Programowanie wizualne .NET 4

Programowanie Wizualne

Paweł Wojciechowski

Instytut Informatyki, Politechniki Poznańskiej

2023

Klasy częściowe

- w C# z założenia definicja/deklaracja/implementacja klasy musi znajdować się w jednym pliku
- słowo kluczowe `partial` mówi o tym, że dla takiej klasy część definicji znajduje się jeszcze gdzieś

```
public partial class Pracownik
```

- wykorzystanie:
  - oddzielenie deklaracji od definicji?
  - mechanizmy korzystające z automatycznego generowania kodu

Kolekcje

- znajdują się w `System.Collections`
- np. ArrayList, Hashtable, Queue, SotredList, Stack

```
ArrayList al = new ArrayList();

al.Add(1);
al.Add("ala ma kota");
al.Add(new Pracownik() { Imie = "Jan", Nazwisko = "Nowak" });
al.Add(false);

foreach( var el in al)
{
    Console.WriteLine(el);
}
```

- używanie kolekcji ogólnych ma dwie wady:
  - wydajność: pakowanie/rozpakowywanie
  - bezpieczeństwo: nie mamy pewności jakiego typu elementy znajdują się w kolekcji

Pytanie 1: jakie są różnice między kolekcjami, a tablicami?

Typy generyczne

- jako typy generyczne mogą być: klasy, struktury, interfejsy i delegaty (enum nie!)

```
public class Point<T>
{
    public T X { get; set; }
    public T Y { get; set; }
}
```

- ograniczenia dla parametru typu
  - `where T : struct` – typ T musi być wartościowy
  - `where T : class` - typ T musi być referencyjny
  - `where T : new()` – typ T musi mieć bezargumentowy konstruktor
  - `where T : KLASA_BAZOWA` – typ T musi dziedziczyć po klasie `KLASA_BAZOWA`
  - `where T : INTERFEJS` – Typ T musi implementować interfejs `INTERFEJS`

Typy i metody generyczne

- można implementować metody generyczne
- w implementacjach nie można używać operatorów
- można za to przeciążać operatory

```
public static string ReverseToString<T>(T val)
{
    string s = val.ToString();

    return new string (s.ToCharArray().Reverse().ToArray());
}
```

- interfejsy generyczne `IEnumerator<T>, IEnumerable<T>, IComparer<T>, IComparable<T>, ICloneable<T>`

Pytanie 2: jaki będzie sens używania interejsu generycznego IEnumerable<T>?

Kolekcje generyczne

- przy tworzeniu kolekcji specyfikujemy jakiego typu elementy będą przechowywane

```
List<Pracownik> lista = new List<Pracownik>()
{  …          };

foreach( var el in lista)
{
    Console.WriteLine(el);
}
```

- zalety: wydajność, bezpieczeństwo typów, można używać gotowych kolekcji dla dowolnych typów – nie trzeba ich pisać samemu ☺
- kolekcje generyczne: `Dictionary<Tkey, Tvalue>`, `LinkedList<T>`, `List<T>`, `Queue<T>`, `SortedDictionary<T>`, `SortedSet<T>`, `Stack<T>`, `SynchronizedCollection<T>`
- Kolekcje (niegeneryczne specjalne): `System.Collections.Specialized`

https://msdn.microsoft.com/pl-pl/library/system.collections.specialized(v=vs.110).aspx

Kolekcje generyczne - przykład

```
SortedDictionary<int, Person> sortedDict = new SortedDictionary<int, Person>()
{
    {1, new Person(){ ID=1, Name="Grzesiek"}},
    {2, new Person(){ ID=2, Name="Jurek"}},
    {3, new Person(){ ID=3, Name="Marek"}}
};

foreach (var p in sortedDict)
{
    Console.WriteLine("Key: {0} Value:{1}", p.Key, p.Value);
}

KeyValuePair<int, Person> kvp = new KeyValuePair<int, Person>(2, new Person() { ID = 2, Name = "Jurek" });

if (sortedDict.Contains(kvp))
{
    Console.WriteLine("zawiera");
}
```

