5 КОЛЕКЦІЇ. ВИКОРИСТАННЯ LINQ.СЕРІАЛІЗАЦІЯ

5.1 Мета роботи

Навчитися створювати та працювати з колекціями, аналізувати колекції використовуючи LINQ та використовувати серіалізацію для зберігання даних.

5.2 Організація самостійної роботи студентів

Під час підготовки до виконання лабораторної роботи необхідно вивчити існуючі колекції, методи для роботи з колекціями. Навчитися обробляти інформацію використовуючи мову запитів LINQ та зберігати дані за допомогою серіалізації об'єктів.

Велика частина класів колекцій міститься в просторах імен *System.Collections* (прості неузагальнені класи колекцій), *System.Collections.Generic* (узагальнені або типізовані класи колекцій) і *System.Collections.Specialized* (спеціальні класи колекцій).

ArrayList

Клас *ArrayList* є колекцією об'єктів, яка зберігає разом різнотипні об'єкти (рядки, числа і т.д.)

Основні методи класу:

– *int Add(object value)*: додає до списку об'єкт *value*.

– *void AddRange(ICollection col)*: додає до списку об'єкти колекції *col*, яка представляє інтерфейс *ICollection* – інтерфейс, реалізований колекціями.

– *void Clear()*: видаляє зі списку всі елементи.

– *bool Contains(object value)*: перевіряє, чи міститься в списку об'єкт *value*. Якщо міститься, повертає *true*, інакше повертає *false*.

– *void CopyTo(Array array)*: копіює поточний список в масив *array*.

– *ArrayList GetRange(int index, int count)*: повертає новий список *ArrayList*, який містить *count* елементів поточного списку, починаючи з індексу *index*.

– *int IndexOf(object value)*: повертає індекс елемента *value*.

– *void Insert(int index, object value)*: вставляє в список за індексом *index* об'єкт *value*.

– *void InsertRange(int index, ICollection col)*: вставляє в список починаючи з індексу *index* колекцію *ICollection*.

– *int LastIndexOf(object value)*: повертає індекс останнього входження в списку об'єкта *value*.

– *void Remove(object value)*: видаляє зі списку об'єкт *value*.

– *void RemoveAt(int index)*: видаляє зі списку елемент за індексом *index*.

– *void RemoveRange(int index, int count)*: видаляє зі списку *count* елементів, починаючи з індексу *index*.

– *void Reverse()*: перевертає список.

– *void SetRange(int index, ICollection col)*: копіює в список елементи колекції *col*, починаючи з індексу *index*.

– *void Sort()*: сортує колекцію.

Крім того, за допомогою властивості *Count* можна отримати кількість елементів у списку.

```
using System.Collections;
static void Main(string[] args)
    {
      ArrayList list = new ArrayList();
      list.Add(55); // додаємо в список об'єкт типу int
      list.AddRange(new string[] { "Hello", "world" }); додаємо в список строковий масив
      // видаляємо перший елемент
      list.RemoveAt(0);
      // перевертаємо список
      list.Reverse();
      // отримуємо елемент за індексом
      Console.WriteLine(list[0]);
      // перебираємо значення і виводимо на екран
      for (int i = 0; i < list.Count; i++)
      {
        Console.WriteLine(list[i]);
      }
      Console.ReadLine();
    }
```

List<T>

Клас *List*<T> з простору імен *System.Collections.Generic* представляє найпростіший список однотипних об'єктів.

Серед його методів можна виділити наступні:

– *void Add(T item)*: додавання нового елемента в список.

– *void AddRange(ICollection collection)*: додавання в список колекції або масиву.

– *int IndexOf(T item)*: повертає індекс першого входження елемента в списку.

– *void Insert(int index, T item)*: вставляє елемент *item* в списку на позицію *index*.

– *bool Remove(T item)*: видаляє елемент *item* зі списку, і якщо видалення пройшло успішно, то повертає *true*.

– *void RemoveAt(int index)*: видалення елемента за вказаною індексу *index*.

– *void Sort()*: сортування списку.

