

Spis treści

1	Zasady	2
2	Literatura	3
3	Język C#	4
4	Podstawy składni C#	5
5	Wyliczenia	9
6	Switch	10
7	Typy referencyjne	15
8	Partial i sealed	16
9	Klasy i struktury	17
10	Tablice	18
11	Operatory	19
12	Modyfikatory dostępu	22
13	Metody i parametry	23
14	Properties	26
15	Indeksatory	27
16	Dziedziczenie	29
17	Użycia new	30
18	Elementy Programowania generycznego	32
19	Delegaty	35
20	Events	37

21	yield	38
22	IComparable	40
23	Lock	43
24	String	43
25	Generowanie dokumentacji	45
26	Atrybuty	47
27	Refleksje	49
28	Instrukcja using	51
29	C++ i C#	53
30	Powtórka	54

1 Zasady

Przedmiot jest podzielony na dwie części. Obie zaliczane oddzielnie. Końcowa ocena jest średnią arytmetyczną obu części zaokrąglaną na korzyść studenta.

Pierwsza część obejmuje elementy języka C#, ma na celu zbudowanie dobrych podstaw do dalszej nauki programowania na platformie .NET.

Zaliczenie części pierwszej

- Zaliczenie laboratorium (50%)
- Zaliczenie wykładu – sprawdzian pisemny (50%)

Kontakt

e-mail: kuszner@sphere.pl

e-mail: deren@sphere.pl

2 Literatura

Książki

- Andrew Troelsen, *Język C# i platforma .NET*, Wydawnictwo Naukowe PWN.

Inne

- Standard ECMA-334 – C# Language Specification – przygotowywana wersja 3.0 języka nie jest zaimplementowana w pakietach dostępnych na uczelni – dostępna jest wersja już wersja Beta pakietu Visual Studio 2008.
- Pomoc pakietu Visual Studio 2005
- <http://binboy.sphere.pl/> - linki do kursów artykułów itp,

Wprowadzenie do programowania na platformie .NET

.NET Framework jest częścią systemu operacyjnego Windows firmy Microsoft. Na platformę .NET składają się dwie części:

- The common language runtime (CLR) oraz
- .NET Framework class library.

CLR jest środowiskiem uruchomieniowym implementującym standard CLI (Common Language Infrastructure, ISO/IEC 23271, ECMA-335). Class Library jest rozbudowaną, zorientowaną obiektowo biblioteką klas, które mogą być użyte do tworzenia różnego rodzaju aplikacji: od tekstowych poprzez graficzne do sieciowych, opartych na ASP.NET.

Języki na platformie .NET

- C#
- Visual C++

- Visual Basic
- Visual J#

Programy narzędziowe

- csc.exe - kompilator C#
- ildasm.exe - deassembler
- AL.exe - assembly linker
- vbc.exe - visual basic compiler
- cl.exe - C, C++ compiler

3 Język C#

Język C# jest językiem stworzonym na potrzeby platformy .NET przez firmę Microsoft. Pierwsza szeroko rozpowszechniona implementacja z roku 2000 była częścią tej właśnie platformy. Za twórców języka uważani są Anders Hejlsberg, Scott Wiltamuth i Peter Golde.

Istnieje ogólnie dostępny standard C# oznaczony ECMA-334, opracowany na podstawie zgłoszenia trzech firm: Microsoft, Intel i Hewlett-Packard.

Implementacje Open Source

Obecnie istnieje kilka projektów implementujących ten standard. Wśród nich kilka z jawnym kodem źródłowym.

- DotGNU Portable.NET - PNET
(<http://www.gnu.org/software/dotgnu/>)
- Mono - projekt sponsorowany przez firmę Novell, obecnie dostępne są kompilatory mcs i gmc (pełna zgodność z .NET 2.0)
(<http://www.mono-project.com/>).
- OCL - biblioteka klas, w której automatycznie wygenerowano interfejsy na podstawie specyfikacji ECMA, implementacja firmy Intel
(<http://sourceforge.net/projects/ocl/>).

- Shared Source Common Language Infrastructure (SSCLI) (poprzednio Rotor) - niekomercyjna implementacja firmy Microsoft (<http://msdn.microsoft.com/>).

Motywacja twórców

Cele, które przyświecały twórcom języka można zawrzeć w kilku punktach:

- C# ma być nowoczesnym, łatwym do opanowania językiem obiektowym ogólnego przeznaczenia.
- Język i jego implementacja powinny zapewniać takie udogodnienia jak: silna kontrola typów, sprawdzanie zakresów tablic, wykrywanie prób użycia niezainicjalizowanych zmiennych i automatyczne zwalnianie pamięci (ang. garbage collection).
- Język ma umożliwiać tworzenie aplikacji działających w środowisku rozproszonym.
- Język ma zapewnić przenośność kodu jak i łatwość jego przyswojenia przez programistów znających C i C++.

Motywacja twórców (2)

- Język ma wspierać tworzenie aplikacji wielojęzycznych (ang. internationalization).
- Mimo że projektanci dołożyli starań aby aplikacje tworzone w C# działały szybko i nie używały zbyt wielu zasobów pamięciowych, to należy pamiętać, że język nie został wymyślony po to, by konkurować pod tym względem z C czy asemblerami.

4 Podstawy składni C#

Pierwszy program

```
/*
Przykładowe rozwiązanie zadania TEST
w systemie SPOJ
```

```

https://spoj.pl/
*/

using System;
public class Test
{
    public static void Main()
    {
        int n;
        while ((n = int.Parse(Console.ReadLine()))!=42)
            Console.WriteLine(n);
    }
}

```

Tokeny w C#

- Identyfikatory (Nazwy stałych, zmiennych, funkcji, klas), np: `Test`, `Main`,
- Słowa kluczowe, np: `public`, `static`
- Literały, np: `42`
- Operatory i separatory, np: `!=`, `{`, `}` `(`, `)`.

Identyfikatory

Reguły, stosują zalecenia: załącznika 15 do standardu Unicode (ang. Unicode Standard Annex 15). Dodatkowo można stosować znak podkreślenia (`_`) również na początku nazwy (jest traktowany jak litera), kody escape, dopuszczalny jest również znak `@`, jako prefiks dla słów kluczowych.

