Wyższa Szkoła Bankowa
Wydział Finansów i Zarządzania
Informatyka Inżynierska
Studia I stopnia, semestr 2
Gdańsk

Programowanie obiektowe
Laboratorium 7 & 8

Autor:
Jakub Głuszek

13 czerwca 2020

Spis treści

1 Laboratorium nr 7 **3**
1.1 Delegaty . . . 3
1.2 Wyrażenia lambda . . . 4
1.3 Zdarzenia . . . 4
1.4 Obsługa wyjątków . . . 5

2 Laboratorium nr 8 **5**
2.1 Refleksja . . . 5
2.2 Atrybuty . . . 7
2.3 Serializacja . . . 8

1 Laboratorium nr 7

1.1 Delegaty

Delegaty są obiektami, które „wiedzą" jak wywołać metodę. Są one podobne do wskaźników na funkcję w C. Aby utworzyć **typ delegacyjny** używamy słowa kluczowego `delegate` po którym określamy typ zwrotny i dalej nazwę delegatu wraz z jego parametrami.

Dodaj do projektu klasę `ExampleTransformer` i umieść w niej typ delegacyjny:

```
public delegate int Transformer (int x);
```

Dodatkowo umieść w niej metodę, która będzie wykonywała „transformację" liczby z wykorzystaniem przekazanego egzemplarza delegatu:

```
public void Transform(Transformer t, int value)
{
  int ret = t(value);
  Console.WriteLine($"Output value is: {ret}");
}
```

Do delegatu możemy przypisać dowolną metodę zgodną z definicją delegatu. Dla powyższego przypadku może to być dowolna metoda, która przyjmuje jeden parametr typu `int` i zwraca typ `int`.

W klasie `Program` utwórz metodę, zgodną z definicją delegatu `Transformer`, zmieniającą przekazaną wartość zmiennej np.:

```
static int Square(int x)     //this method should has exactly the same
    signature as delegate's in ExampleTransformer
{
  return x * x;
}
```

W kodzie klienta (`Main()`) stwórz instancję klasy `ExampleTransformer` i wywołaj metodę `Transform` przekazując jako pierwszy parametr metodę `Square` oraz jako drugi liczbę, która ma zostać poddana zmianie.

Dodaj analogiczną metodę, która będzie podnosić przekazaną wartość do sześcianu.

```
...
ExampleTransformer transformer = new ExampleTransformer();
transformer.Transform(Square, 10);
...
```

W .NET 3.5 zostały wprowadzone predefiniowane/gotowe generyczne delegaty `Action<TParameter>`[1] oraz `Func<TParameter, TOutput>`[2]. Korzystając z nich nie musimy tworzyć własnego typu `delegate` jak w przykładzie powyżej.

Dodaj do klasy `ExampleTransformer` metodę (ponieważ będzie ona posiadała inną sygnaturę może mieć taką samą nazwę), która zamiast delegatu `Transformer` będzie przyjmowała gotowy delegat `Func<>` z dwoma argumentami wejściowymi typu `int` i jednym wyjściowym również typu `int` (pierwsze dwa typy odnoszą się do parametrów wejściowych (ogólnie może ich być maksymalnie 16), ostatni jest zawsze typem zwracanym):

```
public void Transform(Func<int,int,int> t, int value1, int value2)
{
  int ret = t(value1, value2);
  Console.WriteLine($"Output value is: {ret}");
}
```

Analogicznie do metody `Square` dodaj metodę, która będzie zgodna z deklaracją powyższego delegatu (t.j. przyjmowała dwa argumenty typu `int` i zwracała typ `int`) np.:

```
static int Multiply(int x, int y)
{
  return x * y;
}
```

[1] `Action` jest wykorzystywany jeżeli chcemy, aby delegat **nie** zwracał wartości

[2] `Func` jest wykorzystywany jeżeli chcemy, aby delegat zwracał wartość

Wywołaj metodę `Transform(Func<int,int,int> t, int value1, int value2)` w metodzie `Main()` przekazując nowo utworzoną metodę jako egzemplarz delegatu (analogicznie jak w przypadku metody `Square`).

