



КОЛЕКЦІЇ В МОВІ ПРОГРАМУВАННЯ C#

Лекція 07
Об'єктно-орієнтоване програмування

План лекції

- Огляд API колекцій.
- Колекції з простору імен `System.Collections.Generic`.
- Колекції з простору імен `System.Collections`.

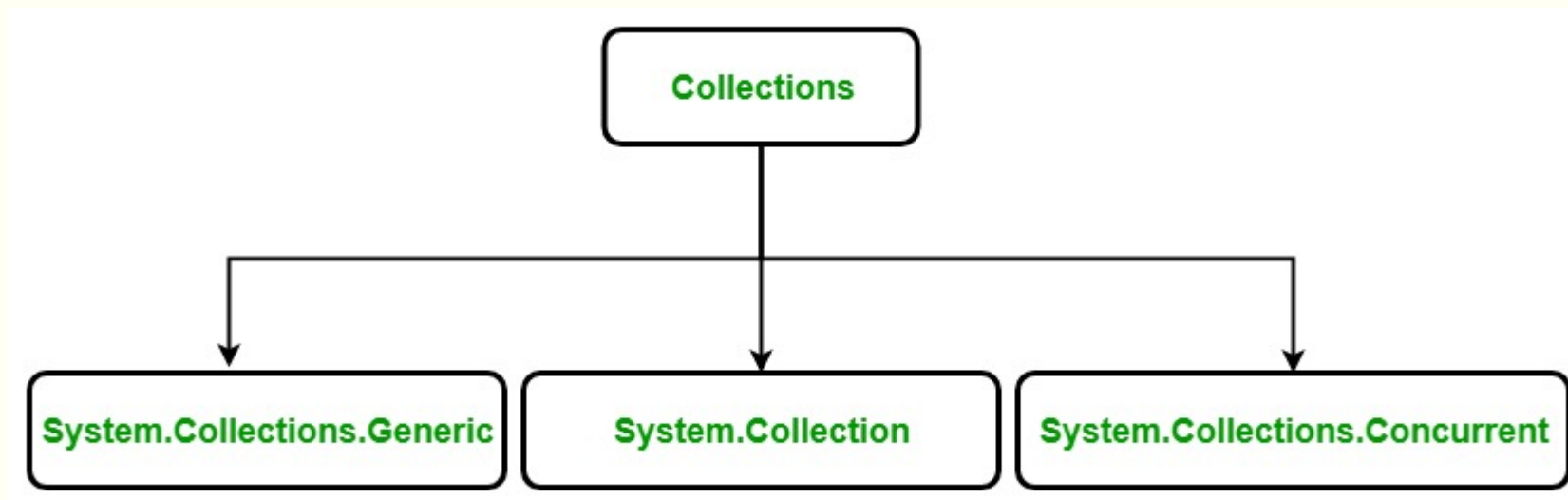


ОГЛЯД АРІ КОЛЕКЦІЙ

Питання 7.1.

Колекції та структури даних

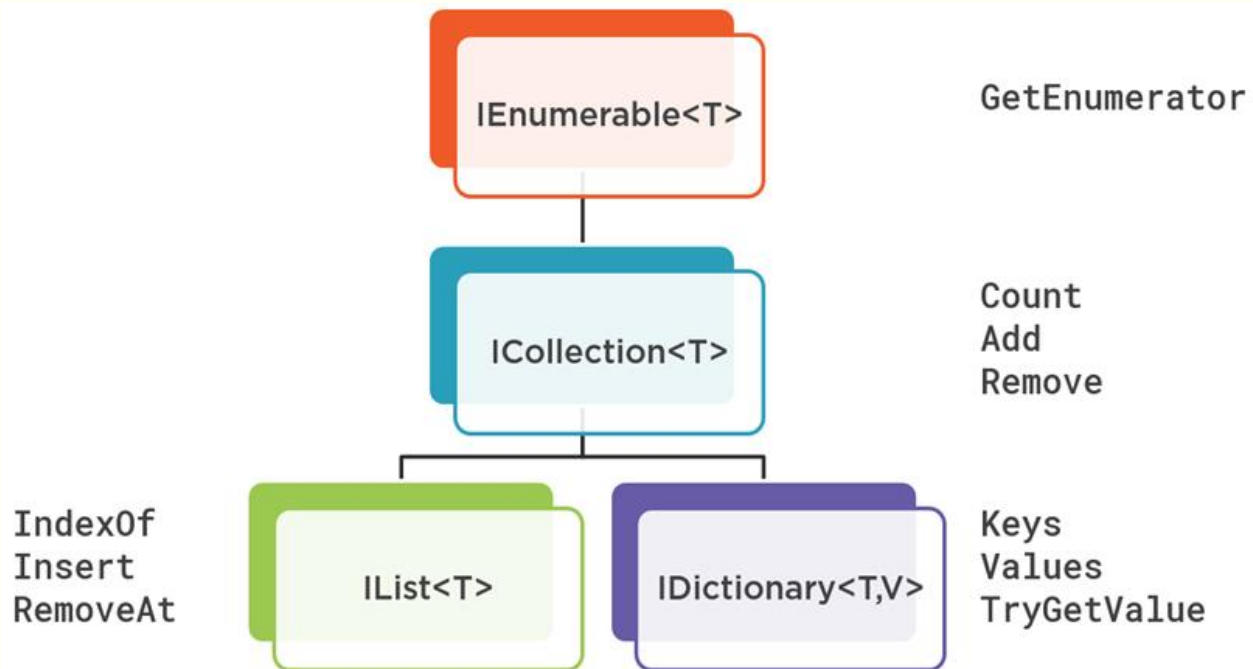
- Подібні дані часто можуть оброблятись більш ефективно при їх зберіганні та обробці в формі колекції.
 - Можуть застосовуватись клас `System.Array` чи класи з просторів імен `System.Collections.Generic`, `System.Collections`, `System.Collections.Concurrent` та `System.Collections.Immutable`, щоб додавати, видаляти чи змінювати окремі елементи або діапазон елементів колекції.