Operatory

- arytmetyczne: `+` `-` `*` `/` `%`
- logiczne: `&&`, `||`, `!`
- porównania: `<`, `<=`, `>`, `>=`, `==`, `!=`
- przypisania: `=`

- bitowe: & || ^ ~ << >>

Dyrektywy preprocesora

- #define i #undef definiowanie symboli dla warunkowej kompilacji
- #if, #elif, #else, and #endif - warunkowe włączanie i wyłączanie fragmentów kodu źródłowego
- #line - kontrola numeracji linii, może być użyteczne przy uruchamianiu
- #error i #warning Zgłoszenie błędu lub ostrzeżenia
- #region i #endregion, zaznaczanie fragmentów kodu
- #pragma informacja dla kompilatora

Typy całkowite

Alias C#	Typ .NET	bity	zakres
sbyte	System.SByte	8	-128 do 127
byte	System.Byte	8	0 do 255
short	System.Int16	16	-32768 do 32767
ushort	System.UInt16	16	0 do 65535
char	System.Char	16	znak unicode, 0 do 65,535
int	System.Int32	32	-2^{31} do $2^{31} - 1$
uint	System.UInt32	32	0 do $2^{32} - 1$
long	System.Int64	64	-2^{63} do $2^{63} - 1$
ulong	System.UInt64	64	0 do $2^{64} - 1$

Typy zmiennoprzecinkowe

Alias C#	Typ .NET	bity
float	System.Single	32
double	System.Double	64
decimal	System.Decimal	128

Inne typy predefiniowane

Alias C#	Typ .NET	bity
bool	System.Boolean	32
object	System.Object	32/64
string	System.String	16 * długość

Instrukcje warunkowe: **if**, **switch**

Przykład 1.

```
switch ( aircraft_ident ) {
    case "C-FESO" :
        Console.WriteLine("Rans_S6S_Coyote");
        break;
    case "C-GJIS" :
        Console.WriteLine("Rans_S12XL_Airaille");
        break;
    default :
        Console.WriteLine("Unknown aircraft");
        break;
}
```

Pętle

Pętle: **while**, **foreach**, **for**, **do...while**

Przykład 2.

```
public class ForEachSample {
    public void DoSomethingForEachItem ()
    {
        string [] itemsToWrite =
            {"Alpha", "Bravo", "Charlie"};
        foreach (string item in itemsToWrite)
            System.Console.WriteLine(item);
    }
}
```

Kontrola przepelnień

- checked(a) - sprawdza przepelnienie
- unchecked(a) - nie sprawdza

```
int square(int i){
    return i * i;
}
void f(){
    checked {
        int i = square(1000000);
    }
}
```

Odnosi się tylko do bieżącego bloku, nie dotyczy instrukcji wewnątrz funkcji square.

Wyliczenia

```
enum Weekday {Monday, Tuesday, Wednesday,
              Thursday, Friday, Saturday, Sunday};
```

Następnie można pisać jak niżej:

```
Weekday day = Weekday.Monday;
if (day == Weekday.Tuesday) {
    Console.WriteLine
    ("Time_sure_flies_by_when_you_program_in_C#!");
}
```

Wyliczenie z innym typem bazowym:

```
enum CardSuit : byte { Hearts,
                      Diamonds, Spades, Clubs };
```

Wyliczenie z innymi wartościami stałych:

```
enum Age { Infant = 0, Teenager = 13, Adult = 18 };
```

5 Wyliczenia

Wyliczenia `enum` (szczegóły)

Deklaracja ma postać:

```
[atrybuty] [modyfikatory] enum identyfikator
    [:typ_bazowy] {lista_elementów};
```

Wyliczenie jest typem, w którego skład wchodzi elementy odziedziczone po typie `System.Enum`

```
public abstract class Enum :
    ValueType, IComparable, IFormattable, IConvertible
```

oraz zdefiniowane elementy. Wyliczenie ma swój typ bazowy (ang underlying type); typem bazowym wyliczenia może być: `byte`, `sbyte`, `short`, `ushort`, `int` (domyślnie), `uint`, `long` lub `ulong`

Dla typów wyliczeniowych predefiniowane są operatory: `==`, `!=`, `>`, `<`, `>=`, `<=`.

Przykład 3. Wyliczenia przykład:

```
class Program {
    private enum Cos : byte {Jeden, Dwa = 2,
                            Trzy, Cztery};
    static void Main(string[] args) {
        Cos x = Cos.Jeden;
        Console.WriteLine(x < Cos.Cztery); // True
        Console.WriteLine(x); // jeden
        Console.WriteLine((byte)x); //0
        x = Cos.Dwa;
        Console.WriteLine((byte)x); //2
        Console.WriteLine(x.GetType()); //Program+Cos
    }
}
```

6 Switch

Instrukcja warunkowa: `switch` (szczegóły)

```
switch (<expression>) <switch block>
```

Możliwe typy dla `<expression>`:

sbyte, byte, short, ushort,
int, uint, long, ulong,
char, string

Można również zastosować typ wyliczeniowy lub inny typ o ile zdefiniowano wcześniej operator konwersji `implicit` do jednego z powyższych.

Switch - cechy

- Programista jest chroniony przed omyłkowym pominięciem `break`
- Obsługę kolejnych przypadków można przedstawiać w kodzie
- Możliwe jest użycie `goto` (niezalecane), co w niektórych wypadkach może uprościć kod.
- możliwa jest obsługa kilku przypadków w jednym bloku
- Musi być możliwa niejawna konwersja z każdego wyrażenia stałego po słowie `case` do typu wyrażenia po `switch`
- Jeśli powtórzy się wyrażenie stałe o tej samej wartości, to wystąpi błąd kompilacji.
- Jeśli wyrażenie stałe nie jest jawnie zadeklarowane jako `unchecked`, to przepełnienie (ang. overflow) przy jego obliczaniu spowoduje błąd kompilacji.

Przykład 4.

```
class Program {  
    private enum Cos : byte { Jeden, Dwa = 2,  
                             Trzy, Cztery };  
    static void Main(string[] args) {  
        Cos x = Cos.Dwa;  
        switch (x){  
            case Cos.Jeden:  
                Console.WriteLine(x.ToString() + "_raz");  
                break;  
            default:  

```

```

        Console.WriteLine(x.ToString() + " _razy");
        break;
    }
}
}

```

Przykład 5.

```

using System;
public struct Digit{
    byte value;
    public Digit(int value) {
        if (value < 0 || value > 9)
            throw new ArgumentException();
        this.value = value;
    }
    public static implicit operator byte(Digit d) {
        return d.value; // no exceptions
    }
    public static explicit operator Digit(int n) {
        return new Digit(n); // possible exception
    }
} // end struct

```

switch i break

Jeśli koniec kodu bieżącego przypadku „może być osiągalny” wystąpi błąd kompilacji. Kolejne przykłady są poprawne:

Przykład 6.

```

switch (i) {
case 0:
    CaseZero();
    goto case 1;
case 1:
    CaseZeroOrOne();
    goto default;
default:

```

```

    CaseAny ();
    break;
}

```

Zamiast `break` i `goto` można użyć innych konstrukcji, które nie dopuszczają przeniesienia sterowania na koniec obsługi danego przypadku:

Przykład 7.

```

switch (i) {
    case 0:
        while (true) F();
    case 1:
        throw new ArgumentException ();
    case 2:
        return;
}

```

Dozwolone jest użycie typu `string`

Przykład 8.

```

void DoCommand(string command) {
    switch (command.ToLower()) {
        case "run":
            DoRun();
            break;
        case "save":
            DoSave();
            break;
        case "quit":
            DoQuit();
            break;
        default:
            InvalidCommand(command);
            break;
    }
}

```

Wielokrotne etykiety w obsłudze pojedynczego przypadku są dozwolone:

```

switch (i) {
    case 0:

```

```

        CaseZero ();
        break;
    case 1:
        CaseOne ();
        break;
    case 2:
    default:
        CaseTwo ();
        break;
}

```

switch i break

Dzięki podanym regułom można dowolnie przestawiać kolejność przypadków w obrębie pojedynczej instrukcji `switch`

Przykład 9.

```

switch (i) {
    default:
        CaseAny ();
        break;
    case 1:
        CaseZeroOrOne ();
        goto default;
    case 0:
        CaseZero ();
        goto case 1;
}

```

Instrukcje iteracyjne

`for`, `while`, `do-while`, `foreach-in`

Przykład 10.

```

for (int i = 1; i <= 5; i++) {
    Console.WriteLine(i);
}

```

Słowa kluczowe `break`, `continue`.