Delegaty

- delegat jest typem
- należy go traktować jak znany z C++ wskaźnik na funkcję
- delegat przechowuje trzy ważne elementy:
  - adres metody, którą ma wywołać,
  - parametry wywołania metody
  - zwracany typ

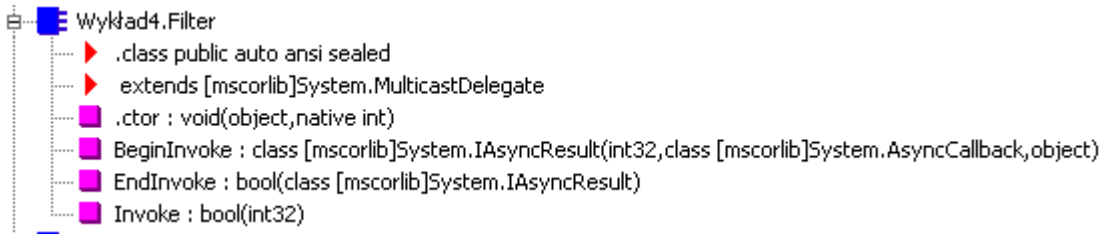
```
public delegate int BinOperator(int i, int j);

public class Foo {
    public int ObjectSub( int i, int j)
    {
        return i - j;
    }
}

public static int StaticAdd(int i, int j)
{
    return i + j;
}

BinOperator bo = new BinOperator( Program.StaticAdd);
Foo f = new Foo();
BinOperator bo2 = new BinOperator( f.ObjectSub);
int res = bo(10, 4);
```

Delegaty (2)

- dla tak zdefiniowanego delegata:

```
public delegate bool Filter(int i);
```

tworzony jest następujący typ:



- dziedziczenie po `MulticastDelegate` – delegat może przechowywać wiele metod do wywołania
- `BeginInvoke i EndInvoke` – do wywoływania asynchronicznego (nieaktualne od C#9)
- `Invoke` – wywołanie synchroniczne

Pytanie 3: wynik której metody zostanie zwrócony po wywołaniu delegata?

```
BinOperator bo = new BinOperator( Program.StaticAdd);
Foo f = new Foo();
bo += f.ObjectSub;
int res = bo(10, 4);
```

Predefiniowane delegaty

- Action<>

  delegat funkcji, który ma do 16 argumentów wejściowych i nic nie zwraca

- Func<>

  delegat funkcji, który ma do 16 argumentów i zwraca wartość. Wartość zwracana jest ostatnim argumentem

```
Func<int, bool>
```

Wykorzystanie delegatów do informowania o zmianie stanu

- mechanizm callback – informowanie o zajściu zdarzenia wymaga (minimum):
  - zdefiniowania delegata

```
public delegate void BudgetAlertHandler(int missingAmount);
```

  - zdefiniowanie pola, które będzie przechowywało metody do wywołania

```
private BudgetAlertHandler budgetAlertHandlers;
```

  - dodania metody pozwalającej na rejestracje metod

```
public void RegisterBudgetAlert( BudgetAlertHandler metoda)
{
    budgetAlertHandlers += metoda;
}
```

Pytanie 4: jak zapewnić obsługę tylko jednej metody?

Pytanie 5: dlaczego nie zdefiniować pola `budgetAlertHandlers` jako publicznego?

Zdarzenia

- słowo kluczowe event

```
public delegate void BudgetAlertHandler(int missingAmount);

public event BudgetAlertHandler BudgetExceeded;

firma.BudgetExceeded += PrzekroczonoBudzet;
```

- w porównaniu do mechanizmu opartego o delegaty, zdarzenie nie wymaga pola do przechowywania metod i metod rejestrujących (wyrejestrowujących)
- nie można wywołać zdarzenia spoza klasy, w której jest zdefiniowane
- zalecana metoda obsługi zdarzenia powinna mieć taki nagłówek:

```
public void MetodaObslugiZdarzenia( object sender, EventArgs e)
```

- dla typowych zdarzeń jest delegat `EventHandler<>` którego typem może być klasa dziedzicząca po `EventArgs`

```
public class BudgetExceededEventArgs: EventArgs{
    public int Amount { get; set; }
}
public event EventHandler<BudgetExceededEventArgs> BudgetExceeded;
```

Metody anonimowe

Pytanie 6: czy można jedną metodę zarejestrować wielokrotnie?