Колекції та структури даних

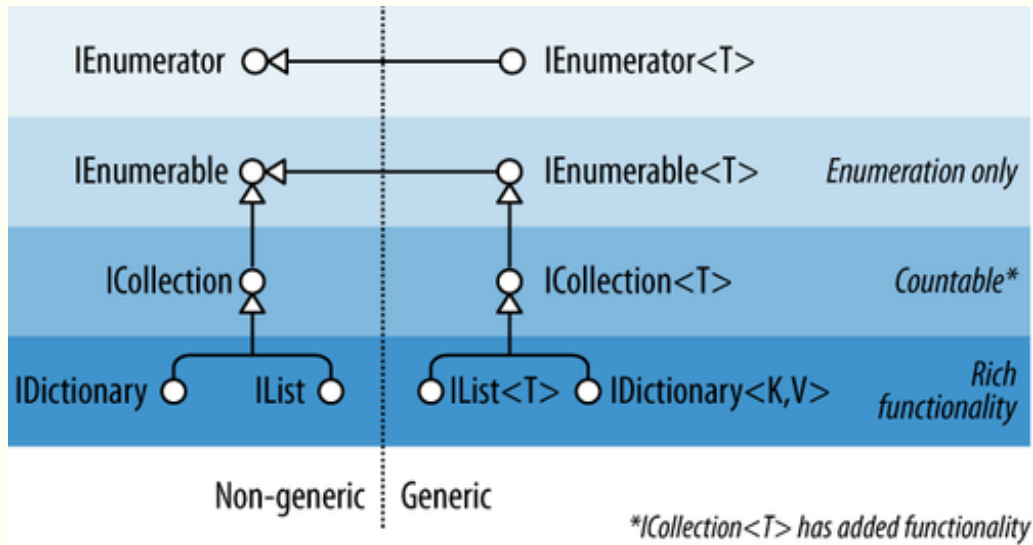
- Розглядають два основних види колекцій: узагальнені та неузагальнені.
- *Узагальнені колекції* типобезпечні при компіляції, тому часто пропонують вищу продуктивність.
 - Вони приймають параметр типу при конструюванні та не вимагають зведення типу від `System.Object` при вставці чи видаленні елементів колекції.
 - Більшість узагальнених колекцій підтримуються в додатках для Windows Store.
- *Неузагальнені колекції* зберігають елементи колекції як об'єкти типу `System.Object`, вимагають зведення типів та переважно не підтримуються додатками для Windows Store.
 - Проте зустрічаються в легасі-кодi.
- Починаючи з .NET Framework 4, колекції з простору імен `System.Collections.Concurrent` забезпечують ефективні потокобезпечні операції доступу до елементів колекції з багатьох потоків.
 - *Незмінювані (immutable) колекції* з простору імен `System.Collections.Immutable` (пакет NuGet) потокобезпечні за своєю суттю, оскільки операції виконуються на копії початкової колекції, а початкову колекцію змінити неможливо.

Спільні риси колекцій



- Всі колекції постачають методи для додавання, видалення та пошуку своїх елементів.
 - Також всі колекції напряду чи опосередковано реалізують інтерфейс `ICollection` або `ICollection<T>`.
- Дані інтерфейси передбачають:
 - Здатність ітерувати (`enumerate`) по колекції
 - Можливість копіювати вміст колекції в масив
- Крім цього, багато класів містять наступні можливості:
 - Властивості `Capacity` і `Count`
 - Синхронізований доступ у багатопоточному контексті
 - Узгоджена нижня межа (`consistent lower bound`)

Ітерування по колекціях



- Інтерфейс `IEnumerator` визначає базовий низькорівневий протокол обходу колекцій *тільки вперед*.
 - Зауважте, що колекції не реалізують еnumератори (лічильники), а радше, постачають їх через інтерфейс `IEnumerable`.
 - Інтерфейс `IEnumerable` є найбільш базовим інтерфейсом, що реалізують .NET-колекції.

```
namespace System.Collections
{
    //
    // Сводка:
    //     Supports a simple iteration over a non-generic collection.
    public interface IEnumerator
    {
        //
        // Сводка:
        //     Gets the element in the collection at the current position of the enumerator.
        //
        // Возврат:
        //     The element in the collection at the current position of the enumerator.
        object? Current { get; }

        //
        // Сводка:
        //     Advances the enumerator to the next element of the collection.
        //
        // Возврат:
        //     true if the enumerator was successfully advanced to the next element; false if
        //     the enumerator has passed the end of the collection.
        //
        // Исключения:
        //     T:System.InvalidOperationException:
        //         The collection was modified after the enumerator was created.
        bool MoveNext();

        //
        // Сводка:
        //     Sets the enumerator to its initial position, which is before the first element
        //     in the collection.
        //
        // Исключения:
        //     T:System.InvalidOperationException:
        //         The collection was modified after the enumerator was created.
        void Reset();
    }
}
```

Ітерування по колекціях

```
namespace System.Collections
{
    //
    // Сводка:
    //     Exposes an enumerator, which supports a simple iteration over a non-generic collection.
    public interface IEnumerable
    {
        //
        // Сводка:
        //     Returns an enumerator that iterates through a collection.
        //
        // Возврат:
        //     An System.Collections.IEnumerator object that can be used to iterate through
        //     the collection.
        IEnumerator GetEnumerator();
    }
}
```