Pętla foreach

Przykład 11.

```
static void Main(string [] args) {  
    foreach (int i in args) {  
        System.Console.WriteLine(i);  
    }  
}
```

7 Typy referencyjne i niereferencyjne

- Typy niereferencyjne (ang. value types): bool, byte, char, decimal, double, enum, float, int, long, sbyte, short, struct, uint, ulong, ushort.
- Typy referencyjne: class, delegate, interface, object, string.

Typy niereferencyjne

Typy niereferencyjne pochodzą od `ValueType`.

```
[SerializableAttribute]  
[ComVisibleAttribute(true)]  
public abstract class ValueType
```

`ValueType` dziedziczy po `Object`.

Typ decimal

- 128 bitów, 28 cyfr (dziesiętnie) znaczących.
- typ .NET `System.Decimal`
- literały z literą `m` lub `M`, przykład: `300.5m`

Niejawne konwersje typów

sbyte	short, int, long, float, double, or decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
char	ushort, int, uint, long, ulong, float, double, decimal
float	double
ulong	float, double, decimal

8 Partial i sealed

Słowo kluczowe `partial` oznacza iż definicja klasy w danym pliku z kodem źródłowym jest niepełna.

Przykład 12.

```
//kod wygenerowany automatycznie plik1.cs
partial class Widget{
    private void InitializeComponent ()...
    ...
}
```

```
//kod tworzony przez programistę plik2.cs
partial class Widget{
    private int value;
    public int Process(object obj) {
        ...
    }
}
```

Klasy zapieczętowane

Klasa oznaczona modyfikatorem `sealed` nie może być dziedziczona.

W związku z tym klasy abstrakcyjne nie mogą być `sealed`.

Klasy statyczne (opatrzone modyfikatorem `static` nie mogą być, ani `sealed`, ani `abstract`, gdyż i tak muszą zachowywać się w taki sposób.

Klasa `string` jest `sealed`.

9 Klasy i struktury

Struktury są typami przekazywanymi przez wartość - ich wartości są przechowywane na stosie, nie można po nich dziedziczyć. Ale posiadają wiele cech klas.

Przykład 13.

```
struct Person{
    string name;
    System.DateTime birthDate;
    int heightInCm;
    int weightInKg;
}
...

Person dana = new Person ();
//użycie new zainicjalizuje zmienne
dana.name = "Dana_Developer";
```

Konstruktory struktur

Język dostarcza dla struktur bezparametrowych konstruktorów domyślnych. Można również dodawać własne:

Przykład 14.

```
struct Point{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Struktury i interfejsy

Struktury nie mogą dziedziczyć, mogą jednak implementować interfejsy.

Przykład 15.

```
using System;
interface IInterface{
    void Method();
}
struct MyStruct : IInterface{
    public void Method(){
        Console.WriteLine("Struct _Method");
    }
}
class MyClient{
    public static void Main(){
        MyStruct s = new MyStruct();
        s.Method();
    }
}
```

10 Tablice w C#

- Indeksowane od zera
- Elementy mogą być dowolnego typu
- Tablice są typami referencyjnymi
- operator `new` tworzy tablicę i nadaje elementom wartości inicjalne.

Tablice jednowymiarowe

Deklaracja:

```
string [] myStringArray;
```

Deklaracja i utworzenie:

```
string [] myStringArray = new string [6];
```

Deklaracja, utworzenie i inicjalizacja niestandardowymi wartościami:

```
int [] myArray = new int [] {1, 3, 5, 7, 9};
```

lub

```
int [] myArray = {1, 3, 5, 7, 9};
```

Tablice wielowymiarowe

Zwykła prostokątna tablica:

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

i tablica tablic (ang. jagged array)

```
int [][] myJaggedArray = new int [3][];
```

Deklaracja, utworzenie i inicjalizacja niestandardowymi wartościami:

```
int [][] myJaggedArray = {  
    new int [] {1,3,5,7,9},  
    new int [] {0,2,4,6},  
    new int [] {11,22}  
};
```

11 Operatory w C#

- arytmetyczne: +, -, *, /, %, ++, --;
- logiczne i bitowe: &, |, ^, !, ~, &&, ||, true, false, <<, >>;
- konkatencji: +;
- Relacyjne: ==, !=, <, >, <=, >=;
- przypisania: =, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=;
- dostępu: .; indeksu: []; rzutowania: (); warunkowy: ?::;
- dodawania i usuwania delegatów +, -;
- tworzenie obiektu: new;

- informacji o typie: `as`, `is`, `sizeof`, `typeof`;
- kontroli przepełnień: `checked`, `unchecked`;
- adresowania: `*`, `->`, `&`;

Operator `is`

Użycie `is` sprawdza, czy obiekt jest kompatybilny z danym typem, tzn. czy rzutowanie nie zgłasza wyjątków.

Uwaga: `is` nie bierze pod uwagę konwersji definiowanych przez użytkownika.

Uwaga: operator `is` nie może być przeciążony

Przykład 16.

```
if (obj is string){
}
```

Operator `is` – składnia

```
expression is type
```

gdzie `expression` to wyrażenie typu referencyjnego

Wynikiem jest `true` jeśli: `expression` jest różne od `null` lub rzutowanie do typu `type` nie powoduje zgłoszenia wyjątku.

Operator `as`

```
expr as type
```

jest równoważne

```
expr is type ? (type)expr : (type)null
```

```
string s = someObject as string;
```

```
if (s != null){
    // someObject is a string.
}
```

Przykład 17.

```
using System;
class Class1 {}
class MainClass {
    static void Main() {
        object [] objArray = new object [4];
        objArray [0] = new Class1 ();
        objArray [1] = "hello";
        objArray [2] = 123.4;
        objArray [3] = null;
        for (int i = 0; i < objArray.Length; ++i) {
            string s = objArray [i] as string;
            Console.WriteLine (" {0}: ", i);
            if (s != null)
                Console.WriteLine (" " + s + " ");
            else
                Console.WriteLine ("not_a_string");
        } } }
/* 0: not a string
   1: 'hello '
   2: not a string
   3: not a string */
```

Operatory adresowania (wskaźnikowe) i unsafe

Przykład 18.

