Programowanie obiektowe w języku C#

Krzysztof Grąbczewski

26 marca 2024

Katedra Informatyki Stosowanej
Uniwersytet Mikołaja Kopernika
Toruń

`http://www.is.umk.pl/~kg/zajecia/PO2/PO2.pdf`

Literatura

- Jesse Liberty i in. „C#. Programowanie”, wiele wydań również po polsku.
- Ben Albahari, Joseph Albahari, „C# 5.0. Leksykon kieszonkowy”
- Joseph Albahari, Ben Albahari, „C# 5.0 in a Nutshell”, O'Reilly Media, 2012, wydanie V.
- `https://docs.microsoft.com/en-us/dotnet/csharp/`
- `https://docs.microsoft.com/en-us/visualstudio/`

Platforma .NET (1)

- .NET Framework = Common Language Runtime (CLR) + Framework Class Library (FCL)
- Środowisko a kompilatory
- MSIL - MicroSoft Intermediate Language (Common Intermediate Language - CIL)
  - wspólny kod wyjściowy dla różnych jezyków programowania (Visual Basic, C#, Managed C++, JScript…)
  - każdy program można łatwo zdekompilować
  - programy utrudniające dekompilację - *obfuscators*
- JIT (Just in Time) compiler
  - tworzy kod maszynowy dla danego procesora (jak w Javie)
  - optymalizacja na dany komputer – bardzo poważna zaleta

Platforma .NET (2)

- Zarządzanie bibliotekami dynamicznymi
  - *Zestaw (assembly)* jako uogólnienie DLL
  - Zezwolenie na wiele wersji tego samego zestawu
- bogactwo biblioteki standardowej .NET (FCL) – przestrzenie nazw nabierają jeszcze większego znaczenia
- projekt MONO `http://www.mono-project.com/` – .NET na Unix'ach
- Xamarin – od 2016 firma zależna Microsoftu
- .NET Core – Windows, Linux, MacOS; .NET Foundation
- .NET MAUI (Multi-platform App UI) – 2021, .NET 6

Visual Studio

Zintegrowane środowisko rozwoju oprogramowania (IDE - Integrated Development Environment)

- Zarządzanie projektami i rozwiązaniami wieloprojektowymi
- Wspomaganie walki z błędami (debugging)
- Graficzne tworzenie interfejsów okienkowych
- Organizacja zasobów
- Organizacja testów tworzonego kodu
- Różne narzędzia ułatwiające tworzenie kodu
  - Intellisense – automatyczne uzupełnianie
  - Refactoring – łatwiejsze zmiany w kodzie
  - import kontrolki ActiveX
- Wsparcie kontroli wersji: git, subversion, TFS

Język C# – podstawy

- Brak plików nagłówkowych, bardzo szybka kompilacja
- Pierwszy program

```
1  class Example
2  {
3     public static void Main() {
4        System.Console.WriteLine("Hello world!");
5     }
6  }
```

- Wszystko wewnątrz klas
- Operator . pobiera składowe (odpowiednik :: z C++)
- Komentarze w stylu C i C++
  - */*komentarz o dowolnej rozpiętości*/*
  - *//komentarz do końca linii*,
- Identyfikatory to ciągi znaków Unicode (mogą zawierać polskie znaki)

Zmienne

- ściśle określony typ (operacje, które można wykonywać)
- każda zmienna musi być zadeklarowana, np.:

```
1 int n;
2 float z = 0.75f;
```

- każda zmienna musi być zainicjowana przed użyciem
- pola są inicjowane automatycznie na 0 lub wartość „najbliższą zeru”
  - numeryczne i wyliczeniowe – 0
  - char – '\0'
  - bool – **false**
  - referencyjne – **null**
- nie ma automatycznej inicjalizacji zmiennych lokalnych, ale kompilator czuwa

Predefiniowane typy wartościowe (1)

- typy całkowite

alias	typ	rozmiar	znak
sbyte	System.SByte	1	tak
short	System.Int16	2	tak
int	System.Int32	4	tak
long	System.Int64	8	tak
byte	System.Byte	1	nie
ushort	System.UInt16	2	nie
uint	System.UInt32	4	nie
ulong	System.UInt64	8	nie

```
1 int ii = 5;
2 uint ui = 5U;
3 long ll = 5L;
4 ulong ul = 5UL;
```

Predefiniowane typy wartościowe (2)

- typy rzeczywiste

alias	typ	rozmiar	zakres
float	System.Single	4	$\pm 1,5 \times 10^{-45}$ – $\pm 3,4 \times 10^{38}$
double	System.Double	8	$\pm 5,0 \times 10^{-324}$ – $\pm 1,7 \times 10^{308}$
decimal	System.Decimal	16	$\pm 1,0 \times 10^{-28}$ – $\pm 7,9228 \times 10^{28}$

```
1 float f = 3.14f;
2 double d = 3.14;
3 decimal m = 123456789.123456789m;
```

typ **float**:

- 7-8 cyfr znaczących

typ **double**:

- 14-15 cyfr znaczących

typ **decimal**:

- 28-29 cyfr znaczących, ale mały zakres
- szczególnie przydatny w finansach

Predefiniowane typy wartościowe (3)

- **char** i **bool**

alias	typ	rozmiar
char	System.Char	2
bool	System.Boolean	1

- **char** reprezentuje znaki w standardzie Unicode
- **bool** to typ logiczny rozróżniający dwie wartości: **true** oraz **false**

Predefiniowane typy wartościowe (4)

Typ wyliczeniowy

- Standardowo wartości 0, 1, 2, …

```
1 enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

- Wartości 1, 2, …

```
1 enum Days {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
```

- Wartości jawnie zadane i typ **long**:

```
1 enum Range : long {Max = 2147483648L, Min = 255L};
```

- Brak konwersji niejawnych - trzeba jawnie

```
1 int i = (int)Days.Sun;
2 long l = (long)Range.Max;
```

- Dostępne operatory: == != < > <= >= + − ^ & | ~ = += −= ++ −− **sizeof**

Predefiniowane typy referencyjne

alias	typ	rozmiar
object	System.Object	0/8
string	System.String	min. 20

- **object** – klasa bazowa dla wszystkich typów, dla referencyjnych 8 bajtów
- **string** – niezmienna sekwencja znaków Unicode
- **string** – zwykła klasa o specjalnym traktowaniu

```
1 string a1 = "\\\\server\\fileshare\\helloworld.cs";
2 string a2 = @"\\server\fileshare\helloworld.cs";
3 Console.WriteLine(a1==a2); // Prints "True"
4 string b1 = "First Line\r\nSecond Line";
5 string b2 = @"First Line
6 Second Line";
7 Console.WriteLine(b1==b2); // Prints "True"
```

Tablice – też referencyjne (1)

- Tablica - wiele obiektów tego samego typu, w ciągłym obszarze pamięci
- Inicjalizacja jako null lub z alokacją:

```
1 int[] liczby = null;
2 float[] ff = new float[7];
3 double[] zz = { 1.2, 4.5, 3 };
```

- Elementy tablic są automatycznie inicjowane
- Dostęp do elementów przez operator [] – indeksowanie od 0

```
1 liczby[0] = liczby[1];
2 ff[9] = 7; // błąd: indeks spoza zakresu
```

Tablice – też referencyjne (2)

- System.Array – zwykła klasa o specjalnym traktowaniu
- syntaktyka podobna do C++, ale z pewnymi różnicami
- tablice prostokątne i „postrzępione” (jagged)

```
1 int[] tab, tab2 = new int[15];
2 int[,] tab2d = new int[5,5];
3 double[][] macierz = new double[5][];
4 for (int i=0; i<5; i++)
5     macierz[i] = new double[10+i];
6 tab2[5] = tab2d[0,3] = 7;
7 macierz[0][3] = 7;
8 int[,] zainicjowana = {{1,2},{3,4}};
```

- kontrola zakresów, wyjątek IndexOutOfRangeException, dzięki optymalizacji nie musi dużo kosztować

Wybrane metody i własności klasy Array

Length	długość tablicy
Rank	liczba wymiarów
IsReadOnly	czy tylko do odczytu
GetLength()	długość w danym wymiarze
GetLowerBound()	dolne ograniczenie danego wymiaru
GetUpperBound()	górne ograniczenie danego wymiaru
Sort()	sortowanie tablic jednowymiarowych
Reverse()	odwracanie kolejności tablic jednowymiarowych
IndexOf()	indeks pierwszego wystąpienia wartości
LastIndexOf()	indeks ostatniego wystąpienia wartości
Copy()	kopiowanie fragmentu
Clear()	ustawianie wartości 0 lub **null**

Operatory, wyrażenia (1)

Operatory według priorytetów:

Pierwszorzędowe	. ?. [] ?[] f() x++ x−− **new typeof checked unchecked default delegate sizeof** −>
Jednoargumentowe	+ − ! ~ ++x −−x (T) **await** & *
Multiplikatywne	* / %
Addytywne	+ −
Przesunięcia	<< >>
Relacyjne i testy typu	< > <= >= **is as**
Porównania	== !=
Koniunkcja	&
Alternatywa wyłączająca	^
Alternatywa	|
Koniunkcja warunkowa	&&
Alternatywa warunkowa	||
Null-coalescing	??
Operator warunku	?:
Przypisania i lambda	= += −= *= /= %= &= |= ^= <<= >>= => ??=

Łączność: operatory dwuargumentowe z wyjątkiem przypisań i ?? mają łączność lewostronną

Operatory, wyrażenia (2)

Wyrażenia

- zawierają identyfikatory zmiennych lub stałych oraz ich złożenia z uwzględnieniem argumentowości, łączności i priorytetów operatorów
- priorytety mają bardzo istotne znaczenie dla sposobu wyliczania wartości wyrażeń
- mają ściśle określony typ
- przykłady:

```
1        x + 5
2        x = y−1
3        z++
4        a <<= −−b & c++
```

Instrukcje (1)

Instrukcje to:

- wybrane wyrażenia zakończone średnikiem
  - przypisania
  - inkrementacje, dekrementacje
  - wywołania funkcji
  - **async**
  - **new**
- inne konstrukcje zdefiniowane w ramach języka:
  - warunkowe: **if else switch case**
  - iteracyjne: **do for foreach in while**
  - skoku: **break continue default goto return yield**
  - obsługi wyjątków: **throw try–catch try–finally try–catch–finally**
  - kontroli błędów: **checked unchecked**
  - inne: **fixed lock**
- składają się na funkcje (metody)

Instrukcje (2)

- przykłady instrukcji:

```
1  x + 5; // błąd, bo to żadne z ww wyrażeń
2  x = y−1; // ok
3  z++; // ok
4  a <<= −−b & c++; // ok;
5  Console.WriteLine("Dzień dobry!");
6  delta = b*b − 4*a*c;
7  pierw = Math.Sqrt(delta);
8  if (delta > 0)
9  {
10    x[0] = (−b − pierw)/2/a;
11    x[1] = (−b + pierw)/2/a;
12 }
13 else if (delta == 0)
14    x[0] = x[1] = −b/2/a;
```

Instrukcje (3)

- Instrukcja warunkowa **if** może zawierać część **else**, ale nie musi

```
1  if (delta < 0)
2  {
3     Console.WriteLine("Uwaga! Brak pierwiastków!");
4     return;
5  }
6  if (delta > 0)
7  {
8     x1 = (−b − Math.Sqrt(delta))/2/a; // niełądnie...
9     x2 = (−b + Math.Sqrt(delta))/2/a; // 2 x Sqrt!
10 }
11 else
12       x1 = x2 = −b/2/a;
```

Instrukcje (4)

- Instrukcja **switch**

```
1  int n;
2  ...
3  switch (n>=0 ? n%2 : -1)
4  {
5      case 0:
6          Console.WriteLine("Liczba naturalna parzysta");
7          break;
8      case 1:
9          Console.WriteLine("Liczba naturalna nieparzysta");
10         break;
11     default:
12         Console.WriteLine("Liczba ujemna");
13         break;
14 }
```

W każdej części musi się pojawić jedna z instrukcji: **break**, **goto case**, **return**, **throw**.

Instrukcje (5)

- Pętla **do–while**
  - najpierw wykonuje instrukcję/blok, a potem sprawdza warunek
  - wykonuje instrukcję/blok co najmniej raz

```
1 char c;
2 Console.Write("Tak czy nie?");
3 do
4       c = Console.ReadKey(true).KeyChar;
5 while (c != 't' && c != 'n');
6 Console.WriteLine(c == 't' ? "\nŚwietnie!"
7       : "\nJak nie, to nie!");
```

Instrukcje (6)

- Pętla **while**
  - najpierw sprawdza warunek, potem wykonuje instrukcję/blok, jeśli prawdziwy
  - może nie wykonać instrukcji/bloku ani razu

```
1 int n = 104, i = 0;
2 while (n%2 == 0)
3 {
4       n /= 2;
5       i++;
6 }
7 Console.WriteLine("2^{0}*{1}", i, n);
```

Instrukcje (7)

- Pętla **for**

```
1 for (inicjalizacja; warunek; iterator)
2       instrukcja
```

wykona to samo co następujący kod:

```
1 inicjalizacja;
2 while (warunek)
3 {
4       instrukcja
5       iterator;
6 }
```

Przykład:

```
1 int n = 104;
2 for (int i=0; n%2 == 0; i++)
3       n /= 2;
```

Instrukcje (8)

- Pętla **foreach** wykonuje instrukcje dla każdego elementu kolekcji

```
1 int[] tab = {2, 5, 6, 4, 7};
2 Console.WriteLine("Parzyste:");
3 foreach (int n in tab)
4   if (n%2 == 0)
5     Console.WriteLine(n);
```

Działa nie tylko z tablicami ale ze wszystkimi typami kolekcji, które są odpowiednio zaimplementowane (mają enumeratory)

Typy wartościowe i referencyjne (1)

- Stos i sterta (stack and heap)
  - stos – obsługa wywołań funkcji i zmiennych lokalnych
  - sterta – miejsce na dynamicznie alokowane zmienne
  - czyszczenie stosu związane z powrotami z funkcji
  - czyszczenie sterty przejmuje system zbierania odpadków (garbage collection)
- Typy wartościowe (typy proste, **struct**, **enum**)

```
1 class Test {
2   static void Main () {
3     int x = 3;
4     int y = x; // assign x to y, y is now a copy of x
5     x++; // increment x to 4
6     System.Console.WriteLine(y); // prints 3
7   }
8 }
```

Typy wartościowe i referencyjne (2)

- Typy referencyjne (klasy, kolekcje, delegacje, interfejsy)
  - wartość i adres jej przechowywania w jednym
  - analogicznie do przekazywania parametrów przez referencję w C++
  - syntaktycznie jak wartości, semantycznie jak wskaźniki

```
1  using System;
2  using System.Text;
3  class Test {
4    static void Main () {
5      StringBuilder x = new StringBuilder("hello");
6      StringBuilder y = x;
7      x.Append(" there");
8      Console.WriteLine(y); // prints "hello there"
9    }
10 }
```

Typy wartościowe i referencyjne (3)

- typy wartościowe i referencyjne - przykład porównujący

```
1  class PointR { // typ referencyjny
2    public int x, y;
3    public PointR(int a, int b) { x = a, y = b }
4  }
5  struct PointV { // typ wartościowy
6    public int x, y;
7    public PointV(int a, int b) { x = a, y = b }
8  }
9  class Test {
10   static void Main() {
11     PointR a; // 4 bajty na stosie
12     PointV b; // 8 bajtów na stosie
13     a = new PointR(1,1); // alokacja 8 bajtów na stercie
14     b = new PointV(2,3); // zawołanie konstruktora
15     a.x = 7;
16     b.x = 7;
17   }
18 }
```

Typy wartościowe i referencyjne (4)

- opakowywanie wartości w obiekty referencyjne i rozpakowywanie (boxing, unboxing)

```
1 class Test {
2   static void Main () {
3     int x = 9;
4     object o = x; // box the int
5     int y = (int)o; // unbox the int
6   }
7 }
```

Garbage collection (1)

- **new** - alokacja/inicjalizacja/aktywacja obiektów
  - typy wartościowe – tylko inicjalizacja obiektów (konstruktor)
  - typy referencyjne – alokacja pamięci i inicjalizacja konstruktorem
- nie zwalniamy pamięci wprost (na wzór **delete** z C++)
- garbage collection system – system zarządzania pamięcią
  - zarządza wszelkimi referencjami do obiektów
  - odzyskuje pamięć po niepotrzebnych obiektach
  - Uwaga! ważne zerowanie „długo żyjących” referencji, bo inaczej wycieki pamięci
  - Uwaga! obiekty mogą zmieniać swoje położenie (adres) bez wiedzy programisty

```
1 public void Test()
2 {
3     Macierz m = new Macierz();
4     ObliczeniaDlaMacierzy(m);
5     m = null; // dla zmiennych lokalnych niepotrzebne, chyba, że dalej jeszcze dużo ...
6     DużoLiczenia();
7 }
```

Garbage collection (2)