- Визначаючи єдиний метод для повернення енумератора інтерфейс `IEnumerable`:
 - дозволяє реалізувати логіку ітерування в окремому класі.
 - дозволяє кільком клієнтам (consumers) обходити колекцію одночасно, не втручаючись у роботу один одного.


```
namespace System.Collections.Generic
```

```
{  
    //  
    // Сводка:  
    //     Supports a simple iteration over a generic collection.  
    //  
    // Параметры типа:  
    //     T:  
    //     The type of objects to enumerate.  
    public interface IEnumerator<out T> : IEnumerator, IDisposable  
    {  
        //  
        // Сводка:  
        //     Gets the element in the collection at the current position of the enumerator.  
        //  
        // Возврат:  
        //     The element in the collection at the current position of the enumerator.  
        T Current { get; }  
    }  
}
```

Ітерування по колекціях

- Узагальнені аналоги інтерфейсів IEnumerator та IEnumerable.
 - Такі інтерфейси підтримують безпеку типів та уникають накладних витрат на упаковання/розпакування значимих типів.
 - Масиви автоматично реалізують IEnumerable<T>.

```
namespace System.Collections.Generic  
{  
    //  
    // Сводка:  
    //     Exposes the enumerator, which supports a simple iteration over a collection of  
    //     a specified type.  
    //  
    // Параметры типа:  
    //     T:  
    //     The type of objects to enumerate.  
    public interface IEnumerable<out T> : IEnumerator  
    {  
        //  
        // Сводка:  
        //     Returns an enumerator that iterates through the collection.  
        //  
        // Возврат:  
        //     An enumerator that can be used to iterate through the collection.  
        IEnumerator<T> GetEnumerator();  
    }  
}
```

Ітерування по колекціях

- Стандартна практика для класів колекцій – публічно розкривати `IEnumerable<T>`, приховуючи неузагальнену версію `IEnumerator`.
 - Проте зверніть увагу, що виклик `GetEnumerator` для масиву повертає неузагальнений `IEnumerator`.
 - Це зроблено для підтримки зворотної сумісності:

```
public void BasicEnumeratorUsage()
{
    // Узагальнені колекції публічно розкривають IEnumerable<T>
    List<int> numbers = new List<int> { 1, 2, 3 };
    IEnumerator<int> enumerator = numbers.GetEnumerator();

    // Масиви публічно розкривають IEnumerable.
    // Тому GetEnumerator повертає неузагальнений IEnumerator
    int[] numbers2 = numbers.ToArray();
    IEnumerator enumerator2 = numbers2.GetEnumerator();

    // Щоб отримати узагальнений IEnumerator, використовуйте зведення типів
    // NOTE: Ви рідко будете використовувати такий код, оскільки зазвичай застосовується foreach
    IEnumerator<int> enumerator3 = ((IEnumerable<int>)numbers2).GetEnumerator();
}
```

Ітерування по колекціях

- `IEnumerator<T>` також реалізує інтерфейс `IDisposable`.
 - Це дозволяє еnumераторам містити посилання на ресурси, зокрема підключення до БД, та забезпечувати вивільнення таких ресурсів після завершення обходу.
 - Код

```
foreach (var item in numbers) { ... }
```

транслюється в

```
using (var enumerator = numbers.GetEnumerator())  
{  
    while (enumerator.MoveNext())  
    {  
        var element = enumerator.Current;  
  
        ...  
    }  
} // IDisposable.Dispose() called here
```

Ітерування по колекціях

- Інтерфейси `IEnumerable` / `IEnumerable<T>` реалізуються з наступних причин:
 - підтримка інструкції `foreach`;
 - взаємодія (`interoperate`) з будь-яким типом чи методом, що очікують стандартну колекцію;
 - підтримка ініціалізаторів колекцій.
- Для реалізації `IEnumerable` / `IEnumerable<T>` в класі необхідно постачати енумератор одним з трьох способів:
 - якщо клас, що реалізує `IEnumerable` / `IEnumerable<T>`, огортає (`wraps`) колекцію, повертати енумератор колекції;
 - використовувати ітератор (оператор `yield return`);
 - інстанціювати клас, що реалізує (`derive`) `IEnumerator` / `IEnumerator<T>`.

Ітерування по колекціях

■ Опція 2: ітератори.

- Метод-ітератор або get-виклик виконують визначене користувачем ітерування по колекції.
- Метод-ітератор використовує інструкцію yield return, щоб повертати за один раз один елемент колекції.
- Коли досягається інструкція yield return, поточне місце в коді запам'ятовується, а виконання перезапускається з нього при наступному виклику функції-ітератора.

■ Порівняємо реалізацію з традиційними циклами та реалізацію з yield return:

```
IEnumerable<int> GenerateWithoutYield()  
{  
    var i = 0;  
    var list = new List<int>();  
    while (i < 5)  
        list.Add(++i);  
    return list;  
}
```

```
IEnumerable<int> GenerateWithYield()  
{  
    var i = 0;  
    while (i < 5)  
        yield return ++i;  
}
```

Традиційні цикли vs yield return

- Версія з традиційними циклами: `foreach (var number in GenerateWithoutYield()) Console.WriteLine(number);`
 - Викликається `GenerateWithoutYield()`.
 - Весь метод виконується і конструюється список.
 - Цикл `foreach` обходить усі значення зі списку.
- Версія з `yield return`: `foreach (var number in GenerateWithYield()) Console.WriteLine(number);`
 - Викликається `GenerateWithYield()`.
 - Вона повертає `Ienumerable`-об'єкт: не список, а «обіцянку» повернути послідовність чисел, коли про цю послідовність запитують (тобто розкриває ітератор, який виконуватиме обіцянку).
 - Кожна ітерація циклу `foreach` викликає метод-ітератор. Коли досягається `yield return`, значення повертається, а поточне місце в коді запам'ятовується. При наступному виклику функції-ітератора відбувається перезапуск з цього місця.

