**Колекції**

**Список List<T>**

Класс ***List<T>*** з простору імен ***System.Collections.Generic*** представляє найпростіший перелік однотипних об'єктів. Клас ***List*** типизується типом, об'єкти якого зберігаються у списку.

Ми можемо створити порожній список:

List<string> people = new List<string>();

У разі об'єкт ***List*** типизується типом ***string*** . А це означає, що зберігати у цьому списку ми можемо лише рядки.

Можна відразу під час створення списку ініціалізувати його початковими значеннями. У цьому випадку елементи списку розміщуються після виклику конструктора у фігурних дужках

List<string> people = new List<string>() { "Tom", "Bob", "Sam" };

У даному випадку до списку поміщаються три рядки

Також можна під час створення списку ініціалізувати його елементами з іншої колекції, наприклад іншого списку:

var people = new List<string>() { "Tom", "Bob", "Sam" };

var employees = new List<string>(people);

Можна поєднати обидва способи:

var people = new List<string>() { "Tom", "Bob", "Sam" };

var employees = new List<string>(people){"Mike"};

У цьому випадку в списку ***employees*** буде чотири елементи ***( { "Tom", "Bob", "Sam", "Mike" })*** - три додаються зі списку ***people*** і один елемент задається при ініціалізації.

Так само можна працювати зі списками інших типів, наприклад:

List<Person> people = new List<Person>()

{

new Person("Tom"),

new Person("Bob"),

new Person("Sam")

};

class Person

{

public string Name { get;}

public Person(string name) => Name = name;

}

**Встановлення початкової ємності списку**

Ще один конструктор класу ***List*** приймає як параметр початкову ємність списку:

List<string> people = new List<string>(16);

Вказівка ​​початкової ємності списку дозволяє в майбутньому збільшити продуктивність і зменшити витрати на виділення пам'яті додавання елементів. Оскільки динамічне додавання до списку може призводити на низькому рівні додаткового виділення пам'яті, що знижує продуктивність. Якщо ж ми знаємо, що список не перевищуватиме певний розмір, то ми можемо передати цей розмір як місткість списку та уникнути додаткових виділень пам'яті.

Також початкову ємність можна встановити за допомогою властивості ***Capacity***, що є у класу ***List***.

**Звернення до елементів списку**

Як і масиви, списки підтримують індекси, за допомогою яких можна звернутися до певних елементів:

var people = new List<string>() { "Tom", "Bob", "Sam" };

string firstPerson = people[0]; // получаем первый элемент

Console.WriteLine(firstPerson); // Tom

people[0] = "Mike"; // изменяем первый элемент

Console.WriteLine(people[0]); // Mike

**Довжина списку**

За допомогою властивості ***Count*** можна отримати довжину списку:

var people = new List<string>() { "Tom", "Bob", "Sam" };

Console.WriteLine(people.***Count***); // 3

**Перебір списку**

C# дозволяє здійснити перебір списку за допомогою стандартного ***foreach*** циклу

var people = new List<string>() { "Tom", "Bob", "Sam" };

***foreach*** (var person in people)

{

Console.WriteLine(person);

}

// Вывод программы:

// Tom

// Bob

// Sam

Також можна використовувати інші типи циклів та в комбінації з індексами перебирати списки:

var people = new List<string>() { "Tom", "Bob", "Sam" };

for (int i = 0; i < people.Count; i++)

{

Console.WriteLine(people[i]);

}

**Методи списку**

Серед його методів можна виділити такі:

* ***void Add(T item)*** : додавання нового елемента до списку
* ***void AddRange(IEnumerable<T> collection)*** : додавання до списку колекції або масиву
* ***int BinarySearch(T item)*** : бінарний пошук елемента у списку. Якщо елемент знайдено, метод повертає індекс цього елемента в колекції. При цьому список має бути відсортований.
* ***void CopyTo(T[] array)*** : копіює список у масив ***array***
* ***void CopyTo(int index, T[] array, int arrayIndex, int count)*** : копіює зі списку починаючи з індексу ***index*** елементи, кількість яких дорівнює ***count***, і вставляє їх у масив ***array*** починаючи з індексу ***arrayIndex***
* ***bool Contains(T item)*** : повертає true, якщо елемент item є у списку
* ***void Clear()*** : видаляє всі елементи зі списку
* ***bool Exists(Predicate<T> match)*** : повертає ***true***, якщо у списку є елемент, який відповідає делегату ***match***
* ***T? Find(Predicate<T> match)*** : повертає перший елемент, який відповідає делегату ***match***. Якщо елемент не знайдено, повертається ***null***
* ***T? FindLast(Predicate<T> match)*** : повертає останній елемент, який відповідає делегату ***match***. Якщо елемент не знайдено, повертається ***null***
* ***List<T> FindAll(Predicate<T> match)*** : повертає список елементів, які відповідають делегату ***match***
* ***int IndexOf(T item)*** : повертає індекс першого входження елемента у списку
* ***int LastIndexOf(T item)*** : повертає індекс останнього входження елемента у списку
* ***List<T> GetRange(int index, int count)*** : повертає список елементів, кількість яких дорівнює ***count***, починаючи з індексу ***index***.
* ***void Insert(int index, T item)*** : вставляє елемент ***item*** до списку індексу ***index***. Якщо такого індексу у списку немає, то генерується виняток
* ***void InsertRange(int index, collection)*** : вставляє колекцію елементів ***collection*** у поточний список, починаючи з індексу ***index***. Якщо такого індексу у списку немає, то генерується виняток
* ***bool Remove(T item)*** : видаляє елемент ***item*** зі списку і якщо видалення пройшло успішно, то повертає ***true***. Якщо у списку кілька однакових елементів, то видаляється лише перший з них
* ***void RemoveAt(int index)*** : видалення елемента за вказаним індексом ***index***. Якщо такого індексу у списку немає, то генерується виняток
* ***RemoveRange(int index, int count)*** : параметр ***index*** задає індекс, з якого треба видалити елементи, а параметр count задає кількість елементів, що видаляються.
* ***int RemoveAll((Predicate<T> match))*** : видаляє всі елементи, які відповідають делегату ***match***. Повертає кількість віддалених елементів
* ***void Reverse()*** : змінює порядок елементів
* ***void Reverse(int index, int count)*** : змінює порядок на зворотний для елементів, кількість яких дорівнює count, починаючи з індексу index
* ***void Sort()*** : сортування списку
* ***Sort(IComparer<T>? comparer)*** : сортування списку за допомогою об'єкта comparer, який передається як параметр

