Lekcja: Kolekcje

Tablice, listy, kolejki, drzewa i różne ich podtypy, zostały w C# nazwane kolekcjami. Co więcej większość typowych kolekcji (a także kilka dodatkowych) została zaimplementowana w platformie .NET i jest gotowa do użycia.

* Do wersji 2.0 .Net została dodana przestrzeń nazw *System.Collections.Generic* zawierajaca parametryzowaną wersję kolekcji.
* Dostępna jest również przestrzeń nazw *System.Collections.Specialized* zawierająca ich wyspecjalizowane wersje.
* Warto zwrócić uwagę na często używaną listę zaimplementowaną w klasie List<> oraz na SortedList<> -- słownik, w którym elementy są automatycznie sortowane.

**Tablice**

Tablica to struktura, która przechowuje elementy tego samego typu. Inną sprawą jest jej implementacja. W C# zrobiono to w taki sposób, aby niemożliwe było popełnienie często występującego błędu w C/C++, w którym dochodzi do odczytu lub zapisu danych za zadeklarowanym obszarem tablicy. W C# indeksy są kontrolowane przez klasę implementującą tablicę. O ile w C/C++ tablica to po prostu fragment pamięci z adresem zapamiętanym we wskaźniku do pierwszego elementu, to w C# jest ona instancją klasy System.Array. Oznacza to, że tablica jest obiektem. Dzięki temu jest możliwe kontrolowanie tego, czy indeksy nie są mniejsze od zera lub czy nie są większe od rozmiaru tablicy (minus jeden).

Pomimo innej implementacji, praktyczne zasady dotyczące tworzenia tablic w C# są niemal identyczne z zasadami rządzącymi dynamicznie tworzonymi tablicami w C++. Możemy zadeklarować tablicę, co oznacza utworzenie referencji do tablicy, oraz ją zdefiniować, czyli zarezerwować pamięć , a następnie zainicjować, czyli zapełnić wartościami.

Tablice w C# mają jeden specjalny atrybut: są zaimplementowane jako obiekty. W deklaracji tablic musimy określić typ wartości jaki ma przechowywać tablica, a także liczbę elementów tablicy. Mogą być tablice jedno lub dwuwymiarowe itd.

Poprawna składnia deklaracji referencji do tablicy z elementami typu int:

*int[] ti;*

A o to przykład deklaracji referencji wraz z utworzeniem obiektu tablicy (rezerwowana jest pamięć na stercie):

*int[] ti = new int[3];*

Po utworzeniu obiektu tablicy (tj. w istocie instancji klasy System.Array) jest ona automatycznie inicjowana wartościami domyślnymi dla danego typu, czyli w przypadku typu int – zerami. Możemy także samodzielnie zainicjować elementy tablicy:

*int[] ti = new int[3] {1,2,3};*

W tablicy zawierającej typy referencyjne domyślne wartości elementów tablicy to null. Konieczna jest wobec tego jawna inicjacja elementów przez wywołanie operatora new dla każdego elementu tablicy i utworzenie odpowiedniej grupy obiektów. Nie należy mylić w tym przypadku wykorzystania operatora new do utworzenia tablicy z użyciem operatora new do utworzenia jej elementów. Polecenia:

**(1)**

*Exception[] te*

*Console.WriteLine(„ Komunikat: „ +te[0].Message);*

**(2)**

*Exception[] te =new Exception[3];*

*Console.WriteLine(„Komunikat: „ + te[0].Message);*

**(3)**

*Exception[] te =new Exception[3] {new Exception(),new Exception(), new Exception() };*

*Console.WriteLine(„Komunikat: „ + te[0].Message);*

1. Polecenie pierwsze w ogóle się nie skompiluje, ponieważ w drugiej linii próbujemy odwołać się do zmiennej te (referencji do tablicy), która nie została zainicjowana.
2. Polecenia te skompilują się program może się nawet uruchomić, ale podczas działania zgłoszony zostanie wyjątek System.NullReferenceException. Dlaczego? Tablica te jest co prawda utworzona (obiekt tablicy powstał) i zainicjowana, ale domyślne wartości referencji, a to one są tu elementami przechowywanymi w tablicy, to null. Nie powstał żaden obiekt klasy Exception, więc elementy tablicy te nie mogą przechowywać żadnych adresów, a co za tym idzie próba odczytania właściwości Message musi skończyć się błędem.
3. Dopiero trzecia możliwość czyli tablica zostaje zapełniona referencjami zawierającymi adresy istniejących obiektów i instrukcja Console.WriteLine(„Komunikat: „+te[0].Message); pokaże domyślny komunikat pierwszego wyjątku z tablicy.