```
// compile with: /unsafe
using System;
class UnsafeTest {
    // unsafe method: takes pointer to int:
    unsafe static void SquarePtrParam (int* p) {
        *p *= *p;
    }
    unsafe public static void Main() {
        int i = 5;
        // unsafe method: uses address-of operator (ℰ)
```

```
        SquarePtrParam (&i);  
        Console.WriteLine (i);  
    }  
}
```

12 Modyfikatory dostępu

- `private` – dostęp wyłącznie z klasy zawierającej
- `public` – bez ograniczeń dostępu
- `protected` – dostęp wyłącznie z klasy zawierającej i jej klas potomnych
- `internal` – dostęp wyłącznie w obrębie pakietu (the same assembly)
- `protected internal` – dostęp w obrębie pakietu lub w klasie potomnej

Dopuszczalność stosowania modyfikatorów

- Za wyjątkiem kombinacji `protected internal` nie można łączyć modyfikatorów.
- Typy niezagnieżdżone (ang. top-level types) mogą być jedynie `internal` lub `public`; domyślnie `internal`.
- elementy wyliczenia (`enum`) domyślnie są `public`; nie dopuszcza się stosowania modyfikatorów.
- elementy klasy (`class`) domyślnie są `private`; dopuszcza się stosowanie wszystkich modyfikatorów.
- elementy struktury (`struct`) domyślnie są `private`; dopuszcza się stosowanie modyfikatorów: `public`, `internal` i `private`.
- elementy interfejsu (`interface`) domyślnie są `public`; nie dopuszcza się stosowania modyfikatorów.
- Nie stosuje się modyfikatorów dostępu do przestrzeni nazw (`namespace`)

13 Metody i parametry

- ref
- out
- params

Przykład użycia ref

Przykład 19.

```
class OutExample{
    static void Method(out int i){
        i = 42;
    }
    static void Main(){
        int value;
        Method(out value);
        // value = 42
    }
}
```

Przykład użycia out

Przykład 20.

```
class RefExample{
    static void Method(ref int i){
        i = 42;
    }
    static void Main(){
        int val = 0;
        //parametr musi być zainicjalizowany
        Method(ref val);
        // val = 42
    }
}
```

Przykład użycia params

Przykład 21.

```
public class MyClass {
    public static void UseParams(params object[] list) {
        for ( int i = 0 ; i < list.Length ; i++ )
            Console.WriteLine(list[i]);
        Console.WriteLine();
    }
    public static void Main() {
        UseParams(1, 'a', "test");
    }
}
```

- params może wystąpić tylko jeden raz.
- params może wystąpić tylko na końcu listy argumentów.

Opakowanie (ang. boxing)

Przykład 22.

```
using System;
```

```
class OutExample{
    static void Method(object i){
        i = 4;
    }
    static void Main(){
        int value = 0;
        Method(value);
        Console.Write(value);
        // 0
    }
}

.method private hidebysig
    static void Method(object i) cil managed
```

```

{
  // Code size      10 (0xa)
  .maxstack 8
  IL_0000: nop
  IL_0001: ldc.i4.4
  IL_0002: box      [mscorlib]System.Int32
  IL_0007: starg.s  i
  IL_0009: ret
} // end of method OutExample::Method

```

Opakowanie c.d.

Przykład 23.

```

using System;

class OutExample{
  static void Method(ref object i){
    i = 4;
  }
  static void Main(){
    int value = 0;
    object x = value;
    Method(ref x);
    value = (int)x;
    Console.WriteLine(value);
    // 4
  }
}

```

Uwaga: rzutowanie nie jest l-wartością – nie można napisać: `Method(ref (Object)value)`

Przeciążanie metod

Możliwe jest przeciążanie metod różniących się sposobem przekazywania argumentu.

```

class RefOutOverloadExample{

```

```

    public void SampleMethod(int i) { }
    public void SampleMethod(ref int i) { }
}

```

Nie mogą to być jednak `ref` i `out`.

```

class CS0663_Example {
    public void SampleMethod(ref int i) { }
    public void SampleMethod(out int i) { }
}

```

Powyższy kod spowoduje błąd kompilacji:

```

compiler error CS0663: "cannot define overloaded
methods that differ only on ref and out"

```

14 Properties

Przykład 24.

```

class TimePeriod {
    private double seconds;
    public double Hours {
        get { return seconds / 3600; }
        set { seconds = value * 3600; }
    }
}

class Program{
    static void Main() {
        TimePeriod t = new TimePeriod();
        t.Hours = 24;
        System.Console.WriteLine("Time in hours: \r"
                                + t.Hours);
    }
}

```

Ograniczanie widzialności punktów dostępu (properties)

Przykład 25.

```
public string Name{
    get{return name;}
    protected set{name = value;}
}
```

Uwaga: jeśli nie zdefiniowano metody `set`, to właściwość jest tylko do odczytu.

15 Indeksatory (ang. indexers)

Przykład 26.

```
class Coll<T>{
    private T[] arr = new T[100];
    public T this[int i]{
        get{return arr[i];}
        set{arr[i] = value;}
    }
}
class Program{
    static void Main(string[] args){
        Coll<string> strColl = new Coll<string>();
        strColl[0] = "Hello ,_World";
        System.Console.WriteLine(strColl[0]);
    }
}
```

Słowo kluczowe `this` jest potrzebne w definicji indeksatora.

Słowo kluczowe `value`, podobnie jak w metodzie `set` dla właściwości, jest podstawianą wartością.

Indeksatory - przykład

Przykład 27.

```

public class YearClass{
    const int StartDate = 1900;
    const int EndDate = 2050;
    int [] arr = new int[EndDate - StartDate + 1];

    public int this[int num]{
        get { return arr[ num - StartDate]; }
        set { arr[num - StartDate] = value; }
    }
}

public class Test{
    public static void Main(){
        YearClass yc = new YearClass();
        yc[1950] = 5;
    }
}

```

Indeksowanie “nie liczbami”

Przykład 28.