- wywołania równoznaczne

```
firma.BudgetExceeded += PrzekroczonoBudzet;
firma.BudgetExceeded += new BudgetAlertHandler(PrzekroczonoBudzet);
```

- zamiast definiować metodę, która jest wykorzystywana tylko do obsługi zdarzenia

```
public static void PrzekroczonoBudzet(int kwota) {
    Console.WriteLine("UWAGA: budżet przekroczono o: {0:C}", kwota);
}
firma.BudgetExceeded += PrzekroczonoBudzet;
```

- można użyć metody anonimowej

```
firma.BudgetExceeded += delegate (int i)
{
    Console.WriteLine("przekroczono kwote o: {0:C}", i);
};
```

- taki zapis musi się kończyć średnikiem
- jeżeli delegat nie ma argumentu można pominąć nawias

Metody anonimowe (2)

```
public static void ZarejsetrujMetodyObslugi( int argument, Firma firma )
{
    int zmiennaLokalna = 7;

    firma.BudgetExceeded += delegate (int a)
    {
        Console.WriteLine($"zmienna lokalna {zmiennaLokalna} {argument}");
        Console.WriteLine( $"przekroczono o kwotę {a:C} ");
    };
}
```

- metoda anonimowa:
  - nie może używać parametrów typu ref/out metody w której jest zdefiniowana
  - nie może mieć zmiennych lokalnych o takiej samej nazwie jak zmienne lokalne w metodzie w której jest zdefiniowana (nieaktualne od C#8)
  - może mieć dostęp do pól klasy w której jest zdefiniowana
  - może przysłonić pola klasy zmiennymi lokalnymi

Wyrażenie lambda

- Definiuje się je następująco:

  ```
  (argumenty) => polecenia
  ```

- nawias przy argumentach może być pominięty jeśli jest jeden argument
- typ argumentu można pominąć – wynika bezpośrednio z zastosowania
- gdy wyrażenie nie ma argumentów stosuje się () =>

Wyrażenia lambda vs delegat

- metoda Where `IEnumerable<TSource> Ienumerable<Tsource>.Where<TSource>(Func<TSource, bool> predicate);`
- dla tablicy chcemy wypisać liczby nieparzyste `int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };`
- wykorzystanie delegata

```
public static bool IsOdd(int i)
{
    return (i % 2 == 1);
}

Func<int, bool> parityChecker = new Func<int, bool>(IsOdd);

foreach( int i in arr.Where( parityChecker))
                Console.WriteLine(i);
```

- metoda anonimowa

```
Func<int, bool> parityChecker = delegate( int i)
{
    return (i % 2 == 1);
};
```

Wyrażenia lambda vs delegat (2)

- metoda anonimowa bez tworzenia obiektu delegata

```
foreach (int i in arr.Where(delegate (int i) { return (i % 2 == 1); }))
{
    Console.WriteLine(i);
};
```

- wyrażenie lambda

```
foreach (int i in arr.Where(i => (i % 2) == 1))
{
    Console.WriteLine( i);
}
```

Metody rozszerzania

- umożliwiają na dodanie funkcjonalności do istniejący już klas
- muszą być zdefiniowane jako statyczne w statycznych klasach
- nie mają dostępu do pól klasy

```
public static int SumOfDigits(this int i)
{
    int sum = 0;
    char[] tab = i.ToString().ToCharArray();
    for (int d = 0; d < tab.Length; d++)
    {
        sum += int.Parse( ""+tab[d]);
    }
    return sum;
}
```

```
int number = 12345678;
Console.WriteLine( number.SumOfDigits());
```

```
IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);
```

Pytanie 6: czy to ma jakieś zalety?