1.2 Wyrażenia lambda

Wyrażenie lambda to metoda bez nazwy wpisana w miejsce egzemplarza delegatu. Jest ona konwertowana na egzemplarz delegatu. Postać wyrażenia lambda jest następująca:
`(parametry)`[3]`=> wyrażenie-lub-blok-instrukcji`.
`x=>x*x` odpowiada przekazywanemu parametrowi, natomiast `x =>x*x` odpowiada typowi zwrotnemu. Najczęściej wyrażeń lambda używamy wraz z delegatami `Func` i `Action`.

Wywołaj teraz metodę `Transform(Transformer t, int value)` (w kodzie klienta, metoda `Main()`) wykorzystując wyrażenie lambda zamiast deklarowania osobnej metody. Przekaż wyrażanie lambda, które będzie podnosić przekazany argument `int` do czwartej potęgi:

```
transformer.Transform((x) => { return x * x * x * x; }, 2);
```

Wyrażenie lambda są szeroko stosowane w zapytaniach LINQ[4]. Przepisz poniższy kod i sprawdź jego działanie. Zwróć uwagę, że metoda `Where` przyjmuje właśnie delegat `Func<string, bool>`.

```
List<string> lst = new List<string>();
lst.Add("Ross Geller");
lst.Add("Chandler Bing");
lst.Add("Joey Tribbiani");
lst.Add("Monica Geller");

var gellers = lst.Where(x => x.Contains("Geller")).ToList();
```

Gdyby nie wyrażenia lambda pisanie takich zapytań byłoby bardziej pracochłonne. Dla chętnych zostawiam sprawdzenie innych standardowych operatorów zapytań `LINQ`.

1.3 Zdarzenia

Specyficznym rodzajem delegatu są zdarzenia[5]. Umożliwiają one klasie lub obiektowi powiadomienie subskrybentów (innych zainteresowanych tym zdarzeniem obiektów) o wystąpieniu czegoś.

Dodaj do klasy `ExampleTransformer` proste zdarzenie wykorzystując `Action` (w poniższym przykładzie został on zdefiniowany bez żadnych argumentów wejściowych). Dodatkowo dodaj wywołanie (poinformowanie wszystkich subskrybentów) tego zdarzenia w metodach `Transform` tak jak zostało to pokazane poniżej:

```
public event Action TransformationCompleted;

public void Transform(Transformer t, int value)
{
  ...
  TransformationCompleted?.Invoke(); //delegate invocation
}
```

W kodzie klienta zasubskrybuj ww. zdarzenie w sposób jak poniżej:

```
transformer.TransformationCompleted += () => { Console.WriteLine("
   Transformation completed!"); };
```

Wyrażenie to dodaje (`+=`) nowego subskrybenta do zdarzenia, w tym przypadku jest to blok instrukcji, który wypisuje tekst na ekranie konsoli. Zwróć uwagę, że powyżej zostało wykorzystane wyrażenie lambda, zamiast tego możliwe jest również przypisanie standardowej metody o odpowiedniej sygnaturze.

[3] Jeżeli lambda ma jeden parametr nawiasy są opcjonalne, inaczej wymagane. Gdy wyrażenie nie posiada parametrów wejściowych używamy pustego nawiasu. W przypadku większej ilości parametrów rodzielane są one przecinkami. Jeżeli kompilator nie może określić ich typów, konieczne jest ich określenie przez programistę.

[4] Technologia .NET pozwalająca na szybkie tworznenie zapytań do obiektów. Wykorzystuje się składnie podobną to SQL.

[5] W przeciwieństwie do delegatów, zdarzenia zapewniają zapobieżenie interakcji między subskrybentami, dodają one pewną dodatkową warstwę abstrakcji/bezpieczeństwa. Klient nie może zmienić listy inwokacji, jedynie dopisać/usunąć subskrybenta. Dodatkowo zdarzenia muszą być zgłoszone „z wnętrza" danej klasy.

1.4 Obsługa wyjątków

Wyjątki pomagają rozwiązać/obsłużyć problem niespodziewanych czy wyjątkowych sytuacji podczas wykonywania programu. Wyjątek może być zgłoszony przez CLR, biblioteki .NET albo przez kod programu. Aby obsłużyć wyjątek należy wykorzystać blok `try`, któremu musi towarzyszyć blok `catch` lub `finally`.

