})$ | 128-bit signed number |
| string | Yes | System.String | Limited by system memory | Represents a set of Unicode characters |
| Object | Yes | System.Object | Can store any data type in an object variable | The base class of all types in the .NET universe |

Typy proste C#

- Typ wspólny i rzutowanie
 - Wszystkie mają wspólny typ bazowy "object" – zmienną każdego typu (w tym typu prostego) można przypisać do zmiennej typu object; Odwrotne przypisanie wymaga rzutowania (konwersji typów):
 - ```
Int32 x;
object o;

x = 13;
o = x;
x = (Int32)o; // ok
```
    - ```
Double y;  
y = (Double)o; // błąd! (obiekt w "o" nie jest Double)
```


Typy proste C#

- Typy proste
 - Służą do przechowywania pojedynczych wartości logicznych, liczb, znaków, łańcuchów znaków
 - Wszystkie są strukturami BCL (Base Class Library), wspólne dla .NET; Struktury dostarczają wielu użytecznych metod i stałych, np. dla Int32:
 - Int32.Parse – metoda statyczna, konwertuje łańcuch na liczbę Int32
 - Int32.MaxValue – stała, maksymalna wartość Int32
 - Int32.ToString – metoda, konwertuje liczbę Int32 na łańcuch
 - Mają aliasy w postaci słów kluczowych, np.
 - Int32 – int
 - Single – float
 - Double – double
 - Boolean – bool
- (aliasy są zapożyczone z C/C++/Java)

Typy proste C#

- Typy proste
 - Wartości logiczne:
 - Boolean (bool)
 - Liczby całkowite:
 - Byte (byte), SByte (sbyte)
 - Int16 (short), UInt16 (ushort)
 - Int32 (int), UInt32 (uint)
 - Int64 (long), UInt64 (ulong)
 - Liczby zmiennoprzecinkowe
 - Single (float)
 - Double (double)
 - Decimal (decimal)
 - Znaki i łańcuchy znaków unicode (tzw. widestring)
 - Char (char)
 - String (string)

Typy proste C#

■ Boolean

- Wartości logiczne – true i false (są to słowa kluczowe);
Do zmiennej Boolean można wstawić wyłącznie stałą (jw.)
albo rezultat operacji lub funkcji (metody), o ile jest typu Boolean
 - `Boolean niskiIndeks = indeks < 0.25D;`
`Boolean sukces = Int32.TryParse("100", out liczba);`
- Nie jest dopuszczalna konwersja z wartości liczbowych i odwrotnie
(co jest typowe dla języków C/C++):
 - `Boolean test;`
`test = (Boolean)1; // błąd!`
 - `Int32 number;`
`number = (Int32>true; // błąd!`

Obowiązuje również dla warunków, np. w instrukcji warunkowej

- `Int32 x = 7, y = 0;`
`if (x) // błąd! (ale w C++ ok)`
`y = 2 * x;`

Typy proste C#

- Liczby całkowite:
 - Wyróżniamy 8 odmian; różnią się obecnością znaku oraz ilością zajmowanego w pamięci miejsca (1, 2, 4 i 8 bajtów, tj. 8, 16, 32 i 64 bity) a w konsekwencji zakresem wartości
 - Byte (byte), SByte (sbyte)
 - Int16 (short), UInt16 (ushort)
 - Int32 (int), UInt32 (uint)
 - Int64 (long), UInt64 (ulong)

Typy proste C#

- Liczby całkowite:
 - Zakres wartości wynosi od -2^{n-1} do $2^{n-1}-1$ dla wersji ze znakiem oraz od 0 do 2^n-1 dla wersji bez znaku, np.:
 - SByte: -128 .. 127, Byte 0 .. 255
 - Int16: -32.768 .. 32.767, UInt16: 0 .. 65.535 ±32000, 0 .. 65000
 - Int32: -2.147.483.648 .. 2.147.483.647 ±2 miliardy
 - UInt32: 0 .. 4,294,967,295 0 .. 4 miliardy
 - Int64: ±9,223,372,036,854,775,807 ±9e18
 - UInt64 0 .. 18,446,744,073,709,551,615 0 .. 18e18

Operacje arytmetyczne (np. "+") są wykonywane na Int32 lub Int64; po dodaniu dwóch zmiennych Int16 trzeba dokonać jawnej konwersji wyniku – niepraktyczne, lepiej używać Int32

- Int16 x=1, y=2;
Console.WriteLine((x).GetType()); // x jest Int16
Console.WriteLine((+x).GetType()); // +x to już Int32
Console.WriteLine((x+y).GetType()); // x+y też Int32

Typy proste C#

- Liczby zmiennoprzecinkowe
 - Single (float), 32 bity
 - Zakres wartości: 0, $\pm 1.5e-45$.. $\pm 3.4e38$, \pm nieskończoność, nie-liczba
 - 7 cyfr znaczących
 - Double (double), 64 bity
 - Zakres wartości: 0, $\pm 5.0e-324$.. $\pm 1.7e308$, \pm nieskończoność, nie-liczba
 - 15 cyfr znaczących
 - Decimal (decimal), 128 bitów
 - Zakres wartości: 0, $\pm 1.0e-28$.. $\pm 7.9e28$
 - 28 cyfr znaczących

Single i Double, po przekroczenia zakresu wartości podczas obliczeń, otrzymają wartość 0 lub $\pm\infty$, a ostatecznie NaN (np. $0/0$, $0*\infty$);

Decimal bardziej tradycyjnie wygeneruje wyjątek programowy;

Decimal ma mniejszy zakres wartości przy większej precyzji

- do specyficznych zastosowań, np. obliczeń finansowych

Typy proste C#

■ Znaki i łańcuchy

- Do zapisu znaków używany jest Unicode (1 znak = 2 bajty, UInt16), łańcuchy to sekwencje znaków
 - Char (char)
 - String (string)

W Unicode mieszczą się znaki wszystkich "żywych" alfabetów
Unicode eliminuje kłopoty ze znakami diakrytycznymi, używaniem stron kodowych itp.

- Obie struktury są wyposażone w liczne metody – większość zadań związanych z przetwarzaniem znaków i łańcuchów jest gotowa
- W przetwarzaniu znaków i łańcuchów są uwzględniane ustawienia regionalne (tzw. lokalizacja) – domyślnie wg ustawień systemu, ale można to zmienić

Typy proste C#

- Znaki i łańcuchy
 - ...
 - W C# łańcuchy są traktowane w specjalny sposób – raz nadana wartość nie może być modyfikowana, zamiast tego tworzona jest nowa struktura (z nową wartością), a stara jest usuwana z pamięci przez GC, zupełnie inaczej niż pozostałe typy proste, w których po prostu zmienia się wartość istniejącej całej czas, tej samej zmiennej.

Programy intensywnie przetwarzające łańcuch mogą z tego powodu działać wolniej – każda najmniejsza zmiana wartości łańcucha, choćby jednego jego znaku, powoduje utworzenie nowej struktury i pozbycie się starej. Aby tego uniknąć należy stosować obiekt `StringBuilder`

Typy proste C#

- Konwersje typów prostych

- Konwersja automatyczna jest możliwa, gdy nie ma ryzyka utraty wartości lub pogorszenia precyzji:

- `Int32 a;`
`Int16 b = 77;`
`a = b; // ok, Int32 <- Int16`

w przeciwnym razie jest błędem składniowym:

- `b = a; // błąd: Int16 <- Int32`
`Single y;`
`y = 1.25; // błąd: Single <- Double`

- Można stosować konwersje wymuszone:

- `b = (Int16) a;`

Przy przekroczeniu zakresu wartości konwersja nie kończy się błędem, ale wynik może być niespodzianką

Typy proste C#

- Deklaracja zmiennych
 - Nazwa typu i lista nazw zmiennych, rozdzielona przecinkami:
 - `Int32 a;`
`Double x, y, z;`
 - Można połączyć deklarację z nadaniem początkowej wartości (może to dotyczyć wszystkich lub tylko wybranych zmiennych):
 - `Int32 a = 7;`
`Double x = 13.0, y = 0.0, z;`
 - Może istnieć tylko jedna zmienna o danej nazwie, ponowna deklaracja jest błędem:
 - `Int32 n;`
`Int32 n = 7; // błąd – zmienna n już istnieje`
`Double n; // też błąd`

Typy proste C#

- Zasięg zmiennej

- Zasięg zmiennej lokalnej jest ograniczony do bloku (od "{" do "}"), w którym została zadeklarowana. Tym blokiem jest zawsze funkcja, jednak może to być węższy blok:

- ```
static void Main(string[] args)
{
 Double x;
```
- ```
{
    Double x; // błąd – zmienna x już istnieje
    Int32 n;
}
```
- ```
n = 7; // błąd – tu już nie ma zmiennej n
```
- ```
{
    Double n; // ok., inny blok, nowa zmienna n
}
```
- ```
}
```

# Typy proste C#

- Stałe liczbowe (szerzej: literały)
  - Stałe liczbowe mają określony typ
    - 12 – Int32
    - 12.0 albo 1.2e1 – Double
    - 0x12 – Int32, notacja szesnastkowa (=18 dziesiętnie)
  - Stałe całkowite są niejawnie konwertowane na Byte i Int16, o ile wartość na to pozwala:
    - `Byte b = 500; // błąd!`
  - Aby jawnie określić typ należy stosować sufiksy:
    - U – uint (UInt32)
    - L – long (Int64)
    - F – float (Single)
    - D – double (Double)
    - M – decimal (Decimal)
  - `Single s = 1.25; // błąd! 1.25F będzie ok`
  - `Decimal d = 1.25; // błąd! 1.25M będzie ok`

# Typy proste C#

- Stałe liczbowe (szerzej: literały)

- Stałe znakowe są zapisywane w apostrofach, na kilka sposobów

- Char a, b, c, d, e;  
a = 'A';  
b = '\u0042'; // "B", numer znaku Unicode  
c = '\x0043'; // "C", notacja szesnastkowa  
d = (Char)68; // "D", konwersja z liczby całkowitej  
e = '\t'; // <tab>, tzw. sekwencja ucieczki

- Stałe łańcuchowe są zapisywane w cudzysłowach:

- String s1, s2;  
s1 = "Hello\r\nworld";  
s2 = @"d:\archiwum\plik.zip";

Normalnie wykrywane są sekwencje ucieczki:

"\t" – tabulator, "\r" – CR, "\n" – LF, "\\" – znak "\"

Użycie "@" wyłącza sekwencje ucieczki – przydatne, gdy w łańcuchu występują znaki "\", np. ścieżkach do plików

# Deklaracja zmiennych

- Przykład:

```
static void Main(string[] args)
{
 int myInt;
 string myString;
}
```

- lub:

```
static void Main(string[] args)
{
 int myInt = 0;
 string myString;
 myString = "This is my character data";
}
```

- lub:

```
static void LocalVarDeclarations()
{
 bool b1 = true, b2 = false, b3 = b1;
}
```

# Domyślna inicjalizacja wartością początkową przy użyciu rozkazu `new`

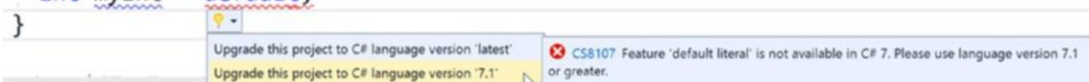
- `bool` domyślnie = `false`
- typy numeryczne 0 lub 0.0
- typ `char` – jeden pusty znak
- `BigInteger` ustawiany na 0
- `DateTime` na 1/1/2001 12:00:00 AM
- obiekty typu `Object` (w tym `String`) – `null`

## ■ Przykłady:

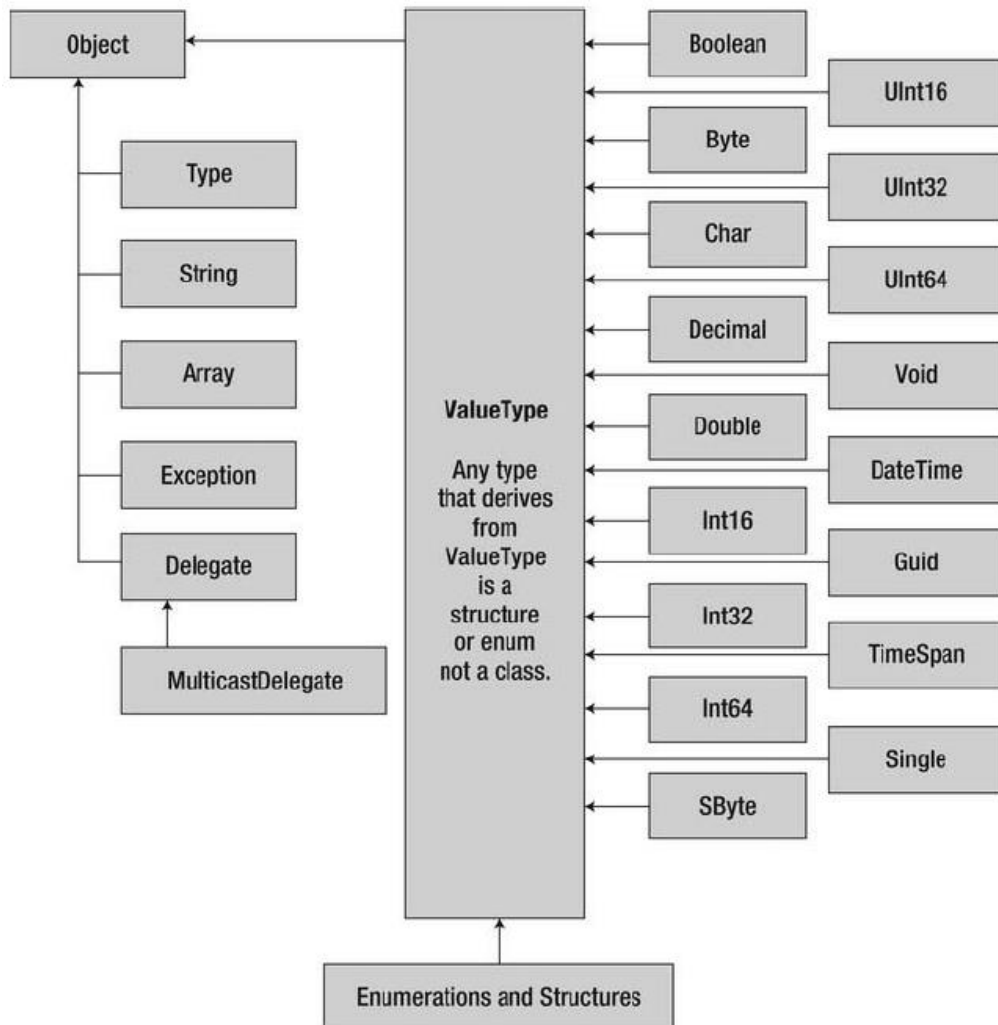
```
bool b = new bool(); // Set to false.
int i = new int(); // Set to 0.
double d = new double(); // Set to 0.
DateTime dt = new DateTime(); // Set to 1/1/0001 12:00:00 AM
```

- A w .NET 7.1 – rozkaz `default` do ustawiania wartości domyślnej.