```
1  // W klasie Macierz
2  public Macierz(Manager mgr)
3  {
4     mgr.Rejestruj(this); // mgr wrzuca this do kolekcji
5     ...
6  }
7  public void Test(Manager mgr)
8  {
9     Macierz m = new Macierz(mgr);
10    ...
11    // m nie będzie zwolniona, bo "trzyma" ją mgr
12 }
```

Klasy i struktury (1)

- Własne typy danych
  - typy wyliczeniowe (**enum**)
  - klasy
  - struktury
- Proste deklaracje klasy i struktury:

```
1 class PointR
2 {
3   public int x, y;
4 }
```

```
1 struct PointV
2 {
3   public int x, y;
4 }
```

- Klasy i struktury mogą zawierać definicje *pól* (zmiennych wewnętrznych klasy), *metod* (funkcji) i *własności* oraz *zagnieżdżonych typów*
- Operator . daje dostęp do składowych (pól, metod, własności):

```
1 PointR p;
2 p.x = 1; p.y = 7;
```

Klasy i struktury (2)

Najważniejsze różnice pomiędzy **klasami** a **strukturami**:

- Struktury są typami **wartościowymi** a klasy **referencyjnymi**
- W strukturach nie można jawnie definiować konstruktora bezargumentowego (realizowany niejawnie, automatycznie)
- Operator **new** inicjalizuje struktury, a klasy alokuje i inicjalizuje
- Wśród struktur nie działa mechanizm dziedziczenia (o tym później)

Klasy i struktury (3)

Tablice obiektów typów wartościowych i referencyjnych

```
1  PointV[] punktyV = new PointV[10];
2  punktyV[5].x = 7; // ok
3
4  PointR[] punktyR = new PointR[10];
5  punktyR[5].x = 7; // błąd!
6
7  for (int i=0; i<punktyR.Length; i++)
8     punktyR[i] = new PointR();
9  punktyR[5].x = 7; // ok
10
11 for (int i=0; i<punktyV.Length; i++)
12    punktyV[i] = punktyV[5];
13
14 for (int i=0; i<punktyR.Length; i++)
15    punktyR[i] = punktyR[5];
```

Modyfikatory dostępu

- **public** – „pełna widoczność”
- **private** – dostęp tylko dla metod tej samej klasy
- **protected** – dostęp dla metod tej samej klasy i klas potomnych
- **internal** – dostęp w ramach zestawu (assembly)
- **protected internal** – suma logiczna warunków dostępu dla **protected** i **internal**
- domyślnie:
  - dla niezagnieżdżonych klas **internal**
  - dla składowych klas **private**
- dziedziczenie zawsze publiczne!

Pola klas i struktur (1)

- zmienne zadeklarowane wewnątrz klasy/struktury
- pola mogą być inicjowane w linii deklaracji (przypisanie nastąpi przed konstruktorem)

```
1 public class TestAli
2 {
3    internal string kto = "Ala";
4    public TestAli(bool oficjalnie) {
5       if (oficjalnie) kto = "Alicja";
6    }
7    public TestAli(int ileKotów) { ... }
8    public string KtoTo() { return kto; }
9 }
```

- dostęp w metodach klasy przez nazwę, a poza klasą w kontekście obiektu:

```
1    TestAli x = new TestAli();
2    Console.WriteLine(x.KtoTo());
```

Pola klas i struktur (2)

- pola **statyczne** to pola wspólne dla wszystkich obiektów
  - nie powiększają poszczególnych obiektów klasy (leżą na zewnątrz)
  - są alokowane i inicjowane dopiero wtedy, gdy są potrzebne
  - dostępne przez nazwę klasy i operator .

```
1 public class Kontrola
2 {
3    internal static int ileRazy = 0;
4    public void Dotknij() { ileRazy++; }
5 }
6    ... // np. na koniec funkcji Main()
7    Console.WriteLine("Obiekty Kontroli dotknięte {0} razy",
8       Kontrola.ileRazy);
```

Pola klas i struktur (3)

- pola **tylko do odczytu**

```
1    readonly int rozmiar = 100;
2    static readonly maxRozmiar = 1000;
```

  - wyznaczane w trakcie działania, nie kompilacji
  - muszą być zainicjowane w linii deklaracji bądź w konstruktorze

- pola **stałe**

```
1 public const double PI = 3.14159265358979323846;
```

  - wyznaczane w trakcie kompilacji
  - statyczne z założenia
  - dozwolone typy: **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, **float**, **double**, **decimal**, **bool**, **char**, **string**, **enum**
  - muszą być zainicjowane w linii deklaracji

Metody klas (1)

- **Metody** to funkcje w klasach
- Każda funkcja w C# to metoda pewnej klasy
- Ogólny schemat definicji metody:

```
1 typ Nazwa(typ1 arg1, typ2 arg2, ...)
2 {
3   deklaracje i instrukcje
4 }
```

- Nagłówek określa typ zwracanej wartości, nazwę metody oraz typy i nazwy parametrów/argumentów funkcji
- Lista argumentów może być pusta
- Wartości są zwracane instrukcją **return** wyrażenie;
- Funkcja może nic nie zwracać, wówczas w miejscu typu występuje słowo kluczowe **void**

Metody klas (2)

- Przykład prostej funkcji – silnia iteracyjnie i rekurencyjnie

```
1  static long Silnia(uint n)
2  {
3     long wynik = 1;
4     for (int i=2; i<=n; i++)
5        wynik *= i;
6     return wynik;
7  }
8
9  static long SilniaRek(uint n)
10 {
11    return n < 2 ? 1 : n * SilniaRek(n-1);
12 }
```

Metody klas (3)

- Zmienna liczba argumentów – słowo kluczowe **params**

```
1  double Suma(params double[] argumenty)
2  {
3     double suma = 0;
4     foreach (double x in argumenty)
5        suma += x;
6     return suma;
7  }
8
9  static void Main() {
10    double[] tab = {1.5, 2.3, 17, 123.45 };
11    ...
12    double wszystko = Suma(tab) + Suma(9.8, 7.6, 5.4);
13 }
```

Metody statyczne

- Podobnie jak pola statyczne, nie są związane z konkretnymi obiektami klasy
- Wywoływane są z nazwą klasy a nie obiektu np: Console.WriteLine(Math.Sqrt(10));
- Definiowane ze słowem kluczowym **static**
- Mogą korzystać z pól i innych metod statycznych, niestatycznych tylko przez konkretne obiekty
- Przykład: funkcja główna progamu (Main())
- Przykład: funkcje liczące silnię

Przekazywanie argumentów do metod (1)

- domyślnie przez wartość (działania na kopii)
- w przypadku typów referencyjnych – kopia referencji
- **ref** – przekazanie przez referencję

```
1 public void zamień(ref int a, ref int b)
2 {
3     int c = a;
4     a = b;
5     b = c;
6 }
```

Przekazywanie argumentów do metod (2)

- **out** – parametr traktowany jak wyjście funkcji, kompilator pilnuje by zmiennej coś przypisać w funkcji

```
1 public void ustaw(out int a, out int b)
2 {
3     a = 5;
4     b = 7; // skomentuj jedną z linii by dostać błąd
5 }
```

Przekazywanie argumentów do metod (3)

- **ref** i **out** konieczne również przy wywołaniach
- zmienna przekazana z modyfikatorem **out** może nie być zainicjowana

```
1 public void test()
2 {
3     int x, y;
4     ustaw(out x, out y);
5     zamień(ref x, ref y);
6 }
```

Przekazywanie argumentów do metod (4)

- Parametry nazwane

```
1 public int Kalkuluj(int a, int b)
2 {
3     return a*a + b*b;
4 }
5
6 public int UżyjKalkulacji()
7 {
8     return Kalkuluj(a:4, b:0) + Kalkuluj(b:3, a:7);
9 }
```

Przekazywanie argumentów do metod (5)

- Domyślne wartości parametrów metod

```
1 public double Logarytm(double wartość, int podstawa=10)
2 {
3   return ...;
4 }
5
6 public int UżyjLogarytmów()
7 {
8   return Logarytm(312, 2) + Logarytm(15);
9 }
```

- Łączne używanie parametrów nazwanych i domyślnych wartości parametrów pozwala znacznie uprościć wywołania wieloargumentowych metod

Dociążanie operatorów (1)

- Dociążanie lub przeciążanie (ang. *operator overloading*)
- Niektóre operatory mają różne implementacje dla różnych kontekstów
- Niektórym operatorom można nadawać nowe konteksty – robi się to przez definicje metod-operatorów
- Operatory, które można dociążać:
  - jednoargumentowe: `+ - ! ~ ++ -- true false`
  - dwuargumentowe: `+ - * / % & | ^ << >> == != > < >= <=`
- Nie można tworzyć nowych operatorów (symboli) ani zmieniać argumentowości

Dociążanie operatorów (2)

- Operatory implementujemy jako metody statyczne o nazwie **operator** OP, gdzie OP to odpowiedni symbol
- Operatorów logicznych && || nie można dociążać, ale można opertory & | **true false**
  - jeśli implementujemy jeden z **true false**, to musimy i drugi
- Operatory porównujące też trzeba definiować w parach:
  - == i !=
  - < i >
  - <= i >=
- Można definiować operatory konwersji typów
  - operatory do konwersji jawnej (**explicit**) i niejawnej (**implicit**)
  - nazwa metody: **operator** Typ
  - dozwolone operatory konwersji w obie strony
- Przykład – klasa UInt128 do arytmetyki na liczbach 128-bitowych

Dociążanie operatorów (3)

```
1  public struct UInt128
2  {
3     ulong hi; // bardziej znaczący ośmiobajt
4     ulong lo; // mniej znaczący ośmiobajt
5     public static UInt128 operator +(UInt128 l1, UInt128 l2)
6     {
7        UInt128 wynik;
8        wynik.lo = l1.lo + l2.lo;
9        ulong przes = wynik.lo < l1.lo || wynik.lo < l2.lo ? 1ul : 0ul;
10       wynik.hi = l1.hi + l2.hi + przes;
11       return wynik;
12    }
13    public static bool operator true(UInt128 x) { return x.hi != 0 || x.lo != 0; }
14    public static bool operator false(UInt128 x) { return x.hi == 0 && x.lo == 0; }
15    public static implicit operator UInt128(ulong x) { return new UInt128() { lo = x }; }
16    public static explicit operator ulong(UInt128 x) { return x.lo; }
17 }
```

Własności (properties)

- implementowane inline

```
1 public class Prostokąt
2 {
3   int lewa, góra, długość, szerokość;
4   public int Prawa {
5     get { return lewa + dlugość; }
6     set { lewa = value − dlugość; }
7   }
8 }
```

- w praktyce para metod get_Prawa() i set_Prawa(**int value**)
- **automatycznie implementowane własności** – od wersji 3 – prywatne pole tworzone automatycznie przez kompilator

```
1 public class Prostokąt
2 {
3   public int Lewa { get; set; }
4   public int Góra { get; set; }
5   public float PrywatnyZapis { get; private set; }
6 }
```

Operatory indeksujące (indexers)

- definiowane podobnie do własności

```
1  public class Tab5Double {
2    double[] tab = new double[5];
3    double suma;
4    public double this[int i] {
5      get { return tab[i]; }
6      set { suma += value−tab[i]; tab[i] = value; }
7    }
8    public double Srednia {
9      get { return suma / 5; }
10   }
11 }
```

- można zdefiniować wiele operatorów o różnych listach argumentów

Konstruktory klas

- jak w C++ tj. metody o nazwie klasy

```
1 public class TabDouble {
2   double[] tab;
3   TabDouble(int n) { tab = new double[n]; }
4   TabDouble() : this(100) {};
5 }
```

- pola inicjowane w linii deklaracji przychodzą do konstruktora już zainicjowane (w kolejności występowania w klasie)
- mogą być opatrzone dowolnym modyfikatorem dostępu

Konstruktor statyczny

- może być tylko jeden w klasie
- wołany przed stworzeniem pierwszego obiektu i przed dostępem do jakiegokolwiek pola statycznego klasy

```
1 public class TabDouble {
2    static TabManager tabmgr;
3    static TabDouble() { tabmgr = new TabManager(); }
4 }
```

- pola statyczne inicjalizowane przed konstruktorem statycznym
- kolejność wołania konstruktorów statycznych jest nieustalona, również dziedziczenie nie ma tu znaczenia

Destruktor

- Jak w C++ metoda o nazwie klasy poprzedzonej tyldą
- Zadania całkiem inne niż w C++ ze względu na garbage collection
- Bardzo rzadko potrzebne
- Czas wykonania destruktora jest nieokreślony
- Kompilowany do funkcji wirtualnej Finalize()

```
1 protected override void Finalize() {
2   ...
3   base.Finalize();
4 }
```

Klasy i struktury zagnieżdżone

- Klasy i struktury można definiować wewnątrz innych klas/struktur
- Dostęp do zagnieżdżonych klas za pośrednictwem nazwy klasy zewnętrznej
- Modyfikatory dostępu mogą ukryć zagnieżdżony typ
- Używać w przypadku gdy typ jest rzeczywiście lokalny dla danego problemu

Przestrzenie nazw

- Deklaracje dokładania do przestrzeni i korzystanie z przestrzeni

```
1 namespace Poczatki {
2   using System;
3   class Example {
4     public static void Main () {
5       Console.WriteLine ("Hello world!");
6     }
7   }
8 }
```

- Można zagnieżdżać przestrzenie nazw tworząc hierarchię
- Przykłady z hierarchii FCL

```
1 System.IO
2 System.Collections.Generic
3 System.Runtime.Serialization.Formatters.Binary
4 System.Security.Cryptography
```

Typ System.Object

- Typ-baza dla wszystkich innych typów

```
1  public class Object
2  {
3      public Object();
4      public Type GetType();
5      protected object MemberwiseClone();
6
7      public virtual bool Equals(object obj);
8      public virtual int GetHashCode();
9      public virtual string ToString();
10
11     public static bool Equals(object objA, object objB);
12     public static bool ReferenceEquals(object objA, object objB);
13 }
```

Dziedziczenie (1)

- Klasa bez jawnego dziedziczenia **dziedziczy** po **object**
- **Wielokrotne** dziedziczenie jest **niedozwolone**
- Prosty przykład:

```
1 public class Kwadrat : Prostokąt {
2   public Kwadrat(int lewa, int góra, int bok)
3           : base(lewa, góra, bok, bok)
4   {
5   }
6   public Okrąg OkrągWpisany();
7 }
```

- **Konwersje niejawne** dozwolone tylko w stronę bazowej

```
1 Kwadrat k = new Kwadrat(0, 0, 5);
2 Prostokąt p = k; // OK
3 Kwadrat w = (Kwadrat)p; // jawna konwersja konieczna
```

Dziedziczenie (2)

- Nieudana **jawna** konwersja oznacza wyjątek InvalidCastException
- Operator **as** (w wyniku **null**, gdy się nie uda)

```
1 Kwadrat w = p as Kwadrat;
```

- Operator **is** (pytanie o dziedziczenie lub implementację interfejsu)

```
1 if (p is Kwadrat)
2    ((Kwadrat)p).OkragWpisany().Rysuj();
```

Klasy abstrakcyjne (1)

- **Polimorfizm** to możliwość realizacji tego samego wywołania w kontekście różnych typów.
- **Dziedziczenie** klas i **implementowanie** interfejsów
- Metody wirtualne

```
1 public class Figura {
2     public virtual void Rysuj() {throw new NotDefined();}
3 }
4 public class Prostokąt : Figura {
5     public override void Rysuj() {...}
6 }
7 public class Kwadrat : Prostokąt {
8     public override void Rysuj() {...}
9 }
```

Klasy abstrakcyjne (2)

- **Klasy abstrakcyjne** mogą zawierać **metody abstrakcyjne**
- **Metody abstrakcyjne** to metody wirtualne bez implementacji:

```
1 public abstract class Figura {
2     public abstract void Rysuj();
3 }
```

- W C++

```
1 class Figura {
2     public:
3         virtual void Rysuj() = 0;
4 }
```

Klasy abstrakcyjne (3)

- Ukrywanie metod wirtualnych – przerywanie wirtualności

```
1 public class Figura {
2    public virtual void Rysuj() {throw new NotDefined();}
3 }
4 public class Prostokąt : Figura {
5    public override void Rysuj() {...}
6 }
7 public class Kwadrat : Prostokąt {
8    public new void Rysuj() {...}
9 }
```

Wówczas:

```
1 Kwadrat k = new Kwadrat(0, 0, 5);
2 k.Rysuj(); // woła Kwadrat.Rysuj()
3 ((Figura)k).Rysuj(); // woła Prostokąt.Rysuj()
```

Klasy abstrakcyjne (4)

- Modyfikator **sealed** – klasy i metody „**zalakowane**” (**zapieczętowane**)
  - **Nie można dziedziczyć** po zalakowanej klasie

```
1 public sealed class Prostokąt : Figura {
2   ...
3 }
4 public class Kwadrat : Prostokąt { // Błąd!
5   ...
6 }
```

  - Pozwala kompilatorowi zmienić wywołania wirtualne na niewirtualne
  - Zwykle dla klas zawierających tylko statyczne metody (np. Math)
  - Można zalakować pojedyńcze metody

```
1 public class Prostokąt : Figura {
2   public sealed override Rysuj() {...}
3 }
```

Słowo kluczowe base

- Daje dostęp do metod klasy bazowej
- Podobne w użyciu jak słowo kluczowe **this**:

```
1  public class Prostokąt : Figura {
2      public Prostokąt(int x, int y, int dług, int szer) {...}
3      public override void Rysuj() {...}
4  }
5  public class Kwadrat : Prostokąt {
6      public Kwadrat(int x, int y, int bok)
7          : base(x, y, bok, bok) {...}
8      public Kwadrat(int bok)
9          : this(0, 0, bok) {...}
10     public override void Rysuj() {
11         base.Rysuj(); ...
12     }
13 }
```

Interfejsy (1)

- **Interfejsy to specyfikacje możliwości** klas (główne narzędzie polimorfizmu w C#)
- Klasy i struktury mogą **implementować interfejsy** tzn. implementować wszystko co wyspecyfikowane w ramach interfejsu
- Konstrukcje podobne do klas z kilkoma bardzo istotnymi różnicami:
  - Interfejsy **nie zawierają implementacji** żadnych metod (są jak klasy abstrakcyjne zawierające wyłącznie deklaracje abstrakcyjnych metod)
  - Klasy i struktury mogą **implementować wiele interfejsów**
  - **Struktury** mogą implementować interfejsy, ale nie mogą dziedziczyć klas

Interfejsy (2)

- Interfejsy mogą zawierać deklaracje:
  - metod,
  - własności,
  - operatorów indeksujących,
  - zdarzeń.
- Definicja interfejsu

```
1 public interface ICloneable {
2   object Clone();
3 }
```

Interfejsy (3)

- Interfejs może **rozszerzać** inne interfejsy:

```
1 public interface ITablica {
2   object this[int i] { get; set; }
3 }
4 public interface ISortowalnaTablica : ITablica {
5   void Sortuj();
6 }
```

- Implementacja interfejsu

```
1 public class TablicaInt : ISortowalnaTablica {
2   public object this[int i] { get { return 0;} set {} }
3   public void Sortuj() { }
4 }
```

Interfejsy (4)

- Implementacja interfejsu **w sposób jawny** (np. żeby uniknąć konfliktu, gdy dwa interfejsy zawierają taką samą składową)

```
1 public class TablicaCalkowitych : ISortowalnaTablica {
2   object ITablica.this[int i] { get {return 1;} set {} }
3   void ISortowalnaTablica.Sortuj() { }
4 }
```

- Dziedziczenie i implementacja – **najpierw klasa, potem interfejsy**

```
1 public class Kwadrat : Prostokąt, ICloneable {
2   ...
3 }
```

Reimplementacja interfejsu (1)

- Jeśli klasa bazowa **implementuje składową interfejsu jako wirtualną**, to klasy potomne mogą je dociążać:

```
1 public class Bazowa : ICloneable {
2    public virtual object Clone() {return new Bazowa();}
3 }
4 public class Potomna : Bazowa {
5    public override object Clone() {return new Potomna();}
6 }
```

- Uwaga na **virtual**/**override**!
- Jeśli w powyższym Clone zabraknie **virtual**/**override**, to:

```
1 Potomna coś = new Potomna();
2 (coś as ICloneable).Clone(); // woła Bazowa.Clone()
```

Reimplementacja interfejsu (2)

- Można też **reimplementować** interfejs:

```
1 public class Bazowa : ICloneable {
2    public object Clone() {return new Bazowa();}
3 }
4 public class Potomna : Bazowa, ICloneable {
5    public object Clone() {return new Potomna();}
6 }
```

Wówczas:

```
1 Potomna coś = new Potomna();
2 (coś as ICloneable).Clone(); // woła Potomna.Clone()
```

Interfejs IDisposable i instrukcja using (1)

- Interfejs IDisposable implementują obiekty-zasoby wymagające dodatkowych operacji na koniec pracy

```
1 public interface IDisposable
2 {
3     void Dispose();
4 }
```

- Przykłady typów implementujących IDisposable:
  - strumienie, np. StreamReader, StreamWriter
  - inne obiekty związane z komunikacją, np. TCPClient
  - Font, Brush

Interfejs IDisposable i instrukcja using (2)

- Instrukcja **using** upraszcza korzystanie z takich obiektów

```
1 using (TextWriter tw = new StreamWriter("Plik.txt"))
2 {
3     tw.WriteLine("Nagłówek raportu\n{0}", raport);
4 }
```

choć to samo można uzyskać przez:

```
1 TextWriter tw = new StreamWriter("Plik.txt");
2 tw.WriteLine("Nagłówek raportu\n{0}", raport);
3 tw.Dispose();
```

a od C# 8 wystarczy

```
1 using TextWriter tw = new StreamWriter("Plik.txt");
2 tw.WriteLine("Nagłówek raportu\n{0}", raport);
```

Wyodrębnianie idei

- **Rozdzielanie** funkcjonalności, **wyodrębnianie** i **uogólnianie** (ang. *abstraction*)
- **Zamykanie**, opakowywanie i **ukrywanie** wewnętrznych danych (ang. *encapsulation*)
- Strategia **dziel i zwyciężaj** (ang. *divide and conquer*)
- Model programowania ***top-down***
- **Hierarchie** typów obiektów (klas i interfejsów)
- **Algorytmy** + **struktury danych** = programy
- Prosty przykład: rozwiązywanie problemu n hetmanów
- Prosty przykład: panel tekstowy lub graficzny do rysowania figur
- Prosty przykład: algorytm genetyczny

Typy ogólne – generics (1)

- Idea ta sama co szablonów w C++: **typy sparametryzowane innymi typami**.
- Główne zastosowanie: różne kolekcje takie jak listy, kolejki, stosy, słowniki…
- Przykład korzystania:

```
1 Stack<int> stack = new Stack<int>();
2 for (int i=0; i<10; i++)
3     stack.Push(i);
4 for (int i=0; i<10; i++)
5     Console.WriteLine(stack.Pop());
```

Typy ogólne – generics (2)

- Przykład listy elementów (dynamicznej tablicy):

```
1 List<double> lista = new List<double>();
2 for (int i=0; i<10; i++)
3   if (InteresującaLiczba(i))
4     lista.Add(i*i);
```

- Podstawowe zasady budowania typów ogólnych:
  - Parametrami mogą być tylko typy.
  - Założenia co do typu parametryzującego muszą być ujawnione w deklaracji klasy ogólnej.
  - Typy kompilowane do kodu pośredniego, typy szczegółowe powstają dopiero w wyniku działania komplatora JIT

Typy ogólne – generics (3)

- Typ Nullable<T> – struktura opakowująca typy wartościowe, by rozszerzyć ich dziedzinę o „brak wartości”

```
Nullable<int> k = null;
...
int z1 = k ?? -1;
int z2 = k != null ? k : -1; // gorzej...
int z3; // najgorzej!
if (k != null)
    z3 = k;
else
    z3 = -1;
```

Skróty do typów przez dodanie znaku ?, np.:

```
int? m; // to samo co Nullable<int> m;
double? d; // to samo co Nullable<double> d;
bool? decyzja; // to samo co Nullable<bool> decyzja; – trzy wartości: null, true, false
```

Typy ogólne – generics (4)

- Kolekcje
  - ICollection
    - Count,
    - CopyTo().
  - IEnumerable
    - GetEnumerator().
  - IEnumerator
    - Current,
    - MoveNext(), Reset().
  - **foreach** ułatwia używanie enumeratorów.
  - wersja 2.0 – iteratory, **yield return** oraz **yield break**

```
1 public IEnumerator GetEnumerator() {
2   for (int i = count - 1; i >= 0; --i) {
3     yield return items[i];
4   }
5 }
```

  - Przestrzeń nazw System.Collections.Generic.

Typy ogólne – generics (5)

- Ogólne klasy
  - Comparer – klasa bazowa dla imlplementacji IComparer.
  - Dictionary – słownik (kolekcja par klucz–wartość).
  - Dictionary.KeyCollection – Kolekcja kluczy słownika.
  - Dictionary.ValueCollection – Kolekcja wartości słownika.
  - EqualityComparer – klasa bazowa dla implementacji IEqualityComparer.
  - KeyNotFoundException – wyjątek dla słowników.
  - LinkedList – lista dwukierunkowa.
  - LinkedListNode – węzeł listy dwukierunkowej.
  - List – lista indeksowana.
  - Queue – kolejka.
  - SortedDictionary – słownik posortowany.
  - SortedDictionary.KeyCollection.
  - SortedDictionary.ValueCollection.
  - SortedList – kolekcja par klucz-wartość posortowanych przy użyciu implementacji IComparer.
  - Stack – stos.

Typy ogólne – generics (6)

- Ogólne interfejsy
  - ICollection<T> : IEnumerable<T>, IEnumerable
    - Count, IsReadOnly,
    - Add(), Clear(), Contains(), CopyTo(), Remove().
  - IComparer<T>
    - Compare()
  - IDictionary<TKey,TValue> : ICollection<KeyValuePair<TKey,TValue>>, IEnumerable<KeyValuePair<TKey,TValue>>, IEnumerable
    - Item[], Keys, Values,
    - Add(), ContainsKey(), Remove(), TryGetValue()
  - IEnumerable<T> : IEnumerable
    - GetEnumerator()
  - IEnumerator<T> : IDisposable, IEnumerator
    - Current,
    - MoveNext(), Reset()
  - IEqualityComparer<T>
    - Equals(), GetHashCode()
  - IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
    - Item[],
    - IndexOf(), Insert(), RemoveAt()

Typy ogólne – generics (7)

- Ogólne struktury
  - Dictionary.Enumerator
  - Dictionary.KeyCollection.Enumerator
  - Dictionary.ValueCollection.Enumerator
  - KeyValuePair
  - LinkedList.Enumerator
  - List.Enumerator
  - Queue.Enumerator
  - SortedDictionary.Enumerator
  - SortedDictionary.KeyCollection.Enumerator
  - SortedDictionary.ValueCollection.Enumerator
  - Stack.Enumerator

Typy ogólne – generics (8)

- Przykład użycia słownika:

```
1 Dictionary<string, int> wzrost = new Dictionary<string, int>();
2 wzrost.Add("Mariusz", 197);
3 wzrost.Add("Marcin", 211);
4 Console.WriteLine("Mariusz ma {0} cm wzrostu, a Marcin {1}",
5       wzrost["Mariusz"], wzrost["Marcin"]);
```

Typy ogólne – generics (9)

- Definiowanie klas ogólnych:

```
1 public class Tablica<T>
2 {
3    T[] tablica;
4    Tablica(int n) {
5       tablica = new T[n];
6    }
7    T this[int i] { return tablica[i]; }
8    ...
9 }
```

Typy ogólne – generics (10)

- Założenia co do typów-parametrów:

```
1  public class Dictionary<K,V>
2        where K: IComparable
3  {
4     public void Add(K key, V value)
5     {
6        ...
7        if (key.CompareTo(x) < 0) {...}
8        ...
9     }
10 }
```

Typy ogólne – generics (11)

- Metody ogólne

```
1 public static swap<T>(ref T x, ref T y)
2 {
3     T z = x;
4     x = y;
5     y = z;
6 }
```

Typy ogólne – generics (12)

- Założenia co do typów-parametrów metod:

```
1 public static T Min<T>(params T[] tab)
2             where T : IComparable<T>
3 {
4    T min = tab[0];
5    for (int i=0; i<tab.Length; i++)
6       if (tab[i].CompareTo(min) < 0)
7          min = tab[i];
8    return min;
9 }
```

Interfejsy i klasy ogólne (ang. *generics*)

- Interfejsy zwykłe i ogólne – przykład IComparable

```
1  namespace System
2  {
3      public interface IComparable
4      {
5          int CompareTo(object obj);
6      }
7
8      public interface IComparable<T>
9      {
10         int CompareTo(T other);
11     }
12 }
```

Interfejsy i klasy ogólne – c.d.

Klasa implementująca oba interfejsy (zwykły i ogólny):

```
1  public class Ułamek : IComparable, IComparable<Ułamek> {
2      public int Licznik { get; private set; }
3      public uint Mianownik { get; private set; }
4      public int CompareTo(object obj)
5      {
6          return CompareTo((Ułamek) obj);
7      }
8      public int CompareTo(Ułamek other)
9      {
10         double d1 = (double)Licznik/Mianownik;
11         double d2 = (double)other.Licznik/other.Mianownik;
12         if (d1 == d2)
13             return 0;
14         return d1 < d2 ? −1 : 1;
15     }
16 }
```

Wyjątki (1)

- Schemat:

```
1  try {
2    ...
3  }
4  catch (ExceptionA e) {
5    ...
6  }
7  catch (ExceptionB e) {
8    ...
9  }
10 finally {
11   ...
12 }
```

- Musi wystąpić przynajmniej jedna sekcja **catch** lub **finally**.
- **catch** i **finally** się wzajemnie nie wykluczają jak w C++.
- Tylko pierwsza pasująca sekcja **catch** jest wykonywana.

Wyjątki (2)

- Klasy wyjątków muszą dziedziczyć po System.Exception.
- Klasa System.Exception zawiera m.in. własności:
  - Message – opis wyjątku,
  - StackTrace – stos wywołań,
  - TargetSite – metoda, która zgłosiła wyjątek,
  - Data – słownik z dodatkowymi informacjami o wyjątku.
- Przykład:

```
1   public class WeightCalculator
2   {
3       public static float CalcBMI(float kilos, float meters) {
4           if (meters < 0 || meters > 3)
5               throw new ArgumentException("Impossible Height", "meters");
6           if (kilos < 0 || kilos > 1000)
7               throw new ArgumentException("Impossible Weight", "kilos");
8           return kilos / (meters * meters);
9       }
10  }
```

Wyjątki (3)

```
11  class Test {
12    static void Main () {
13      TestIt ();
14    }
15    static void TestIt () {
16      try {
17        float bmi = WeightCalculator.CalcBMI(100, 5);
18        Console.WriteLine(bmi);
19      }
20      catch(ArgumentException ex) {
21        Console.WriteLine(ex);
22      }
23      finally {
24        Console.WriteLine("Thanks for running the program");
25      }
26      Console.Read();
27    }
28  }
```

Tryb niechroniony - unsafe (1)

- Metody i bloki niechronione oznacza się słowem kluczowym **unsafe**.
- Używanie wskaźników możliwe jest tylko w trybie niechronionym. Konieczne jest wówczas „przyszpilowanie” obiektu deklaracją **fixed**.
- Przykład:

```
1 unsafe void RedFilter(int[,] bitmap) {
2    const int length = bitmap.Length;
3    fixed (int* b = bitmap) {
4       int* p = b;
5       for(int i = 0; i < length; i++)
6          *p++ &= 0xFF;
7    }
8 }
```

Tryb niechroniony - unsafe (2)

- Przykład mnożenia macierzy:

```
1  public static Macierz operator*(Macierz m1, Macierz m2) {
2    Macierz m3 = new Macierz(m1.LWierszy, m2.LKolumn);
3    unsafe {
4      fixed (double* pocz1 = m1.wartości) {
5        fixed (double* pocz2 = m2.wartości) {
6          fixed (double* pocz3 = m3.wartości) {
7            double *p1 = pocz1,
8              koniec1 = pocz1+m1.LWierszy*m1.LKolumn;
9            while (p1 < koniec1)
10           {
11             ...
12           }
13         }
14       }
15     }
16   }
17 }
```

Zestawy (ang. assembly)

- Nowe podejście do bibliotek alokowanych dynamicznie.
- Różne wersje tej samej biblioteki mogą równolegle funkcjonować w systemie.
- Zestaw może stanowić pojedynczy moduł (plik .dll lub .exe), albo kilka modułów (słabe wsparcie).
- Nie mylić z przestrzeniami nazw (ang. namespaces), choć często ta sama nazwa jest użyta i dla zestawu i przestrzeni nazw.
- Silne nazwy (strong names) – unikalne, generowane przy użyciu klucza prywatnego
- Global Assembly Cache – magazyn zestawów

Domeny aplikacji (application domains) (1)

- **Proces** = uruchomiona aplikacja
- System **chroni** procesy przed innymi procesami
  - osobne przestrzenie danych
  - niezależna ochrona, awaryjne zamykanie procesów itp.
- **Domeny aplikacji**
  - pozwalają na **podobną ochronę** (jak międzyprocesowa) w ramach jednego procesu
  - wymagają mechanizmów wymiany danych podobnych do tych dla osobnych procesów, ale **łatwiejszych** w użyciu
  - wymagają niezależnego **ładowania zestawów**, ale nie przełączania procesów
  - błędy w jednej domenie **nie „psują”** innych domen
- Proces w .NET startuje **z jedną domyślną domeną**, dodatkowe można tworzyć samemu w miarę potrzeby.
- **Domeny a wątki**
  - wiele wątków może biec w ramach jednej domeny
  - wątek może chodzić w ramach różnych domen (nie jednocześnie)

Domeny aplikacji (application domains) (2)

Główne składowe klasy AppDomain:

- CurrentDomain – **statyczna** własność, zwraca **bieżącą domenę** dla bieżącego wątku
- CreateDomain() – tworzy **nową domenę**
- GetCurrentThreadID() – identyfikator bieżącego **wątku**
- Load() – **ładuje** zestaw do domeny
- Unload() – **usuwa** zadaną domenę
- CreateInstance() – **tworzy obiekt** w domenie
- CreateInstanceAndUnwrap() – tworzy obiekt w domenie i woła ObjectHandle.Unwrap()

Domeny aplikacji (application domains) (3)

Przykład:

```
1  AppDomain ad2 = AppDomain.CreateDomain("Shape Domain");
2  ObjectHandle oh = ad2.CreateInstance(
3     "ProgCSharp", // the assembly name
4     "ProgCSharp.Shape", // the type name with namespace
5     false, // ignore case
6     System.Reflection.BindingFlags.CreateInstance, // flag
7     null, // binder
8     new object[] {3, 5}, // args
9     null, // culture
10    null, // activation attributes
11    null ); // security attributes
```

Domeny aplikacji (application domains) (4)

Klasa ObjectHandle

- **„opakowanie" obiektów** (dokładniej: ich „pełnomocników" – ang. ***proxy***) na podróż pomiędzy domenami
- nie wymaga ładowania **meta-danych** opakowanego obiektu na użytek podróży
- daje kontrolę nad **ładowaniem** niezbędnych **zestawów**

Podstawowe biblioteki .NET

- Klasy String i StringBuilder
- Struktury DateTime i TimeSpan
- Kolekcje: List, LinkedList, Queue, Stack, Dictionary, SortedDictionary, ...
- Klasa Math
- Klasa Random
- Klasy TCPClient, TCPListener, NetworkStream, Socket

Atrybuty (1)

- Służą do opisywania elementów składniowych: zestaw, klasa, konstruktor, delegat, typ wyliczeniowy, zdarzenie, pole, interfejs, metoda, moduł, parametr, własność, wartość zwracana, struktura
- Meta-dane o klasach, polach itd.
- Przykłady:

```
1 [assembly: AssemblyKeyFile("c:\\myStrongName.key")]
2 [NoIDispatch]
3 public interface IEksperymentCOM {...}
4 [Serializable]
5 class MySerializableClass { ...}
```

- Wstawia się je bezpośrednio przed deklaracją, której dotyczą, z wyjątkiem tych dotyczących zestawów i modułów.
- Atrybuty wewnętrzne – wsparcie Common Language Runtime, zintegrowane z .NET

Atrybuty (2)

- Atrybuty użytkownika – klasy dziedziczące po System.Attribute

```
1  public class MachineAttribute : Attribute
2  {
3      public string name;
4      public Type configType;
5      public MachineAttribute(string name, Type configType)
6      {
7          this.name = name;
8          this.configType = configType;
9      }
10 }
11
12 [Machine("Decision tree", typeof(DTConfig))]
13 public class DT : DTClassifier, IMachine
14 { ... }
```

Atrybuty (3)

- Konwencja wspierana przez kompilator: nazwy klas kończą się na Attribute
- Koniecznie przynajmniej jeden konstruktor.
- Parametry pozycyjne (argumenty konstruktora)
- Parametry nazwane (własności)

```
1 [Machine("Decision tree", typeof(DTConfig),
2    Description="A standard greedy search decision tree")]
```

- Atrybut atrybutów (meta-atrybut)

```
1 [AttributeUsage(AttributeTargets.Class,
2    AllowMultiple = false)]
3 public class MachineAttribute : Attribute
4 { ... }
```

- Wiele atrybutów można wymienić w jednym nawiasie:

```
1 [Serializable, Machine(...)]
```

Mechanizm refleksji (reflection) (1)

- Dostęp do meta-danych.
- Duże możliwości penetrowania hierarchii klas, wnętrz klas itd.
- Klasa Type – meta-dane o typach i ich wnętrznościach
- Pobieranie informacji o typie

```
1 Type t = zmienna.GetType(); // metoda System.Object
2 Type t = Type.GetType("System.Int32");
3 Type t2 = Type.GetType("MyNamespace.MyType", MyAssembly);
4 Type t3 = typeof(System.Int32);
```

- Przykład penetracji typów na kolejnej stronie…

Mechanizm refleksji (reflection) (2)

```
1  using System;
2  using System.Reflection;
3  class Test {
4     static void Main( ) {
5        DumpTypeInfo((new Object()).GetType( ));
6        DumpTypeInfo(typeof(int));
7        DumpTypeInfo(Type.GetType("System.String"));
8     }
9     static void DumpTypeInfo(Type t) {
10       Console.WriteLine("Type: {0}", t);
11       // Retrieve the list of members in the type
12       MemberInfo[ ] miarr = t.GetMembers( );
13       // Print out details on each of them
14       foreach (MemberInfo mi in miarr)
15          Console.WriteLine(" {0} {1}", mi.MemberType, mi);
16    }
17 }
```

Mechanizm refleksji (reflection) (3)

```
1  void SearchInterfaces(Assembly a, Type theInterface)
2  {
3     Type[] types = a.GetTypes();
4     foreach (Type t in types)
5     {
6        if (t.GetInterface(theInterface.FullName) != null)
7        { /* ... */ }
8     }
9  }
10 void SearchAttributes(Assembly a, Type attributeType)
11 {
12    Type[] types = a.GetTypes();
13    foreach (Type t in types)
14    {
15       if (Attribute.GetCustomAttribute(t, attributeType) != null)
16       { /* ... */ }
17    }
18 }
```

Mechanizm refleksji (reflection) (4)

- Wybrane klasy przestrzeni nazw System.Reflection:
  - Assembly
  - Module
  - ConstructorInfo
  - MethodInfo
  - FieldInfo
  - EventInfo
  - PropertyInfo
  - ParameterInfo
  - CustomAttributeData
  - MethodBody
- Inne możliwości mechanizmów refleksji:
  - *late binding* – ładowanie bibliotek, tworzenie obiektów i wołanie metod na etapie działania programu – Uwaga! znacznie wolniej!
  - Modyfikacja pól prywatnych.
  - Tworzenie typów na etapie działania programu (System.Reflection.Emit), klasy:
    - AssemblyBuilder – dynamiczne tworzenie zestawów,
    - ModuleBuilder – dynamiczne tworzenie modułów,
    - TypeBuilder – dynamiczne tworzenie typów,
    - ILGenerator – dynamiczne tworzenie kodu MSIL.

Delegaty/delegacje (1)

Delegaty to kolekcje referencji do metod (związanych z konkretnym obiektem lub statycznych)

- przykład typu na użytek obsługi zdarzeń alarmowych:
  delegate void AlarmHandler(**object** sender, AlarmEventArgs e);
- własność dla obsługi alarmu i obsługa zdarzenia:

```
1 public AlarmHandler OnDoorOpen { get; set; }
2 void SprawdźAlarmy()
3 {
4    ...
5    if (door.State == State.Open && OnDoorOpen != null)
6       OnDoorOpen(this, null);
7 }
```

Delegaty/delegacje (2)

- Sygnatury metod – typy do przechowywania i wołania metod (pojedyńczych bądź list metod) o określonej liście parametrów

```
1  delegate Figura Transformacja(Figura f);
2  class KolekcjaFigur {
3     Figura[] figury;
4     KolekcjaFigur Transformuj(Transformacja t)
5     {
6        KolekcjaFigur kf = new KolekcjaFigur(liczbaFigur);
7        foreach (Figura f in Figury)
8           kf.Dodaj(t(f));
9        return kf;
10    }
11 }
```

Delegaty/delegacje (3)

- Argumenty mogą być specyfikowane z modyfikatorem **params**

```
1  delegate int Kombinacja(params int[] arg);
2  class Operacje
3  {
4     static public int Suma(params int[] argumenty) {
5        int wynik = 0;
6        foreach (int i in argumenty)
7           wynik += i;
8        return wynik;
9     }
10    static public void Main() {
11       Kombinacja k = new Kombinacja(Suma);
12       System.Console.WriteLine("Wynik = {0}", k(1, 2, 3, 4));
13    }
14 }
```

Delegaty/delegacje (4)

- Operatory += i −= do obsługi delegatów przechowujących listy metod

```
1  delegate int Kombinacja(params int[] arg);
2  class Operacje
3  {
4     static public int Suma(params int[] argumenty) { ... }
5     static public int Iloczyn(params int[] argumenty) { ... }
6     static public void Main() {
7        Kombinacja k = new Kombinacja(Suma);
8        k += Iloczyn;
9        //k -= Suma;
10       System.Console.WriteLine("Wynik = {0}", k(1, 2, 3, 4));
11    }
12 }
```

  - Kolejność wołania taka jak kolejność dodawania.
  - Delegat zwraca wynik ostatniej metody z listy.

Delegaty/delegacje (5)

Gotowe rozwiązania systemowe:

- Typy System.EventHandler i System.EventArgs:
  delegate void EventHandler(**object** sender, EventArgs e);
- Statyczna wartość EventArgs.Empty
- W EventArgs nie ma miejsca na dane opisujące zdarzenie. W razie potrzeby tworzy się klasy dziedziczące po EventArgs.
- klasy Delegate i MulticastDelegate
  - **public virtual** Delegate[] GetInvocationList()
  - metody Combine() i Remove()
  - kompilator tworzy obiekty dziedziczące przy deklaracjach delegatów
  - użytkownik nie może tworzyć klas dziedziczących w sposób jawny
  - metody BeginInvoke() i EndInvoke() są dodawane automatycznie przez kompilator

Zdarzenia (1)

Zdarzenia to efektywny środek kontroli złożonych środowisk

Interfejsy graficzne – typowy przykład

- każdy element wizualny to obiekt, który może **generować** zdarzenia, a także **odpowiadać** na zdarzenia generowane przez
  - system
  - użytkownika
  - inne elementy środowiska
- zdarzenia realizuje się **delegatami** a odpowiadanie na zdarzenia metodami przypisywanymi do tych delegatów
- możliwe są **złożone reakcje** na pojedyncze zdarzenie (wiele różnych reakcji innych komponentów)

Zdarzenia (2)

- Delegaty – doskonały środek do realizacji obsługi zdarzeń.
- Słowo kluczowe **event**
  - oznacza delegaty, które mogą być wywoływane **tylko wewnątrz klasy**
  - z zewnątrz można się do nich podłączać
- Konwencja: pierwszy argument to źródło, drugi to parametry zdarzenia (dziedziczy po System.EventArgs):

```
1 delegate void MoveEventHandler(object source,
2               MoveEventArgs e);
```

Zdarzenia (3)

- Przykład parametrów zdarzenia:

```
1 public class MoveEventArgs : EventArgs
2 {
3     public int newPosition;
4     public bool cancel;
5     public MoveEventArgs(int newPosition)
6     {
7         this.newPosition = newPosition;
8     }
9 }
```

Zdarzenia (4)

- Przykład uruchomienia zdarzenia:

```
1  class Slider {
2     int position;
3     public event MoveEventHandler Move;
4     public int Position {
5        get { return position; }
6        set {
7              if (Move != null) { // if invocation list not empty
8                 MoveEventArgs args = new MoveEventArgs(value);
9                 Move(this, args); // fire event
10                if (args.cancel) return;
11             }
12             position = value;
13       }
14    }
15 }
```

Aplikacje okienkowe dla Windows (1)

- Windows Forms
  - klasy w częściach
  - liczne komponenty
  - edytor własności
  - definiowanie poszczególnych funkcji komponentów
- Własne komponenty
- Kody źródłowe
- WPF, Silverlight, UWP, Xamarin, MAUI

Proste programy

- okienko ze wskaźnikiem postępu i zadaniem w tle
- wizualizacja problemu Peer z zadania ACM z 2008 r.
- program z wizualizacją stanu domu (światła, brama, ogrzewanie)

Aplikacje okienkowe dla Windows (2)

Wielowątkowość aplikacji okienkowych

- dodatkowe wątki potrzebne, by nie blokować interfejsu podczas długotrwałych operacji
- wywołania asynchroniczne
- klasa Thread – bezpośrednia obsługa wątków
- klasa ThreadPool
- klasa Task
- modyfikacje okien możliwe tylko z jednego wątku
- metoda Invoke()
- klasa BackgroundWorker

Asynchroniczne wywołania metod (1)

- Normalne wywołania są **synchroniczne** (uruchomienie funkcji i kontynuacja po jej zakończeniu)
- Wywołanie **asynchroniczne** – zlecenie uruchomienia w tle, możliwość informacji zwrotnej na koniec (ang. *callback*)
- Metody BeginXYZ() i EndXYZ()
- Metody BeginInvoke() i EndInvoke() są dodawane automatycznie przez kompilator przy tworzeniu delegatów
- BeginInvoke() ma argumenty takie jak delegat plus metoda dla informacji zwrotnej **public delegate void** AsyncCallback(IAsyncResult ar) i obiekt z parametrami
- Interfejs IAsyncResult – interfejs obiektów zwracanych przez BeginXYZ() – pozwala monitorować postępy, przekazywać dodatkowe dane itp.

Asynchroniczne wywołania metod (2)

Przykład:

```
1  public class AsyncDemo
2  {
3      public string TestMethod(int callDuration, out int threadId)
4      {
5          Console.WriteLine("Test method begins.");
6          Thread.Sleep(callDuration);
7          threadId = Thread.CurrentThread.ManagedThreadId;
8          return String.Format("My call time was {0}.", callDuration);
9      }
10 }
11
12 public delegate string AsyncCaller(int callDuration, out int threadId);
13
14 public class AsyncMain
15 {
16     static int threadId;
```

Asynchroniczne wywołania metod (3)

```
17      public static void Main()
18      {
19          AsyncDemo ad = new AsyncDemo();
20          AsyncCaller caller = new AsyncCaller(ad.TestMethod);
21          IAsyncResult result = caller.BeginInvoke(3000, out threadId,
22              new AsyncCallback(CallbackMethod), caller);
23          Console.WriteLine("Press Enter to close application.");
24          Console.ReadLine();
25      }
26      static void CallbackMethod(IAsyncResult ar)
27      {
28          AsyncCaller caller = (AsyncCaller)ar.AsyncState;
29          string returnValue = caller.EndInvoke(out threadId, ar);
30          Console.WriteLine("Thread {0}, return value \"{1}\".",
31              threadId, returnValue);
32      }
33  }
```

Obsługa wątków

Wątki a procesy:

- wątki działają w ramach wspólnego procesu, ze wspólną pamięcią (dostęp do wszystkich zmiennych procesu)
- ochrona ze strony systemu operacyjnego na poziomie procesów

Wybrane elementy środowiska .Net:

- Przestrzeń nazw System.Threading
- Klasa Thread
- **lock** i klasa Monitor
- Klasa ReaderWriterLock
- klasa ThreadPool
- klasa Task
- Klasa BackgroundWorker

Klasa Thread

Thread – łatwe tworzenie i kontrola wątków

Podstawowe składowe:

- **delegate void** ThreadStart()
- **delegate void** ParameterizedThreadStart(**object** o)
- Start()
- ThreadState – bieżący stan wątku
- Join() – wstrzymuje bieżący wątek do zakończenia innego
- Interrupt() – przerywa gdy wątek w stanie WaitSleepJoin,
- Abort(), ResetAbort()
- Suspend(), Resume() – trzeba ostrożnie!
- **static** Thread CurrentThread
- IsAlive, IsBackground
- Name, Priority

Prosty przykład użycia klasy Thread

```
1  void Obliczenia()
2  {
3      List<Figura> figury = new List<Figura>;
4      List<Kolor> kolory = new List<Kolor>;
5      while (!koniec)
6      {
7          ...
8          figury.Add(...);
9          kolory.Add(...);
10     }
11     // sygnalizacja zakończenia obliczeń
12 }
13 void Licz() // np. odpowiedź na kliknięcie przycisku
14 {
15     Thread t = new Thread(Obliczenia);
16     t.Start();
17 }
```

Metody anonimowe (1)

- Można definiować metody „w locie” bez potrzeby ich nazywania, określania listy argumentów itp.
- Przykład:

```
1 double[] tablica;
2 ...
3 Thread w = new Thread(delegate()
4 {
5    new QuickSort().Sort(tablica);
6 });
7 w.Start();
8 ...
9 w.Join();
```

Metody anonimowe (2)

- delegaty z argumentami też można w ten sposób definiować

```
1  double[] tablica;
2  ...
3  ParameterizedThreadStart sortowanie = delegate(object o)
4  {
5     ((ISortMethod)o).Sort(tablica);
6  }
7  Thread w = new Thread(sortowanie);
8  w.Start(new QuickSort());
9  ...
10 w.Join();
```

Sekcje krytyczne – wyrażenie lock

