Лекция №2. IEnumerable, IEnumerator, yield return

Снова о циклах

Давайте вспомним, какие циклы есть в языке C# и в каких ситуациях каждый из них нужно использовать:

1. `for` — когда есть явный счётчик.
2. `foreach` — когда нам нужно просто перебрать все элементы в коллекции.
3. `while` — во всех остальных случаях.

Вернёмся к коду нашей самописной очереди с прошлого занятия:

```
public class QueueItem<T>
{
    public T Value { get; set; }
    public QueueItem<T> Next { get; set; }
}

public class Queue<T>
{
    private QueueItem<T> head;
    private QueueItem<T> tail;

    public bool IsEmpty => head == null;

    public void Enqueue(T value)
    {
        var item = new QueueItem<T> { Value = value, Next = null };
        if (IsEmpty)
            tail = head = item;
        else
        {
            tail.Next = item;
            tail = item;
        }
    }

    public T Dequeue()
    {
        if (head == null)
            throw new InvalidOperationException();

        var result = head.Value;
```

```
            head = head.Next;
            if (head == null)
                tail = null;

            return result;
        }
    }
```

Предположим, что нам понадобилось перебрать все элементы в очереди. При текущей реализации это можно сделать только так:

```
// Создадим очередь и добавим в неё элементы
var queue = new Queue<int>();
queue.Enqueue(1);
queue.Enqueue(2);
queue.Enqueue(3);
...

// Достанем все элементы из очереди
while (!queue.IsEmpty)
    Console.WriteLine(queue.Dequeue());
```

У этого кода есть ряд очевидных недостатков. Во-первых, во время перебора элементов мы хотим просто получить все элементы, не изменяя саму коллекцию, а здесь мы достаём элементы из очереди, тем самым изменяя её. Во-вторых, здесь используется цикл `while`, хотя по смыслу больше подходит `foreach`.

IEnumerable и IEnumerator

Мы можем переписать код очереди так, чтобы её элементы можно было перебирать через цикл `foreach`. Для этого нам понадобится реализовать два стандартных интерфейса: `IEnumerable` и `IEnumerator`.

- `IEnumerable` определяет коллекцию, которую можно перебирать (итерировать).
- `IEnumerator` определяет класс (итератор), который будет перебирать элементы из этой коллекции.

Перепишем код очереди:

```
//Наша очередь должна реализовывать интефейс IEnumerable
public class Queue<T> : IEnumerable<T>
{
//Этот метод возвращает Enumerator,
//который будет перебирать элементы в коллекции
    public virtual IEnumerator<T> GetEnumerator()
    {
        return new QueueEnumerator<T>(this);
```

```
    }

//Не будем останавливаться на этом. Просто всегда пишите так.
//Это связано с архитектурными особенностями IEnumerable<T>,
//И требует следующего уровня понимания архитектур
    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
    ...
//Создадим Enumerator для очереди
//Классы в C# можно объявлять внутри других классов
//В этом случае во вложенном классе будут доступны
//Все поля и методы из класса, в который он вложен
    public class QueueEnumerator<T> : IEnumerator<T>
    {
        Queue<T> queue;
        QueueItem<T> currentItem;

        public QueueEnumerator(Queue<T> queue)
        {
            this.queue = queue;
//В самом начале перечислитель не смотрит ни на какой элемент
//Чтобы посмотреть на первый элемент, его нужно сначала переместить.
            this.currentItem = null;
        }

//Свойство Current возвращает текущий элемент
        public T Current { get { return currentItem.Value; } }

//Этот метод вызывается на каждом шаге цикла
        public bool MoveNext()
        {
//Если мы еще никуда не смотрим - посмотреть на голову очереди
            if (currentItem == null)
                currentItem = queue.Head;
//В противном случае, посмотреть на следующий элемент
            else
                currentItem = currentItem.Next;
//Вернуть true, если нам удалось перейти на следующий элемент,
//или false, если не удалось и из foreach нужно выйти
            return currentItem != null;
        }

// Остальные методы нас не волнуют. Просто всегда пишите так.
        public void Dispose() {}
        object IEnumerator.Current { get { return Current; } }
        public void Reset() {}
    }
}
```

