


Programowanie w .NET – Kolekcje i LINQ

Jan Polak
Poznań/21-03-2016
Wersja 1.2

- 
1. Tablice i Listy
 2. IList, ICollection, IEnumerable
 3. Operacje na kolekcjach i wyliczeniach
 4. Extension Method, Lambda, LINQ

Klasy pomocnicze

- Klasy Student i Exam pomogą nam zapoznać się z operowaniem na kolekcjach
 - Student (obiekt klasy Student) zawiera listę egzaminów (obiektów klasy Exam) – kompozycja
- Klasa OurStudents zawiera listę studentów na której będziemy dalej operować
 - Pole prywatne jest ukryte przed bezpośrednim dostępem z zewnątrz; dostęp jest kontrolowany przez właściwości tylko do odczytu – enkapsulacja

<https://github.com/plbo/NETLab05>

Tablice i listy

- W języku C# mamy dostępnych wiele konstrukcji pozwalających na tworzenie zestawów danych
 - Tablice
 - Kolekcje niegeneryczne
 - ArrayList
 - SortedList
 - Hashtable
 - BitArray
 - Queue
 - Stack
 - Generyczne
 - List<T>
 - Dictionary<K, V>

Tablica (T[], System.Array)

- Tablica to najprostsza forma reprezentacji zestawu danych tego samego typu
- Jej deklaracja i obsługa jest standardowym elementem składni języka
 - Deklaracja zmiennej: `string[] tablica;`
 - Inicjalizacja zmiennej: `tablica = new string[8];`
 - Zapis i odczyt elementów przez indeks: `tablica[0] = „wartość”;`
- Dostęp do wybranego elementu tablicy otrzymujemy poprzez jego indeks (numer kolejny) w tablicy; numeracja od 0
- Zalety i wady tablic
 - Stały, szybki dostęp do każdego z elementów tablicy
 - Zmiana rozmiaru tablicy wymaga ponownej alokacji pamięci
- Dodatkowe operacje dla tablic dostępne są przez metody statyczne klasy `System.Array`
 - Sortowanie
 - Wyszukiwanie
 - Kopiowanie

Lista generyczna (System.Collections.Generic.List<T>)

- Lista jest najczęściej używaną kolekcją
- Domyślna implementacja bazuje na tablicy
- Jest typem generycznym, każdy element listy musi być konwertowalny do parametru generycznego podanego podczas tworzenia listy
- Automatycznie rozszerza się do potrzeb
 - Rozmiar zwiększany jest skokowo
 - Domyślnie alokowana jest pamięć na 4 elementy; można to zmienić podając w konstruktorze parametr *capacity*
- Dostarcza funkcjonalności niedostępne standardowo w tablicy*, jak sortowanie w miejscu, wyszukiwanie elementów
- Deklaracja i inicjalizacja:
 - Tworzenie zmiennej: `List<string> lista;`
 - Inicjalizacja zmiennej: `lista = new List<string>();`

Inicjalizacja kolekcji

- Większość kolekcji możemy zainicjować wartościami poprzez specjalną składnię
- Może to być zrobione poza blokiem metody, dzięki temu deklaracja obiektów może być prostsza i bardziej czytelna
- Deklaracja tablicy:
 - `int[] tablica = { 1, 2, 5, 7, 11, 13, 17 };`
- Deklaracja listy
 - `List<int> lista = new List<int> { 1, 2, 5, 7, 11, 13, 17 };`
- W obu przypadkach rozmiar zostanie dostosowany do liczby podanych elementów
- Inicjalizacja może być zagnieżdżana i wielolinijkowa, można stworzyć za jej pomocą rozbudowany graf obiektów

ICollection, IEnumerable

- Standardowe kolekcje implementują interfejsy na których możemy operować w celu uwolnienia się od konieczności podawania konkretnej implementacji
- Interfejsy te pozwalają na implementowanie własnych kolekcji i innych wyliczeń, które mogą być wykorzystywane w standardowych metodach dostępnych na platformie
- Dostępne są zarówno w wersjach generycznych jak i niegenerycznych co pozwala na operowanie na kolekcjach niezależnie od typu zawartości oraz dostarcza kompatybilność z innymi platformami niewspierającymi generycznych typów
- Interfejsy mają hierarchię która pozwala na stopniowe dodawanie funkcjonalności:
 - IList dziedziczy z ICollection
 - ICollection dziedziczy z IEnumerable
- Uwaga – nie wszystkie metody i właściwości kolekcji są dostępne w interfejsach!