```
1  List<Figura> figury = new List<Figura>();
2  List<Kolor> kolory = new List<Kolor>();
3  object locker = new object();
4  void Obliczenia()
5  {
6      lock (locker) {
7        figury.Clear();
8        kolory.Clear();
9      }
10     while (!koniec) {
11       ...
12       lock (locker) {
13         figury.Add(...);
14         kolory.Add(...);
15       }
16     }
17     // sygnalizacja zakończenia obliczeń
18 }
```

Klasa Monitor

Synchronizacja dostępu do danych (**lock** jest realizowany właśnie poprzez wykorzystanie tej klasy)

Podstawowe **składowe**:

- Enter(), TryEnter() – początek sekcji krytycznej (*critical section*)
- Exit() – koniec sekcji krytycznej
- Wait() – zwalnia blokadę i czeka na sygnał (Pulse)
- Pulse(), PulseAll() – wysyłają sygnały do jednego wątku / wszystkich wątków oczekujących takiego sygnału

Klasa ReaderWriterLock

Wiele wątków może jednocześnie czytać, tylko jeden – pisać (gdy jeden zapisuje inne nie mogą również czytać)

- metoda szybsza niż użycie klasy Monitor, gdy można pozwalać na jednoczesny odczyt wielu wątkom

Podstawowe **składowe**:

- AcquireReaderLock()
- AcquireWriterLock()
- DowngradeFromWriterLock(), UpgradeToWriterLock()
- ReleaseLock() – zwalnia blokadę niezależnie od liczby zablokowań
- ReleaseReaderLock(), ReleaseWriterLock – zmniejszają liczniki zablokowań
- RestoreLock() – przywraca sytuację sprzed wywołania ReleaseLock()
- WriterSeqNum, AnyWritersSince()
- IsReaderLockHeld, IsWriterLockHeld

Klasa ThreadPool

Statyczna klasa wspierająca „wątkowanie”, asynchroniczne wejście/wyjście, oczekiwanie na wątki itp.

- Metody QueueUserWorkItem()
- Metoda SetMaxThreads()
- Metody RegisterWaitForSingleObject() – uruchanianie opóźnione do czasu:
  - pojawienia się sygnału od skojarzonego obiektu
  - przeterminowania

Inne przydatne klasy:

- Klasa WaitHandle
- Klasa ThreadStaticAttribute

Zadania (klasa Task) (1)

- .NET wersja 4.0 – dodatkowe narzędzia do wielowątkowej obsługi zadań – przestrzeń nazw System.Threading.Tasks
- Schemat użycia test klasy Task

```
1 public static void TaskTest()
2 {
3       Task t = new Task(Obliczenia);
4       t.Start();
5       ...
6       t.Wait();
7       ...
8 }
```

Zadania (klasa Task) (2)

- Tworzenie i start w jednym: Task t = Task.Factory.StartNew(Zwiększaj);
- Inne przydatne klasy:
  - Task<TResult> i własność Result
  - TaskFactory i TaskFactory<TResult>
  - TaskScheduler
- Inne ciekawe zagadnienia
  - zadania kontynuacji (Task.ContinueWith())
  - przerywanie zadań (*task cancellation*), żetony przerywania
  - słowa kluczowe **async** i **await**

Metoda Invoke()

- Bezpośrednia modyfikacja komponentów graficznych jest możliwa tylko w ramach wątku, który stworzył ten komponent
- Wyjątek InvalidOperationException z komunikatem: *"Nieprawidłowa operacja między wątkami: do formantu ... uzyskiwany jest dostęp z wątku innego niż wątek, w którym został utworzony."*
- Każdy typ komponentów graficznych posiada metodę Invoke()
- Invoke() wywołuje wskazany delegat w odpowiednim wątku

Klasa BackgroundWorker

Najważniejsze składowe:

Zdarzenia:

- DoWork
- ProgressChanged
- RunWorkerCompleted

Metody:

- ReportProgress()
- RunWorkerAsync()
- CancelAsync()

Własności:

- CancellationPending
- WorkerReportsProgress
- WorkerSupportsCancellation

Programowanie statyczne a dynamiczne

Wiązanie (ang. *binding*) – proces określania typów, składowych i operacji

- **statyczne** – w trakcie kompilacji,
- **dynamiczne** – odroczone do czasu działania programu

Cechy dynamicznego wiązania:

- Przydatne, gdy **kompilator nie zna** typów czy metod, które będą dostępne w *runtime* (np. współpraca z językami dynamicznymi lub obiektami COM).
- Można wołać metody, których **istnienie trudno wykazać** kompilatorowi.
- Ewentualne **błędy** wyjdą dopiero **w *runtime***.
- Nie nadużywać! Dynamiczne wiązanie **spowalnia** działanie programu.

Programowanie dynamiczne (1)

- Typ **dynamic**

```
1  static void Main(string[] args) {
2        ExampleClass ec = new ExampleClass();
3        ec.exampleMethod1(10, 4); // błąd kompilatora
4        dynamic dynamic_ec = new ExampleClass();
5        dynamic_ec.exampleMethod1(10, 4); // błąd w runtime
6        dynamic_ec.someMethod("some argument", 7, null); // j.w.
7        dynamic_ec.nonexistentMethod(); // j.w.
8        dynamic_ec.exampleMethod1(104); // ok
9  }
10 class ExampleClass {
11    public ExampleClass() { }
12    public void exampleMethod1(int i) { }
13 }
```

Programowanie dynamiczne (2)

Inny przykład:

```
1 dynamic x = "hello";
2 Console.WriteLine (x.GetType().Name); // String
3 x = 123;
4 Console.WriteLine (x.GetType().Name); // Int32
```

var vs **dynamic**

- **var** to tylko pozostawienie kompilatorowi zadania określenia typu zmiennej – wiązanie statyczne

```
1 var x = "hello"; // to samo co string x = "hello";
2 Dictionary<Klucz,Wartość> słownik = ...
3 foreach (var paraKW in słownik)
4     ... // paraKW ma typ KeyValuePair<Klucz,Wartość>
```

- **dynamic** – określenie typu dopiero w *runtime*

Programowanie dynamiczne (3)

Dynamicznie **nie można wołać**:

- metod rozszerzających (wywołanych jak metody obiektu),
- składowych interfejsu, jeśli potrzebna jest jawna konwersja,
- ukrytych metod klas bazowych.

Wiązanie dynamiczne

- **użytkownika** – interfejs IDynamicMetaObjectProvider
- **językowe**

Programowanie dynamiczne (4)

Wiązanie użytkownika

```
1  static void Main()
2  {
3     dynamic d = new Duck();
4     d.Quack(); // Quack method was called
5     d.Waddle(); // Waddle method was called
6  }
7  public class Duck : DynamicObject
8  {
9     public override bool TryInvokeMember(
10    InvokeMemberBinder binder, object[] args, out object result)
11    {
12       Console.WriteLine(binder.Name + " method was called");
13       result = null;
14       return true;
15    }
16 }
```

Programowanie dynamiczne (5)

Klasa DynamicObject

- TryInvokeMember – metody
- TryGetMember, TrySetMember – własności lub pola
- TryGetIndex,TrySetIndex – operatory indeksujące
- TryUnaryOperation – operatory jednoargumentowe
- TryBinaryOperation – operatory dwuargumentowe
- TryConvert – operatory konwersji typów
- TryInvoke – operatory ()

Jeśli zwracają **false**, to próbowane są runtime'owe wiązania danego języka (zwykle skończy się wyjątkiem).

Programowanie dynamiczne (6)

Klasa ExpandoObject – typ dla obiektów, których składowe można dodawać i usuwać

```
1  dynamic employee = new ExpandoObject();
2  employee.Name = "John Smith";
3  employee.Age = 33;
4
5  foreach (var property in (IDictionary<String, Object>)employee)
6      Console.WriteLine(property.Key + ": " + property.Value);
7
8  ((IDictionary<String, Object>)employee).Remove("Name");
9  // pokaż ponownie
10 employee.Name = "Johnnie Walker";
11 // pokaż ponownie
```

Programowanie dynamiczne (7)

Wiązanie językowe

```
1 static dynamic Mean (dynamic x, dynamic y)
2 {
3   return (x + y) / 2;
4 }
5 static void Main()
6 {
7   int x = 3, y = 4;
8   Console.WriteLine (Mean (x, y));
9 }
```

Wyrażenia regularne (1)

- **Wyrażenia regularne** (ang. *regular expressions*) to bardzo przydatne narzędzie do przetwarzania tekstów
- Główne funkcje:
  - **wyszukiwanie** wzorców wewnątrz tekstów (z możliwością uwzględniania powtórzeń grup znaków)
  - **dzielenie** tekstów na części
  - **zastępowanie** fragmentów innymi
- System prawie identyczny ze stosowanym w Perlu 5

Wyrażenia regularne (2)

Wyrażenie regularne to napis składający się z **literałów i metaznaków**

- literały to znaki traktowane dosłownie
- metaznaki to symbole o specjalnych funkcjach np.:
  - `^` – początek napisu lub linii,
  - `$` – koniec napisu lub linii,
  - `+` – ciąg złożony z jednego lub więcej symboli występujących bezpośrednio przed,
  - `*` – jak +, ale 0 lub więcej symboli,
  - `|` – logiczna alternatywa,
  - `[]` – znak ze zbioru,
  - `.` – dowolny znak,
  - `\d` – dowolna cyfra, `\s` – dowolny odstęp itp.
  - `()` – nawiasy oznaczają grupy, które mogą być wyszukiwane, zastępowane itp.,
- Krótki przegląd np. w MSDN:
  `http://msdn.microsoft.com/en-us/library/az24scfc.aspx`

Wyrażenia regularne (3)

- Przestrzeń nazw System.Text.RegularExpressions
- Główne klasy: RegEx, MatchCollection, Match, GroupCollection, Group, CaptureCollection, Capture
- Klasa RegEx – podstawowa w systemie, wszelka analiza wyrażeń
  - metody **statyczne** i **lokalne** (konstruktor z wyrażeniem regularnym)
  - Escape() – ignorowanie metaznaków
  - Unescape() – odwrotnie…
  - Split() – podział napisu na tablicę napisów wg wzorca
  - Replace() – zastępowanie części zgodnych z wyrażeniem czym innym
  - IsMatch() – czy wyrażenie pasuje do napisu
  - Match() – wyszukiwanie pasującego fragmentu
  - Matches() – wyszukiwanie pasujących fragmentów

Wyrażenia regularne (4)

- Klasa MatchCollection – zawiera kolekcje wyników wyszukiwań z powtórzeniami
  - **int** Count – liczba dopasowań
  - Match Item – kolejne dopasowania
- Klasa Match – pojedyncze dopasowanie
  - GroupCollection Groups – odnalezione grupy (otoczone okrągłymi nawiasami w wyrażeniu)
  - CaptureCollection Captures – kolekcja wszystkich odnalezionych grup (również zagnieżdżonych)
  - **int** Index – pozycja pierwszego znaku dopasowania
  - **int** Length – liczba znaków dopasowania
  - **bool** Success – czy znaleziono
  - **string** Value – dopasowany podciąg znaków
  - NextMatch() – następne dopasowanie

Wyrażenia regularne (5)

- Klasa GroupCollection – kolekcja grup znalezionych w pojedynczym dopasowaniu
  - **int** Count – liczba grup
  - Group Item – kolejne grupy (wg indeksów bądź nazw)
- Klasa Group – odnaleziona grupa jako kolekcja obiektów typu Capture (wielość ze względu na kwantyfikatory)
  - CaptureCollection Captures – kolekcja odnalezionych napisów
  - **int** Index – pozycja pierwszego znaku dopasowania
  - **int** Length – liczba znaków dopasowania
  - **bool** Success – czy znaleziono
  - **string** Value – dopasowany podciąg znaków

Wyrażenia regularne (6)

- Klasa CaptureCollection – kolekcja dopasowanych napisów
  - **int** Count – liczba dopasowanych napisów
  - Capture Item – kolejne napisy
- Klasa Capture
  - **int** Index – pozycja pierwszego znaku dopasowania
  - **int** Length – liczba znaków dopasowania
  - **string** Value – dopasowany podciąg znaków

Zadanie: Na stronie `http://www.fizyka.umk.pl/wfaiis/?q=node/17` wyszukaj:

- wszystkie adresy emailowe
- adresy emailowe wszystkich Ann
- nazwiska wszystkich profesorów o parzystym numerze telefonu

LINQ – Language-Integrated Query (1)

Uwaga: Wymagany **.NET Framework 3.5** lub nowszy

Trzy części:

1. **Źródło danych**
2. **Definicja** zapytania
3. **Realizacja** zapytania

Realizacja zapytania dopiero **wtedy, gdy konieczne** (przy wyliczaniu, pytaniu o liczbę elementów w wynikowej kolekcji)

Obsługiwane źródła danych:

- obiekty implementujące IEnumerable<T> lub IEnumerable (przez rozszerzenie OfType<T>)
- bazy danych serwera SQL
- dokumenty XML
- bazy danych ADO.NET

LINQ – Language-Integrated Query (2)

Przykład:

```
1 int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };
2
3 var numQuery =
4       from num in numbers
5       where (num % 2) == 0
6       select num;
7
8 foreach (int num in numQuery)
9       Console.Write("{0,1} ", num);
```

LINQ – Language-Integrated Query (3)

Przykład z użyciem **metod rozszerzających**:

```
1 int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };
2
3 var numQuery =
4       numbers.Where(num => (num % 2) == 0);
5
6 foreach (int num in numQuery)
7       Console.Write("{0,1} ", num);
```

Bogata **dokumentacja** Microsoftu:

- LINQ
- Standard Query Operators Overview
- Słowa kluczowe zapytań
- Linq samples w dystrybucji Visual Studio
- White papers w dystrybucji Visual Studio 2008

LINQ – Language-Integrated Query (4)

Technologie wspierające LINQ (do wykorzystania nie tylko z LINQ):

- ukryte typy zmiennych (**var**)

```
1 var number = 5;
2 var name = "Virginia";
3 var query = from str in stringArray
4                 where str[0] == 'm'
5                 select str;
```

- łatwiejsza inicjalizacja obiektów i kolekcji

```
1 Customer cust =
2       new Customer { Name = "Mike", Phone = "555−1212" };
```

- anonimowe typy

```
1 select new {name = cust.Name, phone = cust.Phone};
```

LINQ – Language-Integrated Query (5)

- metody rozszerzające

```
1 public static class Enumerable {
2   public static IEnumerable<T> Where<T>(
3     this IEnumerable<T> source, Func<T, bool> predicate)
4   {
5       foreach (T item in source)
6         if (predicate(item))
7           yield return item;
8   }
9 }
```

LINQ – Language-Integrated Query (6)

- lambda wyrażenia i drzewa wyrażeń

```
1 Func<int, bool> f = n => n < 5;
2 Expression<Func<int, bool>> e = n => n < 5;
3
4 BinaryExpression body = (BinaryExpression)e.Body;
5 ParameterExpression left = (ParameterExpression)body.Left;
6 ConstantExpression right = (ConstantExpression)body.Right;
7 Console.WriteLine("{0} {1} {2}",
8       left.Name, body.NodeType, right.Value);
```

- automatyczne (auto-implementujące) własności

```
1 public string Name {get; set;}
```

- interfejs IQueryable<T>
  - wszystkie operatory korzystają z drzew wyrażeń
  - opóźnione wyliczanie (*deferred evaluation*) bardzo przydatne w rozwiązaniach typu LINQ to SQL

LINQ – Language-Integrated Query (7)

Klasa Enumerable i standardowe operatory zapytań (*Standard Query Operators*)

- **Where** Restriction operator based on predicate function
- **Select/SelectMany** Projection operators based on selector function
- **Take/Skip/TakeWhile/SkipWhile** Partitioning operators based on position or predicate function
- **Join/GroupJoin** Join operators based on key selector functions
- **Concat** Concatenation operator
- **OrderBy/ThenBy/OrderByDescending/ThenByDescending** Sorting operators sorting in ascending or descending order based on optional key selector and comparer functions
- **Reverse** Sorting operator reversing the order of a sequence
- **GroupBy** Grouping operator based on optional key selector and comparer functions
- **Distinct** Set operator removing duplicates
- **Union/Intersect** Set operators returning set union or intersection
- **Except** Set operator returning set difference

LINQ – Language-Integrated Query (8)

- **AsEnumerable** Conversion operator to IEnumerable<T>
- **ToArray/ToList** Conversion operator to array or List<T>
- **ToDictionary/ToLookup** Conversion operators to Dictionary<K,T> or Lookup<K,T> (multi-dictionary) based on key selector function
- **OfType/Cast** Conversion operators to IEnumerable<T> based on filtering by or conversion to type argument
- **SequenceEqual** Equality operator checking pairwise element equality
- **First/FirstOrDefault/Last/LastOrDefault/Single/SingleOrDefault** Element operators returning initial/final/only element based on optional predicate function
- **ElementAt/ElementAtOrDefault** Element operators returning element based on position
- **DefaultIfEmpty** Element operator replacing empty sequence with default-valued singleton sequence
- **Range** Generation operator returning numbers in a range
- **Repeat** Generation operator returning multiple occurrences of a given value

LINQ – Language-Integrated Query (9)

- **Empty** Generation operator returning an empty sequence
- **Any/All** Quantifier checking for existential or universal satisfaction of predicate function
- **Contains** Quantifier checking for presence of a given element
- **Count/LongCount** Aggregate operators counting elements based on optional predicate function
- **Sum/Min/Max/Average** Aggregate operators based on optional selector functions
- **Aggregate** Aggregate operator accumulating multiple values based on accumulation function and optional seed

LINQ – Language-Integrated Query (10)

Wyrażenia dynamiczne (*dynamic expressions*) pozwalają na konstruowanie i wykonywanie zapytań w czasie działania programu (podobnie jak w SQL).

- Przestrzeń nazw System.Linq.Dynamic
- Źródła w LinqSamples w Visual Studio – projekt DynamicQuery
- Metody DynamicExpression.ParseLambda i DynamicExpression.Parse – dynamiczna analiza napisów i tworzenie drzew wyrażeń

```
1 LambdaExpression e = DynamicExpression.ParseLambda(
2       typeof(Customer), typeof(bool),
3       "City = @0 and Orders.Count >= @1",
4       "London", 10);
```

LINQ – Language-Integrated Query (11)

- Metody DynamicExpression.CreateClass – dynamiczne tworzenie klas danych

```
1 DynamicProperty[] props = newDynamicProperty[] {
2       new DynamicProperty("Name", typeof(string)),
3       new DynamicProperty("Birthday", typeof(DateTime)) };
4 Type type = DynamicExpression.CreateClass(props);
5 object obj = Activator.CreateInstance(type);
6 t.GetProperty("Name").SetValue(obj, "Albert", null);
7 t.GetProperty("Birthday").SetValue(obj, new DateTime(1879, 3, 14), null);
8 Console.WriteLine(obj);
```

- Rozszerzenia IQueryable – klasa DynamicQueryable

C# 6 (1)

- interpolacja napisów
  - łatwiejsze i czytelniejsze formatowanie napisów
  - zagnieżdżone wyrażenia w napisie
  - znak $ poprzedza napis interpolowany

```
1 class Osoba
2 {
3     public string Imię { get; set; }
4     public string Nazwisko { get; set; }
5     public override string ToString() { return $"{Imię} {Nazwisko}"; }
6 }
```

  - możliwe bardziej złożone wywołania i formatowanie: $"Łączna kwota: {pozycje.Suma():F2}"

C# 6 (2)

- automatycznie implementowane własności **tylko do odczytu**

```
1  public class Prostokąt
2  {
3     public int Lewa { get; set; }
4     public int Góra { get; set; }
5     public int Szerokość { get; }
6     public int Wysokość { get; }
7     public Prostokąt(int szer, int wys)
8     {
9        Szerokość = szer;
10       Wysokość = wys;
11    }
12 }
```

- automatycznie implementowane własności już w wersji 3 – str. 51

C# 6 (3)

- Inicjalizacja automatycznych właściwości

```
public IList<float> Wartości { get; } = new List<float>();
```

- Metody definiowane lambda-wyrażeniami (ang. *Expression-bodied function members*)

```
public override string ToString() => $"{LastName}, {FirstName}";
public string FullName => $"{FirstName} {LastName}";
```

- Operatory warunkowe nieprzypisania (ang. *null-conditional operators*) ?. i ?[]
  - używane często razem z operatorem ??