Приклад створення і використання *List* <T>:

```
List<int> numbers = new List<int>() { 1, 2, 3, 45 };
numbers.Add(6); // додавання елементу
numbers.AddRange(new int[] { 7, 8, 9 });
numbers.Insert(0, 666); // додаємо на перше місце в списку число 666
numbers.RemoveAt(1); // видаляємо другий елемент
```

LinkedList<T>

Клас *LinkedList*<T> представляє двохзв'язний список, в якому кожен елемент зберігає посилання одночасно на наступний і на попередній елемент. Якщо в простому списку *List*<T> кожен елемент являє собою об'єкт типу T, то в *LinkedList* <T> кожен вузол являє собою об'єкт класу *LinkedListNode*<T>. Цей клас має наступні властивості:

– *Value*: саме значення вузла, представлене типом T.

– *Next*: посилання на наступний елемент типу *LinkedListNode*<T> в списку. Якщо наступний елемент відсутній, то має значення *null.*

– *Previous*: посилання на попередній елемент типу *LinkedListNode*<T> в списку. Якщо попередній елемент відсутній, то має значення *null.*

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

– *AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode)*: вставляє вузол *newNode* в список після вузла *node*.

– *AddAfter(LinkedListNode<T> node, T value)*: вставляє в список новий вузол із значенням *value* після вузла *node*.

– *AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode)*: вставляє в список вузол *newNode* перед вузлом *node*.

– *AddFirst(T value)*: вставляє новий вузол із значенням *value* в початок списку.

– *AddLast(T value)*: вставляє новий вузол із значенням *value* в кінець списку.

– *RemoveFirst()*: видаляє перший вузол зі списку. Після цього новим першим вузлом стає вузол, наступний за видаленим.

– *RemoveLast()*: видаляє останній вузол зі списку.

Приклад використання *LinkedList* <T>:

```
LinkedList<int> numbers = new LinkedList<int>();
numbers.AddLast(1); // вставляємо вузел зі значенням 1 на останнє місце
numbers.AddFirst(2); // вставляємо вузел зі значенням 2 на перше місце
numbers.AddAfter(numbers.Last, 3); // додаємо після останнього вузла новий вузол зі
//значенням 3 маємо список з наступною послідовністю: 2, 1, 3
```

Queue<T>

Клас *Queue*<T> представляє звичайну чергу, працює за алгоритмом FIFO («перший прийшов – першим вийшов»).

В класі *Queue*<T> можна відзначити наступні методи:

– *Dequeue*: витягує і повертає перший елемент черги.

– *Enqueue*: додає елемент в кінець черги.

– *Peek*: просто повертає перший елемент з початку черги без його видалення.

Stack<T>

Клас *Stack*<T> представляє колекцію, яка використовує алгоритм LIFO («останній прийшов – першим вийшов»). При такій організації кожен наступний доданий елемент розміщується поверх попереднього. Витяг з колекції відбувається в зворотному порядку – витягується той елемент, який знаходиться вище всіх в стеці.

У класі *Stack* можна виділити два основні методи, які дозволяють управляти елементами:

– *Push*: додає елемент в стек на перше місце.

– *Pop*: витягує і повертає перший елемент з стека.

– *Peek*: просто повертає перший елемент з стека без його видалення.

Dictionary<T, V>

Ще один поширений тип колекції представляють словники. Словник зберігає об'єкти, які представляють пару ключ-значення. Кожен такий об'єкт є об'єктом структури *KeyValuePair<TKey, TValue>*. Завдяки властивостям *Key* і *Value*, які є у цієї структури, ми можемо отримати ключ і значення елемента в словнику.

У класі *Dictionary<TKey, TValue>* визначається також ряд методів:

– *Add(TKey, TValue)*: додає до словника пару "ключ-значення", яка визначається параметрами *key* і *value*. Якщо ключ *key* вже знаходиться в словнику, то його значення не змінюється, і генерується виключення *ArgumentException*.

– *ContainsKey(TKey)*: повертає логічне значення *true*, якщо словник містить об'єкт *key* як ключ; а інакше – логічне значення *false.*

– *ContainsValue(TValue)*: повертає логічне значення *true*, якщо словник містить значення *value*; в іншому випадку – логічне значення *false.*

– *Remove(TKey)*: видаляє ключ *key* зі словника. При вдалому результаті операції повертається логічне значення *true*, а якщо ключ *key* відсутній в словнику – логічне значення *false.*

Крім того, в класі *Dictionary<TKey, TValue>* визначаються власні властивості:

– *Comparer*: отримує метод порівняння для викликає словника.