```

class DayCollection{
    string [] days =
    {"Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };
    private int GetDay(string testDay){
        int i = 0;
        foreach (string day in days){
            if (day == testDay) return i;
            i++;
        }
        return -1;
    }
    public int this[string day]{
        get{return (GetDay(day));}
    }
}

class Program{

```

```

static void Main(string [] args){
    DayCollection week = new DayCollection ();
    System.Console.WriteLine(week [ " Fri" ] );
    System.Console.WriteLine(week [ "Made-up-Day" ] );
}
}

```

Własności indeksatorów

- Indeksator może mieć więcej niż jeden parametr
- Indeksatory można przeciążać

16 Dziedziczenie

Konieczność użycia: `virtual` i `override`

```

public class Employee{
    public virtual void GetPayCheck(){
    }
    public void Work(){
    }
    ...
}
public class Executive : Employee{
    public override void GetPayCheck(){
        //...
    }
    public void AdministerEmployee()
    {
        //...
    }
}

```

Cechy dziedziczenia

- W C# nie ma dziedziczenia mnogiego

- Dopuszczalna jest implementacja wielu interfejsów
- Nie ma znanego z C++ dziedziczenia publicznego, chronionego i prywatnego.

Użycie override

Metoda bazowa nie może być `sealed`, `static`

Metoda bazowa musi być `virtual`, `override` lub `abstract`.

Metoda bazowa o tej samej sygnaturze musi istnieć (typ zwracany, lista parametrów i specyfikatory dostępu muszą być takie same).

17 Użycia new

Modyfikatora `new` używamy aby zaznaczyć, że przesłonięcie (ang. hide) pola, metody jest zamierzone. Nie umieszczenie `new` powoduje zgłoszenie przez kompilator ostrzeżenia (ang. warning).

Przykład 29.

```
using System;
public class BaseC {
    public static int x = 55;
    public static int y = 22;
}
public class DerivedC : BaseC {
    new public static int x = 100;
    static void Main() {
        Console.WriteLine(x);
        Console.WriteLine(BaseC.x);
        Console.WriteLine(y);
    }
}

using System;
public class BaseC {
    public class NestedC {
        public int x = 200;
        public int y;
```

```

    }
}
public class DerivedC : BaseC{
    new public class NestedC{
        public int x = 100;
        public int y;
        public int z;
    }
    static void Main() {
        NestedC c1 = new NestedC ();
        BaseC.NestedC c2 = new BaseC.NestedC ();
        Console.WriteLine(c1.x); //100
        Console.WriteLine(c2.x); //200
    }
}

using System;
public class Base{
    public void B(){ C(); }
    public void C(){
        System.Console.WriteLine("Base.C()");
    }
}
public class Derived : Base {
    new public void C(){
        System.Console.WriteLine("Derived.C()");
    }
}
class Test
{
    static void F() {
        System.Console.WriteLine("Test.F");
    }
    static void Main() {
        Derived d = new Derived ();
        d.B(); // Base.C
        d.C(); // Derived.C
    }
}

```

```
}
```

18 Elementy Programowania generycznego

Metody generyczne

```
static void Swap<T>(ref T lhs , ref T rhs){  
    T temp;  
    temp = lhs;  
    lhs = rhs;  
    rhs = temp;  
}
```

Przykład 30.

```
public static void TestSwap(){  
    int a = 1;  
    int b = 2;  
  
    Swap<int>(ref a, ref b);  
    System.Console.WriteLine(a + " " + b);  
}
```

Można również:

```
Swap(ref a, ref b);
```

Przykład 31.

```
void SwapIfGreater<T>(ref T lhs , ref T rhs)  
    where T : System.IComparable<T>{  
    T temp;  
    if (lhs.CompareTo(rhs) > 0){  
        temp = lhs;  
        lhs = rhs;  
        rhs = temp;  
    }  
}
```

Każda jednowymiarowa tablica automatycznie implementuje IList.

```

class Program
{
    static void Main(){
        int [] arr = { 0, 1, 2, 3, 4 };
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
            list.Add(x);
        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }
    static void ProcessItems<T>(IList<T> coll){
        foreach (T item in coll){
            System.Console.Write(item.ToString() + " ");
        }
        System.Console.WriteLine();
    }
}

```

Słowo kluczowe default

Przykład 32.

```

public class GenericList<T>{
    private class Node{
        //...
        public Node Next;
        public T Data;
    }
    private Node head;
    //...
    public T GetNext(){
        T temp = default(T);
        Node current = head;
        if (current != null){
            temp = current.Data;
            current = current.Next;
        }
    }
}

```

```

        return temp;
    }
}

```

Nullable Types

Przykład 33.

```

class NullableExample{
    static void Main(){
        int? num = null;
        if (num.HasValue == true)
            System.Console.WriteLine("num==_" + num.Value);
        else
            System.Console.WriteLine("num==_Null");
        //y is set to zero
        int y = num.GetValueOrDefault();
        // num.Value throws an InvalidOperationException
        // if num.HasValue is false
        try{ y = num.Value; }
        catch (System.InvalidOperationException e){
            System.Console.WriteLine(e.Message);
        }
    }
}

```

Składnia T? jest krótszą wersją System.Nullable<T>.

Przykłady typów nullable

```

int? i = 10;
double? d1 = 3.14;
bool? flag = null;
char? letter = 'a';
int?[] arr = new int?[10];

```

19 Delegaty (ang. Delegates)

Przykład 34.

```
delegate void SimpleDelegate ();
class Test{
    static void F() {
        System.Console.WriteLine("Test.F");
    }
    static void Main() {
        SimpleDelegate d = new SimpleDelegate(F);
        d();
    }
}
```

Delegaty – deklaracja

```
[atrybuty] [modyfikatory]
    delegate typ-wyniku nazwa ([parametry]);
```

- dopuszczalne modyfikatory to `new` i modyfikatory dostępu.
- dalej jak w deklaracji metody

Delegaty – cechy

- Delegaty można traktować jak bezpieczne wskaźniki do funkcji.
- Sygnatura podstawianej funkcji musi być taka jak deklaracja delegata.
- Podstawiać można zarówno metody statyczne jak i niestacyjne (ang. instance methods)
- Delegaty są typami referencyjnymi
- Obiekty typów delegowanych nie mogą być zmieniane (są ang. immutable)

```

delegate void Procedure ();

class DelegateDemo{
    static void Meth1(){
        Console.WriteLine("Method_1");
    }
    static void Meth2(){
        Console.WriteLine("Method_2");
    }
    void Meth3(){
        Console.WriteLine("Method_3");
    }
    static void Main(){
        Procedure procs = null;
        procs += new Procedure(DelegateDemo.Meth1);
        procs += new Procedure(DelegateDemo.Meth2);
        DelegateDemo demo = new DelegateDemo();
        procs += new Procedure(demo.Meth3);
        procs();
    }
}

```

Delegaty – łączenie

- Delegaty można łączyć – podstawić kilka metod pod to samo wystąpienie
- Połączone metody wykonywane są jedna po drugiej w zadanej kolejności
- Jeśli parametr był `ref` i uległ zmianie, to taki zmieniony parametr jest przekazywany do kolejnej metody
- Jeśli parametr był `out` (wbrew [http://msdn2.microsoft.com/en-us/library/900fyy8e\(VS.71\).aspx](http://msdn2.microsoft.com/en-us/library/900fyy8e(VS.71).aspx) może tak być), to otrzymamy wartość z ostatniego wywołania, podobnie z wartością zwracaną (ang. return value).

- Jedna metoda może być dodana kilkakrotnie, wtedy usunięcie jej powoduje usunięcie ostatniego wystąpienia.
- Próba usunięcia nieistniejącej metody nie powoduje błędu.

Delegaty a interfejsy

Delegaty i interfejsy są podobne w tym sensie, że separują deklarację od implementacji.

Interfejsy są użyteczne, gdy:

- definiujemy zbiór powiązanych ze sobą metod
- typowo wewnątrz klasy potrzebujemy jednej implementacji każdej z metod
- spodziewamy się użyteczności tworzenia interfejsów i klas pochodnych

Delegaty a interfejsy

Delegaty są użyteczne, gdy:

- potrzebujemy kilku implementacji danej metody wewnątrz klasy
- dla obiektu wywołującego nie jest istotne, gdzie (w jakiej klasie) metoda została zdefiniowana.
- ma sens podstawianie metod statycznych

20 Events

Events - mogą być umieszczane wewnątrz klas. Bez względu na specyfikator dostępu wywołanie może nastąpić tylko z klasy zawierającej.

Przykład 35.