```
string imię = osoba != null ? osoba.imię : null;
string imię = osoba?.Imię;
int? wiek = osoba?.Wiek;
decimal doWypłaty = pracownik?.Wynagrodzenie ?? 0;
```

C# 6 (4)

- Filtry wyjątków (ang. *exception filters*)

```
1 try { ... }
2 catch (Wyjątek w) when (w.Szczegół == −15) { ... }
```

- Wyrażenie nameof

```
1 if (IsNullOrWhiteSpace(lastName))
2       throw new ArgumentException(message: "Cannot be blank",
3                                   paramName: nameof(lastName));
```

- **await** można używać również w sekcjach **catch** i **finally**

C# 6 (5)

- Nowa syntaktyka dla inicjalizacji tablic asocjacyjnych:

```
private var webErrors = new Dictionary<int, string>
{
    [404] = "Page not Found",
    [302] = "Page moved, but left a forwarding address.",
    [500] = "The web server can't come out to play today."
};
```

Wcześniej tylko:

```
private var messages = new Dictionary<int, string>
{
    { 404, "Page not Found"},
    { 302, "Page moved, but left a forwarding address."},
    { 500, "The web server can't come out to play today."}
};
```

C# 7.0 (1)

- Deklaracje parametrów wyjściowych mogą deklarować zmienne

```
1 float maks = tablica.Maksimum(out int indeks);
2 if (int.TryParse(input, out int result))
3       Console.WriteLine(result);
4 else
5       Console.WriteLine("Could not parse input");
```

C# 7.0 (2)

- **Wsparcie języka dla krotek** (ang. *tuples*)
  - Typ Tuple z własnościami Item1, Item2, … był dostępny już wcześniej.
  - Teraz można więcej i prościej, np:

```
1 (string Imię, string Nazwisko) jb = ("Jan", "Bąk");
2 var jb2 = (Imię: "Jan Drugi", Nazwisko: "Bąk");
```

  - **Dekonstrukcja** krotki:

```
1 (int max, int min) = Range(numbers);
2 var (max, min) = Range(numbers);
```

  - Można **pomijać/ignorować elementy** krotek:

```
1 var (_, min) = Range(numbers);
```

    Podobnie można **pomijać wyjściowe argumenty metod**.

C# 7.0 (3)

- Można tworzyć dekonstruktory dla swoich typów:

```
1 public class Point
2 {
3     public double X { get; }
4     public double Y { get; }
5     public Point(double x, double y) => (X, Y) = (x, y);
6     public void Deconstruct(out double x, out double y) => (x, y) = (X, Y);
7 }
8 var p = new Point(3.14, 2.71);
9 (double X, double Y) = p;
```

C# 7.0 (4)

- Dopasowywanie do wzorców (ang. *Pattern Matching*)
  - dla operatora **is**:

```
1 if (input is int count)
2     sum += count;
```

  - dla instrukcji **switch**: typ wyrażenia może być dowolny; wcześniej tylko całkowite, wyliczeniowe, **string** lub Nullable<> od nich;

```
1 foreach (object i in sequence)
2     switch (i)
3     {
4         case 0: ...
5         case IEnumerable<int> childSequence: ...
6         case int n when n > 0: ...
7         case null: ...
8         default: ...
9     }
```

C# 7.0 (5)

- **Referencyjne** zmienne **lokalne** i wartości **zwracane**

```
1 public static ref int Find(int[,] matrix, Func<int, bool> predicate)
2 {
3     for (int i = 0; i < matrix.GetLength(0); i++)
4         for (int j = 0; j < matrix.GetLength(1); j++)
5             if (predicate(matrix[i, j]))
6                 return ref matrix[i, j];
7     throw new InvalidOperationException("Not found");
8 }
```

Wykorzystanie:

```
1 ref var item = ref MatrixSearch.Find(matrix, (val) => val == 42);
2 Console.WriteLine(item);
3 item = 24;
4 Console.WriteLine(matrix[4, 2]);
```

C# 7.0 (6)

- **Funkcje lokalne**
  - można definiować prywatne metody do wykorzystania tylko w jednej metodzie,
  - głównie z myślą o metodach z opóźnionym działaniem - iteratorach i asynchronicznych, by np. kontrola błędów odbyła się bezzwłocznie, a reszta klasycznie.
- Więcej **składowych definiowanych wyrażeniami** – wcześniej tylko metody i własności tylko do odczytu

```
1 public ExampleClass(string label) => this.Label = label;
2 ~ExampleClass() => Console.Error.WriteLine("Finalized!");
3 private string label;
4 public string Label
5 {
6     get => label;
7     set => this.label = value ?? "Default label";
8 }
```

C# 7.0 (7)

- **throw tworzy wyrażenia**, nie instrukcje – dzięki temu, może być wołany w wielu nowych miejscach.
- Nowy typ ValueTask<TResult> dla usprawnienia wywołań asynchronicznych – by nie alokować obiektów typu Task, gdy zwracamy wartość.
- Ulepszenia notacji stałych numerycznych: stałe binarne i separator cyfr.

```
1 public const int Sixteen = 0b0001_0000;
2 public const int ThirtyTwo = 0b0010_0000;
3 public const int SixtyFour = 0b0100_0000;
4 public const int OneHundredTwentyEight = 0b1000_0000;
5
6 public const long BillionsAndBillions = 100_000_000_000;
7 public const double AvogadroConstant = 6.022_140_857_747_474e23;
8 public const decimal GoldenRatio =
9   1.618_033_988_749_894_848_204_586_834_365_638_117_720_309_179M;
```

C# 7.1 (1)

- Asynchroniczna funkcja Main()
  - Main() może zwracać Task lub Task<> i być oznaczona jako asynchroniczna (**async**),
  - pozwala to korzystać z **await** w Main().

```
1 static async Task<int> Main()
2 {
3       return await DoAsyncWork();
4 }
```

zamiast

```
1 static int Main()
2 {
3       return DoAsyncWork().GetAwaiter().GetResult();
4 }
```

C# 7.1 (2)

- Usprawnienia w stosowaniu słowa **default**. Np. zamiast

```
1 Func<string, bool> whereClause = default(Func<string, bool>);
```

  można pisać

```
1 Func<string, bool> whereClause = default;
```

- **Dedukowanie nazw elementów krotek**. Np. zamiast

```
1 var pair = (count: count, label: label);
```

  wystarczy

```
1 var pair = (count, label);
```

- **Dopasowywanie do wzorców** z uwzględnieniem parametrów typów ogólnych, w wyrażeniach **is** oraz instrukcjach **switch**.

C# 7.2 (1)

- Techniki obsługi typów wartościowych używając **semantyki typów referencyjnych** czyli dalsza część powrotu do referencji rodem z C++.
  - Modyfikator **in** parametru metody – przekazanie przez referencję, parametr niemodyfikowany przez metodę.
  - Modyfikator **ref readonly** wartości zwracanej – wartość zwracana przez referencję, bez możliwości modyfikacji.
  - Deklaracja **readonly struct** – oznacza strukturę jako niezmienną, przekazywaną do metod jak parametr **in**.
  - Deklaracja **ref struct** zapewnia alokację na stosie ciągłych obszarów pamięci. W szczególności, takie struktury nie mogą być składowymi klas.

C# 7.2 (2)

- **Parametry nazwane** niekoniecznie na końcu. Mogą zastępować pozycyjne, jeśli są z nimi zgodne.
- **Podkreślnik** może rozpoczynać stałą numeryczną, np. 0b_0101_0101.
- **Nowy modyfiktor dostępu**: **private protected** – dostęp tylko w klasie i jej potomnych zdefiniowanych w tym samym zestawie.
- Wyrażenia warunkowe (?:) mogą zwracać referencje:

```
1 ref var r = ref (arr != null ? ref arr[0] : ref otherArr[0]);
```

C# 7.3 (1)

Zmiany dla sprawniejszego działania kodu chronionego:

- można używać pól **fixed** bez szpilowania,
- **można zmienić** wartość referencyjnej zmiennej lokalnej,
- można **używać inicjalizatorów** dla tablic alokowanych na stosie przez stackalloc,
- **fixed** akceptuje więcej typów,
- rozszerzone deklaracje **ograniczeń** dla parametrów typów ogólnych.

C# 7.3 (2)

Usprawnienia wcześniejszych funkcji:

- operatory == i != **dla krotek**,
- szersze możliwości deklaracji zmiennych przy **out** (inicjalizacja pól, własności, konstruktorów),
- **atrybuty** dla pól generowanych dla automatycznych własności,
- lepsze rozstrzyganie pomiędzy przeciążonymi metodami różniącymi się modyfikatorem **in**.

Nowe opcje kompilatora:

- `-publicsign` – przypisanie klucza publicznego kompilowanemu zestawowi,
- `-pathmap` – przemapowanie ścieżek do źródeł.

C# 8 (1)

- oficjalnie od września 2019
- **domyślne implementacje metod interfejsów**
  - więcej niż metody rozszerzające, bo pozwalają na nadpisywanie,
- nowe aspekty **dopasowywania wzorców** (ang. *pattern matching*): **switch**, property, tuple,

```
1  public enum Rainbow { Red, Orange, Yellow, Green, Blue, Indigo, Violet }
2  public static RGBColor FromRainbow(Rainbow colorBand) =>
3  colorBand switch
4  {
5      Rainbow.Red => new RGBColor(0xFF, 0x00, 0x00),
6      Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),
7      Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),
8      Rainbow.Green => new RGBColor(0x00, 0xFF, 0x00),
9      Rainbow.Blue => new RGBColor(0x00, 0x00, 0xFF),
10     Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),
11     Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),
12     _ => throw new ArgumentException(message: "invalid enum value", paramName: nameof(colorBand)),
13 };
```

C# 8 (2)

- deklaracje **using**

```
1  using var file = new System.IO.StreamWriter("file.txt");
2  ...
```

zamiast

```
1  using (var file = new System.IO.StreamWriter("file.txt"))
2  {
3      ...
4  }
```

  - wygodne w kontekście deklarowania innych zmiennych lokalnych,
- zerowalne typy referencyjne - (ang. *nullable reference types*)
  - domyślnie wyłączone,
  - po włączeniu nie można użyć referencji bez niezerowego przypisania,
- obiekty użyte w **using** mogą implementować IDisposable lub IAsyncDisposable,

C# 8 (3)

- Indeksy i zakresy
  - typy System.Index i System.Range,
  - operatory ^ i ..
  - przykłady: tab[^1] == tab[tab.Length−1], zakres = 1..4, tab[4..^2], tab[..4],
- operator przypisania warunkowego ??= (ang. *null-coalescing assignment*)
  - przypisanie tylko gdy wartość **null**,
- niezarządzane typy ogólne (ang. *unmanaged constructed types*)
  - niezarządzane, gdy wszystkie pola niezarządzane,
- dowolny porządek znaków $ i @ w interpolowanych napisach dosłownych,
- statyczne funkcje lokalne,
- Dispose() w **ref struct**,
- strumienie asynchroniczne i **async foreach**,
- stackalloc w wyrażeniach zagnieżdżonych.

C# 9 (1)

- Wydany razem z **.NET 5** – 10 listopada 2020
- **init** - akcesor **tylko do inicjalizacji**

```
1 public class Osoba {
2       public string Imię { get; init; }
3       public string Nazwisko { get; init; }
4 }
```

może **modyfikować pola tylko do odczytu**:

```
1 public class Osoba {
2       private readonly string nazwisko;
3       public string Nazwisko { get => nazwisko; init => nazwisko = value; }
4 }
```

C# 9 (2)

Rekordy

```
1 public record Osoba {
2       public string Imię { get; init; }
3       public string Nazwisko { get; init; }
4 }
5 public record Osoba(string Imię, string Nazwisko);
```

- Typy referencyjne
- Zwykle niezmienne obiekty, ale niekoniecznie
- Porównywanie jak dla struktur
- Metoda ToString() stosująca format

  ```
  Osoba { Imię = iii, Nazwisko = nnn }
  ```

- **Syntaktyka pozycyjna** $\Longrightarrow$ właściwości „init-only”, konstruktor z prametrami, metoda Deconstruct
- Wyrażenia **with** (*nondestructive mutation*)

```
1 Osoba o2 = o1 with { Imię = "Jan" };
2 Osoba o3 = o1 with { DataUrodzenia = new DateTime(2000,1,1) };
```

- Rekordy mogą **dziedziczyć** po innych rekordach, nie po klasach ani klasy po rekordach.

C# 9 (3)

- Rozszerzenia **dopasowywania wzorców**

```
1  if (x is not null) { }
2
3  public static bool IsLetter(this char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z');
4
5  static string GetCalendarSeason(DateTime date) => date.Month switch {
6      3 or 4 or 5 => "spring",
7      >= 6 and <= 8 => "summer",
8      9 or 10 or 11 => "autumn",
9      12 or (>= 1 and < 3) => "winter",
10     _ => throw new ArgumentOutOfRangeException(nameof(date), $"Month error: {date.Month}."),
11 };
12
13 public record Point(int X, int Y);
14 static Point Transform(Point point) => point switch {
15     var (x, y) when x < y => new Point(-x, y),
16     var (x, y) when x > y => new Point(x, -y),
17     var (x, y) => new Point(x, y)
18 };
```

C# 9 (4)

- Instrukcje **poza klasami**
  - Tylko w jednym pliku
  - Zamiast funkcji Main() – błąd gdy obok niej
  - Pełny program może wyglądać tak:

```
1 System.Console.WriteLine("Hello World!");
```

- Usprawnienia na rzecz wydajności i zgodności
  - Natywne typy całkowite – nint, nuint
  - Wskaźniki do funkcji – **delegate***
  - SkipLocalsInitAttribute wyłącza flagę localsinit

C# 9 (5)

Różne drobiazgi

- Wywołania **new**() gdzie typ wynika z kontekstu
- Wyrażenia warunkowe ?: – **typy wyrażeń nie muszą być jednakowe** – wystarczy odpowiednia zgodność
- **Statyczne** funkcje anonimowe i lambda wyrażenia – ochrona przed niepożądanym przechwyceniem zmiennych
- **Kowariantność** typów zwracanych przez metody wirtualne – przy nadpisywaniu można zwracać typ potomny
- **foreach** akceptuje **rozszerzające metody** GetEnumerator
- Lambda wyrażenia akceptują **argumenty ignorowane** (ang. *discard* – _)
- Można stosować **atrybuty do funkcji lokalnych**

Wzorce projektowe (1)

Wzorce projektowe

- to **ogólne schematy rozwiązań** często spotykanych problemów architektury oprogramowania,
- to zestawy reguł opisujących **jak osiągnąć pewne cele** z dziedziny rozwoju oprogramowania,
- określają abstrakcyjne pojęcia/komponenty rozwiązań **na wyższym poziomie ogólności** niż konkretne klasy i obiekty.

Bardzo popularny przykład: **Model-Widok-Kontroler** (ang. MVC: Model-View-Controller)

- stworzony w 1979 roku dla języka Smalltalk (jako Model-View-Editor)
- używany głównie do interfejsów użytkownika (szczególnie MDI: Multiple Document Interface)

Wzorce projektowe (2)

Bibliografia:

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, „Inżynieria oprogramowania: Wzorce projektowe” WNT, Wyd. II, Warszawa, 2008
- Judith Bishop, „C# 3.0 Design Patterns”, O’Reilly, 2008
- James W. Cooper, „C# Design Patterns. A Tutorial”, Addison-Wesley, 2003
- Steven John Metsker, „C#. Wzorce projektowe”, Helion, 2005

Język UML (1)

Język **UML – Unified Modeling Language**

- **graficzne** przedstawienie związków pomiędzy fragmentami kodu
- **wiele wersji** języka
- **standard ISO** od 2000 roku
- oznaczenia są dość **intuicyjne**, więc nietrudno rozumieć schematy bez wyuczenia się szczegółów standardu na pamięć

Język UML (2)

Elementy języka UML:

- trzy rodzaje ramek
  - **klasy**
  - **interfejsy**
  - **pakiety**
- relacje obrazowane liniami i strzałkami:
  - **dziedziczenie** – trójkątny grot w kierunku klasy bazowej
  - **implementacja interfejsu** – przerywana linia i trójkątny grot w kierunku interfejsu
  - **wzajemny dostęp** pomiędzy komponentami – ciągła linia
  - **dostęp jednokierunkowy** – strzałka w kierunku udostępnionego komponentu
  - **agregacja** – „referencja" do zewnętrznego komponentu, który funkcjonuje niezależnie – linia zapoczątkowana pustym rombem
  - **posiadanie składnika**/zależnego komponentu (ang. composition) – linia zapoczątkowana pełnym rombem

Systematyka wzorców projektowych

Podział wg typu relacji pomiędzy kluczowymi komponentami

- **strukturalne** (ang. structural)
- **kreacyjne** (ang. creational)
- **czynnościowe** (ang. behavioral)

Podział wg rodzaju komponentów

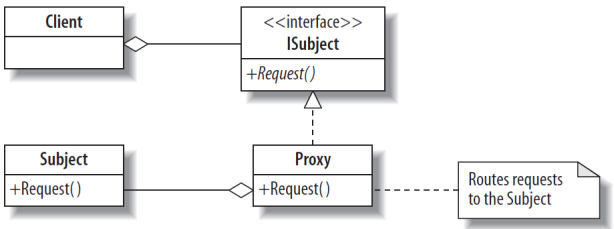
- **klasowe**
- **obiektowe**

Wzorzec *pośrednik* (ang. proxy)

Klasyfikacja: **strukturalny**, **obiektowy**

Cele:

- kontrola tworzenia i dostępu do obiektów
- pośrednik to mały publiczny obiekt, który pełni rolę właściwego dużego prywatnego obiektu



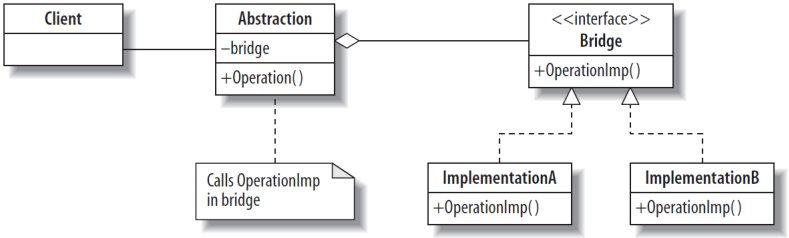
Przykład: system stron internetowych wymagający rejestracji i logowania w celu dostępu do odpowiednich stron (obiektów)

Wzorzec *most* (ang. bridge)

Klasyfikacja: **strukturalny**, **klasowy**

Cele:

- Wybór i przełączanie implementacji w trakcie działania programu
- Rozdział rozwoju interfejsu i jego implementacji



Przykłady:

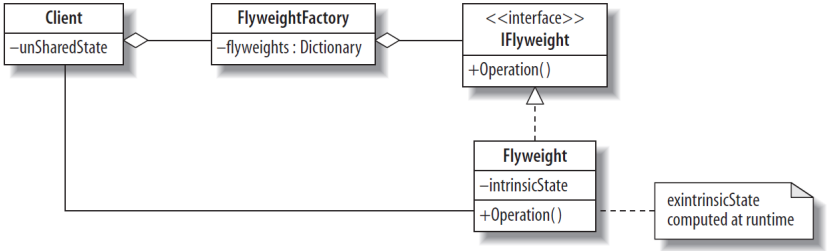
- w nowej wersji programu zmienia się pewna funkcjonalność, ale chcemy zachować i tę pierwotną
- ścieżka systemowa określa lokalizację biblioteki, której należy użyć

Wzorzec *waga musza* (ang. flyweight)

Klasyfikacja: **strukturalny**, **obiektowy**

Cele:

- obsługa wielu małych obiektów
- redukcja pamięci przez unikanie zduplikowanych obiektów



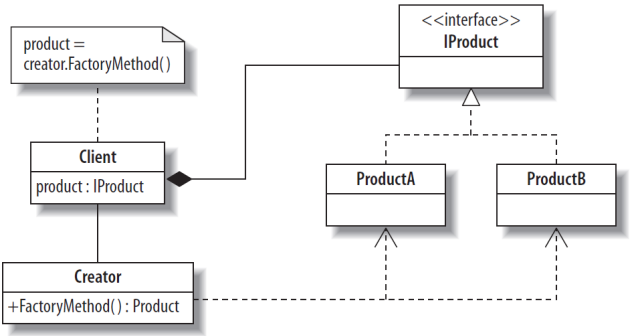
Przykład: biblioteka obrazów, z których każdy może należeć do wielu grup

Wz. *metoda wytwórcza* (ang. factory method)

Klasyfikacja: **kreacyjny**, **klasowy**

Cele:

- udostępnienie interfejsu do tworzenia obiektów (konkretne klasy implementują interfejs)



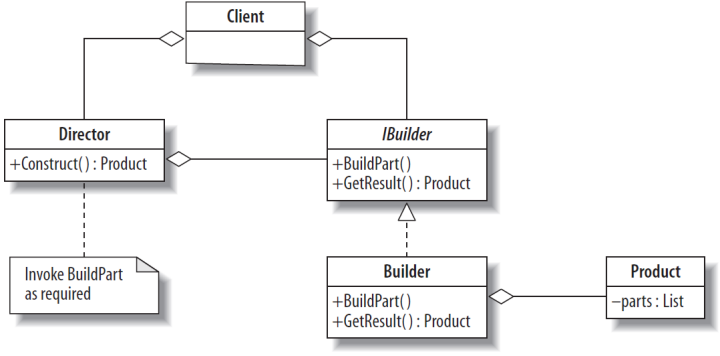
Przykład: dostawca owoców awokado importuje z różnych miejsc

Wzorzec *budowniczy* (ang. builder)

Klasyfikacja: **kreacyjny**, **obiektowy**

Cele:

- rozdział specyfikacji złożonego obiektu i jego tworzenia
- klient zleca zadanie nadzorcy i od niego odbiera produkt



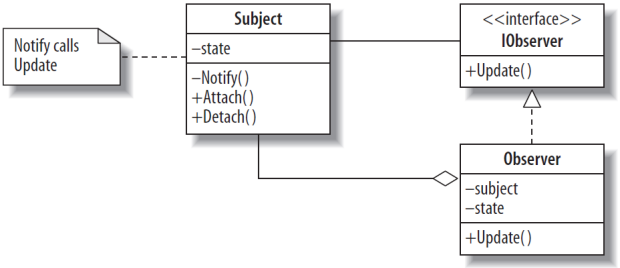
Przykład: produkcja torebek z różnych komponentów różnych producentów

Wzorzec *obserwator* (ang. observer)

Klasyfikacja: **czynnościowy**, **obiektowy**

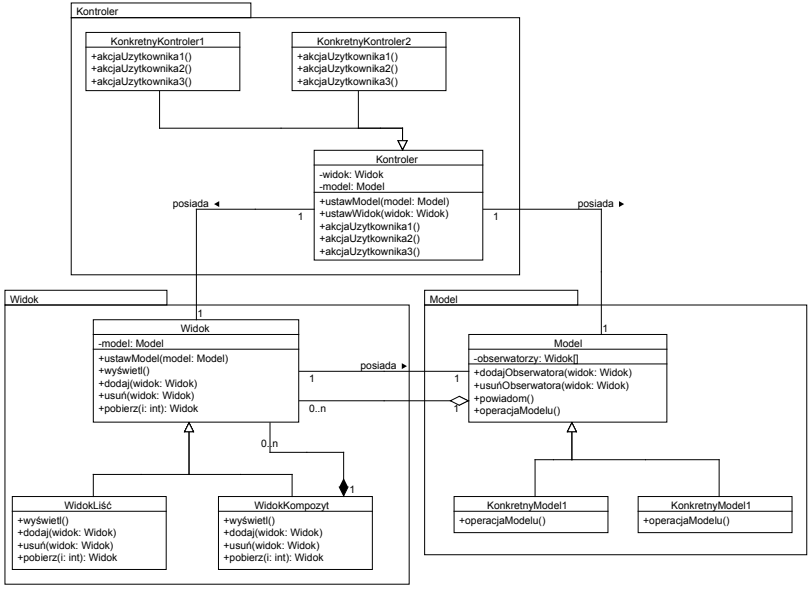
Cele:

- schemat relacji pomiędzy obiektami, w której zmiana jednego obiektu skutkuje przekazaniem informacji o zmianie wszystkim pozostałym obiektom



Przykład: środowiska sterowane zdarzeniami

MVC – złożony wzorzec projektowy



Strumienie (1)

- Abstrakcyjna klasa Stream i jej pochodne – niskopoziomowy zapis i odczyt
  - CanRead, CanWrite, CanSeek
  - Read(), Write() – binarny odczyt i zapis
  - metody Seek(), SetLength() i własności Position, Length
  - BeginRead(), EndRead(), BeginWrite(), EndWrite() – operacje asynchroniczne
  - Flush(), Close()

Strumienie (2)

- Klasy potomne:
  - Microsoft.JScript.COMCharStream
  - Microsoft.WindowsMobile.DirectX.GraphicsStream
  - System.IO.BufferedStream
  - System.IO.Compression.DeflateStream
  - System.IO.Compression.GZipStream
  - System.IO.FileStream
  - System.IO.MemoryStream
  - System.IO.UnmanagedMemoryStream
  - System.Net.Security.AuthenticatedStream
  - System.Net.Sockets.NetworkStream
  - System.Security.Cryptography.CryptoStream

Strumienie (3)

- Klasy usprawniające pisanie i czytanie
  - BinaryReader, BinaryWriter,
  - StreamReader, StreamWriter, Encoding,
  - StringReader, StringWriter,
  - TextReader, TextWriter – abstrakcyjne klasy bazowe dla StreamXXX i StringXXX.

Strumienie (4)

- Podstawowe klasy obsługi wejścia/wyjścia
  - Directory – metody statyczne
  - DirectoryInfo – metody w kontekście obiektu
  - DriveInfo – metody obsługi napędu
  - File – metody statyczne
  - FileInfo – metody w kontekście obiektu
  - FileSystemInfo – klasa abstrakcyjna, bazowa dla FileInfo i DirectoryInfo
  - Path – obsługa ścieżek (wieloplatformowa)
  - SerialPort – obsługa portów szeregowych
  - File, FileInfo, DriveInfo, Path, Directory, DirectoryInfo – sealed

Strumienie (5)

- Podstawowy przykład użycia strumieni:

```
1 using System.IO;
2 class StreamFun {
3    static void Main() {
4       Stream s = new FileStream("foo.txt", FileMode.Create);
5       s.WriteByte(67);
6       s.WriteByte(35);
7       s.Close();
8    }
9 }
```

Strumienie (6)

- Buforowanie odczytu i zapisu:

```
1  void Run( )
2  {
3      Stream inStream = File.OpenRead("plik");
4      Stream outStream = File.OpenWrite("plik.kopia");
5      BufferedStream bufIn = new BufferedStream(inStream);
6      BufferedStream bufOut = new BufferedStream(outStream);
7
8      byte[] buffer = new Byte[1024];
9      int bytesRead;
10     while ( (bytesRead = bufIn.Read(buffer, 0, 1024)) > 0 )
11         bufOut.Write(buffer, 0, bytesRead);
12
13     bufOut.Flush( );
14     bufIn.Close( );
15     bufOut.Close( );
16 }
```

Strumienie (7)

- Bardziej zaawansowany przykład użycia strumieni:

```
1  using System;
2  using System.IO;
3  using System.Security.Cryptography;
4  class EncoderFun {
5    static void Main() {
6      Stream stm = new FileStream("foo.txt",
7            FileMode.Open, FileAccess.Read);
8      ICryptoTransform ict = new ToBase64Transform();
9      CryptoStream cs = new CryptoStream(stm,
10           ict, CryptoStreamMode.Read);
11     TextReader tr = new StreamReader(cs);
12     string s = tr.ReadToEnd();
13     Console.WriteLine(s);
14   }
15 }
```

Strumienie (8)

- Binarny zapis i odczyt:

```
1  public class Student {
2      public string Name;
3      public int Age;
4      public double GPA;
5  }
6
7  void SaveToStream(Stream stm, Student s) {
8      BinaryWriter bw = new BinaryWriter(stm);
9      bw.Write(s.Name);
10     bw.Write(s.Age);
11     bw.Write(s.GPA);
12     bw.Flush(); // Ensure the BinaryWriter buffer is empty
13 }
```

Strumienie (9)

- Binarny odczyt:

```
1 void ReadFromStream(Stream stm, Student s) {
2     BinaryReader br = new BinaryReader(stm);
3     s.Name = br.ReadString();
4     s.Age = br.ReadInt32();
5     s.GPA = br.ReadDouble();
6 }
```

Strumienie (10)

- Strumienie tekstowe:

```
1  void SaveToStream(Stream stm, Student s) {
2      TextWriter tw = new StreamWriter(stm);
3      tw.WriteLine(s.Name);
4      tw.WriteLine(s.Age);
5      tw.WriteLine(s.GPA);
6      tw.Flush(); // Ensure the TextWriter buffer is empty
7  }
8  void ReadFromStream(Stream stm, Student s) {
9      TextReader tr = new StreamReader(stm);
10     s.Name = tr.ReadLine();
11     s.Age = Int32.Parse(tr.ReadLine());
12     s.GPA = Double.Parse(tr.ReadLine());
13 }
```

Strumienie (11)

- Łańcuchy znaków jako strumienie:

```
1  using System;
2  using System.IO;
3  using System.Text;
4  class Test {
5     static void Main() {
6        StringBuilder sb = new StringBuilder();
7        StringWriter sw = new StringWriter(sb);
8        WriteHello(sw);
9        Console.WriteLine(sb);
10    }
11    static void WriteHello(TextWriter tw) {
12       tw.Write("Hello, String I/O!");
13    }
14 }
```

Strumienie (12)

- Asynchroniczne wejście/wyjście:

```
1  using System;
2  using System.IO;
3  using System.Text;
4  class AsyncReadFun {
5    class AsyncReadState {
6      public Stream stm; // Underlying stream
7      public byte[] buf = new byte[256]; // Read buffer
8      public StringBuilder sb = new StringBuilder(); // Result buffer
9      public AsyncCallback acb = new AsyncCallback(ReadCallback);
10   }
11   static void AReadCallback(IAsyncResult iar) {
12     AsyncReadState ars = (AsyncReadState)iar.AsyncState;
13     int bytes = ars.stm.EndRead(iar); // Get count of bytes read
14     if (bytes > 0) { // Copy read bytes and restart read
15       ars.sb.Append(Encoding.ASCII.GetString(ars.buf, 0, bytes));
16       Console.WriteLine("Read chunk of {0} bytes", bytes);
```

Strumienie (13)

```
17            ars.stm.BeginRead(ars.buf, 0, ars.buf.Length, ars.acb, ars);
18         }
19      }
20      static void Main(string[] args) {
21         Stream s = File.OpenRead(args[0]);
22         AsyncReadState ars = new AsyncReadState();
23         ars.stm = s; // Save the stream reference
24         IAsyncResult iar = s.BeginRead(ars.buf, 0, ars.buf.Length, ars.acb, ars);
25         // Download a file while we're reading the stream
26         System.Net.WebClient wc = new System.Net.WebClient();
27         wc.DownloadFile("http://www.oreilly.com", "index.html");
28         Console.WriteLine("Finished downloading index.html.");
29         // Wait until the async read is done
30         iar.AsyncWaitHandle.WaitOne();
31         Console.WriteLine(ars.sb);
32      }
33   }
```

Serializacja (1)

- Konwersja złożonych hierarchicznych struktur danych do strumieni danych
- Deserializacja – operacja odwrotna
- Przestrzenie nazw System.Runtime.Serialization.*, System.Xml.Serialization.*.
- Podział zadań: strumienie, obiekty formatujące, obiekty serializowane.
- Jawna serializacja:

```
1 public void SerializeGraph(string file, object root) {
2    Stream stm = new FileStream(file, FileMode.Create);
3    IFormatter fmt = new BinaryFormatter( );
4    fmt.Serialize(stm, root);
5    stm.Flush( );
6    stm.Close( );
7 }
```

Serializacja (2)

- Jawna deserializacja:

```
1 public object DeserializeGraph(string file) {
2    Stream stm = new FileStream(file, FileMode.Open);
3    IFormatter fmt = new SoapFormatter( );
4    object o = fmt.Deserialize(stm);
5    stm.Close( );
6    return o;
7 }
```

Serializacja (3)

- Niejawna serializacja
  - gdy serializujemy obiekt zawierający jako pola inne serializowalne obiekty,

```
1  [Serializable]
2  public sealed class Person {
3      public string Name;
4      public int Age;
5  }
6  [Serializable]
7  public sealed class Team {
8      public string Name;
9      public Person[ ] Players;
10 }
```

  - gdy przekazujemy argumenty metodzie wołanej w innej domenie aplikacji,

```
1  public Team MergeTeams(Team teamOne, Team teamTwo) {...}
```

  - w innych przypadkach zdalnego uruchamiania.

Serializacja (4)

- Atrybut Serializable
  - Ustawia odpowiedni bit w meta-danych o typie.
  - Nie jest dziedziczony – klasy potomne muszą redeklarować atrybut.
  - Wszystkie składowe muszą być też serializowalne.
- Atrybut NonSerialized – wyłącza pola z serializacji.

