Лабораторна робота №8

Тема: Використання колекцій для роботи з масивами у C#.

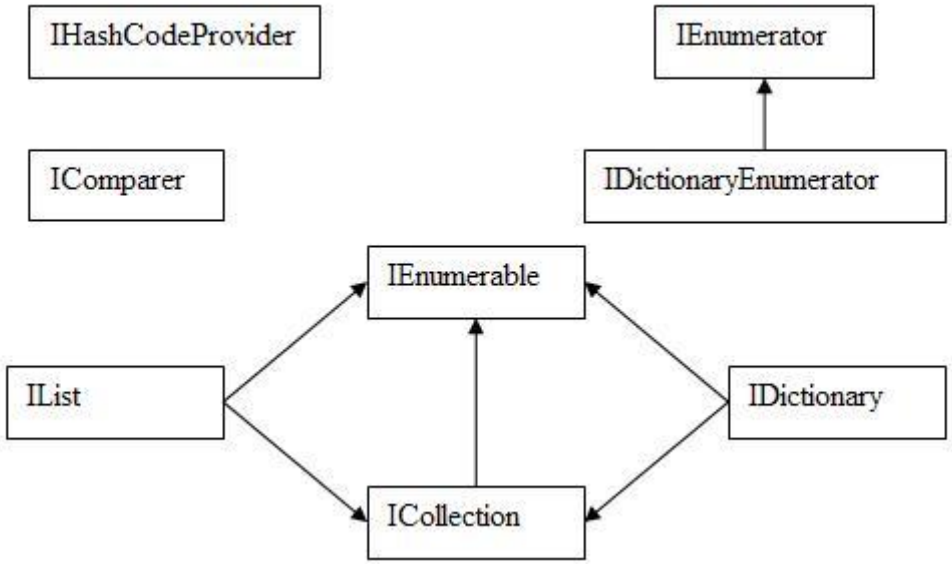
Мета роботи: Набуття навичок використання узагальнених колекцій з масивами.

Теоретичні відомості. Під ***колекцією*** (також, ***контейнер***) мають на увазі групу об'єктів. Така група об'єктів реалізується як об'єкт певного *класу*, що містить у собі колекцію об'єктів іншого *класу*. Технічно це можна реалізувати за допомогою *класу*, у якого є *поле*, що є *посиланням* на *масив об'єктів*, що формують *колекцію*. Тобто, у практичному плані ми могли б самі без особливих проблем описати подібний *клас*. Тобто сама собою можливість «заховати» в одному об'єкті групу інших об'єктів не є унікальною. Зручність та ефективність використання *колекцій* багато в чому пов'язані з наявністю *методів*, що полегшують роботу з групами об'єктів, що містяться у *колекції*. Інша важлива особливість *колекцій*, що виділяє їх на тлі звичайних *масивів*, – можливість додавати елементи до *колекції* та видаляти елементи з *колекції*.

Масиви є гарним інструментом групування даних. Проте, *масиви* зберігають фіксовану кількість об'єктів, інколи ж заздалегідь невідомо, скільки потрібно об'єктів. І в цьому випадку набагато зручніше використовувати *колекції*. Після створення *масиву* його розмір не можна змінити. Але ми можемо зробити так: створюємо новий *масив* потрібного розміру, заповнюємо його правильним чином і посилання на цей *масив* записуємо в змінну *масиву*. Створюється ілюзія, що *масив* змінив розмір. Хоча це, звісно, не так. В роботі з *колекціями* процес «зміни» розміру *масиву*, що є основою *колекції*, автоматизований, що досить зручно.

Ще один плюс *колекцій* полягає в тому, що деякі з них реалізує стандартні структури даних, наприклад, *стек, черга, словник*, які можуть стати в нагоді для вирішення різних спеціальних завдань. Більшість *класів колекцій* міститься в просторах імен *System.Collections* (прості неузагальнені *класи колекцій*), *System.Collections.Generic* (*узагальнені* або *типізовані класи колекцій*) та *System.Collections.Specialized* (*спеціальні класи колекцій*). Також для забезпечення паралельного виконання завдань та багатопотокового доступу застосовуються *класи колекцій* із *простору імен System.Collections.Concurrent.*

Крім того, у *System.Collections* визначено набір стандартних інтерфейсів. Деякі з цих інтерфейсів об'єднані в ієрархії, в той час, як деякі існують незалежно від інших.