Wygenerujmy wyjątek `DivideByZeroException` przekazując wyrażenie lambda wykonujące dzielenie do metody `Transform(Func<int,int,int>, int value1, int value2)`. Przekaż `0` jako trzeci paramter tej metody, aby wykonać niedozwoloną operację dzielenia przez zero. Zauważ, że podczas wykonywania programu CLR zgłosił wyjątek, a program został przedwcześnie zakończony:

```
transformer.Transform((x,y) => { return x / y; }, 2, 0);
```

Umieść teraz operację wywołania delegatu w `Transform` w bloku `try-catch`. *W praktyce lepiej byłby wprost sprawdzić czy dzielnik nie ma wartości zero.*

```
try
{
  int ret = t(value1, value2);
  Console.WriteLine($"Output value is: {ret}");
  TransformationCompleted?.Invoke(); //delegate invocation
}
catch(Exception ex)
{
  Console.WriteLine(ex.Message);
}
```

Istnieje możliwość dodawania wielu bloków `catch` przechwytujących konkretne wyjątki. Te bardziej precyzyjne powinny być powyżej tych bardziej ogólnych. Dodaj do powyższego fragmentu kodu blok `catch` dla wyjątku `DivideByZeroException` (umieść go powyżej ogólniejszego bloku `catch(Exception ex)`):

```
catch(DivideByZeroException ex)
{
  Console.WriteLine(ex.Message); //log exception
  //do something extra...
}
```

Blok `finally` wykonywany jest zawsze, niezależnie czy wyjątek zostanie zgłoszony czy nie. Najczęściej w blokach `finally` wpisuje się kod porządkujący. Za ostatnim blokiem `catch` dodaj blok `finally`:

```
finally //usually used for clean up
{
  Console.WriteLine("Always called!");
}
```

W bloku `catch` możemy ponownie zgłosić wyjątek, aby przekazać go dalej. Do bloku `catch`, w którym zostają przechwytywane wszystkie wyjątki dodaj słowo kluczowe `throw` jak zostało to pokazane poniżej:

```
catch(Exception e)
{
  Console.WriteLine(e.Message); //log exception
  throw; //throw exception further to the caller
}
```

W takiej sytuacji wyjątek zostanie przekazany do podmiotu, który wywołaj funkcję. Jeśli tam nie zostanie on obsłużony nastąpi przerwanie programu.

2 Laboratorium nr 8

2.1 Refleksja

Refleksja polega na przeglądaniu metadanych i kodu skompilowanego, podczas wykonywania programu w środowisku uruchomieniowym. Wiele usług jest uzależnionych od obecności metadanych.

Dodaj do projektu nową klasę, niech zawiera kilka właściwości oraz metod. Za przykład niech posłuży nam klasa `Configuration` (w dalszej części laboratorium będziemy ją poddawać serializacji):

```
public class Configuration
{
  public int MaxUsersCount { get; set; }
  public int MinUserNameLength { get; set; }
  public int MaxUserNameLength { get; set; }
  public int MinUserPasswordLength { get; set; }
  public int MaxUserPasswordLength { get; set; }

  public static void Save(Configuration configuration, string
   configFilePath)
  {
    throw new NotImplementedException();
  }

  public static Configuration Load(string configFilePath)
  {
    throw new NotImplementedException();
  }
}
```

Następnie w metodzie `Main` stwórz obiekt typu `Type`[6] w następujący sposób:

```
Type configType = typeof(Configuration);
```

Teraz sprawdź jakie składowe zawiera typ `Type`. Możesz je podejrzeć wpisując słowo `configType` i kropkę, VS powinien wyświetlić Ci wszystkie dostępne składowe klasy `Type`. Wykorzystując mechanizm refleksji sprawdź jakie klasa `Configuration` zawiera metody wywołując `GetMethods()` (aby móc z niej skorzystać dodaj przestrzeń nazw System.Reflection):

```
MethodInfo[] methodInfo = configType.GetMethods();
Console.WriteLine("The methods of the Configuration class are: ");
foreach (MethodInfo temp in methodInfo)
{
  Console.WriteLine(temp.Name);
}
```

W analogiczny sposób sprawdź jakie nasza nowo utworzona klasa posiada właściwości:

```
PropertyInfo[] propertiesInfo = configType.GetProperties();
Console.WriteLine("The properties of the Configuration class are:");
foreach (PropertyInfo temp in propertiesInfo)
{
  Console.WriteLine(temp.Name);
}
```