ICollection i IList

■ IList

- Zawiera większość właściwości i metod dostępnych w klasie List
- Dostarcza funkcjonalność zbliżoną do tablicy
 - Dostęp do elementu po jego indeksie
 - Sprawdzanie indeksu elementu
 - Wstawianie i usuwanie elementu we wskazanym miejscu

■ ICollection, ICollection<T>

- ICollection definiuje kolekcję o określonym rozmiarze – udostępnia właściwość Count; implikuje to założenie że wyliczenie jest skończone
- ICollection<T> umożliwia modyfikowanie kolekcji:
 - Dodawanie elementów
 - Usuwanie elementów
 - Czyszczenie kolekcji

IEnumerable

- Dostarcza ogólną abstrakcję wyliczenia
- Definiuje, że implementująca ją klasa dostarcza enumerator (IEnumerator)
 - Enumerator posiada trzy składowe:
 - Aktualnie dostępny obiekt (właściwość Current)
 - Przejście do następnego elementu (metoda MoveNext)
 - Zresetowanie wyliczenia (metoda Reset)
- Jest podstawą konstrukcji języka (np. foreach) i bazą dla wielu jego rozszerzeń (np. LINQ)
- Wyliczenia mogą być nieskończone – nie mają określonego rozmiaru
- Konsument wyliczenia nie może bezpośrednio go modyfikować
- Wyliczenia mogą operować zarówno na kolekcjach dostępnych w pamięci, jak i obiektach zewnętrznych (np. plikach, danych z sieci, danych z urządzeń itp.)

Słowniki (System.Collections.Generic.Dictionary<K,V>)

- Słownik dostarcza funkcjonalność kolekcji indeksowanej (prawie) dowolnym kluczem – key-value storage
- Wykorzystuje podstawowe mechanizmy obiektów do optymalizacji dostępu do danych (hashcode obiektu)
- Dostęp do elementów słownika jest w stałym czasie
- Jest typem generycznym – to my definiujemy co będzie dla nas kluczem (pierwszy parametr), a co wartością (drugi parametr):
 - `var slownik = new Dictionary<string, Student>();`
- Dostęp do elementów słownika odbywa się poprzez indeks, w którym podajemy klucz do oczekiwanej wartości:
 - `var kowalski = slownik["Kowalski"];`
 - `slownik["Kowalski"] = new Student();`
- Słownik możemy modyfikować – dodawać, usuwać elementy, zmieniać wartość przechowywaną pod danym kluczem

Operacje na kolekcjach i wyliczeniach

- Język C# dostarcza kilka przydatnych konstrukcji które umożliwiają nam wygodne operowanie na wyliczeniach
 - Standardowe pętle:
 - for – umożliwia proste operowanie na tablicach i listach poprzez indeksy
 - while, do...while – umożliwiają na tworzenie bardziej zaawansowanych bloków operacji np. na enumeratorach
 - Pętla foreach
 - Upraszcza operowanie na elementach wyliczenia
 - Daje wygodny dostęp do aktualnego elementu wyliczenia
 - Prosta składnia: foreach (TYP element in kolekcja) { ... }
 - Najczęściej wykorzystywana pętla przy przetwarzaniu wyliczeń
 - Słowo kluczowe yield
 - Ułatwia tworzenie własnych wyliczeń
 - Umożliwia tworzenie nieskończonych i leniwie ewaluowanych wyliczeń

Extension Method

- Sposób na rozszerzanie możliwości istniejących klas bez konieczności ich modyfikowania
- Umożliwiają dodawanie zachowań nie przewidzianych przez autora klasy
- Definiuje się je w osobnych, statycznych, niegenerycznych klasach, jako metody statyczne których pierwszym parametrem jest obiekt typu który chcemy rozszerzyć; konieczne jest poprzedzenie parametru słowem kluczowym `this`. Przykład metody rozszerzającej klasę `Student`:
 - `public static int PassedExamsCount(this Student student)`
 - Wywołanie: `int passedCount = someStudent.PassedExamsCount();`
- Można je też wywoływać jak standardowe metody – nie ma obowiązku odwoływania się do nich jako `extension method`:
 - `int passedCount = StudentExtensions.PassedExamsCount(someStudent);`
- Ze względu na to że są zdefiniowane w zewnętrznych klasach nie mają dostępu do prywatnych i chronionych elementów klasy którą rozszerzają (obowiązuje pełna enkapsulacja zgodnie z OOP)
- Uwaga! Zbyt dużo `extension method` może spowodować chaos w kodzie i problemy z niejednoznacznością metod (jeżeli np. mamy metodę o tej samej nazwie zdefiniowaną w różnych przestrzeniach nazw, które używamy w tym samym bloku kodu)

Funkcje jako obiekty

- Każda funkcja w języku C# jest również obiektem
- Dostęp do obiektu funkcji odbywa się poprzez jej nazwę (poprzedzoną nazwą klasy gdy jest zdefiniowana poza aktualną klasą oraz ewentualnie przestrzenią nazw)
- Obiekty funkcji można przekazywać jako parametry do innych funkcji, przypisywać do zmiennych, zwracać jako wynik funkcji
- Do definiowania typu zmiennej która przechowuje uchwyt do funkcji używane jest słowo kluczowe `delegate`
- W .NET jest zdefiniowanych kilka typów które umożliwiają proste operowanie na obiektach funkcji:
 - `Action`, `Action<P1>`, `Action<P1, P2>`, ...
 - Opisuje funkcję która nie posiada wyniku (`void`); kolejne parametry generyczne określają typy parametrów wejściowych do funkcji
 - `Func<TResult>`, `Func<P1, TResult>`, `Func<P1, P2, TResult>`, ...
 - Opisuje funkcję która zwraca wynik typu `TResult`; kolejne parametry generyczne określają typy parametrów wejściowych do funkcji, ostatni – typ wyniku