Наступні інтерфейси реалізовані у більшості колекцій:

Інтерфейс	**Призначення**
ICollection	Визначає загальні характеристики класу-набору елементів
IComparer	Дозволяє порівнювати два об'єкти
IDictionary	Дозволяє представляти вміст об'єкта у вигляді пар «ім'я-значення»
IDictionaryEnumerator	Використовується для нумерації вмісту об'єкта, що підтримує *IDictionary*
IEnumerable	Повертає інтерфейс *IEnumerator*
IEnumerator	Зазвичай використовується для підтримки конструкції *foreach* щодо об'єктів
IHashCodeProvider	Повертає *хеш-код* для реалізації *типу* із застосуванням користувачем *алгоритму хешування*
IList	Забезпечує методи для додавання, видалення та індексування у списку об'єктів

Списки – List<T>

Клас List<T> є найпростішим із *класів колекцій*. Його можна використовувати практично так само, як *масив*, посилаючись на існуючий у *колекції List<T>* елемент з використанням звичайної для *масивів* системи запису з квадратними дужками та індексом елемента. Можна додати елемент до кінця *колекції List<T>*, скориставшись наявним у її *класі методом Add*, якому надається елемент, що додається. Розмір *колекції* збільшується *List<T>* автоматично.

```
List<int> list = new List<int>();
list.Add(3);
list.Add(1);
list[0]=list[1]+3;
```

Вказувати розмір *колекції List<T>* під час її створення не обов'язково. *Колекція* може змінювати свої розміри при додаванні (або видаленні) елементів. Але треба зауважити, що на фізичне додавання елементів йде час процесора, і при необхідності потрібно вказати початковий розмір. Але якщо він буде перевищений, то через необхідність *колекція List<T>* просто розшириться. Якщо перший рядок прикладу змінити на

```
List<int> list = new List<int>(2);
```

то результат буде той самий, але останній варіант відпрацює швидше. І тут команда *Add* – це ініціалізація чергового елемента наприкінці *списку*. Для видалення з *колекції List<T>* вказаного елемента можна скористатися методом *Remove*. Елементи *колекції List<T>* автоматично перебудуються, закриваючи пусте місце, що утворилося. За допомогою методу *RemoveAt* також можна видалити елемент, вказавши його позицію в *колекції List<T>*. Можна вставити елемент у середину *колекції List<T>*, скориставшись при цьому методом *Insert*. При цьому розмір *колекції <T>* також зміниться автоматично. Розмір *колекції*, тобто. кількість ініціалізованих елементів можна отримати через властивість (тільки на читання) *Count*, а ємність колекції – через властивість (читання та запис) *Capacity*.

Двозв'язні списки - LinkedList<T>

Клас *колекцій LinkedList* реалізує двозв'язний список. У кожному елементі списку міститься значення елемента з посиланням на наступний елемент *списку* (*Властивість Next*) та його попередній елемент (*Властивість Previous*).

У *класі LinkedList<T>* записи, властиві масивам, не підтримуються. Вставка елементів здійснюється відмінним від *List<T>* способом. Можна скористатися *методом AddFirst* для вставки елемента на початок *списку* з переміщенням попереднього першого елемента далі за списком та установки як значення його *властивості Previous* посилання на новий елемент. Аналогічно для вставки елемента в кінець списку можна скористатися *методом AddLast*. Для вставки елемента перед вказаним елементом *списку* або після нього можна скористатися *методами AddBefore* та *AddAfter*.

Перший елемент колекції LinkedList<T> можна знайти, запросивши значення властивості First, а властивість Last надасть посилання на останній елемент списку. Для послідовного обходу елементів *зв'язного списку* можна приступити до цієї операції з одного кінця і крок за кроком застосовувати посилання з *властивості Next* або *Previous*, поки не буде знайдений елемент, у якого ця властивість має значення *null*. Звичайно ж, краще скористатися інструкцією *foreach*, яка виконає послідовний обхід елементів вперед за *списком LinkedList<T>*-об'єкта, автоматично зупинившись в кінці.

Видалення елемента з *колекції LinkedList<T>* здійснюється за допомогою *методів Remove, RemoveFirst* та *RemoveLast*.

Перевага *зв'язаного списку* в тому, що операція вставки елемента всередину виконується дуже швидко. Це відбувається за рахунок того, що тільки *посилання Next* (наступний) попереднього елемента і *Previous* (попередній) наступного елемента повинні бути змінені так, щоб вказувати на елемент, що вставляється. У *класі List<T>* при вставці нового елемента всі наступні мають бути зсунуті.