Dodając do wywołania metody `GetMethods()` odpowiednie flagi można uzyskać informację o dowolnych składowych. Wywołaj metodę `GetMethods()` z parametrem `BindingFlags.Public|BindingFlags.Static`, aby otrzymać informacje o metodach, które są publiczne oraz statyczne:

```
MethodInfo[] methodInfo = configType.GetMethods(BindingFlags.Public|
   BindingFlags.Static);
```

Wykorzystując refleksję możemy również utworzyć instancję danej klasy:

```
Console.WriteLine("Specify full name of the class to create an object:");
string className = Console.ReadLine(); //one has to provide full name (
   with namespace!)
object ret = Activator.CreateInstance(Type.GetType(className));
```

[6] Reprezentuje deklarację typu. Zmienna typu `Type` pozwala na przechowywanie informacji o konkretnej klasie, interfejsie, typie wyliczeniowym itd.

```
Console.WriteLine(ret.GetType().FullName);
```

Za pomocą metody `SetValue` typu `PropertyInfo` możliwe jest również zmienienie wartości właściwości np.:

```
PropertyInfo propertyInfo = ret.GetType().GetProperty("MaxUsersCount");
propertyInfo.SetValue(ret, 12345);
Console.WriteLine("MaxUsersCount value has been set to: " + (ret as
    Configuration)?.MaxUsersCount); //before one has to cast to
    configuration (?. skip if null)
```

2.2 Atrybuty

Atrybuty pozwalają na związanie metadanych z kodem (zestawem, typem, metodą czy właściwością). Wykorzystując technikę zwaną refleksją możliwe jest „odpytywanie" o atrybuty w trakcie wykonywania programu. Omówiona w następnym podpunkcie serializacja jest jednym z wielu przykładów wykorzystania atrybutów oraz refleksji.

Wszystkie zestawy posiadają pewien zestaw metadanych, które opisują dany typ i jego składowe. Atrybuty mogą przyjmować parametry tak samo jak klasy czy właściwości. Możemy również tworzyć i dodawać swoje własne atrybuty.

Dodaj do klasy `Program` statyczną metodę i umieść przy niej atrybut warunkowy[7] tak jak pokazano poniżej:

```
[Conditional("TRACE_ON")]
public static void Log(string logMsg)
{
  Console.WriteLine(logMsg);
}
```

Teraz w klasie `Program` dodaj dyrektywę preprocesora w następujący sposób:

```
#define TRACE_ON
using System;
using System.Diagnostics;
...
```

Następnie w metodzie `Main()` wywołaj metodę `Log` dwukrotnie. Raz skompiluj program wraz z zdefiniowaną dyrektywą `#define` i drugi raz bez niej. **Jaka jest różnica? (odpowiedź umieść w komentarzu)**

Do typu/metody można dodać kilka argumentów warunkowych. Wtedy metoda zostanie wywołana jeżeli choćby jeden z nich będzie spełniony.

Innym argumentem, który sprawdzimy jest `Obsolete Attribute`, który oznacza pewną część kodu (cały typ/metoda itp.) jako przestarzałą/niezalecaną do korzystania z niej. Jeżeli zostanie taki typ/metoda użyta kompilator zgłosi ostrzeżenie, bądź błąd. Dodaj pokazany poniżej atrybut obok atrybutu warunkowego przy metodzie `Log(string logMsg)`:

```
[Obsolete("Legacy implementation, use ... instead")]    //warning
```

albo

```
[Obsolete("Legacy implementation, use ... instead", true)]  //error
```

Możliwe jest również tworzenie własnych atrybutów. Aby to zrobić dodaj do projektu w osobnym pliku klasę dziedziczącą po klasie `Attribute`. Dodając do naszej klasy atrybut `AttributeUsage` określamy do jakich elementów atrybut może być stosowany (w poniższym przypadku do klas):

```
[AttributeUsage(AttributeTargets.Class)]
public class CustomAttribute : Attribute
{
  public string Author { get; }
  public string Description { get; }

