**Доступ к коллекции с помощью перечислителя**

К элементам коллекции нередко приходится обращаться циклически, например,

для отображения каждого элемента коллекции. С этой целью можно, с одной стороны,

организовать цикл foreach, как было показано в приведенных выше примерах,

а с другой — воспользоваться перечислителем. *Перечислитель* — это объект, который

реализует необобщенный интерфейс IEnumerator или обобщенный интерфейс

IEnumerator<Т>.

В интерфейсе IEnumerator определяется одно свойство, Current, необобщенная

форма которого приведена ниже.

object Current { get; }

А в интерфейсе IEnumerator<T> объявляется следующая обобщенная форма

свойства Current.

Т Current { get; }

В обеих формах свойства Current получается текущий перечисляемый элемент

коллекции. Но поскольку свойство Current доступно только для чтения, то перечислитель

может служить только для извлечения, но не видоизменения объектов в коллекции.

В интерфейсе IEnumerator определяются два метода. Первым из них является метод

MoveNext(), объявляемый следующим образом.

bool MoveNext()

При каждом вызове метода MoveNext() текущее положение перечислителя смещается

к следующему элементу коллекции. Этот метод возвращает логическое значение

true, если следующий элемент коллекции доступен, и логическое значение false,

если достигнут конец коллекции. Перед первым вызовом метода MoveNext() значение

свойства Current оказывается неопределенным. (В принципе до первого вызова метода

MoveNext() перечислитель обращается к несуществующему элементу, который

должен находиться перед первым элементом коллекции. Именно поэтому приходится

вызывать метод MoveNext(), чтобы перейти к первому элементу коллекции.)

Для установки перечислителя в исходное положение, соответствующее началу коллекции,

вызывается приведенный ниже метод Reset().

void Reset()

После вызова метода Reset() перечисление вновь начинается с самого начала коллекции.

Поэтому, прежде чем получить первый элемент коллекции, следует вызвать

метод MoveNext().

В интерфейсе IEnumerator<T> методы MoveNext() и Reset() действуют по тому

же самому принципу.

Необходимо также обратить внимание на два следующих момента. Во-первых,

перечислитель нельзя использовать для изменения содержимого перечисляемой с

его помощью коллекции. Следовательно, перечислители действуют по отношению к

коллекции как к доступной только для чтения. И во-вторых, любое изменение в перечисляемой

коллекции делает перечислитель недействительным.

**Применение обычного перечислителя**

Прежде чем получить доступ к коллекции с помощью перечислителя, необходимо

получить его. В каждом классе коллекции для этой цели предоставляется метод

GetEnumerator(), возвращающий перечислитель в начало коллекции. Используя

этот перечислитель, можно получить доступ к любому элементу коллекции по очереди.

В целом, для циклического обращения к содержимому коллекции с помощью

перечислителя рекомендуется придерживаться приведенной ниже процедуры.

1. Получить перечислитель, устанавливаемый в начало коллекции, вызвав для этой

коллекции метод GetEnumerator().

2. Организовать цикл, в котором вызывается метод MoveNext(). Повторять цикл до

тех пор, пока метод MoveNext() возвращает логическое значение true.

3. Получить в цикле каждый элемент коллекции с помощью свойства Current.

Ниже приведен пример программы, в которой реализуется данная процедура.

В этой программе используется класс ArrayList, но общие принципы циклического

обращения к элементам коллекции с помощью перечислителя остаются неизменными

для коллекций любого типа, в том числе и обобщенных.

(***glava25\_27***)

class EnumeratorDemo

{

static void Main()

{

ArrayList list = new ArrayList(1);

for (int i = 0; i < 10; i++)

list.Add(i);

//use enum to access to list

IEnumerator etr = list.GetEnumerator();

while (etr.MoveNext())

Console.Write(etr.Current + " ");

Console.WriteLine();

//repeat enum list

etr.Reset();

while (etr.MoveNext())

Console.Write(etr.Current + " ");

Console.WriteLine();

}

}

Вообще говоря, для циклического обращения к элементам коллекции цикл

foreach оказывается более удобным, чем перечислитель. Тем не менее перечислитель

предоставляет больше возможностей для управления, поскольку его можно при

желании всегда установить в исходное положение.

**Применение перечислителя типа IdictionaryEnumerator**

Если для организации коллекции в виде словаря, например типа Hashtable,

реализуется необобщенный интерфейс IDictionary, то для циклического обращения

к элементам такой коллекции следует использовать перечислитель типа

IDictionaryEnumerator вместо перечислителя типа IEnumerator. Интерфейс

IDictionaryEnumerator наследует от интерфейса IEnumerator и имеет три дополнительных

свойства. Первым из них является следующее свойство.

DictionaryEntry Entry { get; }

Свойство Entry позволяет получить пару "ключ-значение" из перечислителя в

форме структуры DictionaryEntry. Напомним, что в структуре DictionaryEntry

определяются два свойства, Key и Value, с помощью которых можно получать доступ

к ключу или значению, связанному с элементом коллекции. Ниже приведены два других

свойства, определяемых в интерфейсе IDictionaryEnumerator.

object Key { get; }

object Value { get; }

С помощью этих свойств осуществляется непосредственный доступ к ключу или

значению.