Obiekty, do których odnoszą się referencje przechowywane w tablicy, nie muszą być tworzone w tej samej linii co deklaracja tablicy, można wykorzystać do tego pętlę.

*Exception[] te =new Exception[3];*

*for (int i = 0; i<te.Length;i++)*

*{*

*te[i] = new Exception(“Komunikat “+i);*

*te [i].HelpLink =* [*www.youtob.com*](http://www.youtob.com)*;*

*}*

Elementy tablicy indeksowane od 0. Ostatni element ma zatem indeks rozmiar – 1. W każdej chwili rozmiar tablicy można odczytać z właściwości Length. Rozmiaru tablicy nie można zmienić – jeżeli potrzebujemy takiej możliwości, musimy wybrać inną kolekcję, np. Listę *List<>*.

Dostęp do elementów tablicy jest realizowany za pomocą operatora[] z indeksem elementu wewnątrz, czyli w sposób identyczny jak w C czy C++.

Nieco inaczej niż w C czy C++ definiuje się tablice wielowymiarowe. Nie jest to tablica tablic, a nadal obiekt klasy System.Array, którego elementy są odpowiednio poukładane. Przykład:

Int [,] ti2 = new[2,3]{{0,1,2},{3,4,5}};

**Zadanie 1**

Napisz program, który w 10-elementowej tablicy jednowymiarowej o nazwiedane umieszcza liczby z przedziału od 0 do 9 (zobacz reprezentację graficzną tablicy zamieszczoną po tym poleceniu).

Indeks tablicy Wartość tablicy

0 0

1 1

2 2

3 3

4 4

5 5

6 6

7 7

8 8

9 9

*static void Main(string[] args)*

*{*

*int n = 10, i;*

*int[] dane = new int[n];* // Deklaracja tablicy typu int o rozmiarze n.

*Console.WriteLine("Program wyświetla zawartość tablicy´jednowymiarowej " + n + " - elementowej.");*