```

[7] Atrybut warunkowy sprawia, że wywołanie metody jest zależne od dyrektywy preprocesora. Jeżeli dyrektywa ta nie zostanie dodana, metoda nie zostanie wywołana. W tym przypadku alternatywą może być wykorzystanie bloku `#if Debug ... #endif`

```
  public CustomAttribute(string author, string description)
  {
    Author = author;
    Description = description;
  }
}
```

Dodaj utworzony atrybut do klasy `Configuration`:

```
[CustomAttribute("JG", "Foolin'around with custom attributes...")]
public class Configuration
{
  public int MaxUsersCount { get; set; }
  ...
...
```

W kodzie klienta odczytaj wartości parametrów atrybutu:

```
var customAttributes = typeof(Configuration).GetCustomAttributes();
foreach(var attribute in customAttributes)
{
  Console.WriteLine((attribute as CustomAttribute)?.Author);
  Console.WriteLine((attribute as CustomAttribute)?.Description);
}
```

2.3 Serializacja

Przykład serializacji[8]pokażemy na przykładzie serializacji XML, która serializuje pola publiczne i właściwości w strumieniu XML (C# wspiera również serializację JSON[9]). Pozwala to na konwersję pomiędzy obiektem w programie, a plikiem XML.

Zaimplementuj metodę `Save(Configuration configuration, string configFilePath)` tak, aby wykonywała serializację obiektu typu `Configuration` to pliku `xml` zlokalizowanego w `configFilePath`:

```
public static void Save(Configuration configuration, string configFilePath
    )
{
  XmlSerializer xmlSerializer = new XmlSerializer(typeof(Configuration));
  using (var streamWriter = new StreamWriter(configFilePath))
  {
    xmlSerializer.Serialize(streamWriter, configuration);
  }
}
```

W pierwszej kolejności utworzony został obiekt XmlSerializer, do konstruktora został przekazany typ który będzie poddawany serializacji. Następnie w instrukcji `using`[10]został utworzony obiekt `StreamWriter`, który będzie odpowiedzialny za pisanie znaków do strumienia (w naszym przypadku pliku `xml`). Na końcu wywoływana jest metoda wykonująca faktyczną serializację.

W metodzie `Main()` utwórz obiekt typu `Configuration`, przypisz właściwościom pewne wartości i wywołaj `Configuration.Save()` przekazując odpowiednie argumenty:

```
Configuration.Save(configuration, "Config.xml");
```

Sprawdź w folderze `Debug` w swoim projekcie czy został utworzony plik `Config.xml` wraz z odpowiednia zawartością.

Zaimplementuj teraz metodę `Configuration Load(string configFilePath)`, która dokona operacji odwrotnej:

[8]Serializacja jest mechanizmem, który pozwala na zamianę obiektu na strumień bajtów. Taki obiekt możemy w zmienionej postaci przesłać przez sieć, zapisać do pamięci, bazy danych albo pliku. Proces odwrotny jest nazywany deserializacją.

[9]JSON jest formatem tekstowym, który bazuje na podzbiorze języka JavaScript. Jest szeroko wykorzystywany do udostępniania danych w Internecie. Nazwy klas do (de)serializacji znajdują się w przestrzeni nazw `System.Text.Json`.

[10]Niektóre klasy posiadają niezarządzalne zasoby, takie jak: uchyty do plików, uchwyty graficzne czy połączenie sieciowe albo z bazami danych. Klasy te implementują interfejs `IDisposable`, który implementuje metodę `Dispose` służącą do zwlaniania wykorzysytywanych zasobów. `using` zapewnia elegancką składnie do wywoływania metody `Dispose`.

```
public static Configuration Load(string configFilePath)
{
  XmlSerializer xmlSerializer = new XmlSerializer(typeof(Configuration));
  using (var streamReader = new StreamReader(configFilePath))
  {
    return (Configuration)xmlSerializer.Deserialize(streamReader);
  }
}
```

Ręcznie zmień wartości elementów w pliku `*.xml` i sprawdź czy są one zgodne z właściwościami obiektu zwróconego przez metodę `Load`.

Bibliografia

[1] Ben Albahari Joseph Albahari. C# 7.0 w pigułce. Helion, Gliwice 2018.

[2] Microsoft. Serializacja (C#). [Online; 08-05-2020]. URL: https://docs.microsoft.com/pl-pl/dotnet/csharp/programming-guide/concepts/serialization/.