```
delegate void ButtonClickedHandler ();
```

```
class Button{
    public event ButtonClickedHandler ButtonClicked;
```

```

public void SimulateClick(){
    if (ButtonClicked != null){
        ButtonClicked();
    }
}
...
}

```

Przykład 36.

```

delegate bool Action(Node n);
static void Walk(Node n, Action a) {
    while (n != null && a(n)) n = n.Next;
}
//Wywołanie Walk używa metody anonimowej,
//co upraszcza kod:
Walk(list,
    delegate(Node n) {
        Console.WriteLine(n.Name);
        return true;
    }
);

```

Metody anonimowe mają dostęp do tych samych zmiennych lokalnych co funkcja zawierająca.

21 yield

Przykład 37.

```

using System;
using System.Collections;
public class List {
    public static IEnumerable Power(int number, int exponent) {
        int counter = 0;
        int result = 1;
        while (counter++ < exponent) {
            result = result * number;
            yield return result;
        }
    }
}

```

```

    }
}

static void Main(){
    foreach (int i in Power(2, 8)) {
        Console.WriteLine("{0}_", i);
        //output : 2 4 8 16 32 64 128 256
    }
}
}

```

yield

Postać:

```

yield return <expression>;
lub
yield break;

```

Słowo kluczowe `yield`:

- Może wystąpić wewnątrz metody, w bloku instrukcji iteracyjnej.
- Metoda nie może mieć parametrów `ref` i `out`.
- Metoda nie może zawierać bloku `unsafe`.
- Nie może wystąpić w bloku `finally`.
- Nie może się pojawić w metodach anonimowych.

yield

Przykład 38.

```

using System.Collections.Generic;
using System;

```

```

public class YieldSample {
    static IEnumerable<DateTime> GenerateTimes() {

```

```

    DateTime limit = DateTime.Now +
                    new TimeSpan(0,0,12);
    while (DateTime.Now < limit)
        yield return DateTime.Now;
    yield break;
}
static void Main() {
    foreach (DateTime d in GenerateTimes()){
        System.Console.WriteLine(d);
        System.Console.Read();
    }
}
}

```

22 Interface IComparable

Przykład 39.

```

class Employee:IComparable
{
    private int Id;
    private string Name;
    public Employee(int id ,string name){
        this.Id=id;
        this.Name=name;
    }
    public int CompareTo(object obj){
        Employee temp=(Employee)obj;
        if(this.Id>temp.Id) return 1;
        else{
            if(temp.Id==this.Id) return 0;
            else return -1;
        }
    }
}

public static void Main(){
    Employee [] employees=new Employee [5];
}

```

```

    Console.WriteLine("Before_Sort:");
    for (int i=0;i<employees.Length;i++){
        employees[i]=new Employee(5-i,"Employee#" + i);
        Console.Write(employees[i].Id + ",");
    }
    Console.WriteLine();

    Array.Sort(employees);
    Console.WriteLine("After_Sort:");
    for (int i=0;i<employees.Length;i++){
        Console.Write(employees[i].Id + ",");
    }
    Console.WriteLine();
}
} // class Employee

```

Przykład 40.

```

//.Net Framework 1.1
System.Collections.ArrayList list
    = new System.Collections.ArrayList();
// Add an integer to the list.
list.Add(3);
// Add a string to the list.
// This will compile, but may cause an error later.
list.Add("It_is_raining_in_Redmond.");

int t = 0;
// This causes an InvalidCastException to be returned.
foreach (int x in list) t += x;

//.NET Framework 2.0
List<int> list1 = new List<int>();

// No boxing, no casting:
list1.Add(3);

// Compile-time error:
// list1.Add("It is raining in Redmond.");

```

Delegacje generyczne

```
public delegate void Del<T>(T item);  
public static void Notify(int i) { }
```

```
Del<int> m1 = new Del<int>(Notify);  
// lub krócej: Del<int> m2 = Notify;
```

Przykład 41.

```
using System;  
using System.Collections.Generic;  
  
private static void  
    Add<T,S>(Dictionary<T, List<S>> values, T key, S value) {  
    if (!values.ContainsKey(key)) {  
        List<S> l = new List<S>();  
        l.Add(value);  
        values.Add(key, l);  
    }  
    else values[key].Add(value);  
}  
  
public static class Program {  
    public static void Main() {  
        Dictionary<String, List<String>> values =  
            new Dictionary<String, List<String>>();  
        Add(values, "Gry_Planszowe", "Warcaby");  
        Add(values, "Maskotki", "Kłapouchy");  
        Add(values, "Maskotki", "Tygrysek");  
        Add(values, "Gry_Planszowe", "Chińczyk");  
        foreach (List<String> list in values.Values) {  
            foreach (String elem in list)  
                Console.Write(elem + ", ");  
            Console.WriteLine();  
        }  
    }  
}
```

23 Wątki i słowo kluczowe Lock

Tworzymy nowy wątek.:

```
System.Threading.Thread newThread;  
newThread = new System.Threading.Thread(anObject.AMethod);
```

Uruchamiamy go:

```
newThread.Start();
```

Lock

Przykład 42.

```
public void Function(){  
    System.Object lockThis = new System.Object();  
    lock(lockThis)  
    {  
        // Blok kodu dostępny jednocześnie  
        // tylko dla jednego wątku.  
    }  
}
```

24 Łańcuchy znaków

Typ `string` przechowuje ciągi znaków. Przykłady deklaracji:

```
string hello = "Hello\nWorld!";  
char letter = hello[1]; // 'e';  
string a = "\u0068ello_"; // hello  
string path = @"c:\Docs\Source\a.txt"  
//zamiast "c:\\Docs\\Source\\a.txt"
```

Przykład 43.

