Metody rozszerzające (ang. *extensions methods*)

Krzysztof Molenda	2019-11-10	C#

Klasycznym sposobem rozszerzania funkcjonalności klas jest dziedziczenie. Jednak co zrobić, jeśli klasa jest zapieczętowana (`sealed`)?

Język C# dostarcza mechanizmu **metod rozszerzających** -- czyli możliwości dodania do istniejącego typu pewnych metod, bez konieczności modyfikowania tego typu.

Metoda rozszerzająca to specjalna metoda, zaprogramowana w specjalnym miejscu i w specjalny sposób, jednak jej użycie jest takie samo, jak metod zaimplementowanych w typie -- wywoływane są na rzecz instancji.

Przykład

Potrzebujemy funkcji zwracającej liczbę wyrazów w zadanym napisie (przyjmujemy, że wyraz, to ciąg znaków oddzielony spacją).

Na przykład w napisie `"Ala ma kota, As to Ali pies"` jest 7 wyrazów.

W klasie `string` nie ma takiej funkcjonalności.

Podejście 1

Definiujemy statyczną, niegeneryczną i niezagnieżdżoną klasę o nazwie `Utility` w przestrzeni nazw `MyExtensionMethods`

```
//file: Utility.cs
namespace MyExtensionMethods
{
    public static class Utility
    {

    }
}
```

W klasie tej umieszczamy metodę `WordCount` . Do rozbicia napisu na wyrazy wykorzystujemy metodę `Split` z klasy `string` .

```
public static class Utility
{
    public static int WordCount(string napis)
    {
        return napis.Split(' ').Length;
    }
}
```

W programie głównym sprawdzamy jej działanie:

```
string napis = "Ala  ma kota, as to Ali pies";
Console.WriteLine( Utility.WordCount(napis) );
```

```
8
```

Drobny kłopot -- wyrazów jest `7` , a program zwraca liczbę `8` . Powodem jest podwójna spacja między wyrazami *Ala* oraz *ma*. Drobna korekta kodu:

```
    public static class Utility
    {
        public static int WordCount(string napis)
        {
5           return napis.Split(' ', StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
```

Podejście 2

Do kodu tak opracowanej funkcji dodajemy słowo kluczowe `this` przed typ, który będzie rozszerzany:

```
    public static class Utility
    {
3       public static int WordCount(this string napis)
```

```
        {
            return napis.Split(' ', StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
```

Możemy teraz użyć tak opracowanej funkcji po staremu -- jako metody statycznej klasy `Utility` oraz po nowemu -- jako metody uruchamianej na rzecz instancji. Wcześniej jednak należy użyć dyrektywy `using` importując przestrzeń nazw z klasą definiującą metody rozszerzające:

```
    using System;
    using MyExtensionMethods;

    namespace KM.ExtensionsMethodsExample
    {
        class Program
        {
            static void Main(string[] args)
            {
                string napis = "Ala  ma kota, as to Ali pies";
                Console.WriteLine(Utility.WordCount(napis)); // static method
                Console.WriteLine(napis.WordCount());        // extension method
            }
        }
    }
```

Podejście 3

Separatorem może by nie tylko spacja, ale każdy inny biały znak czy znaki przestankowe `' '`, `'\t'`, `'\n'`, `'.'`, `','`, `'!'`, `'?'` i.t.p.

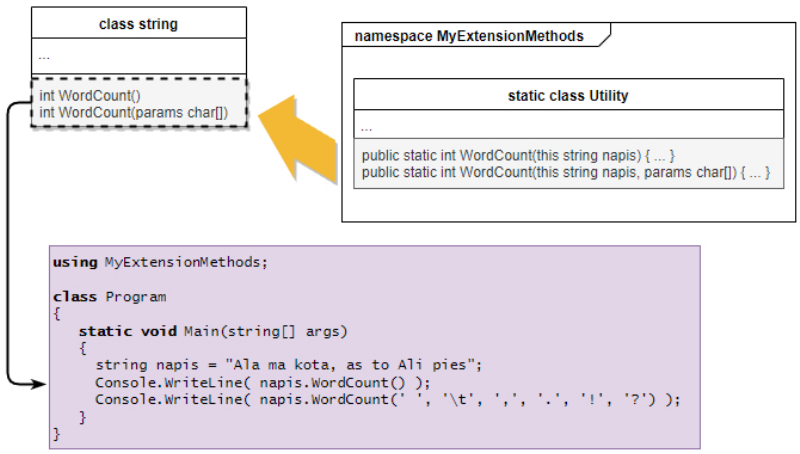
Dodamy przeciążony wariant metody `WordCount` przyjmującej jako parametr tablicę znaków mogących być separatorem.

W klasie `Utility` umieszczamy drugi wariant metody:

```
public static int WordCount(this string napis, params char[] delimiters)
{
    return napis
            .Split(delimiters, StringSplitOptions.RemoveEmptyEntries)
            .Length;
}
```

i testujemy w programie:

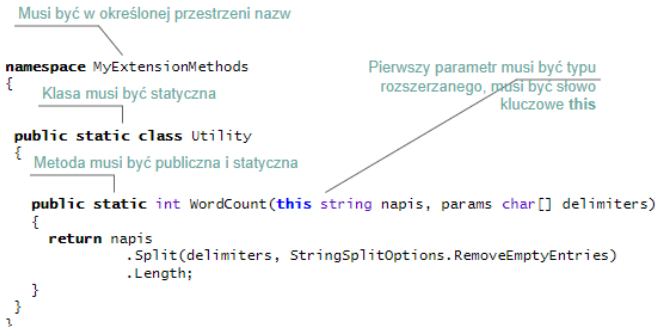
```
Console.WriteLine( Utility.WordCount(napis, ' ', '\t', ',', '.', '!', '?') );
Console.WriteLine( napis.WordCount( delimiters: new char[] {' ', '\t', ',', '.', '!', '?'} ) );
Console.WriteLine( napis.WordCount( ' ', '\t', ',', '.', '!', '?') ); // bo params
```