```
static void DefaultDeclarations()
{
 Console.WriteLine("=> Default Declarations:");
 int myInt = default;
}
```



# Hierarchia typów danych



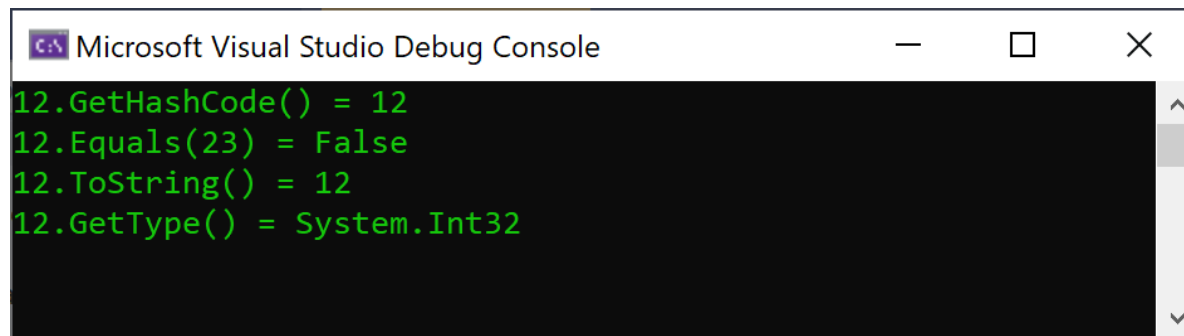
- Warto zaznaczyć, że skoro wszystkie klasy dziedziczą z **System.Object**, to mają dostęp m.in. do metod:
    - **ToString()**
    - **Equals()**
    - **GetHashCode**
- z tejże klasy



# Przykład metod z System.Object

```
// A C# int is really a shorthand for System.Int32,
// which inherits the following members from System.Object.
Console.WriteLine("12.GetHashCode() = {0}", 12.GetHashCode());
Console.WriteLine("12.Equals(23) = {0}", 12.Equals(23));
Console.WriteLine("12.ToString() = {0}", 12.ToString());
Console.WriteLine("12.GetType() = {0}", 12.GetType());
Console.WriteLine();
```

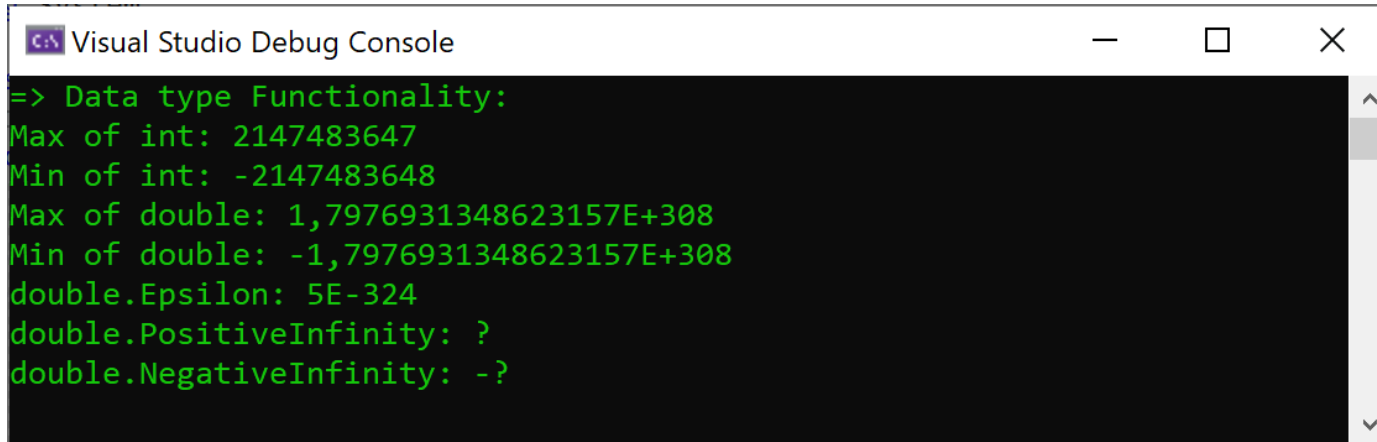
- Efekty działania:



```
Microsoft Visual Studio Debug Console
12.GetHashCode() = 12
12.Equals(23) = False
12.ToString() = 12
12.GetType() = System.Int32
```

# Typy liczbowe – przykłady

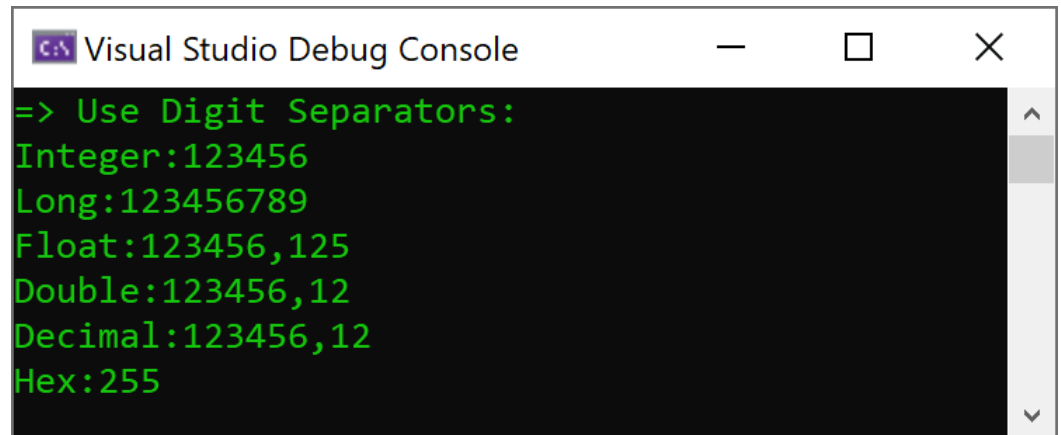
```
Console.WriteLine("=> Data type Functionality:");
Console.WriteLine("Max of int: {0}", int.MaxValue);
Console.WriteLine("Min of int: {0}", int.MinValue);
Console.WriteLine("Max of double: {0}", double.MaxValue);
Console.WriteLine("Min of double: {0}", double.MinValue);
Console.WriteLine("double.Epsilon: {0}", double.Epsilon);
Console.WriteLine("double.PositiveInfinity: {0}", double.PositiveInfinity);
Console.WriteLine("double.NegativeInfinity: {0}", double.NegativeInfinity);
Console.WriteLine();
Console.ReadLine();
```

A screenshot of the Visual Studio Debug Console window. The window title is "Visual Studio Debug Console". The output text is as follows:

```
=> Data type Functionality:
Max of int: 2147483647
Min of int: -2147483648
Max of double: 1,7976931348623157E+308
Min of double: -1,7976931348623157E+308
double.Epsilon: 5E-324
double.PositiveInfinity: ?
double.NegativeInfinity: -?
```

# C# 7.0 i 7.2 – separator części tysięcznych

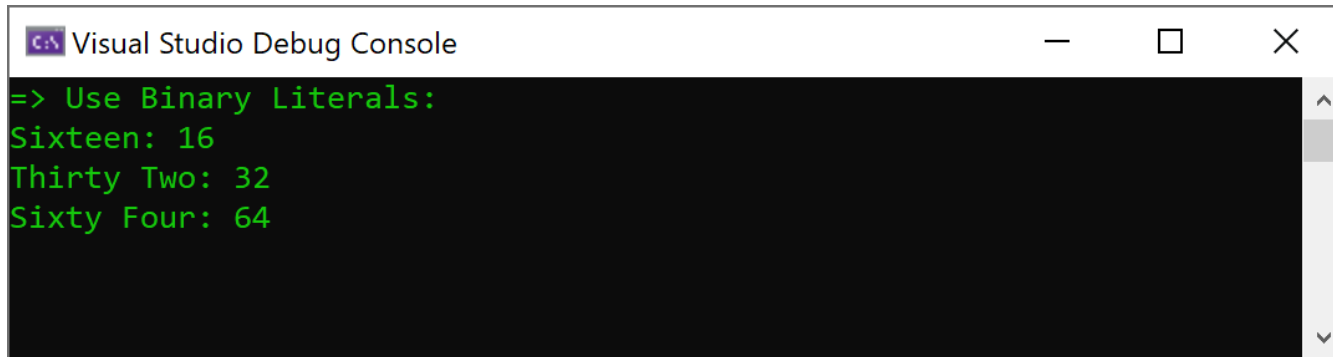
```
Console.WriteLine("=> Use Digit Separators:");
Console.Write("Integer:");
Console.WriteLine(123_456);
Console.Write("Long:");
Console.WriteLine(123_456_789L);
Console.Write("Float:");
Console.WriteLine(123_456.1234F);
Console.Write("Double:");
Console.WriteLine(123_456.12);
Console.Write("Decimal:");
Console.WriteLine(123_456.12M);
//Updated in 7.2, Hex can begin with _
Console.Write("Hex:");
Console.WriteLine(0x_00_00_FF);
```



```
Visual Studio Debug Console
=> Use Digit Separators:
Integer:123456
Long:123456789
Float:123456,125
Double:123456,12
Decimal:123456,12
Hex:255
```

# C# 7.0 i 7.2 – zapis binarny