Теперь мы можем перебирать элементы через `foreach`:

```
var queue = new Queue<int>();
queue.Enqueue(1);
queue.Enqueue(2);
queue.Enqueue(3);
...

foreach (var item in queue)
    Console.WriteLine(item);
```

yield return

Как вы могли заметить, реализация `IEnumerable` и `IEnumerator` занимает много места. К счастью, есть способ сделать то же самое намного проще через `yield return`:

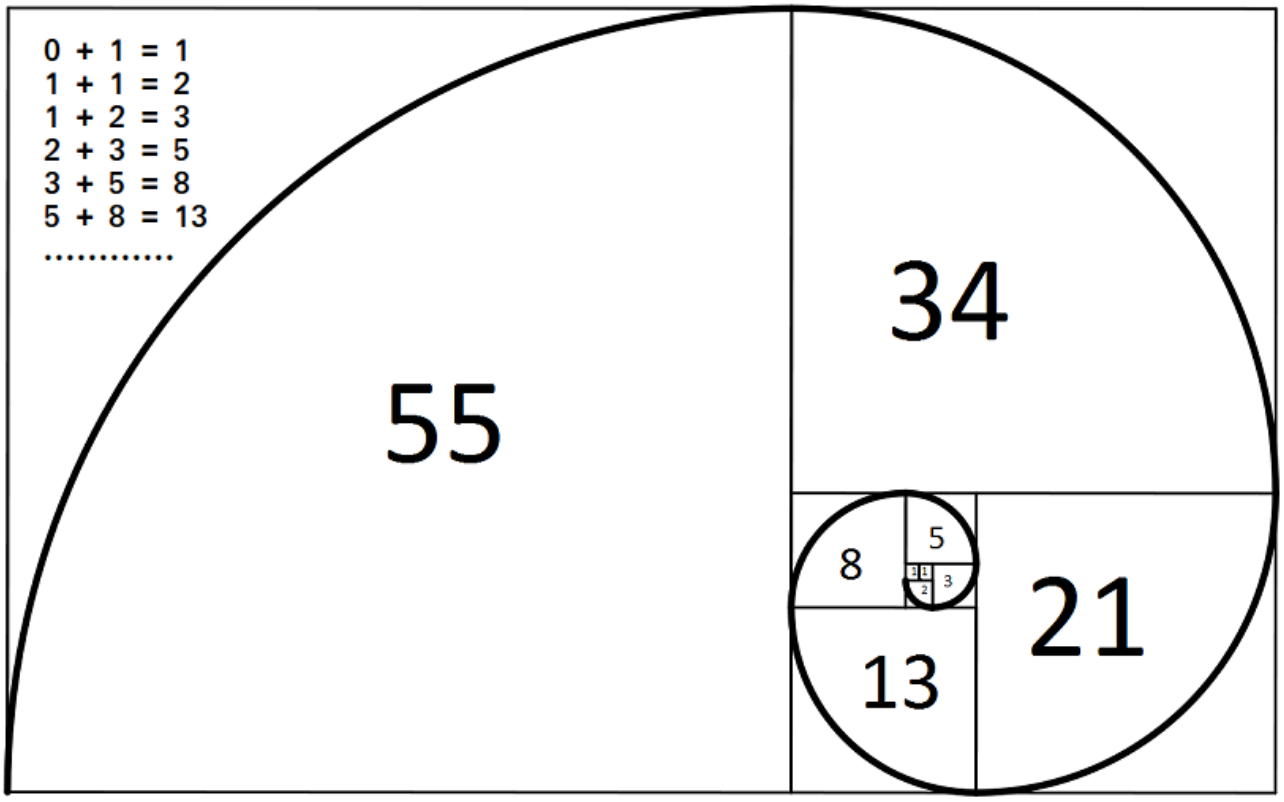
```
public class Queue<T> : IEnumerable<T>
{
    //То же самое, записанное с помощью yield return
    public IEnumerator<T> GetEnumerator()
    {
        var current = Head;
        while (current != null)
        {
            yield return current.Value;
            current = current.Next;
        }
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }

    ...
}
```

Получившийся код намного короче, т.к. нам не понадобилось создавать `QueueEnumerator`. Но как это работает? Если мы запустим этот код через debugger то заметим, что при первом вызове метода, `yield return` не отличается от простого `return`, он также вернет значение и выйдет из метода. Но при следующем вызове, выполнение метода начнется не сначала, а продолжится с `yield return`.