⚠️ -- Podsumowanie -- ⚠️

1. Metody rozszerzające mogą być dodane do klas wbudowanych środowiska .NET lub klas własnych. Dotyczy to również struktur i interfejsów.
2. Metoda rozszerzająca musi być umieszczona w klasie statycznej, niezagnieżdżonej i niegenerycznej. Nazwa klasy nie ma znaczenia. Klasa musi być widoczna dla kodu klienta.
3. Klasa, w której umieszczona jest metoda rozszerzająca, znajduje się w pewnej przestrzeni nazw. Jej nazwa nie ma znaczenia.
4. Metoda rozszerzająca może być użyta w kodzie w dowolnym miejscu, jeśli tylko wskazano przestrzeń nazw w której została zdefiniowana

5. Pierwszy parametr metody rozszerzającej jest typu rozszerzanego oraz poprzedzony jest słowem kluczowym `this`.
6. Sygnatura metody rozszerzającej musi być inna niż te, które są już zaimplementowane w typie. Jeśli jednak tak się zdarzy, Twoja metoda nie będzie uruchamiana.
7. Zalecenie: definiuj metody rozszerzające oszczędnie, tylko wtedy, kiedy musisz lub znacząco uprości to Twój kod. Generalnym sposobem rozszerzania typu jest dziedziczenie.
8. Zalecenie: grupuj metody rozszerzające dla danego typu w jednej klasie. Będzie Ci łatwiej zarządzać kodem.
9. Uwaga: koncepcja metod rozszerzających nie może być zastosowana do pól (*fields*), właściwości (*properties*) i zdarzeń (*events*).




```
namespace MyExtensionMethods
{
  public static class Utility
  {
    public static int WordCount(this string napis, params char[] delimiters)
    {
      return napis
               .Split(delimiters, StringSplitOptions.RemoveEmptyEntries)
               .Length;
    }
  }
}
```

Referencje

- Extension Methods (C# Programming Guide)
- EXTENSIONMETHOD.NET

Zadania

Zadanie 1

Napisz metodę rozszerzającą klasę `string` o nazwie `BezSamoglosek`, zwracającą napis przekazany jako parametr, ale pozbawiony samogłosek.

Zadanie 2

W języku VisualBasic jest dostępna funkcja `IsNumeric(ByVal x As Object) As Boolean` zwracająca prawdę, jeśli przekazany argument ma postać liczby (może być przekonwertowany na liczbę).

Napisz w C# metodę rozszerzającą typ `string` o nazwie `IsNumeric`, zwracającą prawdę, jeśli argument typu `string` jest konwertowalny na liczbę całkowitą. Zadbaj, aby nie zwracała ona wyjątków.

Zadanie 3

Listy w C# nie mają zdefiniowanego przesłonięcia metody `ToString()`. W efekcie, wywołanie `WriteLine` dla listy skutkuje wypisaniem informacji o typie:

```
var lista = new List<int> {0, 1, 2, 3, 4};
Console.WriteLine( lista );
```

```
System.Collections.Generic.List`1[System.Int32]
```

Nie możesz rozszerzyć listy o metodę `ToString()`, ponieważ jest ona zdefiniowana w klasie `List<T>` (odziedziczona z `object`). Dokładniej -- nawet, jak zdefiniujesz, to i tak się ona nie uruchomi, ponieważ pierwszeństwo w wywołaniu ma metoda zaimplementowana w klasie.

Napisz metodę rozszerzającą interfejs `IList<T>` o nazwie `Dump` zwracającą napis zawierający wszystkie elementy listy, oddzielone od siebie przecinkiem i spacją. Elementy te powinny być ujęte w nawiasy klamrowe. Np. dla wcześniejszego kodu wywołanie

```
Console.WriteLine( lista.Dump() );
```

powinno wypisać na konsoli

```
{0, 1, 2, 3, 4}
```

Zadanie 4

Napisz generyczną metodę `void PrintLn<T>()` rozszerzającą typ `IEnumerable<T>`, wypisującą na konsolę kolejne elementy sekwencji w oddzielnych wierszach.

Zadanie 5

Zaprojektuj metodę rozszerzającą sekwencję liczb typu `int` wyznaczającą medianę (wartość środkową w ciągu posortowanym).

Zadanie 6

Zaprojektuj generyczną metodę rozszerzającą o nazwie `Between<T>`, przyjmującą parametry `T lower` oraz `T upper` i zwracającą `true` jeśli argument zawarty jest między `lower` i `upper` włącznie.

Przykłady użycia:

```
if( liczba.Between(0,9) ) ...
if( napis.Between("ala", "baba") ) ...
...
```

Uwaga: aby można było porównywać elementy typu `T`, musi on definiować naturalny porządek (być `IComparable<T>`).

Zadanie 7

Zaprojektuj rozszerzony wariant metody z zadania poprzedniego, uwzględniający parametr typu delegatowego `IComparison<T>`, wykorzystany jako zewnętrzny sposób porównywania. Typ `T` wtedy nie musi być `IComparable<T>`.