Традиційні цикли vs yield return

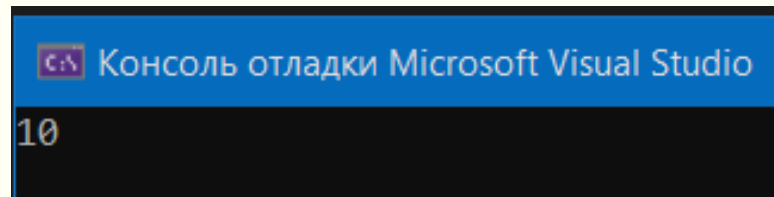
- Нескінченні цикли показують чітку відмінність:
 - Традиційний цикл буде зациклюватись у такій реалізації.
 - Ітератор завдяки методу Take() нормально відпрацює.

```
IEnumerable<int> GenerateWithoutYield()
{
    var i = 0;
    var list = new List<int>();
    while (true)
        list.Add(++i);
    return list;
}

foreach (var number in GenerateWithoutYield().Take(5))
    Console.WriteLine(number);
```

```
IEnumerable<int> GenerateWithYield()
{
    var i = 0;
    while (true)
        yield return ++i;
}

foreach (var number in GenerateWithYield().Take(5))
    Console.WriteLine(number);
```



Множинні ітерації

```
class Program
{
    static void Main(string[] args)
    {
        IEnumerable<Invoice> GetInvoices()
        {
            for (var i = 1; i < 11; i++)
                yield return new Invoice { Amount = i * 10 };
        }

        void DoubleAmounts(IEnumerable<Invoice> invoices)
        {
            foreach (var invoice in invoices)
                invoice.Amount *= 2;
        }

        var invoices = GetInvoices();
        DoubleAmounts(invoices);
        Console.WriteLine(invoices.First().Amount);
    }
}

class Invoice { public double Amount { get; set; } }
```

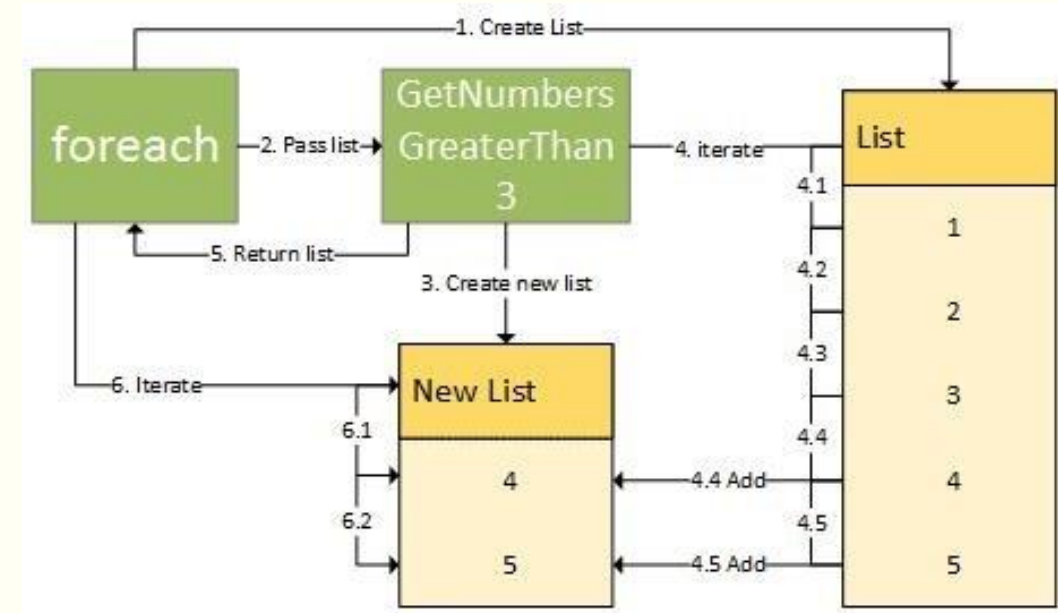
- Інший побічний ефект ітераторів: множинні виклики (invocations) призведуть до різних ітерацій.
 - Коли виконується рядок `var invoices = GetInvoices();` ми не отримуємо список інвойсів, а отримуємо генератор (state machine), який може створювати інвойси.
 - Даний генератор передається в `DoubleAmounts()`, всередині якого він генерує інвойси, суми яких будуть подвоєні.
 - Усі інвойси будуть створені та відкинуті, оскільки до них немає звернень.
 - Повернувшись у метод `Main()`, ми все ще маємо посилання на генератор.
 - Викликавши метод `First`, ми знову просимо згенерувати інвойси, які створюються заново.
 - Таким чином, виводить 10, а не 20.