*Console.WriteLine();*

*for (i = 0; i < dane.Length; i++) // Długość tablicy (liczbę*

*// elementów) odczytujemy, korzystając z właściwości Length.*

*{*

*dane[i] = i; // Wpisywanie do tablicy kolejnych liczb od 0 do 9.*

*Console.WriteLine("dane[" + i + "] = " + dane[i]);*

*// Wyświetlenie zawartości tablicy.*

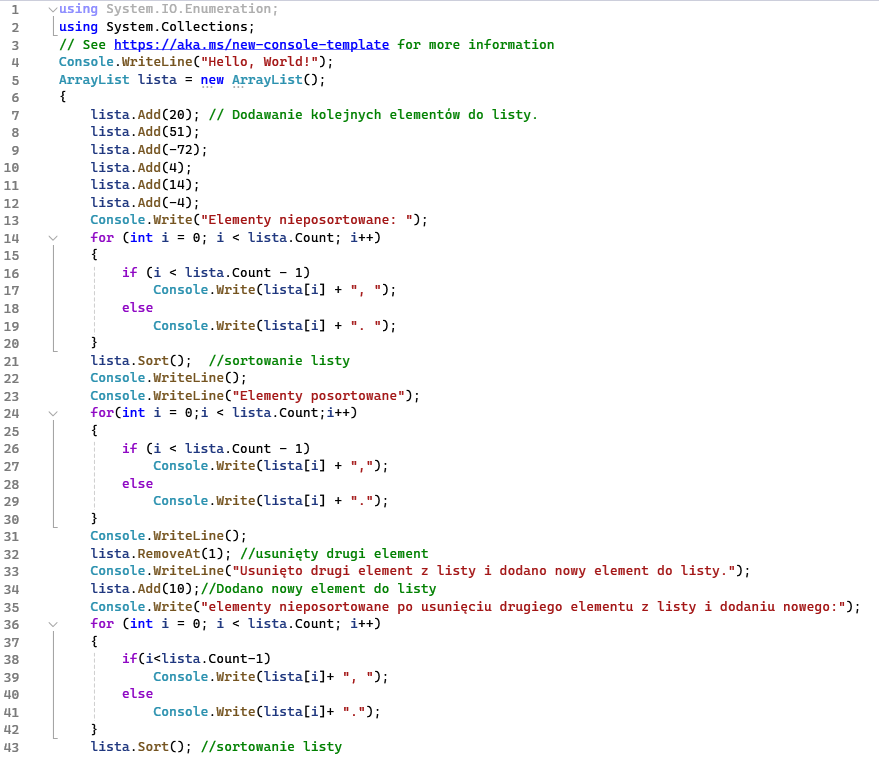
*}*

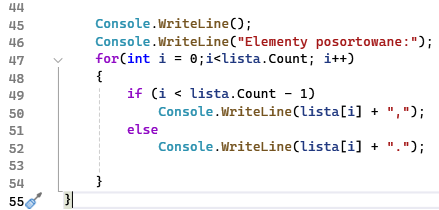
**Kolekcje**

Kolekcja jest to obiekt przechowujący inne obiekty np.: tablice. Najbardziej znaną i najczęściej stosowaną kolekcją w języku C# jest lista *ArrayList()*. Tak jak tablica może przechowywać wiele elementów, jest jednak łatwiejsza i wygodniejsza w obsłudze. W przeciwieństwie do tablic lista *ArrayList* nie wymaga deklaracji ani określenia, z ilu elementów ma się składać. Kolejne elmenty dodajemy do listy lub z niej usuwamy, korzystając z metod Add() i RemoveAt(). Kolekcja ArrayList() ma wiele innych metod, np. Sort(), które upraszczają jej obsługę i zwiększają funkcjonalność

**Zadanie 2.**

Korzystając z właściwości kolekcji ArrayList(), napisz program, który do kolekcji tego typu dodaje sześć liczb, a następnie je sortuje. Po wykonaniu tej operacji należy dokonać usunięcia drugiego elementu z listy, dodać nowy element do listy i dokonać ponownego sortowania.

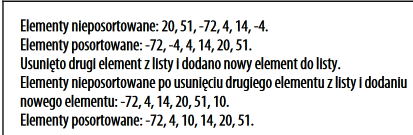




Do programu dołączono przestrzeń nazw:

*using System.Collections*.

Rezultat działania programu:



**Zadanie3.**

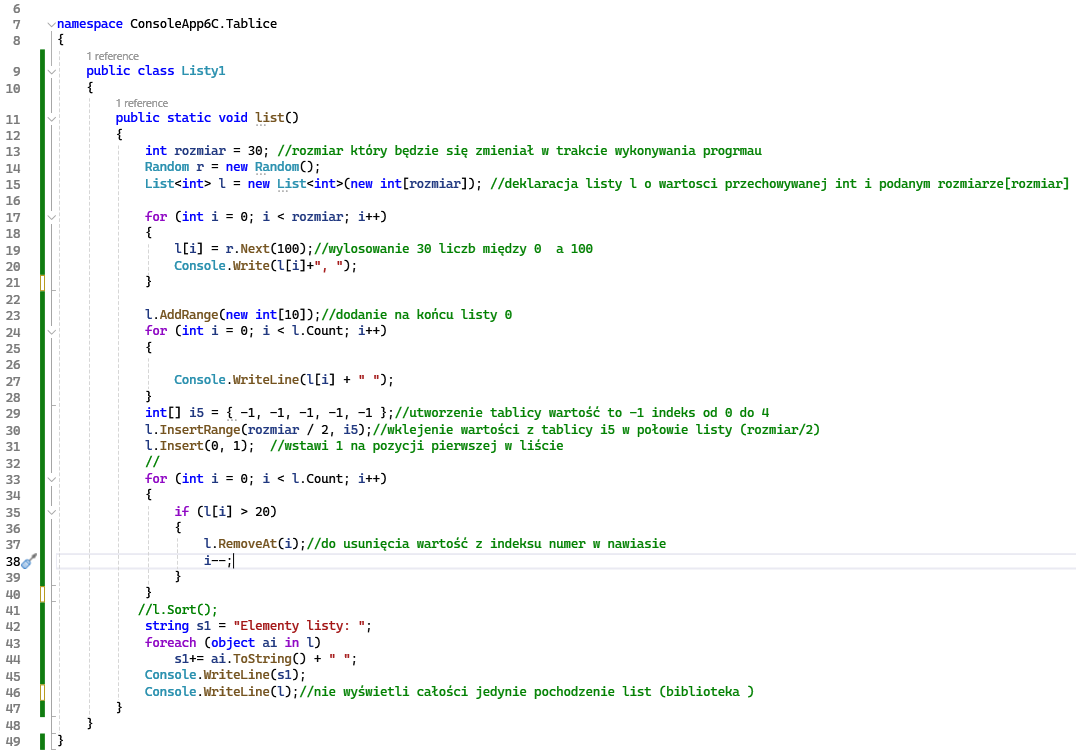
Korzystając z właściwości kolekcji ArrayList() napisz program, który do kolekcji tego typu dodaje sześć imion a następnie je sortuje.