– *Keys*: отримує колекцію ключів.

– *Values*: отримує колекцію значень.

Ініціалізація словників:

```
Dictionary<string, string> countries = new Dictionary<string, string>
{
   {"Франція", "Париж"},
   {"Германія", "Берлін"},
   {"Великобританія", "Лондон"}
};
```

Для додавання необов'язково застосовувати метод *Add(TKey, TValue)*, можна використовувати скорочений варіант:

```
Dictionary<char, Person> people = new Dictionary<char, Person>();
people.Add('b', new Person() { Name = "Bill" });
people['a'] = new Person() { Name = "Alice" };
```

LINQ

LINQ (Language-Integrated Query) представляє просту і зручну мову запитів до джерела даних. Як джерело даних може виступати об'єкт, який реалізує інтерфейс *IEnumerable* (наприклад, стандартні колекції, масиви), набір даних *DataSet*, документ XML.

Список методів розширення LINQ:

– *Select*: визначає проекцію обраних значень.

– *Where*: визначає фільтр вибірки.

– *OrderBy*: впорядковує елементи за зростанням.

– *OrderByDescending*: впорядковує елементи за спаданням.

– *ThenBy*: задає додаткові критерії для упорядкування елементів за зростанням.

– *ThenByDescending*: задає додаткові критерії для упорядкування елементів за спаданням.

– *Join*: з'єднує дві колекції за певною ознакою.

– *GroupBy*: групує елементи по ключу.

– *ToLookup*: групує елементи по ключу, при цьому всі елементи додаються в словник.

– *GroupJoin*: виконує одночасно з'єднання колекцій і угруповання елементів по ключу.

– *Reverse*: змінює елементи в зворотному порядку.

– *All*: визначає, чи всі елементи колекції задовольняють певній умові.

– *Any*: визначає, задовольняє хоча б один елемент колекції певній умові.

– *Contains*: визначає, чи містить колекція певний елемент.

– *Distinct*: видаляє дубльовані елементи з колекції.

– *Except*: повертає різницю двох колекцій, тобто ті елементи, які створюються тільки в одній колекції.

– *Union*: об'єднує дві однорідні колекції.

– *Intersect*: повертає перетин двох колекцій, тобто ті елементи, які зустрічаються в обох колекціях.

– *Count*: підраховує кількість елементів колекції, які задовольняють певній умові.

– *Sum*: підраховує суму числових значень в колекції.

– *Average*: підраховує середнє значення числових значень в колекції.

– *Min*: знаходить мінімальне значення.

– *Max*: знаходить максимальне значення.

– *Concat*: об'єднує дві колекції.

– *Zip*: об'єднує дві колекції відповідно до визначеної умови.

– *First*: вибирає перший елемент колекції.

– *FirstOrDefault*: вибирає перший елемент колекції або повертає значення за замовчуванням.

– *Single*: вибирає єдиний елемент колекції, якщо колекція має більше або менше одного елемента, то генерується виключення.

– *SingleOrDefault*: вибирає перший елемент колекції або повертає значення за замовчуванням

– *ElementAt*: вибирає елемент послідовності за певним індексом.

– *ElementAtOrDefault*: вибирає елемент колекції за певним індексом або повертає значення за замовчуванням, якщо індекс поза допустимого діапазону.

– *Last*: вибирає останній елемент колекції.

– *LastOrDefault*: вибирає останній елемент колекції або повертає значення за замовчуванням.

Виберемо з масиву рядки, що починаються на певну букву і відсортуємо отриманий список:

```
string[] teams = {"Баварія", "Борусія", "Реал Мадрид", "Манчестер Сіті", "ПСЖ", "Барселона"};
var selectedTeams = from t in teams // визначає кожен об'єкт з teams як t
                    where t.ToUpper().StartsWith("Б") //фільтрація за критерієм
```

```
                    orderby t //сортування за зростанням
                    select t; // обираємо об'єкт
foreach (string s in selectedTeams)
   Console.WriteLine(s);
```

Вираз *from t in teams* проходить по всіх елементах масиву *teams* і визначає кожен елемент як *t*. Використовуючи змінну *t* ми можемо проводити над нею різні операції. Незважаючи на те, що ми не вказуємо тип змінної *t*, вираз LINQ є строго типізованим. Тобто середовище автоматично розпізнає, що набір *teams* складається з об'єктів *string*, тому змінна *t* буде розглядатися як ще один рядок.