Власний ітератор

- Наприклад, для виводу всіх чисел послідовності, більших за трійку.
 - *Традиційна реалізація:*

```
IEnumerable<int> GetNumbersGreaterThan3(List<int> numbers)
{
    var theNumbers = new List<int>();
    foreach (var nr in numbers)
    {
        if (nr > 3)
            theNumbers.Add(nr);
    }
    return theNumbers;
}
```

```
foreach (var nr in GetNumbersGreaterThan3(new List<int> { 1, 2, 3, 4, 5 }))
    Console.WriteLine(nr);
```

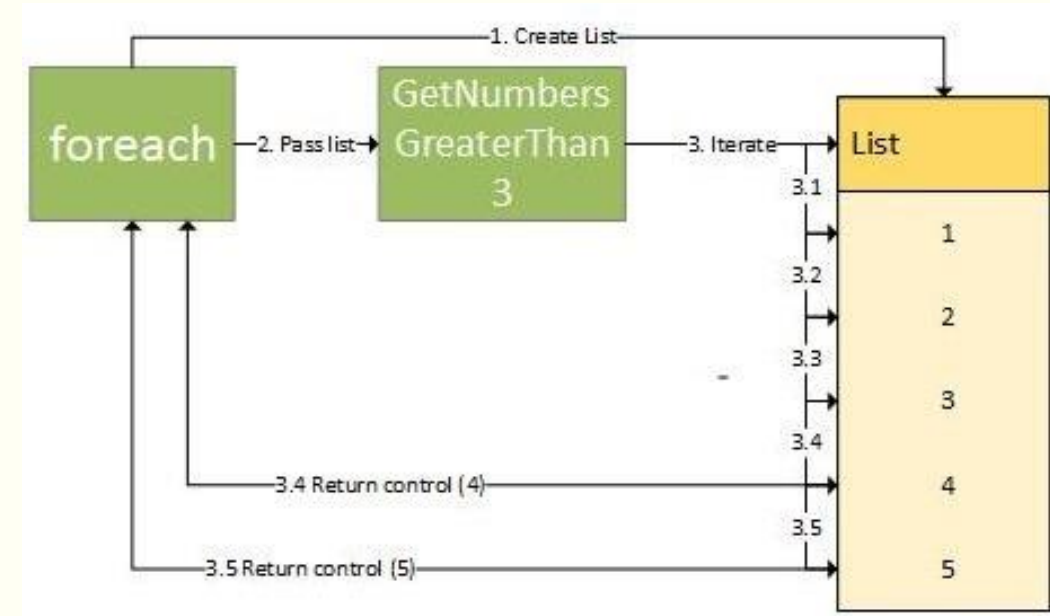


Власний ітератор

- Наприклад, для виводу всіх чисел послідовності, більших за трійку.
 - *Реалізація з ітератором:*

```
IEnumerable<int> GetNumbersGreaterThanOr3(List<int> numbers)
{
    foreach (var nr in numbers)
    {
        if (nr > 3)
            yield return nr;
    }
}
```

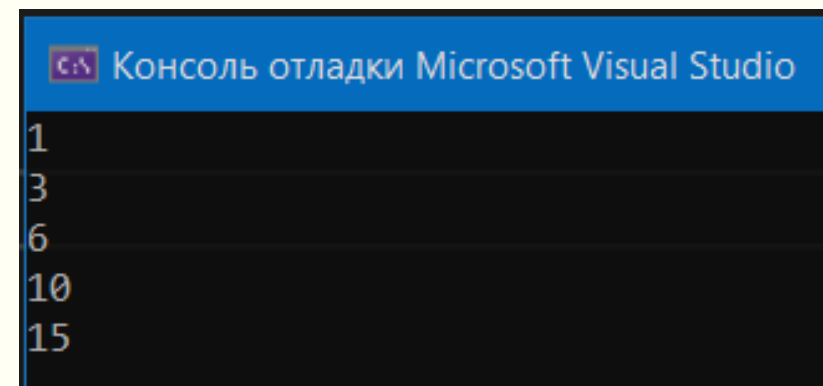
```
foreach (var nr in GetNumbersGreaterThanOr3(new List<int> { 1, 2, 3, 4, 5 }))
    Console.WriteLine(nr);
```



Ітератори зберігають стан

- Оскільки метод, що містить інструкцію `yield return`, призупиняє та відновлює роботу там, де знаходиться дана інструкція, таким чином підтримується стан всередині методу.

```
IEnumerable<int> Totals(List<int> numbers)
{
    var total = 0;
    foreach (var number in numbers)
    {
        total += number;
        yield return total;
    }
}
foreach (var total in Totals(new List<int> { 1, 2, 3, 4, 5 }))
    Console.WriteLine(total);
```



- Через поведінку призупинення/відновлення змінна `total` зберігатиме значення між ітераціями.
- Це може бути корисним для виконання stateful-обчислень.

Відкладене (deferred) виконання

- За допомогою механізму відкладеного виконання деякі методи можна спростити, пришвидшити та навіть зробити те, що було неможливим до цього (зокрема, нескінченний генератор чисел).
 - Вся підсистема LINQ у мові C# будується навколо відкладеного виконання.
- Приклад: нехай є 1000 товарів та LINQ-запит:

```
var dollarPrices = FetchProducts().Take(10)
                                .Select(p => p.CalculatePrice())
                                .OrderBy(price => price)
                                .Take(5)
                                .Select(price => ConvertTo$(price));
```

 - Без відкладеного виконання довелось би отримувати всі 1000 товарів, обчислювати ціни їх усіх, замовляти їх усі, перетворювати всі ціни в долари і брати топ-5 з цих цін.
 - Завдяки відкладеному виконанню процес спрощується: отримується 10 товарів, обчислюється їх ціни, 10 товарів замовляються, а ціни 5ти з них конвертуються в долари.

Додаткові особливості yield return

- Неможливо включати інструкції `yield return` або `yield break` у
 - лямбда-вирази та анонімні методи.
 - методи, в яких є `unsafe`-блоки.
- Інструкція `yield return` (на відміну від `yield break`) не може розташовуватись у блоці `try-catch`.

