Programowanie pod Windows
Zestaw 3

Język C# - refleksja, typy generyczne

2023-03-14

Liczba punktów do zdobycia: **10/26**
Zestaw ważny do: 2023-03-28

1. (**2p**) W trakcie wykładu przedstawiono szkic ogólnego generatora zapytań SQL/struktury XML/struktury JSON dla dowolnych obiektów. Państwa zadaniem będzie odtworzyć ten przykład na przykładzie generatora struktury XML. W pierwszym podejściu - przy pomocy interfejsu za pomocą którego można z obiektu dla którego generuje się XML pobrać informację o jego strukturze.

   Formalnie: generator to klasa z metodą generującą

```
public class XMLGenerator
{
        public string GenerateXML( IClassInfo dataObject )
        {
                // uzupełnić implementację
                throw new NotImplementedException();
        }
}
```

   W celu pobrania informacji o strukturze obiektu dla którego ma zostać wygenerowany XML, generator wykorzysta interfejs `IClassInfo`:

```
public interface IClassInfo
{
        string[] GetFieldNames();
        object GetFieldValue( string fieldName );
}
```

   Proszę zwrócić uwagę jak zaprojektowany jest ten interfejs: jedna z jego metod zwraca listę wszystkich pól klasy, druga zwraca wartość konkretnego pola.

   Jeśli teraz ktoś chciałby wygenerować XML dla zadanej klasy, na przykład takiej:

```
public class Person
{
        public string Name { get; set; }
        public string Surname { get; set; }
}
```

   to po pierwsze, klasa musiałaby implementować interfejs `IClassInfo`

```
public class Person : IClassInfo
{
        public string Name { get; set; }
        public string Surname { get; set; }
```

```
        public string[] GetFieldNames()
        {
                return new[] { "Name", "Surname" };
        }

        public object GetFieldValue( string fieldName )
        {
                switch ( fieldName )
                {
                        case "Name":
                                return this.Name;
                        case "Surname":
                                return this.Surname;
                        default:
                                return null;
                }
                throw new NotImplementedException();
        }
}
```

a po drugie - należałoby właśnie (co jest treścią zadania!) zaimplementować metodę `GenerateXML` generatora.

Wtedy można by napisać fragment kodu:

```
Person person =
        new Person()
        {
                Name = "Jan",
                Surname = "Kowalski"
        };

XMLGenerator generator = new XMLGenerator();

string xml = generator.GenerateXML( person );
```

2. (**2p**) W drugim podejściu do generatora XML luzuje się wymagania - zakładamy że klasa która ma być zapisywana do XML (w poprzednim przykładzie klasa `Person`) nie musi implementować żadnego interfejsu.

   Jak w takim razie generator ma dostać się do listy pól w klasie i wartości konkretnych pól?

   Za pomocą refleksji.

   Formalnie, zmieniamy definicję generatora

```
public class XMLGenerator
{
        public string GenerateXML( object dataObject )
        {
                // uzupełnić implementację
                throw new NotImplementedException();
        }
}
```

   i nadal chcemy móc napisać

```
Person person =
        new Person()
        {
                Name = "Jan",
                Surname = "Kowalski"
        };

XMLGenerator generator = new XMLGenerator();

string xml = generator.GenerateXML( person );
```

3. (**1p**) Generator oparty na refleksji ma pewną wadę - refleksja podczas enumeracji składowych klasy uwzględnia wszystkie składowe o takiej samej charakterystyce (na przykład wszystkie pola publiczne). A co jeśli chciałoby się **pominąć** jakieś pole?

   Należałoby je oznaczyć atrybutem.

   Formalnie, chcemy móc zdefiniować atrybut pozwalający pominąć pole podczas generacji:

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

        [IgnoreInXML]
        public string Surname { get; set; }
}
```

   a kod generatora zmodyfikować w taki sposób żeby podczas enumeracji składowych klasy wykrywał właściwości znakowane tym konkretnym atrybutem i pomijał je w trakcie generowania XML

4. (**1p**) Zademonstrować w działaniu metody `ConvertAll`, `FindAll`, `ForEach`, `RemoveAll` i `Sort` klasy `List<T>` używając anonimowych delegacji o odpowiednich sygnaturach.

5. (**1p**) We własnej klasie `ListHelper` zaprogramować statyczne metody `ConvertAll`, `FindAll`, `ForEach`, `RemoveAll` i `Sort` o semantyce zgodnej z odpowiednimi funkcjami z klasy `List<T>` i sygnaturach rozszerzonych względem odpowiedników o instancję obiektu `List<T>` na którym mają operować.

```
public class ListHelper
{
  public static List<TOutput> ConvertAll<T, TOutput>(
    List<T> list,
    Converter<T, TOutput> converter );
  public static List<T> FindAll<T>(
    List<T> list,
    Predicate<T> match );
  public static void ForEach<T>( List<T>, Action<T> action );
  public static int RemoveAll<T>(
    List<T> list,
    Predicate<T> match );
  public static void Sort<T>(
    List<T> list,
    Comparision<T> comparison );
}
```

6. (**3p**) Napisać klasę `BinaryTreeNode<T>`, która będzie modelem dla węzła drzewa binarnego. Węzeł powinien przechowywać informację o danej typu `T` oraz swoim lewym i prawym synu.

   Klasa powinna zawierać dwa enumeratory, dla przechodzenia drzewa w głąb i wszerz, zaprogramowane z wykorzystaniem słowa kluczowego `yield`.

   Wskazówka: choć implementacja bez `yield` *może wydawać się trudna, w rzeczywistości jest również stosunkowo prosta. Należy wykorzystać pomocnicze struktury danych, przechowującą informację o odwiedzanych węzłach. Każdy* `MoveNext` *ogląda bieżący węzeł, a jego podwęzły, lewy i prawy, umieszcza w pomocniczej strukturze danych. Każdy* `Current` *usuwa bieżący węzeł z pomocniczej struktury i zwraca jako wynik. Strukturę danych dobiera się w zależności od tego czy chce się implementować przechodzenie wszerz czy wgłąb (jakie struktury danych należy wybrać dla każdego z tych wariantów?)*

Wiktor Zychla