**Kolekcja list**

Najczęściej wykorzystywana kolekcja to System.Collections.Generics.List, czyli implementacja **listy.** Należy ona do grupy typów ogólnych (**ang**. *generic types*), nazywanych też typami parametrycznymi. Ma tę zaletę w porównaniu ze zwykłą tablicą (System.Array), że liczba elementów może być zmieniana już po utworzeniu listy – można dodawać elementy na koniec, wstawiać do środka i usuwać dowolnie wybrany element. Jednocześnie możliwy jest dostęp do dowolnego elementu, tak samo jak w przypadku tablicy.

**Zadanie2**

Utworzymy kolekcję składających się ze stu losowo wybranych liczb całkowitych. Następnie na jej koniec dodamy dziesięć liczb o wartości 0, a do środka wstawimy kolejnych pięć o wartości – 1. Wreszcie na pierwszej pozycji ustawimy 1. Na koniec usuniemy wszystkie te liczby, które są większe od 20 i posortujemy resztę.



Zadanie 2

Przykład demonstracyjny użycia własności i metod klasy List.

Uwagi:

List<T>.RemoveAt(Int32)

RemoveAt(Int32) – usuwa element w określonym indeksie elementu List<T>

**Parametry** : index Int32

**Wyjątki**: [ArgumentOutOfRangeException](https://learn.microsoft.com/pl-pl/dotnet/api/system.argumentoutofrangeexception?view=net-8.0) (wartość indeksu jest mniejsza niż 0 ) lub (indekx jest równy lub większy niż Count –(czyli ostatni indeks Listy))

List<T>.AddRange(IEnumerable<T>) -

Add.Range(System.Collections.Generic.IEnumerable<T>collection); - Kolekcja której element należy dodać na końcu elementu List<T>. Sama kolekcja nie może być null ale może zawierać elementy które są null jeśli typ T jest typem odwołania.

**Parametry:** Collection IEnumerable<T>

**Wyjątki**: collection to null kolekcja nuli.

List<T>.Insert(int startIndex, string value);

Zwraca ciąg w którym wstawiono określony ciąg przy określonej pozycji indeksu.

public string Insert (int startIndex, string value);

**Parametry:** (Stary indeks, ciąg do wstawienia) (indeks, Int32), (value, string)

Zwraca (Returns) string

**Wyjątki** : [ArgumentNullException](https://learn.microsoft.com/pl-pl/dotnet/api/system.argumentnullexception?view=net-8.0) (value is null)

[ArgumentOutOfRangeException](https://learn.microsoft.com/pl-pl/dotnet/api/system.argumentoutofrangeexception?view=net-8.0) (indeks jest ujemny albo większy niż długość tej instancji

List<T>.InsertRange(Int32, IEnumerable<T>)

Wstawia elementy kolekcji do List<T> określonego indeksu

public void InsertRange (int index, System.Collections.Generic.IEnumerable<T> collection);

Parametry: index, Int32

Wyjątki:

[ArgumentNullException](https://learn.microsoft.com/pl-pl/dotnet/api/system.argumentnullexception?view=net-8.0) collection to null

[ArgumentOutOfRangeException](https://learn.microsoft.com/pl-pl/dotnet/api/system.argumentoutofrangeexception?view=net-8.0) index wartość jest mniejsza od 0 lub większa od indeksu (ostatniego) całej kolekcji

List<T>RemoveRange(Int32,Int32) - usuwa zakres elementów, począwszy od określonego indeksu i zawierający określoną liczbę elementów z tej niezmiennej listy.