Funkcje anonimowe (lambdy)

- Umożliwiają stworzenie funkcji „ad hoc” bez konieczności umieszczania jej w jakiejś klasie
- Zachowują zbliżoną funkcjonalność do metod definiowanych w klasach (mogą przyjmować parametry wejściowe, mogą zwracać wynik, mogą zawierać wiele linii kodu)
- Nie posiadają własnej nazwy, ale mogą być podstawiane do zmiennych i przekazywane jako parametry funkcji (podobnie jak standardowe funkcje)
- Mogą być definiowane w tej samej linii w której są używane
- Pełna składnia lambda:
 - `(TYPE p1, TYPE p2, ...) => { ... }`
 - Po lewej stronie mamy listę parametrów wejściowych (podobnie jak w standardowych funkcjach); może ich być 0 lub więcej
 - Po prawej stronie mamy ciało metody
 - Jeżeli lambda zwraca wynik, ostatnią instrukcją musi być słowo kluczowe `return` i wyrażenie reprezentujące zwracaną wartość
- Najprostsza, bezparametrowa lambda która nic nie wykonuje i nic nie zwraca: `() => {}`

Funkcje anonimowe (lambdy) – skrócone formy

- Gdy kompilator jest w stanie wywnioskować typ parametru z kontekstu (np. gdy przekazujemy lambdę jako parametr do funkcji wymagającej konkretnej sygnatury funkcji), możemy pominąć typy parametrów wejściowych:
 - `(p1, p2, ...) => { ... }`
- Gdy ciało naszej metody ma tylko jedną instrukcję, która jest wyrażeniem zawierającym wynik funkcji, możemy pominąć nawiasy klamrowe i słowo `return`:
 - Zamiast: `(p1, p2) => { return p1 + p2; }` możemy napisać: `(p1, p2) => p1 + p2`
- Gdy nasza funkcja ma tylko jeden parametr którego typ jest oczywisty dla kompilatora:
 - `p1 => { ... }`
- Funkcja przyjmująca parametr która nic nie robi, tylko zwraca go z powrotem:
 - `p1 => p1`
- Gdy nasza funkcja nie przyjmuje żadnych parametrów: `() => { ... }`

LINQ (System.Linq)

- LINQ to biblioteka dostarczająca zestaw rozszerzeń ułatwiających pracę z kolekcjami
- Udostępnia dwa rodzaje składni:
 - Składnia zbudowana na extension method, pozwalająca na łańcuchowe wywoływanie metod i przekazywanie ich pomiędzy kontekstami (tzw. fluent interface)
 - Składnia przypominająca SQL pozwalająca na tworzenie kodu podobnego do zapytań do baz danych
- Bazuje na extension method i dodatkowych interfejsach, a także na przekazywaniu funkcji jako parametrów (zarówno nazwanych jak i anonimowych – lambda)
- Rozszerzenia są zdefiniowane w klasie Enumerable, dostępnej w przestrzeni System.Linq; w celu korzystania z metod Linq należy dołączyć tę przestrzeń za pomocą klauzuli using
- Wywołania metod można ze sobą łączyć w łańcuchy które można budować warunkowo w zależności od kontekstu
- Większość metod jest leniwie ewaluowana: dopóki nie zaczniemy odwoływać się do elementów wynikowej kolekcji, żaden kod ze zbudowanych zapytań nie zostanie wywołany; pozwala to na optymalizację wywołań i zużycia pamięci

Rozszerzenia dostępne w LINQ

- Filtrowanie (selekcja i projekcja) – Where, Select, OfType, Distinct, Take, TakeWhile, Skip, SkipWhile
- Sortowanie – OrderBy, OrderByDescending
- Odnajdywanie pojedynczych elementów – First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault, Contains
- Operacje skalarne na zbiorach – All, Any, Sum, Average, Aggregate, Max, Min
- Konwersje: ToList, ToArray, ToDictionary, Cast, Reverse
- Łączenie: Concat, Join, GroupBy, GroupJoin, Union, Zip
- ...i wiele innych

Podsumowanie

Zostały omówione:

- Podstawowe kolekcje dostępne w systemie
- Operowanie na kolekcjach i wyliczeniach
- Wyliczenia i ich zastosowania
- Słowniki i operacje na nich
- Extension Method
- Funkcje anonimowe (lambdy)
- LINQ – co to jest, do czego się wykorzystuje, jakie są podstawowe metody

Pytania kontrolne

- Czy się różni wyliczenie od kolekcji?
- Czy bezpieczne jest używanie pętli foreach dla każdego IEnumerable?
- Kiedy warto używać list, a kiedy tablic?
- Co może być kluczem dla elementu słownika?
- Kiedy zostanie wywołana nasza funkcja przekazana jako parametr do metody Where z LINQ?
- Czy się różni lambda od metody zdefiniowanej w klasie?
- Czy za pomocą extension method możemy modyfikować prywatne właściwości rozszerzanej klasy?