Przeciążanie operatorów

- operatory, które można przeciążyć
  - unary: `+, -, !, ~, ++, --, true, false`
  - binary: `+, -, *, /, %, &, |, ^, <<, >>`
  - porównawcze: `==, !=, >, <, >=, <=`
- operatory zawsze są metodami statycznymi
- operatory true/false,

```
public static bool operator true(Point p)
{
    Console.WriteLine("true op");
    if ((p.X == 0) && (p.Y == 0))
        return false;
    else
        return true;
}
```

Przeciążanie operatorów - przykłady

```
public static Point operator +(Point p)
{
    p.X = p.X + 1;
    return p;
}

public static Point operator +(Point p1, Point p2)
{
    return new Point() { X = p1.X + p2.X, Y = p1.Y + p2.Y };
}
```

```
Point p = new Point();
p.X = 1;

if (p)
{}

if ( p && p)
{}

Point p2 = +p;
```

operator true

operator false

operator unary+

Przy definiowaniu operatorów typ drugiego argumentu może być inny niż pierwszego

```
public static Point operator +(Point p1, int m) {
    return new Point() { X = p1.X + m, Y = p1.Y + m };
}
```

Niestandardowe konwersje typów

- niejawna konwersja typów

```
byte z = 10;
int i = z;
```

- jawna konwersja typów

```
short s = (short)i;
```

- tworząc klasę możemy zdefiniować dla niej czy i w jaki sposób ma być rzutowana na inne typy

```
public static explicit operator Point(int p)
{
    return new Point() { X = p, Y = p };
}

public static explicit operator int(Point p)
{
    return p.X;
}

int i = (int) p;
Point p3 = (Point)i;
```

Niestandardowe operatory rzutowania (2)

- rzutowanie niejawne

```
public static implicit operator int(Point p)
{
    return p.Y;
}

int i =  p;
```

- uwaga na niejawne rzutowanie!!! Dla obiektów klasy Point przy próbie wywołania:

```
Point punkt = new Point();
Console.WriteLine(punkt);
```

obiekt punkt zamiast wywołać przeciążoną metodę `ToString()` rzutuje punkt na `int`

- nie można tworzyć jednocześnie operatorów jawnych i niejawnych dla takich samych parametrów, natomiast można wywoływać jawne jeśli zdefiniowany jest tylko niejawny

Typy anonimowe

- można zdefiniować typ anonimowy

```
var car1 = new { Name = "Skoda", Model = "Fabia", Year = 1990 };
```

- typ taki zawsze dziedziczy po object
- kompilator automatycznie wygeneruje typ AnonymousType0’1 z metodą Equals() i ToString()

```
var car1 = new { Name = "Skoda", Model = "Fabia", Year = 1990 };
var car2 = new { Name = "Skoda", Model = "Fabia", Year = 1990 };

if ( car1 == car2 )
    Console.WriteLine("== the same");

if ( car1.Equals(car2))
    Console.WriteLine("Equals the same");
```

Typy anonimowe (2)

- typy anonimowe mogą zawierać typy anonimowe
- właściwości/pola są tylko do odczytu
- nie można dziedziczyć po typach anonimowych
- nie mogą zawierać zdarzeń, własnych metod, operatorów, konstruktorów

Indekser

- Pozwala dla klasy albo struktury zdefiniować dostęp do elementów kolekcji za pomocą operatora indeksowania [].
- indekser definiuje się jako właściwość np.:

```
public object this[int index]
```

- indekser nie musi być int-em, mogą to być różne typy np. string.

```
public Person this[string name]
```

- indekser może być wielowymiarowy, może też być zbudowany na wielu parametrach

```
public object this[int x, int y]
```

```
public object this[int x, bool order, Person p]
```

- można przeciążyć indeksery dla danej klasy
- indekser może być elementem interfejsu

Indekser – przykład

```
public class Person
{
    public string Name { get; set; }
    public int ID { get; set; }
}

public class Office
{
    private Dictionary<string, Person> workers = new Dictionary<string, Person>();

    public Office()
    {
        …
    }

    public Person this[string name]
    {
        get => workers[name];
        set => workers[name] = value;
    }

    public Person this[int index]
    {
        get => workers.ElementAt(index).Value;

    }
}
```