Zwraca – nowa lista z usuniętymi elementami

List<T>Capacity – Pobiera lub ustawia całkowitą liczbę elementów, które struktura może przechowywać

public int Capacity { get; set; }

Wartość właściwości: Liczba elementów, które mogą zawierać przed zmianą rozmiaru List<T>, jest wymagana

GetRange(Int32,Int32) – Tworzy kopię zakresu elementów w źródle List<T>

public System.Collections.Generic.List<T> GetRange (int index, int count);

Zwraca List<T> kopia zakresu elementów



Słowniki

W przestrzeni nazw System.Collections.Generics są dostępne struktury:

* słowniki :Dictionary<,>;
* SortedDictionary<,>;
* SortedList<,>.

Przechowują listy par klucz-wartosć (elementy typu KeyValuePair<,>).

Różnią się optymalizacją zużycia pamięci czasu potrzebnego na wstawianie posortowanych i nieposortowanych danych czy wyszukiwania.

Zajmijmy się SortedList, ale wszystkie trzy kolekcje mają taki sam zestaw metod wymuszanych przez interfejs IDictionary<,>, czyli korzysta się z nich w podobny sposób.

Każda para w słowniku przechowuje klucz i wartość (właściwości Key i Value typu KeyValuePair<,>). Przy wstawianiu elementu do kolekcji SortedList jest ona automatycznie umieszczana na pozycji odpowiadającej posortowanym kluczom. Dzięki temu wyszukiwanie elementu przy odczytywaniu może być szybsze. Typowym zastosowaniem tej kolekcji są różnego typu listy z kluczem np.: słowniki tłumaczeń słów, listy haseł encyklopedycznych, cenniki itp.

Zadanie 2

Wykorzystując SortedList z oboma parametrami typu string. Przechowuję w niej imiona lub nazwiska bohaterów oraz miejsca z nimi zwiazane. Imiona lub nazwiska będą pełniły funkcję kluczy , a miejscowość wartość. Kolekcja będzie automatycznie sortowana zgodnie z kolejnością alfabetyczną kluczy.



**Kolejka** (ang. queue) i stos (ang. stack). – zdefiniowane w przestrzeni: System.Collections.Generic. Brak swobodnego dostępu do elementów kolekcji, możliwość dodawania i zdejmowania pojedynczych obiektów.

Różnica pomiędzy nimi to: to który element możemy zdjąć. Kolejka jest strukturą typu FIFO (ang. first in first out) oznacza to, że element położony jako pierwszy może być pierwszy zdjęty.

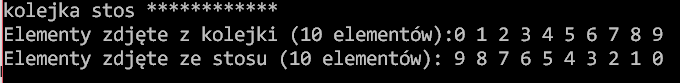
Kolejkę można sobie wyobrażać jako rurę, do której wkłada się piłki z jednego jej końca i wyjmuje z drugiego.

Stos to struktura typu FILO (ang. first in,last out) – element dodany jako pierwszy dostępny będzie jako ostatni. Nazwa tej struktury dobrze oddaje też to, jak można sobie ją wyobrażać – jako stos np. kartek. To co położymy na stosie jako pierwsze, będzie przykrywane przez następne elementy i tym samym dostępne dopiero, gdy zdejmiemy wszystkie elementy położone później.

**Przykład (użycie stosu i kolejki)**

gdzie w przypadku kolejki otrzymamy liczby w tej samej kolejności w jakiej włożyliśmy, a w przypadku stosu ich kolejność będzie odwrócona.





Def.

*Queue<T>.Enqueue(T)* Metoda – Dodaje obiekt na końcu obiektu Queue<T>

public void Enqueue (T item);

Parametry item T Obiekt ,który ma zostać dodany do obiektu Queue<T> klasy ogólnej w tym metodą *Enqueue*