Далі за допомогою оператора *where* проводиться фільтрація об'єктів, і якщо об'єкт відповідає критерію (в даному випадку початкова буква повинна бути «Б»), то цей об'єкт передається далі. Оператор *orderby* впорядковує за зростанням, тобто сортує вибрані об'єкти. Оператор *select* передає обрані значення в результуючу вибірку, яка повертається LINQ-виразом. В даному випадку результатом виразу LINQ є об'єкт *IEnumerable* <T>.

Крім стандартного синтаксису *from .. in .. select* для створення запиту LINQ ми можемо застосовувати спеціальні методи розширення, які визначені для інтерфейсу *IEnumerable*. Як правило, ці методи реалізують ту ж функціональність, що і оператори LINQ типу *where* або *orderby*.

```
string[] teams = { "Баварія", "Борусія", "Реал Мадрид", "Манчестер Сіті", "ПСЖ", "Барселона" };
var selectedTeams = teams.Where(t=>t.ToUpper().StartsWith("Б")).OrderBy(t => t);
foreach (string s in selectedTeams)
   Console.WriteLine(s);
```

Серіалізація

Щоб об'єкт певного класу можна було серіалізувати, треба цей клас помітити атрибутом *Serializable*:

```
[Serializable]
class Person
{
   public string Name {get; set; }
   public int Year {get; set; }

   public Person (string name, int year)
   {
      Name = name;
      Year = year;
   }
}
```

При відсутності даного атрибуту об'єкт *Person* не зможе бути серіалізований, і при спробі серіалізації буде викинуто виключення *SerializationException*. Серіалізація застосовується до властивостей і полів класу. Якщо ми не хочемо, щоб якесь поле класу серіалізувалося, то ми його помічаємо атрибутом *NonSerialized*.

При наслідуванні подібного класу, слід враховувати, що атрибут *Serializable* автоматично не успадковується. І якщо ми хочемо, щоб похідний клас також міг

би бути серіалізований, то знову ж таки ми застосовуємо до нього атрибут *Serializable*.

Для бінарної серіалізації застосовується клас *BinaryFormatter*:

```
static void Main(string[] args)
    {
      // об'єкт для серіалізації
      Person person = new Person("Tom", 29);
      Console.WriteLine("Объект создан");
      // створюємо об'єкт BinaryFormatter
      BinaryFormatter formatter = new BinaryFormatter();
      // отримуємо потік, куди будемо записувати серіалізований об'єкт
      using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))
      {
        formatter.Serialize(fs, person);
        Console.WriteLine("Объект сериализован");
      }
      // десеріализація з файлу people.dat
      using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))
      {
        Person newPerson = (Person)formatter.Deserialize(fs);
        Console.WriteLine("Объект десериализован");
        Console.WriteLine($"Имя: {newPerson.Name} --- Возраст: {newPerson.Age}");
      }
      Console.ReadLine();
    }
```

Робота з JSON

Ключовим типом є клас *JsonSerializer*, який і дозволяє серіалізувати об'єкт в *json* і, навпаки, десеріалізувати код *json* в об'єкт C #.

Для збереження об'єкта в *json* в класі *JsonSerializer* визначено статичний метод *Serialize()*, який має ряд перевантажених версій. Деякі з них:

– *string Serialize(Object obj, Type type, JsonSerializerOptions options)*: серіалізується об'єкт *obj* типу *type* і повертає код *json* у вигляді рядка. Останній необов'язковий параметр *options* дозволяє задати додаткові опції серіалізації.

– *string Serialize <T>(T obj, JsonSerializerOptions options)*: типізована версія серіалізує об'єкт *obj* типу T і повертає код *json* у вигляді рядка.

– *object Deserialize(string json, Type type, JsonSerializerOptions options)*: десеріалізує рядок *json* в об'єкт типу *type* і повертає десеріалізованний об'єкт. Останній необов'язковий параметр *options* дозволяє задати додаткові опції десеріалізації.