Інтерфейс ICollection

```
namespace System.Collections
{
    //
    // Сводка:
    //     Defines size, enumerators, and synchronization methods for all nongeneric collections
    public interface ICollection : IEnumerable
    {
        ...int Count { get; }
        ...bool IsSynchronized { get; }
        ...object SyncRoot { get; }

        ...void CopyTo(Array array, int index);
    }
}
```

- Передбачає 3 доступних тільки зчитування властивості та один метод.

```
// Сводка:
//     Gets the number of elements contained in the System.Collections.ICollection.
//
// Возврат:
//     The number of elements contained in the System.Collections.ICollection.
int Count { get; }
```

```
// Сводка:
//     Gets a value indicating whether access to the System.Collections.ICollection
//     is synchronized (thread safe).
//
// Возврат:
//     true if access to the System.Collections.ICollection is synchronized (thread
//     safe); otherwise, false.
bool IsSynchronized { get; }
```

Інтерфейс ICollection

```
// Сводка:
//     Gets an object that can be used to synchronize access to the System.Collections.ICollection.
//
// Возврат:
//     An object that can be used to synchronize access to the System.Collections.ICollection.
object SyncRoot { get; }

// Сводка:
//     Copies the elements of the System.Collections.ICollection to an System.Array,
//     starting at a particular System.Array index.
//
// Параметры:
//     array:
//         The one-dimensional System.Array that is the destination of the elements copied
//         from System.Collections.ICollection. The System.Array must have zero-based indexing.
//
//     index:
//         The zero-based index in array at which copying begins.
//
// Исключения:
//     T:System.ArgumentNullException:
//         array is null.
//
//     T:System.ArgumentOutOfRangeException:
//         index is less than zero.
//
//     T:System.ArgumentException:
//         array is multidimensional. -or- The number of elements in the source System.Collections.ICollection
//         is greater than the available space from index to the end of the destination
//         array. -or- The type of the source System.Collections.ICollection cannot be cast
//         automatically to the type of the destination array.
void CopyTo(Array array, int index);
```

- Більш спеціалізовані інтерфейси IDictionary та IList реалізують (extend) ICollection.
 - Реалізація IDictionary – це колекція з пар «ключ-значення», на зразок класу Hashtable.
 - Реалізація IList – це колекція значень, доступ до членів колекції відбувається за індексом, як для класу ArrayList.
 - Деякі колекції, які обмежують доступ до своїх елементів, зокрема класи Queue і Stack, напряду реалізують інтерфейс ICollection.

Інтерфейс ICollection<T>

```
namespace System.Collections.Generic
{
    //
    // Сводка:
    //     Defines methods to manipulate generic collections.
    //
    // Параметри типа:
    //     T:
    //     The type of the elements in the collection.
    public interface ICollection<T> : IEnumerable<T>, IEnumerable
    {
        ...int Count { get; }
        ...bool IsReadOnly { get; }

        ...void Add(T item);
        ...void Clear();
        ...bool Contains(T item);
        ...void CopyTo(T[] array, int arrayIndex);
        ...bool Remove(T item);
    }
}
```

Значно відрізняється від неузагальненої версії.

- Додано методи для додавання, видалення елементів колекції, її очистки та перевірки наявності елемента в колекції.
- Ймовірно, причина в різному часі появи: ICollection представлено в .NET 1.1, а ICollection<T> - в .NET 2.0.
- IDictionary<TKey,TValue> та IList<T> - більш спеціалізовані інтерфейси, які розширюють ICollection<T>.
- Реалізація IDictionary<TKey,TValue> - це колекція пар «ключ-значення» на зразок класу Dictionary<TKey,TValue>.
- Реалізація IList<T> - це колекція значень, дотуп до яких відбувається по індексу, на зразок класу List<T>.

Клас Collection<T>

- Клас Collection<T> з простору імен System.Collections.ObjectModel надає базову реалізацію узагальненої колекції, де T – тип елементів колекції.
- Характеристики класу Collection<T>:
 - може використовуватись негайно після створення екземпляру одного з типів, побудованих на базі нього.
 - Постачає захищені методи, які можуть використовуватись для кастомізації його поведінки вставки та видалення елементів, очистки колекції чи присвоєння значення існуючому елементу.
 - Більшість об'єктів типу Collection<T> можна змінювати. Проте Collection-об'єкт, ініціалізований як read-only IList<T> об'єкт змінювати не можна.
 - Доступ до елементів цієї колекції здійснюється за допомогою цілочисельного індексу, починаючи з 0.
 - Приймає null як коректне значення посилальних типів та дозволяє мати дубльовані елементи.
- Конструктори:
 - Collection<T>() новий екземпляр порожньої колекції.
 - Collection<T>(IList<T>) новий екземпляр колекції-обгортки навколо конкретного списку.

Клас Collection<T>

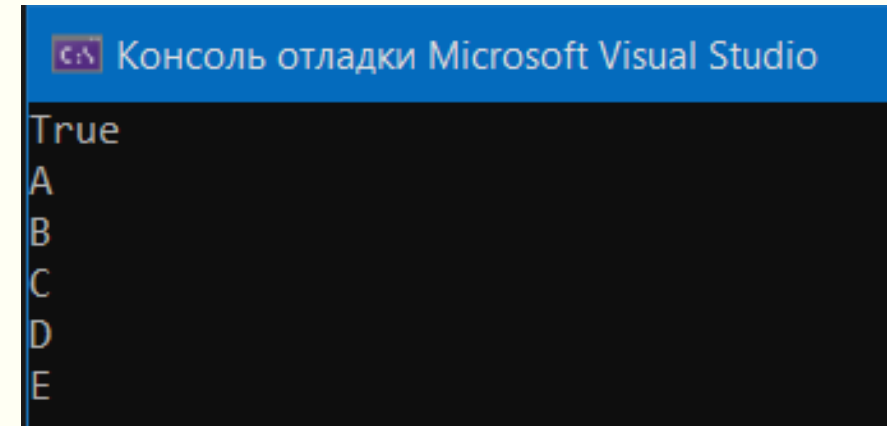
```
class Program
{
    static void Main(string[] args)
    {
        Collection<string> myColl = new Collection<string>();

        myColl.Add("A");
        myColl.Add("B");
        myColl.Add("C");
        myColl.Add("D");
        myColl.Add("E");

        Console.WriteLine(myColl.Contains("A"));

        string[] myArr = new string[myColl.Count];
        myColl.CopyTo(myArr, 0);

        // виведення елементів з myArr
        foreach (string str in myArr) { Console.WriteLine(str); }
    }
}
```