```
1 [Serializable]
2 public sealed class Person {
3   public string Name;
4   public DateTime DateOfBirth;
5   [NonSerialized] public int Age; // Can be calculated
6   // Rest of class...
7 }
```

Serializacja (5)

- Interfejs IDeserializationCallback – pozwala ustawić po deserializacji pola wyłączone z serializacji

```
1  [Serializable]
2  public sealed class Person : IDeserializationCallback {
3     public string Name;
4     public DateTime DateOfBirth;
5     [NonSerialized] public int Age; // Can be calculated
6     public void OnDeserialization(object o) {
7        TimeSpan ts = DateTime.Now - DateOfBirth;
8        Age = ts.Days/365; // Rough age in years
9     }
10    // Rest of class...
11 }
```

Serializacja (6)

- Interfejs ISerializable
  - Pozwala na więcej kontroli nad szczegółami serializacji.
  - Potrzeba implementacji konstruktora deserializacyjnego:

```
1  [Serializable]
2  public sealed class Team : ISerializable {
3     public string ChangedName;
4     public Person[ ] Players;
5     public void GetObjectData(SerializationInfo si, StreamingContext sc) {
6           si.AddValue("Name", ChangedName);
7           si.AddValue("Players", Players);
8     }
9     private Team(SerializationInfo si, StreamingContext sc) {
10          ChangedName = si.GetString("Name");
11          Players = (Person[ ])si.GetValue("Players",
12                                 typeof(Person[ ]));
13    }
14 }
```

.NET remoting (1)

Idea **RPC** (ang. *Remote Procedure Call*) – komunikacja na poziomie obiektów i metod a nie strumieni

Obiekty **przekazywalne zdalnie** (ang. *remotable*)

- **przekazywane przez wartość** (ang. *marshal-by-value*)
  - obiekty serializowalne
    - obiekty są serializowane po jednej i deserializowane po drugiej stronie
    - szybciej gdy trzeba wykonać wiele operacji na obiekcie, wolniej, gdy trzeba przesłać duży obiekt
    - działania na kopii obiektu
- **przekazywane przez referencję** (ang. *marshal-by-reference*)
  - obiekty typów dziedziczących po MarshalByRefObject
    - obiekty-pełnomocniki (*proxies*) pośredniczą w operacjach
    - szybciej, gdy duży obiekt a mało wywołań
    - działania na zdalnym obiekcie (oryginale)

.NET remoting (2)

.NET remoting pozwala działać **na zdalnych obiektach jak na lokalnych** z pewnymi **wyjątkami**:

- **pola statyczne** nie są przenoszone zdalnie
- pola obiektów przekazywanych **przez referencję** mogą być przekazywane **przez wartość** i odwrotnie
- **prywatne metody** nie mogą być uruchamiane zdalnie, delegaty zawierające prywatne metody nie moga być przenoszone zdalnie

.NET remoting (3)

Definiowanie serwera:

- rejestracja **kanału** komunikacji (obsługa protokołu sieciowego i formatów serializacji)
- rejestracja **typów** serwowanych obiektów w systemie .NET remoting

Klasy kanałów: Channel (bazowa), HTTPChannel, TCPChannel, IPCChannel

Definiowanie klienta:

- rejestracja **kanału** komunikacji
- rejestracja **typów** z identyfikacją serwera

Klient **tworząc obiekt** zarejestrowanego typu, w rzeczywistości **tworzy proxy** obiektu serwera

Integracja z Pythonem – IronPython (1)

Czynności wstępne:

- instalacja IronPythona,
- referencje do bibliotek .NET: IronPython oraz Microsoft.Scripting.

```
1  static void Main()
2  {
3     int result = (int)Calculate("2 * 3");
4     Console.WriteLine(result); // 6
5     var list = (IEnumerable) Calculate("[1, 2, 3] + [4, 5]");
6     foreach (int n in list) Console.Write(n); // 12345
7  }
8  static object Calculate(string expression)
9  {
10    Microsoft.Scripting.Hosting.ScriptEngine engine =
11       IronPython.Hosting.Python.CreateEngine();
12    return engine.Execute(expression);
13 }
```

Integracja z Pythonem – IronPython (2)

Można także:

- ustawiać i pobierać zmienne Pythona,
- przekazywać Pythonowi obiekty C#.

```
1  public static void TestPython2()
2  {
3      ScriptEngine engine = Python.CreateEngine();
4      ScriptScope scope = engine.CreateScope();
5
6      scope.SetVariable("taxPaidLastYear", 20000m);
7      scope.SetVariable("taxPaidThisYear", 8000m);
8
9      ScriptSource source = engine.CreateScriptSourceFromString(
10         "taxPaidLastYear / taxPaidThisYear > 2", SourceCodeKind.Expression);
11
12     bool auditRequired = (bool)source.Execute(scope);
13     Console.WriteLine(auditRequired); // True
14
```

Integracja z Pythonem – IronPython (3)

```
15    scope.SetVariable("input", 2);
16    source = engine.CreateScriptSourceFromString("result = input * 3",
17       SourceCodeKind.SingleStatement);
18    source.Execute(scope);
19    Console.WriteLine(scope.GetVariable("result")); // 6
20
21    string code = @"sb.Append(""World"")";
22    var sb = new StringBuilder("Hello");
23    scope.SetVariable("sb", sb);
24    source = engine.CreateScriptSourceFromString(code,
25       SourceCodeKind.SingleStatement);
26    source.Execute(scope);
27    Console.WriteLine(sb.ToString()); // HelloWorld
28 }
```

Współpraca z Win32 (1)

Mechanizm importu funkcji z bibliotek DLL stworzonych dla Win32:

```
1  using System.Runtime.InteropServices;
2  class MsgBoxTest
3  {
4     [DllImport("Kernel32", EntryPoint = "GetCurrentThreadId")]
5     static extern Int32 GetCurrentWin32ThreadId();
6     [DllImport("user32.dll")]
7     static extern int MessageBox (IntPtr hWnd,
8        string text, string caption, int type);
9
10    public static void Main()
11    {
12       MessageBox(IntPtr.Zero,
13          "Bieżący wątek ma id = "+GetCurrentWin32ThreadId(),
14          "Informacja", 0);
15    }
16 }
```

Współpraca z Win32 (2)

Przekazywanie klas i struktur

```
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

```
[StructLayout(LayoutKind.Sequential)]
class SystemTime
{
    public ushort Year;
    public ushort Month;
    public ushort DayOfWeek;
    public ushort Day;
    public ushort Hour;
    public ushort Minute;
    public ushort Second;
    public ushort Milliseconds;
}
```

Współpraca z Win32 (3)

```
1 [DllImport("kernel32.dll")]
2 static extern void GetSystemTime(SystemTime t);
3
4 static void Main()
5 {
6    SystemTime t = new SystemTime();
7    GetSystemTime(t);
8    Console.WriteLine(t.Year);
9 }
```

Współpraca z Win32 (4)

Wywołania zwrotne:

```
1  class CallbackFun {
2    delegate bool EnumWindowsCallback(IntPtr hWnd, IntPtr lParam);
3
4    [DllImport("user32.dll")]
5    static extern int EnumWindows(EnumWindowsCallback hWnd, IntPtr lParam);
6
7    static bool PrintWindow(IntPtr hWnd, IntPtr lParam)
8    {
9      Console.WriteLine(hWnd.ToInt64());
10     return true;
11   }
12   static void Main()
13   {
14     EnumWindows(PrintWindow, IntPtr.Zero);
15   }
16 }
```

Współpraca z COM (1)

Podstawy:

- Łatwiej niż z Win32, bo dostępne meta-dane.
- Od pierwszych wersji, specjalne wsparcie .NETu dla COM.
- COM interface – rozdział interfejsu od implementacji
- IUnknown – podstawowy interfejs COM (wszystkie obiekty muszą go implementować)
  - AddRef() i Release() – kontrola pamięci (alokacji/dealokacji)
  - QueryInterface() – zwraca obiekty implementujące interfejsy
- IDispatch – umożliwia programowanie dynamiczne

Współpraca z COM (2)

Korzystanie z komponentów COM w C#

- *Runtime-Callable Wrappers* (RCWs) – dbają o odpowiednie konwersje danych
- *COM interop types* – typy w C# generowane na podstawie COM
- parametry nazwane i wartości domyślne parametrów pozwalają znacznie skrócić kod i zwiększyć czytelność (workBook.SaveAs z następnej strony ma 12 parametrów)
- narzędzie `tlimp.exe` lub Visual Studio
- niezbędne czynności:
  - dodać referencję do biblioteki COM
  - korzystać z klas dostępnych w odpowiedniej przestrzeni nazw

Współpraca z COM (3)

```
1  using System;
2  using Excel = Microsoft.Office.Interop.Excel;
3
4  class Program
5  {
6     static void Main()
7     {
8        var excel = new Excel.Application();
9        excel.Visible = true;
10       Workbook workBook = excel.Workbooks.Add();
11       excel.Cells [1, 1].Font.FontStyle = "Bold";
12       excel.Cells [1, 1].Value2 = "Hello World";
13       workBook.SaveAs(@"d:\temp.xlsx");
14    }
15 }
```

Współpraca z COM (4)

Udostępnianie obiektów .NETu systemowi COM

- *COM-Callable Wrappers* (CCWs) – zapewniają konwersję do/z COM, implementują IUnknown i (opcjonalnie) IDispatch
- Atrybut ComVisible dla zestawu i klas
- narzędzie `tlbexp.exe` generuje plik `.tlb` (ang. *COM type library*)

Narzędzia do testowania pisanego kodu

- Projekt metod testujących
- Dostęp do danych wewnętrznych biblioteki/programu – plik AssemblyInfo.cs, atrybut System.Runtime.CompilerServices.InternalsVisibleTo
- Klasy i metody testujące, atrybut TestMethod
- Przestrzeń Microsoft.VisualStudio.TestTools.UnitTesting
- Klasy weryfikujące: Assert, CollectionAssert, StringAssert
- Visual Studio – menu „Tests”

Śledzenie przebiegu programu

Przestrzeń nazw: System.Diagnostics

Klasy Debug i Trace

- Atrybut Conditional (np. "DEBUG", "TRACE") uwzględnia:
  - opcje kompilatora (np. /define:DEBUG)
  - zmienne środowiskowe (np. set DEBUG=1)
  - dyrektywy (np. **#define** DEBUG)
- Metody Write(), WriteIf(), WriteLine(), WriteLineIf()

Śledzenie warunkowe:

- Klasy BooleanSwitch i TraceSwitch
- Przełączanie kodem lub w pliku konfiguracyjnym

Definiowanie strumieni wyjściowych śledzenia:

- Statyczne kolekcje Debug.Listeners i Trace.Listeners typu TraceListenerCollection
- Klasy ConsoleTraceListener, TextWriterTraceListener, EventLogTraceListener

Klasa TcpClient (1)

Specyfika protokołu TCP (dla dopełnienia formalności):

- interfejs komunikacyjny jest określony przez **adres IP** oraz **port TCP** (liczba 16-bitowa bez znaku, czyli w zakresie 0–65535)
- serwer **nasłuchuje** połączeń **na** wybranym **porcie**
- klient łączy się wychodząc do sieci również przez pewien port, choć często port klienta jest wybierany automatycznie przez system

Podstawy obsługi **TCP w .NET**

- Przestrzeń nazw System.Net.Sockets
- Klasa TcpClient
- Klasa NetworkStream
- Klasa TcpListener
- Klasa Socket

Klasa TcpClient (2)

Schemat działania:

- **tworzymy** obiekt
- **łączymy** z serwerem (można już w konstruktorze)
- **wymieniamy dane** przez strumień (klasa NetworkStream) lub gniazdo (klasa Socket)

Podstawowe składowe:

- Connect(...) – kilka wersji
- NetworkStream GetStream()
- Close() – „zwalnia” obiekt, ale nie kończy połączenia! Konieczne NetworkStream.Close()
- własności Connected, ReceiveBufferSize, ReceiveTimeout, SendBufferSize, SendTimeout

Prosty klient TCP

```
1  TcpClient client = new TcpClient(server, 80);
2  string message = "GET / HTTP/1.1\nHost: " + server + "\n\n";
3  byte[] data = System.Text.Encoding.ASCII.GetBytes(message);
4  NetworkStream stream = client.GetStream();
5  stream.Write(data, 0, data.Length);
6  data = new byte[256];
7  string odp = string.Empty;
8  int bytes;
9  do
10 {
11     bytes = stream.Read(data, 0, data.Length);
12     odp += System.Text.Encoding.ASCII.GetString(data, 0, bytes);
13 }
14 while (bytes > 0);
15 Console.WriteLine("Pobrane: {0}", odp);
16 stream.Close();
17 client.Close();
```

Prosty klient TCP z użyciem klasy Socket

Wystarczy w kodzie z poprzedniej strony **zamienić**:

```
NetworkStream stream = client.GetStream();
stream.Write(data, 0, data.Length);
```

na

```
Socket socket = client.Client;
socket.Send(data, 0, data.Length, SocketFlags.None);
```

oraz

```
bytes = stream.Read(data, 0, data.Length);
```

na

```
bytes = socket.Receive(data, 0, data.Length, SocketFlags.None);
```

Klasa NetworkStream

- Realizuje odczyt i zapis przez strumienie sieciowe
- Nie pozwala na wyszukiwanie (CanSeek == **false**)
- Nie ma buforowania, Flush() nic nie robi

Podstawowe składowe:

- **int** Read(**byte**[] buffer, **int** offset, **int** size)
- **int** ReadByte()
- **void** Write(**byte**[] buffer, **int** offset, **int** size)
- **void** WriteByte(**byte value**)
- własności:
  - logiczne: CanRead, CanWrite, DataAvailable
  - w milisekundach: ReadTimeout, WriteTimeout,
  - Socket

Klasa TcpListener

- Nasłuchiwanie portu, akceptowanie połączeń, odpowiadanie na zapytania
- Klienci: TcpClient lub Socket
- można nasłuchiwać pod konkretnym adresem bądź pod dowolnym (System.Net.IPAddress.Any)

Podstawowe składowe:

- **void** Start()
- **void** Stop()
- Socket AcceptSocket()
- TcpClient AcceptTcpClient()
- **bool** Pending()

Prosty serwer TCP

```
1  server = new TcpListener(IPAddress.Any, 12345);
2  server.Start();
3  byte[] bytes = new byte[256];
4  while (true) {
5      TcpClient client = server.AcceptTcpClient();
6      string data = null;
7      int len, nl;
8      NetworkStream stream = client.GetStream();
9      while ((len = stream.Read(bytes, 0, bytes.Length)) > 0) {
10         data += Encoding.ASCII.GetString(bytes, 0, len);
11         while ((nl = data.IndexOf('\n')) != -1) {
12             string line = data.Substring(0, nl + 1);
13             data = data.Substring(nl + 1);
14             byte[] msg = Encoding.ASCII.GetBytes(PingPong(line));
15             stream.Write(msg, 0, msg.Length);
16         }
17     }
18     client.Close();
19 }
```

Ping-pong – fragment klienta TCP

```
1  string message = "ping 1024\n";
2  byte[] data = Encoding.ASCII.GetBytes(message);
3
4  stream.Write(data, 0, data.Length);
5  Console.Write("Wysłane: {0}", message);
6
7  byte[] response = new byte[256];
8  string responseStr = string.Empty;
9  int bytes;
10 do
11 {
12       bytes = stream.Read(response, 0, response.Length);
13       responseStr += Encoding.ASCII.GetString(response, 0, bytes);
14 }
15 while (stream.DataAvailable);
16
17 Console.WriteLine("Pobrane: {0}", responseStr);
```