То есть можно представить, что `yield return` как бы на время прерывает выполнение метода, а при следующем вызове продолжает с этого же места. Конечно же никакой метод на самом деле не прерывается, компилятор языка C# неявно переписывает `yield return` через `IEnumerable` и `IEnumerator`, но для простоты можно считать так.

Ленивые коллекции

Есть такая последовательность как числа Фибоначчи, они задаются очень простым правилом: каждое следующее число равно сумме двух предыдущих, а первые два числа это 1 и 1. Вот пример: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ...



Предположим, мы захотим написать программу которая будет последовательно выводить числа Фибоначчи. Самый простой способ - просто создать массив этих чисел и выводить их в цикле, но тогда перед нами встанет проблема - чисел Фибоначчи бесконечно много, а создать бесконечный массив мы не можем. Что делать?

Можно просто воспользоваться `yield return`:

```
public static class Sequences
{
    //Если свойство или метод возвращает IEnumerable<T>,
    //то в нем можно писать yield return,
    //так же в можно использовать несколько yield return
    public static IEnumerable<int> Fibonacci
    {
        get
        {
```

```
            var a = 1;
            var b = 1;
            //Вернет число при первом вызове
            yield return 1;
            //Вернет число при втором вызове
            yield return 1;
            while (true)
            {
                var c = a + b;
                a = b;
                b = c;
            //Вернет все остальные числа
                yield return c;
            }
        }
    }
}
```

Теперь можно использовать `foreach` для вывода:

```
foreach (var item in Sequences.Fibonacci)
    Console.WriteLine(item);
```

Такая коллекция называется **ленивой**, потому что ее элементы изначально не заданы, а генерируются "на лету". Ленивая коллекция позволяет экономить память, т.к. ей не нужно хранить все значения, а также она позволяет создавать бесконечные коллекции, которые невозможно создать обычным способом.

Рекурсивные методы и yield return

В прошлом семестре мы разбирали рекурсивный способ перебрать все возможные подмножества. Давайте вспомним как это работает, у нас есть несколько элементов и нужно перебрать все способы выбрать их, обозначим `1` элемент, который мы достали, и `0` элемент, который мы оставили. Тогда например для множества из 3 элементов будет всего 9 способов выбрать элементы: 000, 001, 010, 011, 100, 101, 110, 111.

```
void MakeSubsets(bool[] subset, int position)
{
//Если дошли до последнего элемента - выводим подможество
    if (position == subset.Length)
    {
        foreach(var e in subset)
            Console.WriteLine(e ? 1 : 0);
        return;
    }
//Не берем текущий элемент и перебираем
//все возможные подмножества оставшихся элементов
```

```
    subset[position] = false;
    MakeSubsets(subset, position + 1)
//Берем текущий элемент и перебираем
//все возможные подмножества оставшихся элементов
    subset[position] = true;
    MakeSubsets(subset, position + 1)
}
```

Этот метод можно переписать так, чтобы он возвращал нам подмножества, как ленивая коллекция. Для этого нам нужно будет воспользоваться операторами `yield return` и `yield break`. Оператор `yield break` прерывает выполнение итератора и указывает, что последовательность больше не имеет элементов, он используется как точка выхода из рекурсии.

```
//Изменяем возвращаемый тип на IEnumerable
IEnumerable<bool[]> MakeSubsets(bool[] subset, int position)
{
    if (position == subset.Length)
    {
        yield return subset;
        yield break;
    }
    subset[position] = false;
    foreach (var e in MakeSubsets(subset, position + 1))
        yield return e;
    subset[position] = true;
    foreach (var e in MakeSubsets(subset, position + 1))
        yield return e;
}
```

Теперь мы можем перебирать все подмножества в цикле `foreach`:

```
foreach (var subset in MakeSubsets(new bool[elementNumber], 0))
{
    foreach (var e in subset)
        Console.Write(e ? 1 : 0);
    Console.WriteLine();
}
```