Реалізація власної колекції на базі класу Collection<T>

- Створимо клас Training, який описуватиме кілька властивостей, а потім зробимо колекцію елементів цього типу.

- Узагальнений клас Collection<T> реалізує ICollection<T>, забезпечуючи впровадження операцій вставки елементів, їх видалення та ін.
- За бажанням можна кастомізувати використання вставки та видалення елементів.

```
public class Training {
    public string Name { get; set; }
    public int Cost { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        var trainings = new Trainings();
        trainings.Insert(0, new Training { Name = "C#", Cost = 10 });
        trainings.Add(new Training() { Name = "Java", Cost = 10 });

        trainings.ForEach(Console.WriteLine);
        Console.ReadKey();
    }
}
```

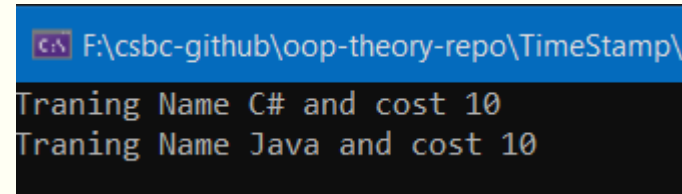
10.11.2020

```
public class Trainings : Collection<Training>
{
    public Training this[string name]
    {
        get { return this.Items.First(s => string.Equals(s.Name, name,
            StringComparison.OrdinalIgnoreCase)); }
    }

    public IEnumerable<string> All => this.Items.Select(s => s.Name);

    protected override void InsertItem(int index, Training item)
    {
        // validation before adding in common place.
        if (item.Cost > 0)
        {
            base.InsertItem(index, item);
        }
    }

    public void ForEach(Action<string> action)
    {
        foreach (var item in Items)
        {
            action($"Traning Name {item.Name} and cost {item.Cost}");
        }
    }
}
```



```
F:\csbc-github\oop-theory-repo\TimeStamp\
Traning Name C# and cost 10
Traning Name Java and cost 10
```

```

public class TrainingsCollection<T> : ICollection<T> where T : Training
{
    public TrainingsCollection()
    {
        List = new List<T>();
    }

    protected IList List { get; }
    public T this[int index] => (T)List[index];

    public int Count => this.List.Count;
    public bool IsReadOnly => throw new NotImplementedException();

    public void Add(T item)
    {
        if (!string.IsNullOrEmpty(item.Name))
        {
            this.List.Add(item);
        }
    }

    public void Clear()
    {
        this.List.Clear();
    }

    public bool Contains(T item)
    {
        return this.List.Contains(item);
    }

    public IEnumerator<T> GetEnumerator()
    {
        return new TrainingsEnumerator<T>(this);
    }
}

```

Реалізація власної узагальненої колекції на основі інтерфейсу ICollection<T>

```

public bool Remove(T item)
{
    if (item != null && Contains(item))
    {
        this.List.Remove(item);
        return true;
    }

    return false;
}

IEnumerator IEnumerable.GetEnumerator()
{
    return new TrainingsEnumerator<T>(this);
}

public void CopyTo(T[] array, int arrayIndex)
{
    this.List.CopyTo(array, arrayIndex);
}

public void ForEach(Action<string> action)
{
    foreach (var item in List.OfType<T>())
    {
        action($"Traning Name {item.Name} and cost {item.Cost}");
    }
}

```

- Дві реалізації GetEnumerator() потрібні для підтримки ітерування по власній колекції за допомогою циклу foreach

Реалізація власної узагальненої колекції на основі інтерфейсу ICollection<T>

```
public class TrainingsEnumerator<T> : IEnumerator<T> where T : Training
{
    private readonly TrainingsCollection<T> collection;
    public int Counter = -1;

    public TrainingsEnumerator(TrainingsCollection<T> collection) {
        this.collection = collection;
    }

    public object Current => collection[Counter];
    T IEnumerator<T>.Current => collection[Counter];

    public void Dispose() {
        Counter = -1;
    }

    public bool MoveNext() {
        ++Counter;
        if (collection.Count > Counter)
        {
            return true;
        }

        return false;
    }

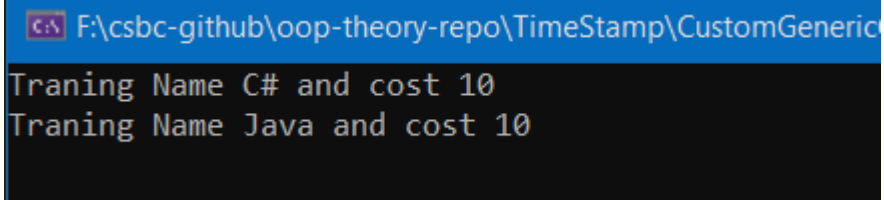
    public void Reset() {
        Counter = -1;
    }
}
```