```
using System;  
  
class StringExample {  
    static void Method(String s){
```

```

    s+=" world";
    Console.WriteLine(s);
    //hello world
}
static void Main(){
    string s="hello";
    Method(s);
    Console.WriteLine(s);
    //hello
}
}

```

- Obiektu klasy `string` nie można zmienić - operatory podstawienia i konkatenacji tworzą nowe obiekty.
- `string` jest inną nazwą dla `String`.
- Pomimo że `string` jest typem referencyjnym, to operatory `==` i `!=` dotyczą wartości, a nie referencji.

StringBuilder

Klasa `StringBuilder` jest zaimplementowana jako rozszerzalna tablica. Dołączanie znaków na koniec łańcucha powinno działać w zamortyzowanym czasie stałym.

Przykład 44.

```

using System;
using System.Text;

public sealed class App {
    static void Main() {
        StringBuilder sb = new StringBuilder("ABC", 50);
        //początkowy rozmiar 50 znaków.
        sb.Append(new char[] { 'D', 'E', 'F' });
        Console.WriteLine("{0} chars: {1}",
            sb.Length, sb.ToString());
        // 6 chars: ABCDEF
    }
}

```

```
}  
}
```

Wyrażenia regularne

Przykład 45.

```
class TestRegularExpressionValidation {  
    static void Main() {  
        string [] numbers = {  
            "123-456-7890",  
            "444-234-22450",  
            "690-203-6578",  
            "146-893-232",  
            "146-839-2322",  
            "4007-295-1111",  
        };  
        string sPattern = "^\\d{3}-\\d{3}-\\d{4}$";  
        foreach (string s in numbers) {  
            System.Console.Write("{0,14}", s);  
            if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern))  
                System.Console.WriteLine(" _ _ valid");  
            else  
                System.Console.WriteLine(" _ _ invalid");  
        }  
    }  
}
```

25 Generowanie dokumentacji

Przykład 46.

```
/// <summary>  
/// Odpowiedź na pytanie o istotę wszechświata.  
/// </summary>  
public class MyClass{  
    static void Main(){}
```

```

}
    /// - oznacza przetwarzany fragment komentarza. Do oznaczenia wielu
linii można stosować: /** ... */
    Po kompilacji z opcją /doc otrzymamy plik:
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>xmlprb</name>
    </assembly>
    <members>
        <member name="T:MyClass">
            <summary>
                Odpowiedź na pytanie o istotę wszechświata.
            </summary>
        </member>
    </members>
</doc>

```

Wybrane, rekomendowane znaczniki

- `<c>text</c>` – wiersz tekst oznaczony jako kod.
- `<code>tekst ... tekst</code>` – jak wyżej, ale więcej linii kodu.
- `<para>` Paragraf tekstu `</para>`.
- `<list>` – definiuje listę.
- `<include>` – pozwala dołączyć plik tekstowy.
- `<summary>` – przeznaczony do opisu typu lub jego elementów.
- `<param name='nazwa'>opis parametru </param>`
- `<returns>opis wartości zwracanej</returns>`

26 Atrybuty

Atrybuty pozwalają skojarzyć z fragmentami kodu informacje, które można potem wykorzystać w fazie wykonania.

Przykład 47.

```
[System.Serializable]
public class SampleClass {
    // Obiekty tego typu mają nadany
    // atrybut System.Serializable
}
```

Przykład 48.

```
//ten atrybut można stosować tylko do klas i struktur
[System.AttributeUsage(System.AttributeTargets.Class |
                        System.AttributeTargets.Struct)
]
public class Author : System.Attribute {
    private string name;
    public double version;
    public Author(string name) {
        this.name = name;
        version = 1.0;
    }
}
```

Tak zdefiniowany atrybut można teraz użyć w następujący sposób:

Przykład 49.

```
[Author("H. Ackerman", version = 1.1)]
class SampleClass {
    // H. Ackerman's code goes here...
}
```

Powszechnie używane atrybuty

- `Obsolete` - definiuje fragment kodu jako niezalecany, jego użycie powoduje ostrzeżenie przy kompilacji, można zażądać, błędu.

- `Conditional` umożliwia włączenie oznaczonej metody tylko pod warunkiem zadeklarowania pewnej stałej (można używać podobnie jak `#ifdef`, `#endif`).

Przykład 50.

```
using System;
[AttributeUsage(AttributeTargets.Class |
                AttributeTargets.Struct,
                AllowMultiple = true)
 // multiuse attribute
]
public class Author : Attribute {
    private string name;
    public double version;
    public Author(string name) {
        this.name = name;
        version = 1.0;
    }
}
```

Teraz można:

```
[Author("H. Ackerman", version = 1.1)]
[Author("M. Knott", version = 1.2)]
class SampleClass {
    // H. Ackerman's code goes here...
    // M. Knott's code goes here...
}
```

Atrybuty i refleksja

Jak uzyskać informację o autorze w fazie wykonania?

Przykład 51.

```
using System;
[Author("H. Ackerman")]
private class FirstClass {
    // ... }
}
```

```

// No Author attribute
private class SecondClass {
    // ... }

[Author("H. Ackerman"),
 Author("M. Knott", version = 2.0)]
private class ThirdClass {
    // ... }

class TestAuthorAttribute {
    static void Main() {
        AuthorInfo(typeof(FirstClass));
        AuthorInfo(typeof(SecondClass));
        AuthorInfo(typeof(ThirdClass));
    }
    private static void AuthorInfo(Type t) {
        Console.WriteLine("Author information for {0}", t);
        Attribute[] attrs =
            Attribute.GetCustomAttributes(t); // reflection

        foreach (Attribute attr in attrs) {
            if (attr is Author) {
                Author a = (Author)attr;
                Console.WriteLine("{0}, version {1:f}",
                    a.GetName(), a.version);
            }
        }
    }
}

```

27 Refleksje

Możliwości:

- Informacja o typie obiektu w trakcie wykonania
- Dostęp do atrybutów
- Tworzenie nowych typów w trakcie wykonania

Dynamiczna informacja o typie

Przykład 52.

```
using System;

class Testowa{
    static void Main(){
        int i = 42;
        Type type = i.GetType();
        Console.WriteLine(type.ToString());
    } //System.Int32
}
```

Utworzenie obiektu zadanego typu

Przykład 53.

```
using System;

class Testowa{
    static void Main(){
        int i = 42;
        Type type = i.GetType();
        //otworzenie obiektu konstruktorem domyślnym
        object obj = Activator.CreateInstance(type);
        Console.WriteLine(obj.GetType());
    } //System.Int32
}
```

Wywołanie metody o zadanej nazwie

Przykład 54.

```
using System;
using System.Reflection;

class Testowa{
```

```

static void Main(){
    int i = 42;
    Type type = i.GetType();
    object obj = Activator.CreateInstance(type);
    Console.WriteLine(obj.GetHashCode());
    object [] Params = null;
    string methodName = "GetHashCode";
    int res = (int)type.InvokeMember(methodName,
        BindingFlags.InvokeMethod,
        null, obj, Params);
    Console.WriteLine(res);
} //
}

```

28 Instrukcja using

Instrukcja `using` określa w jakim momencie obiekt używający pewnych zasobów powinien je zwolnić.

Przykład 55.