**Додавання до списку**

List<string> people = new List<string> () { "Tom" };

people.***Add***("Bob"); // добавление элемента

// people = { "Tom", "Bob" };

people.***AddRange***(new[] { "Sam", "Alice" }); // добавляем массив

// people = { "Tom", "Bob", "Sam", "Alice" };

// также можно было бы добавить другой список

// people.AddRange(new List<string>(){ "Sam", "Alice" });

people.***Insert***(0, "Eugene"); // вставляем на первое место

// people = { "Eugene", "Tom", "Bob", "Sam", "Alice" };

people.***InsertRange***(1, new string[] {"Mike", "Kate"}); // вставляем массив с индекса 1

// people = { "Eugene", "Mike", "Kate", "Tom", "Bob", "Sam", "Alice" };

// также можно было бы добавить другой список

// people.InsertRange(1, new List<string>(){ "Mike", "Kate" });

**Видалення зі списку**

var people = new List<string> () { "Eugene", "Mike", "Kate", "Tom", "Bob", "Sam", "Tom", "Alice" };

people.***RemoveAt(1)***; // удаляем второй элемент

// people = { "Eugene", "Kate", "Tom", "Bob", "Sam", "Tom", "Alice" };

people.***Remove("Tom")***; // удаляем элемент "Tom"

// people = { "Eugene", "Kate", "Bob", "Sam", "Tom", "Alice" };

// удаляем из списка все элементы, длина строки которых равна 3

people.***RemoveAll***(person => person.Length == 3);

// people = { "Eugene", "Kate", "Alice" };

// удаляем из списка 2 элемента начиная с индекса 1

people.***RemoveRange(1, 2)***;

// people = { "Eugene"};

// полностью очищаем список

people.***Clear***();

// people = { };

**Пошук та перевірка елемента**

var people = new List<string> () { "Eugene", "Mike", "Kate", "Tom", "Bob", "Sam" };

var containsBob = people.***Contains***("Bob"); // true

var containsBill = people.***Contains***("Bill"); // false

// проверяем, есть ли в списке строки с длиной 3 символа

var existsLength3 = people.***Exists***(p => p.Length == 3); // true

// проверяем, есть ли в списке строки с длиной 7 символов

var existsLength7 = people.Exists(p => p.Length == 7); // false

// получаем первый элемент с длиной в 3 символа

var firstWithLength3 = people.***Find***(p => p.Length == 3); // Tom

// получаем последний элемент с длиной в 3 символа

var lastWithLength3 = people.***FindLast***(p => p.Length == 3); // Sam

// получаем все элементы с длиной в 3 символа в виде списка

List<string> peopleWithLength3 = people.***FindAll***(p => p.Length == 3);

// peopleWithLength3 { "Tom", "Bob", "Sam"}

**Отримання діапазону та копіювання в масив**

List<string> people = new List<string>() {"Eugene", "Tom", "Mike", "Sam", "Bob" };

// получаем диапазон со второго по четвертый элемент

var range = people.***GetRange***(1, 3);

// range = { "Tom", "Mike", "Sam"};

// копируем в массив первые три элемента

string[] partOfPeople = new string[3];

people.***CopyTo***(0, partOfPeople, 0, 3);

// partOfPeople = { "Eugene", "Tom", "Mike"};

**Розташування елементів у зворотному порядку**

var people = new List<string> () { "Eugene", "Tom", "Mike", "Sam", "Bob" };

// переворачиваем весь список

people.***Reverse***();

// people = { "Bob","Sam", "Mike", "Tom", "Eugene"};

var people2 = new List<string>() { "Eugene", "Tom", "Mike", "Sam", "Bob" };

// переворачиваем часть только 3 элемента с индекса 1

people2.Reverse(1, 3);

// people2 = { "Eugene","Sam", "Mike", "Tom", "Bob" };