Звісно, *зв'язані списки* мають і свої недоліки. Так, наприклад, всі елементи таких *списків* доступні лише один за одним. Тому для знаходження елемента, що знаходиться в середині або наприкінці *списку*, потрібно досить багато часу. *Зв'язаний список* не може просто зберігати елементи у собі. Разом з кожним із них йому необхідно мати інформацію про наступний та попередній елементи. Тому *LinkedList<T>* містить елементи типу *LinkedListNode<T>*. З допомогою *класу LinkedListNode<T>* з'являється можливість звернутися до попереднього та наступного елементів *списку*. *Клас LinkedListNode<T>* визначає *властивості List, Next, Previous* та *Value*. *Властивість List* повертає об'єкт *LinkedList<T>*, асоційований із *вузлом*. *Властивості Next* і *Previous* призначені для ітерацій за *списком* та для доступу до наступного та попереднього елементів. *Властивість Value типу T* повертає елемент, що відповідає *вузлу*. Якщо у *простому списку List<T>* кожен елемент представляє об'єкт *типу T*, то у *LinkedList<T>* кожен *вузол* крім *T* містить також об'єкт *класу LinkedListNode<T>*. Цей клас має такі властивості:

- *Value* – значення *вузла*, представлене *типом T*.
- *Next* – посилання на наступний елемент *типу LinkedListNode<T>* у *списку*. Якщо наступний елемент відсутній, має значення *null*.
- *Previous* – посилання на попередній елемент *типу LinkedListNode<T>* у *списку*. Якщо попередній елемент відсутній, має значення *null*.

Використовуючи *методи класу LinkedList<T>*, можна звертатися до різних елементів як наприкінці, так і на початку *списку*:

- *AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode)*: вставляє *вузол newNode* у *список* після *вузла node*.
- *AddAfter(LinkedListNode<T> node, T value)*: вставляє до *списку* новий *вузол* зі значенням *value* після вузла *node*.
- *AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode)*: вставляє в *список вузол newNode* перед *вузлом node*.
- *AddBefore(LinkedListNode<T> node, T value)*: вставляє до *списку* новий *вузол* зі значенням *value* перед *вузлом node*.
- *AddFirst(LinkedListNode<T> node)*: вставляє новий *вузол* на початок *списку*.
- *AddFirst(T value)*: вставляє новий *вузол* зі значенням *value* на початок *списку*.
- *AddLast(LinkedListNode<T> node)*: вставляє новий *вузол* у кінець *списку*.

- *AddLast(T value)*: вставляє новий *вузол* зі значенням *value* до кінця *списку*.
- *RemoveFirst()*: видаляє перший *вузол* зі *списку*. Після цього новим першим *вузлом* стає *вузол*, що йде за видаленим.
- *RemoveLast()*: видаляє останній *вузол* зі *списку*.

Словники – Dictionary<TKey, TValue>

Масив та об'єкти *типу List<T>* надають спосіб відображення на елемент *цілого індексу*. *Цілочисельний індекс* вказується за допомогою квадратних дужок (наприклад, [1]), і витягується елемент за індексом 1, будучи фактично другим. Але іноді може знадобитися реалізація відображення, у якому використовується інший, *нецілочисельний тип*, наприклад *string, double* чи *DateTime*. В інших мовах програмування така організація зберігання даних часто називається *асоціативним масивом*. Ця функціональна можливість реалізується в *класі Dictionary<TKey, TValue>* шляхом внутрішнього обслуговування двох *масивів*, один з яких призначений для *ключів*, від яких виконується відображення на одне з значень, що відображаються. Коли в *колекцію Dictionary<TKey, TValue>* вставляється пара «*ключ-значення*», *клас* автоматично відслідковує належність *ключа* до *значення*, дозволяючи швидко та легко отримувати *значення*, пов'язане із зазначеним *ключем*. У конструкції *класу Dictionary<TKey, TValue>* є низка важливих особливостей.

У *колекції Dictionary<TKey, TValue>* не можуть бути продубльовані *ключі*. Якщо додавання вже наявного в *масиві ключа* викликається *метод Add*, видається *виняток*. Але для додавання пари «*ключ-значення*» можна скористатися системою запису з використанням квадратних дужок, не побоюючись при цьому видачі *виключення*, навіть якщо *ключ* вже був доданий: будь-яке *значення* з таким самим *ключем* буде переписано новим *значенням*. Протестувати наявність у *колекції Dictionary<TKey, TValue>* конкретного *ключа* можна за допомогою *методу ContainsKey*.

За внутрішнім пристроєм *колекція Dictionary<TKey, TValue>* є розрядженою структурою даних, що працює найбільш ефективно, коли в її розпорядженні є досить великий обсяг пам'яті. У міру вставки елементів розмір *колекції Dictionary<TKey, TValue>* у пам'яті може швидко збільшуватися.

Коли для послідовного обходу елементів *колекції Dictionary<TKey, TValue>* використовується інструкція *foreach*, елемент *KeyValuePair<TKey*, TValue> повертається. Це структура, що містить копію елементів *ключа* та *значення*, що знаходяться в *колекції Dictionary<TKey, TValue>*, і доступ до кожного елемента можна отримати через *властивості Key* та *Value*. Ці елементи доступні лише для читання, і їх не можна використовувати для зміни даних у *колекції Dictionary<TKey, TValue>*.

Відзначимо, що є подібна *неузагальнена колекція* – *Hashtable*, що має такий самий функціонал. Однак ця *колекція* програє у швидкості під час роботи з однотипними об'єктами і використовується, як і всі *неузагальнені колекції*, для угруповання різних об'єктів.