■ Визначимо власний узагальнений Enumerator-об'єкт, який виконуватиме прохід по циклу.

```
class Program
{
    static void Main(string[] args)
    {
        var trainings = new TrainingsCollection<Training>();
        trainings.Add(new Training { Name = "C#", Cost = 10 });
        trainings.Add(new Training() { Name = "Java", Cost = 10 });

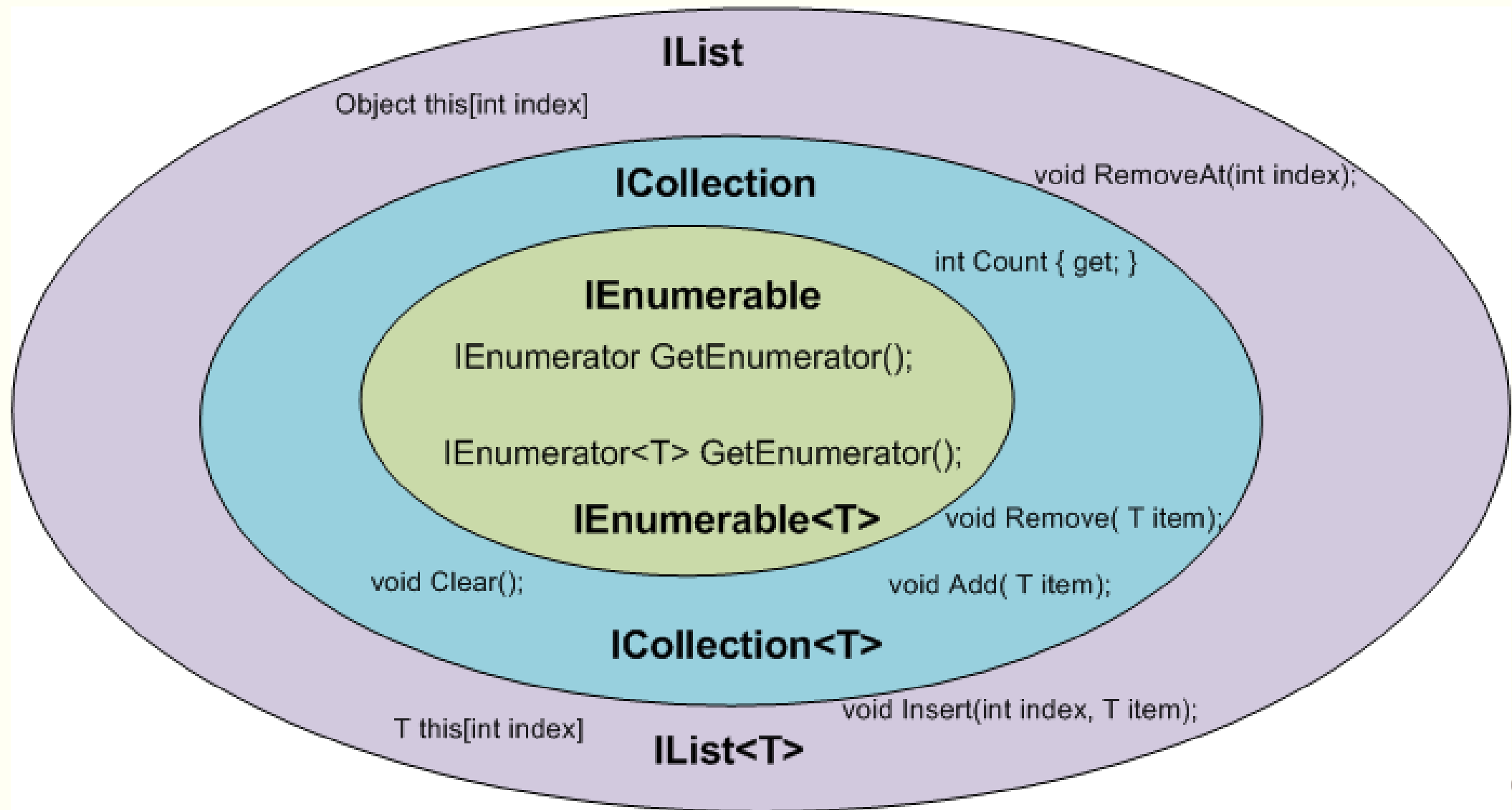
        // Цикл з викликом IEnumerator<T> GetEnumerator() з власного класу TrainingsEnumerator
        foreach (var item in trainings)
        {
            Console.WriteLine($"Traning Name {item.Name} and cost {item.Cost}\n");
        }

        Console.ReadKey();
    }
}
```



```
F:\csbc-github\oop-theory-repo\TimeStamp\CustomGeneric
Traning Name C# and cost 10
Traning Name Java and cost 10
```

Загальний погляд на ядро архітектури колекцій



Рекомендації щодо використання інтерфейсів

Інтерфейс	Сценарій використання
IEnumerable, IEnumerable<T>	Єдина потреба – ітерувати по елементах колекції. Потрібний тільки read-only доступ до колекції.
ICollection, ICollection<T>	Бажано змінювати колекцію або важливий розмір колекції.
IList, IList<T>	Потрібні можливості змінювання колекції, важливе впорядкування та/або позиціонування елементів у колекції.
List, List<T>	В ООП рекомендується залежність від абстракцій, а не реалізацій, тому краще не мати членів власних реалізацій з конкретним типом List/List.

- <https://www.monitis.com/blog/how-c-ienumerable-can-kill-your-sites-performance/>



ДЯКУЮ ЗА УВАГУ!

Наступне запитання: Колекції з простору імен `System.Collections.Generic`