```

using (Font font1 = new Font("Arial", 10.0f)){
    // use font1 }
}

```

lub

```

Font font2 = new Font("Arial", 10.0f);
using (font2){
    // use font2 }
}

```

Przykład 56.

```

using System;

class C : IDisposable{
    public void UseLimitedResource(){
        Console.WriteLine("Using_limited_resource...");
    }
    void IDisposable.Dispose(){

```

```

        Console.WriteLine("Disposing_limited_resource.");
    }
}
class Program{
    static void Main(){
        using (C c = new C()){
            c.UseLimitedResource();
        }
        Console.WriteLine("Now_outside_using_statement.");
        Console.ReadLine();
    }
}

```

Dyrektywa using

```

using namespace;
using alias = type|namespace;

using MyAlias = MyCompany.Proj.Nested;
// synonim

namespace MyCompany.Proj {
    public class MyClass {
        public static void DoNothing() {
        }
    }
}

using System;
using AliasToMyClass = NameSpace1.MyClass;

namespace NameSpace1 {
    public class MyClass {
        public override string ToString(){
            return "You_are_in_NameSpace1.MyClass";
        }
    }
}

```

```

}
namespace NameSpace2 {class MyClass {}}
namespace NameSpace3 {
    using NameSpace1;
    using NameSpace2;
    class MainClass{
        static void Main() {
            AliasToMyClass someth = new AliasToMyClass ();
            Console.WriteLine(someth);
            //You are in NameSpace1.MyClass
        }
    }
}
}

```

29 C++ i C#

Wybrane różnice pomiędzy obydwoma językami.

Typy

- long - w C# 64 bity w C++ 32.
- bool - w C++ właściwie liczba całkowita, W C# nie.
- typy tablicowe w C# są obiektami.
- w C# nie ma pól bitowych (bit fields)

Cechy języka C#

- indeksatory, właściwości
- typy delegowane
- blok `finally` (obsługa wyjątków)
- zmienne lokalne nie mogą być użyte przed inicjalizacją
- kolektor nieużytków (ang. Garbage Collection)

Dziedziczenie

- W C# nie ma dziedziczenia mnogiego, jednak klasy mogą implementować wiele interfejsów.
- W C# występują słowa kluczowe `new`, `override` i `base`, - jawne przesłanianie, modyfikowanie i wołanie metod przesłoniętych.

C++ i C# - różnica

- `static` - w C# używane w węższym znaczeniu.
- `extern` - w C# używane również w innym znaczeniu, do tworzenia synonimów w kodzie niezarządzanym.
- Metoda `Main` w C# nazywa się z wielkiej litery.
- W C# występują słowa kluczowe: `foreach`, `in`.
- W C# metody i zmienne globalne nie są dozwolone.
- W C# nie ma wartości domyślnych dla parametrów metod.
- Występuje słowo kluczowe `as` podobne do `static_cast` w C++.

30 Powtórka

Metody i parametry

- `ref`
- `out`
- `params`

Struktury

- Struktury są typami przekazywanymi przez wartość - ich wartości są przechowywane na stosie, nie można po nich dziedziczyć. Ale posiadają wiele cech klas.

- Język dostarcza dla struktur bezparametrowych konstruktorów domyślnych. Można również dodawać własne.
- Struktury mogą implementować interfejsy.

Switch - cechy

- Obsługę kolejnych przypadków można przestawiać w kodzie
- Możliwe jest użycie `goto` (niezalecane), co w niektórych wypadkach może uprościć kod.
- Możliwa jest obsługa kilku przypadków w jednym bloku
- Musi być możliwa niejawną konwersja z każdego wyrażenia stałego po słowie `case` do typu wyrażenia po `switch`
- Jeśli powtórzy się wyrażenie stałe o tej samej wartości, to wystąpi błąd kompilacji.
- Jeśli wyrażenie stałe nie jest jawnie zadeklarowane jako `unchecked`, to przepełnienie (ang. `overflow`) przy jego obliczaniu spowoduje błąd kompilacji.

Poznane słowa kluczowe

- `partial`
- `sealed`
- `checked`, `unchecked`
- `unsafe`

Właściwości

Przykład 57.

```

class TimePeriod {
    private double seconds;
    public double Hours {
        get { return seconds / 3600; }
        set { seconds = value * 3600; }
    }
}

class Program{
    static void Main() {
        TimePeriod t = new TimePeriod();
        t.Hours = 24;
        System.Console.WriteLine("Time in hours: "
            + t.Hours);
    }
}

```

Indeksatory

Przykład 58.

```

public class YearClass{
    const int StartDate = 1900;
    const int EndDate = 2050;
    int [] arr = new int [EndDate - StartDate + 1];

    public int this [int num]{
        get { return arr [ num - StartDate]; }
        set { arr [num - StartDate] = value; }
    }
}

public class Test{
    public static void Main(){
        YearClass yc = new YearClass ();
        yc [1950] = 5;
    }
}

```

Cechy dziedziczenia

- W C# nie ma dziedziczenia mnogiego
- Dopuszczalna jest implementacja wielu interfejsów
- Nie ma znanego z C++ dziedziczenia publicznego, chronionego i prywatnego.
- Słowa kluczowe `new`, `virtual` i `override`

Elementy generyczne

- Metody generyczne
- Klasy generyczne
- Słowo kluczowe `default`

Nullable Types

Przykład 59.

```
class NullableExample{
    static void Main(){
        int? num = null;
        if (num.HasValue == true)
            System.Console.WriteLine("num=_=" + num.Value);
        else
            System.Console.WriteLine("num=_=Null");
            //y is set to zero
            int y = num.GetValueOrDefault();
            // num.Value throws an InvalidOperationException
            // if num.HasValue is false
            try{ y = num.Value; }
            catch (System.InvalidOperationException e){
                System.Console.WriteLine(e.Message);
            }
        }
    }
}
```

Składnia T? jest krótszą wersją `System.Nullable<T>`.

Delegaty

- Delegaty można traktować jak bezpieczne wskaźniki do funkcji.
- Sygnatura podstawianej funkcji musi być taka jak deklaracja delegata.
- Podstawiać można zarówno metody statyczne jak i niestacyjne (ang. instance methods).
- Delegaty są typami referencyjnymi.
- Obiekty typów delegowanych nie mogą być zmieniane (są ang. immutable).

Atrybuty klas i System.Reflection

- `System.AttributeUsage` - pozwala określić do jakich klas można stosować dany atrybut.
- Klasa `Type` - opisuje typ.
- `Attribute.GetCustomAttributes(Type t)` pobiera atrybuty typu `t`.
- `Activator.CreateInstance(Type t)` tworzy obiekt typu `t`.
- `InvokeMember` metoda w klasie `Type` pozwala wywołać metodę o zadanej nazwie i parametrach.