– *T Deserialize<T>(string json, JsonSerializerOptions options)*: десеріалізує рядок *json* в об'єкт типу T і повертає його.

Приклад серіалізації і десеріалізації найпростішого об'єкта:

```
class Person
  {
    public string Name { get; set; }
    public int Age { get; set; }
  }
  class Program
  {
```

```
    static void Main(string[] args)
    {
        Person tom = new Person { Name = "Tom", Age = 35 };
        string json = JsonSerializer.Serialize<Person>(tom);
        Console.WriteLine(json);
        Person restoredPerson = JsonSerializer.Deserialize<Person>(json);
        Console.WriteLine(restoredPerson.Name);
    }
}
```

Об'єкт, який піддається десеріалізації, повинен мати конструктор без параметрів. Серіалізації підлягають тільки публічні властивості об'єкта (з модифікатором *public*). Атрибут *JsonIgnore* дозволяє виключити з серіалізації певну властивість. А *JsonPropertyName* дозволяє заміщати оригінальну назву властивості.

За замовчуванням *JsonSerializer* серіалізує об'єкти в мініміфіцированій код. За допомогою додаткового параметра типу *JsonSerializerOptions* можна налаштувати механізм серіалізації/десеріалізації, використовуючи властивості *JsonSerializerOptions*. Деякі з його властивостей:

– *AllowTrailingCommas*: встановлює, чи треба додавати після останнього елемента в *json* кому. Якщо значення *true*, кома додається.

– *IgnoreNullValues*: встановлює, чи будуть серіалізовані/десеріалізовані в *json* об'єкти і їх властивості зі значенням *null*.

– *IgnoreReadOnlyProperties*: аналогічно встановлює, чи будуть серіалізовані властивості, призначені тільки для читання.

– *WriteIndented*: встановлює, чи будуть додаватися в *json* пробіли (умовно кажучи, для краси). Якщо значення *true* встановлюються додаткові пробіли.

5.3 Порядок виконання роботи

1. Ознайомитися з теоретичним матеріалом.
2. Виконати індивідуально завдання з пункту 5.4.
3. Оформити звіт.
4. Здати практичну частину.

5.4 Індивідуальні завдання

Завдання №1

Створити клас *Student* з полями (властивостями) ім'я – *Name* (*String*), вік – *Age* (*Int*), колекція предмет-оцінка – *Marks* (*Dictionary<String*, *Float>*) та конструктором для створення об'єктів. Вважати, що у кожного студента три предмета: «Mathematics», «Philosophy», «English», а оцінка варіюється від 1 до 100. В головній функції програми створити колекцію об'єктів *Student* (не менше 10).

Завдання №2

Для створеної колекції об’єктів із завдання 1 провести наступні операції використовуючи LINQ:

– вибрати з колекції всі елементи, для яких довжина ім’я більше ніж 4 символи та вік менше 20. Результат відсортувати за ім’ям в зворотному напрямі;

– відсортувати студентів за оцінкою по математиці;

– отримати кількість студентів, які не склали хоча б один іспит;

– розрахувати та вивести середній бал по кожному з предметів;

– згрупувати студентів за віком та відобразити результат з вказанням кількості елементів в кожній групі;

Завдання №3

Серіалізувати колекцію об’єктів створену в завд. 1 в *json* файл. Врахувати, що поле *Marks* не серіалізується, а для поля *Name* замістити оригінальну назву на *FirstName*. Провести десеріалізацію отриманого файлу та вивести результат на екран.

5.5 Зміст звіту

Звіт має містити:

– мету роботи;

– індивідуальне завдання;

– код програми;

– результат виконання програми;

– висновки.

5.6 Контрольні питання та завдання

1. Назвіть колекції, які використовуються в мові програмування C#?
2. Що таке словник? Як він використовується і для чого потрібен?
3. Назвіть основні методи для роботи з колекцією *List*<T>?
4. Що таке LINQ? Для чого його використовують?
5. Як проводиться сортування та фільтрація даних з використанням LINQ?
6. Як групуються та об’єднуються дані з використанням LINQ?
7. Що таке серіалізація? Які формати серіалізації існують?