Щодо продуктивності розглянутих тут *колекцій* зауважимо, що додавання нового об'єкта (*Add*) швидше робить *List<T>*, повільніше – *Dictionary<TKey, TValue>*. На пошук елемента йде приблизно однаковий час, проте пошук по *ключу* істотно швидше *Dictionary<TKey, TValue>*. Видалення об'єкта повільніше робиться у *класі List<T>*.

Завдання для самостійної роботи

Вхідні дані записуємо зі сформованого відповідно варіанту завдання текстового файлу в масив, наприклад:

```
string text = File.ReadAllText($@"{Environment.CurrentDirectory}\some.txt");
// read everything in text
char[] symbols = text.ToCharArray(); // convert to char array
```

Масив передається методу *Main* як аргумент.

***Варіанти завдань*:**

1. Написати програму, яка обчислює кількість голосних та приголосних літер у файлі. Вміст текстового файлу заноситься до масиву символів. Кількість голосних та приголосних літер визначається проходом по масиву. Передбачити метод, вхідний параметр якого є масив символів. Метод обчислює кількість голосних та приголосних букв. Завдання також виконати за допомогою *колекції List<T>*.
2. Написати програму, яка обчислює кількість прописних літер у файлі. Вміст текстового файлу заноситься до масиву символів. Кількість прописних літер визначається проходом по масиву. Передбачити метод, вхідний параметр якого є масив символів. Метод обчислює кількість прописних букв. Завдання також виконати за допомогою *колекції List<T>*.
3. Написати програму, яка обчислює кількість знаків пунктуації у файлі. Вміст текстового файлу заноситься до масиву символів. Кількість знаків пунктуації визначається проходом по масиву. Передбачити метод, вхідний параметр якого є масив символів. Метод обчислює кількість знаків пунктуації. Завдання також виконати за допомогою *колекції List<T>*.
4. Написати програму, яка обчислює кількість слів у файлі. Вміст текстового файлу заноситься до масиву символів. Кількість слів визначається проходом по масиву. Передбачити метод, вхідний параметр якого є масив символів. Метод обчислює кількість слів. Завдання також виконати за допомогою *колекції List<T>*.
5. Написати програму, що реалізує множення двох матриць, заданих як двовимірні масиви. У програмі передбачити два методи: метод множення матриць (на вхід дві матриці, значення, що

повертається – матриця), метод виводу матриці на екран. Завдання також виконати за допомогою *колекцій LinkedList<LinkedList<T>>*.

6. Написати програму, що реалізує векторний добуток, заданих як одновимірні масиви. У програмі передбачити два методи: метод векторного добутку (на вхід два вектори, значення, що повертається – вектор), метод виводу векторного добутку на екран. Завдання також виконати за допомогою *колекцій LinkedList<LinkedList<T>>*.
7. Написати програму, що реалізує зовнішній векторний добуток, заданих як одновимірні масиви. У програмі передбачити два методи: метод зовнішнього векторного добутку (на вхід два вектори, значення, що повертається – матриця), метод виводу зовнішнього векторного добутку на екран. Завдання також виконати за допомогою *колекцій LinkedList<LinkedList<T>>*.
8. Написати програму, що з довільного двовимірного масиву, утвореного нулями та одиницями, утворює два одновимірних масиви: один з нулів, другий – з одиниць, що містяться у вхідному масиві. У програмі передбачити два методи: метод розділення (на вхід двовимірний масив, значення, що повертається – два одновимірних масиви), метод виводу отриманих масивів на екран. Завдання також виконати за допомогою *колекцій LinkedList<LinkedList<T>>*.
9. Написати програму, яка обчислює середню температуру протягом року. Створити двовимірний рандомний масив *temperature*[12,30], в якому зберігається температура для кожного дня місяця (передбачається, що в кожному місяці 30 днів). Згенерувати значення температури випадковим чином. Для кожного місяця надрукувати середню температуру. Для цього написати метод, який за масивом *temperature*[12,30] для кожного місяця обчислює середню температуру в ньому, і як результат повертає масив середніх температур. Отриманий масив середніх температур відсортувати за зростанням. Завдання також виконати з допомогою *класу Dictionary<TKey, TValue>*. Як ключі вибрати рядки – назви місяців, а як значення – масив значень температур по днях.

Контрольні запитання

1. У чому відміна колекцій від масивів?
2. Назвіть основні властивості та методи класу System.Array.
3. Наведіть приклади опису масивів та колекцій.
4. Як передавати та повертати масиви та колекції з методів.
5. Поясніть принцип роботи циклу foreach.
6. Розкажіть про плюси та мінуси використання двозв'язних списків.
7. Наведіть практичні приклади ефективного використання розглянутих колекцій.