```
//Updated in 7.2, Binary can begin with _
Console.WriteLine("=> Use Binary Literals:");
Console.WriteLine("Sixteen: {0}", 0b_0001_0000);
Console.WriteLine("Thirty Two: {0}", 0b_0010_0000);
Console.WriteLine("Sixty Four: {0}", 0b_0100_0000);
```

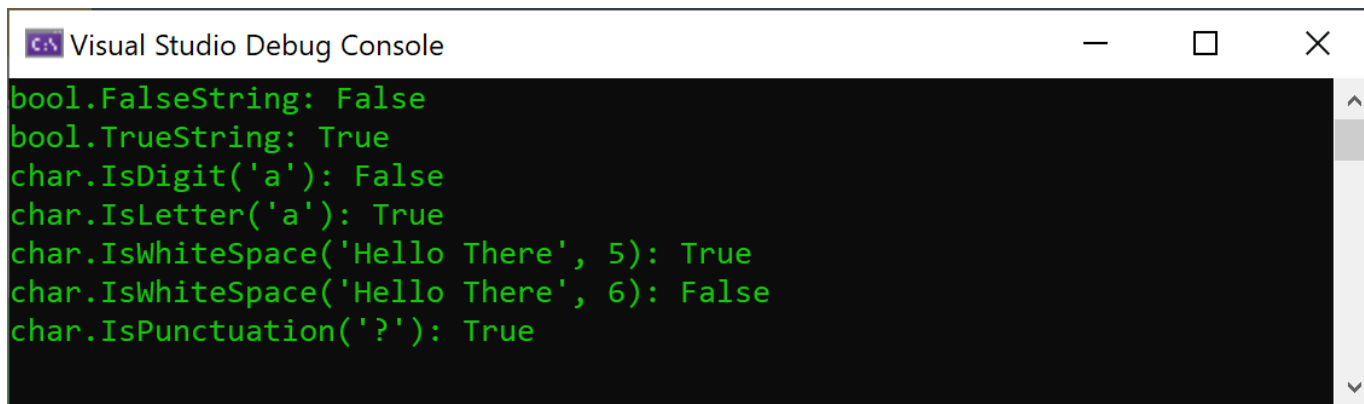


The screenshot shows the Visual Studio Debug Console window. The title bar reads "Visual Studio Debug Console". The console output is as follows:

```
=> Use Binary Literals:
Sixteen: 16
Thirty Two: 32
Sixty Four: 64
```

# Typy bool i char – przykład

```
Console.WriteLine("bool.FalseString: {0}", bool.FalseString);
Console.WriteLine("bool.TrueString: {0}", bool.TrueString);
char myChar = 'a';
Console.WriteLine("char.IsDigit('a'): {0}", char.IsDigit(myChar));
Console.WriteLine("char.IsLetter('a'): {0}", char.IsLetter(myChar));
Console.WriteLine("char.IsWhiteSpace('Hello There', 5): {0}",
 char.IsWhiteSpace("Hello There", 5));
Console.WriteLine("char.IsWhiteSpace('Hello There', 6): {0}",
 char.IsWhiteSpace("Hello There", 6));
Console.WriteLine("char.IsPunctuation('?'): {0}",
 char.IsPunctuation('?'));
Console.ReadLine();
```

A screenshot of the Visual Studio Debug Console window. The window title is "Visual Studio Debug Console". The output text is as follows:

```
bool.FalseString: False
bool.TrueString: True
char.IsDigit('a'): False
char.IsLetter('a'): True
char.IsWhiteSpace('Hello There', 5): True
char.IsWhiteSpace('Hello There', 6): False
char.IsPunctuation('?'): True
```

# Typy proste C#

- Typy wyliczeniowe
  - Służą do definiowania zbioru dopuszczalnych wartości
    - **enum** Color {  
    Red,  
    Green,  
    Blue  
}
    - Color color = Color.Red;

# Typy proste C#

- Typy wyliczeniowe

- Można by definiować stałe "luzem", ale grozi to pomyłkami. Dzięki ścisłej kontroli typów w C#, typy wyliczeniowe gwarantują używanie zawsze poprawnych wartości

Np. kolory konsoli i aplikacji okienkowych są liczbami innego typu. Dzięki typom wyliczeniowym nie ma możliwości pomylenia ich:

- `Console.ForegroundColor = ConsoleColor.Yellow; // ok`  
`Label1.ForeColor = Color.Yellow; // ok`
- `Console.ForegroundColor = Color.Yellow; // błąd`
- `// niezgodny typ, komunikat w oknie błędów głosi:`  
`// Cannot implicitly convert type 'System.Drawing.Color'`  
`// to 'System.ConsoleColor'`

# Operator C#

- Podstawowe (primary) (lewe)
  - `x.y f(a) t[n] x++ x-- new`
- Jednoargumentowe (prawe)
  - `+x -x !x ~x ++x --x (typ)x`
- Mnożenia i dodawania (lewe)
  - `* / %`  
`+ -`
- Przesuwania bitów (lewe)
  - `<< >>`
- Relacji i równości (lewe)
  - `< <= > >= is as`  
`== !=`
- Bitowe i logiczne (lewe)
  - `&`  
`^`  
`|`  
`&&`  
`||`
- Warunkowy (prawe)
  - `?:`
- Przypisania i rozszerzone przypisania (prawe)
  - `= *= /= %= += -= <<= >>= &= ^= |=`



# Operatory C#

- Hierarchia operatorów

Określona dla danego języka programowania kolejność wykonywania operatorów w wyrażeniu

Np. \* jest wyżej w hierarchii niż +, więc w wyrażeniu

- $y = a + b * c;$

najpierw będzie wykonane mnożenie

Różne języki programowania mają różne hierarchie operatorów.

O ile mnożenie jest we wszystkich językach przed dodawaniem, to operatory logiczne mogą być w hierarchii wyżej (np. Pascal) albo niżej (np. C/C++/Java/C#) niż arytmetyczne

Kolejność wynikającą z hierarchii można zmieniać używając nawiasów, wyłącznie "(" i ")", które mogą być zagnieżdżone

- $Y = ((a + b) / (c - d));$

# Operator C#

- Wiązanie operatorów

Określona dla danego języka programowania kolejność wykonywania operatorów o tej samej hierarchii

Większość operatorów ma wiązanie lewe (są wykonywane od lewej)

- $y = a / b * c;$  //  $(a/b)*c$ ; c w liczniku!

Tylko operatory przypisania mają wiązanie prawe

- $x = y = z = 7;$

Kolejność wynikającą z wiązania można zmieniać używając nawiasów, wyłącznie "(" i ")", które mogą być zagnieżdżone

- $y = a / (b * c);$  //  $a/(b*c)$ ; c w mianowniku

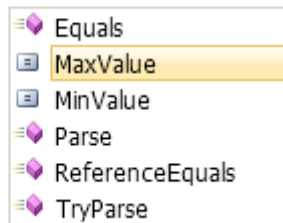
# Operator C#

- Operator dostępu: x.y

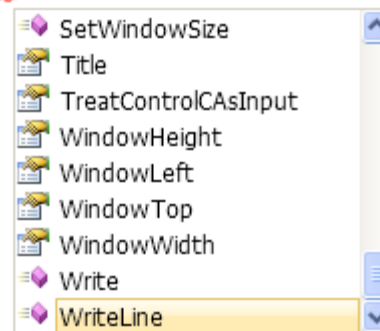
Daje dostęp do metod, pól, właściwości; Wpisanie "." aktywuje Intellisense – widoczność elementów jest zależna od kontekstu, środowisko pamięta też co

O! `Int32 test = 13;`

`test = Int32.`



Console.



# Operator C#

- Operatory zwiększania i zmniejszania

Przyrostkowe operatory ++ oraz -- dostarczają rezultat PRZED zmianą wartości:

- `Int32 x = 0, y;`  
`y = x++; // y=0, x=1`

- Operatory jednoargumentowe zwiększania i zmniejszania

Przedrostkowe operatory ++ oraz -- dostarczają rezultat PO zmianie wartości:

- `Int32 x = 0, y;`  
`y = ++x; // y=1, x=1`

Kolejność ++ oraz x sugeruje kolejność czynności:

++x najpierw zwiększa, potem zwraca x, x++ odwrotnie

Operatory przyrostkowe i przedrostkowe mają inną hierarchię

# Operator C#

- Operatory arytmetyczne (dwuargumentowe):  
mnożenia: \* mnożenie, / dzielenie, % reszta z dzielenia  
dodawania: + -

Działają (na ogół) zgodnie z intuicyjnym rozumieniem;

Działanie i wynik zależy od typów argumentów: jeżeli oba są całkowite, to działanie jest wykonywane jak dla liczb całkowitych, jeżeli chociaż jeden argument jest zmiennoprzecinkowy, to działanie też:

- `Int32 i = 3 / 4; // i = 0`  
`Double d = 3.0 / 4D; // d = 0.75`  
`Double z = 3 / 4; // z = 0.0`

Dla typów całkowitych wynik jest przynajmniej `Int32`, nawet jeżeli oba argumenty są `Byte` lub `Int16`

Reszta z dzielenia działa też dla liczb zmiennoprzecinkowych (inaczej niż w C/C++)

Dla typu `String` jest zdefiniowany operator `+`, oznacza konkatencję

# Operator C#

- Operatory relacji i równości:

relacji: <, <=, >, >=

równości: == !=

Ich sens jest inny niż w matematyce: nie są stwierdzeniem, lecz pytaniem:

- Boolean czyWiększe =  $x > y$ ;  
Boolean czyNierówne =  $x != y$ ;

Rezultat jest true (jeżeli relacja jest prawdziwa) albo false (jeżeli nie)

# Operator C#

- Operatory logiczne:

negacji (not): !

koniunkcja (and): &&

alterantywa (or): ||

Służą do budowania bardziej złożonych wyrażeń logicznych;

Należy pamiętać o hierarchii i wiązaniu, a w razie potrzeby stosować nawiasy

- Boolean warunek = `x > y && !(x > 13 || y < 7);`

Warto zwrócić uwagę na hierarchię – operatory && i || są nisko w hierarchii (za operatorami arytmetycznymi, relacji i równości), więc na ogół nie potrzeba nawiasów, za to "!" jest znacznie wyżej i nawiasy są zwykle potrzebne:

- `W1 = x > 0 && y > 0; // (x>0) && (y>0)`  
`W2 = x < y + 2 || x == y; // (x<(y+2)) || (x==y)`  
`W3 = !(x < y); // bez () jest błąd, (!x) < y`

# Operator C#

- Operatory logiczne:

negacji (not): !

Operator negacji jest rzadko używany w wyrażeniach – zawsze można zmienić znak relacji, np. `!(x>0)` odpowiada `x<=0`, albo skorzystać z praw De Morgana, np. `!(x>0 && y>0)` odpowiada `x<=0 || y<=0`

Operatera negacji używa się w odniesieniu do wartości zwracanych przez funkcje lub właściwości obiektów – ponieważ nie można ani nie warto zmieniać ich działania, np.:

- ```
while (!stream.EndOfStream)
{
    x = stream.ReadLine();
    // ...
}
```


Operator C#

- Operatory logiczne:

koniunkcja (and): `&&`, alternatywa (or): `||`

Program na ogół optymalizuje obliczanie koniunkcji i alternatywy: jeżeli w wyrażeniu `(a && b)` czynnik `a` ma wartość `false`, to całość będzie `false`, niezależnie od wartości `b` – zatem `b` nie jest wyznaczane

Np. poniżej jeżeli `Link.IsOpen` ma wartość `false`, to funkcja `ReadData` nie zostanie wywołana:

- `ReadOK = Link.IsOpen && Link.ReadData(out buffer);`

Czasami ta właściwość jest wykorzystywana dla skrócenia kodu, ponieważ odpowiednio użyta zastępuje instrukcję `if-else`:

- ```
if (!Link.IsOpen)
 ReadOK = false;
else
 ReadOK = Link.ReadData(out buffer);
```

# Operator C#

- Operatory logiczne:  
koniunkcja (and): &&, alternatywa (or): ||  
Podobnie jest optymalizowane wyznaczanie operatora ||

Przykład:

```
■ if (Link.IsOpen || Link.Reconnect())
 {
 Link.ReadData(out buffer);
 }
 else
 {
 Console.WriteLine(" ??? ");
 }
```

Kiedy funkcja Reconnect zostanie wykonana, a kiedy nie?  
Jaki komunikat powinien się pojawić w miejsce "???" ?

# Operator C#

- Operatory przypisania:

zwykły: =

rozszerzone: \*=, +=, ...

Operator przypisania nie jest deklaratywny, jest poleceniem wykonywanym w określonym momencie

- `Int32 a = 7, b = 13, w;`  
`w = a + b; // w=20`  
`a = 13; // w=20`

Rozszerzone operatory przypisania są tylko skróceniem zapisu:

- `w += 1; // dokładnie jak w = w + 1;`  
`w *= b; // dokładnie jak w = w * b;`

Należy pamiętać, że wiązanie jest prawe:

- `x = y = z = 7; // x=7`  
`x += y *= z -= 3; // z=4, y=28, x=35`

# Operator C#

- Operatory przypisania:

Wszystkie operatory przypisania mają skutek (wstawienie wartości do zmiennej) oraz rezultat (wstawiona wartość), np. rezultatem `y=3` jest wartość 3 (ważne: wartość, a nie zmienna lub referencja do niej)

Dzięki temu poniższy kod jest poprawny:

- `x = y = 3;`

Z uwagi na wiązanie prawe operatora = jest to równoważne temu:

- `X = (y = 3);`

zatem do x jest wstawiany rezultat operacji `y=3`, czyli 3

# Operator C#

- Operatory przypisania:

Fakt, że operator = posiada rezultat, pozwala robić np. takie konstrukcje:

- `X = 2 * (y = 3); // ok, y=3, x=6`

To działa, ale kod jest mało czytelny, co może prowadzić do błędów, dlatego takie użycie operatora = nie jest zalecane. Ważne są tu nawiasy, ponieważ operator "=" jest najniżej w hierarchii, zaś bez nawiasów będzie błąd składniowy:

- `X = 2 * y = 3; // błąd, (2*y) = 3, (2*y) nie jest zmienną`

Czasami dopuszcza się takie nadużycie, np. w pętli while:

- `while ("+-*".IndexOf( key = Console.ReadKey(true).KeyChar ) < 0);`

Wplecenie przypisania do wywołania funkcji jest konieczne, ponieważ w warunku pętli można użyć tylko jednego wyrażenia.

Przy okazji – rzadki przykład instrukcji pustej, która nie jest błędem