Перечислитель типа IDictionaryEnumerator используется аналогично обычному

перечислителю, за исключением того, что текущее значение в данном случае получается

с помощью свойств Entry, Key или Value, а не свойства Current. Следовательно,

приобретя перечислитель типа IDictionaryEnumerator, необходимо вызвать метод

MoveNext(), чтобы получить первый элемент коллекции. А для получения остальных

ее элементов следует продолжить вызовы метода MoveNext(). Этот метод возвращает

логическое значение false, когда в коллекции больше нет ни одного элемента.

В приведенном ниже примере программы элементы коллекции типа Hashtable

перечисляются с помощью перечислителя типа IDictionaryEnumerator.

(***glava25\_28***)

class IDicEnumDemo

{

static void Main()

{

//create hash table

Hashtable ht = new Hashtable();

//add elements in table

ht.Add("Ken", "555-7756");

ht.Add("Mary", "555-9876");

ht.Add("Tom", "555-3456");

ht.Add("Todd", "555-3452");

//use enumerator

IDictionaryEnumerator etr = ht.GetEnumerator();

Console.WriteLine("Show info with Entry.");

while (etr.MoveNext())

Console.WriteLine(etr.Entry.Key + ": " + etr.Entry.Value);

Console.WriteLine();

Console.WriteLine("Show info with Key and Value.");

etr.Reset();

while (etr.MoveNext())

Console.WriteLine(etr.Key + ": " + etr.Value);

}

}

**Реализация интерфейсов IEnumerable и Ienumerator**

Как упоминалось выше, для циклического обращения к элементам коллекции зачастую

проще (да и лучше) организовать цикл foreach, чем пользоваться непосредственно

методами интерфейса IEnumerator. Тем не менее ясное представление о

принципе действия подобных интерфейсов важно иметь по еще одной причине: если

требуется создать класс, содержащий объекты, перечисляемые в цикле foreach, то в

этом классе следует реализовать интерфейсы IEnumerator и IEnumerable. Иными

словами, для того чтобы обратиться к объекту определяемого пользователем класса в

цикле foreach, необходимо реализовать интерфейсы IEnumerator и IEnumerable

в их обобщенной или необобщенной форме. Правда, сделать это будет нетрудно, поскольку

оба интерфейса не очень велики.

В приведенном ниже примере программы интерфейсы IEnumerator и

IEnumerable реализуются в необобщенной форме, с тем чтобы перечислить содержимое

массива, инкапсулированного в классе MyClass.

(***glava25\_29***)

class MyClass : IEnumerable, IEnumerator

{

char[] chrs = { 'A', 'B', 'C', 'D' };

int idx = -1;

//realize interface IEnumerable

public IEnumerator GetEnumerator()

{

return this;

}

//next method for IEnumerator

//return curent obj

public object Current

{

get

{

return chrs[idx];

}

}

//move to next

public bool MoveNext()

{

if (idx == chrs.Length - 1)

{

Reset(); //set enum at the begin

return false;

}

idx++;

return true;

}

//set enum at the begining

public void Reset() { idx = -1; }

}

class EnumeratorImplDemo

{

static void Main()

{

MyClass me = new MyClass();

//show content of me

foreach (char ch in me)

Console.Write(ch + " ");

Console.WriteLine();

//show again

foreach (char ch in me)

Console.Write(ch + " ");

Console.WriteLine();

}

}

В данной программе сначала создается класс MyClass, в котором инкапсулируется

небольшой массив типа char, состоящий из символов А-D. Индекс этого массива хранится

в переменной idx, инициализируемой значением -1. Затем в классе MyClass реализуются

оба интерфейса, IEnumerator и IEnumerable. Метод GetEnumerator()

возвращает ссылку на перечислитель, которым в данном случае оказывается текущий

объект. Свойство Current возвращает следующий символ в массиве, т.е. объект, указываемый

по индексу idx. Метод MoveNext() перемещает индекс idx в следующее

положение. Этот метод возвращает логическое значение false, если достигнут конец

коллекции, в противном случае — логическое значение true. Напомним, что перечислитель

оказывается неопределенным вплоть до первого вызова метода MoveNext().

Следовательно, метод MoveNext() автоматически вызывается в цикле foreach перед

обращением к свойству Current. Именно поэтому первоначальное значение переменной

idx устанавливается равным -1. Оно становится равным нулю на первом шаге

цикла foreach. Обобщенная реализация рассматриваемых здесь интерфейсов будет

действовать по тому же самому принципу.

Далее в методе Main() создается объект mc типа MyClass, и содержимое этого

объекта дважды отображается в цикле foreach.

**Применение итераторов**

Как следует из предыдущих примеров, реализовать интерфейсы IEnumerator и

IEnumerable нетрудно. Но еще проще воспользоваться *итератором,* который представляет

собой метод, оператор или аксессор, возвращающий по очереди члены совокупности

объектов от ее начала и до конца. Так, если некоторый массив состоит из

пяти элементов, то итератор данного массива возвратит все эти элементы по очереди.

Реализовав итератор, можно обращаться к объектам определяемого пользователем

класса в цикле foreach.

Обратимся сначала к простому примеру итератора. Приведенная ниже программа

является измененной версией предыдущей программы, в которой вместо явной реализации

интерфейсов IEnumerator и IEnumerable применяется итератор.

(***glava25\_30***)

class MyClass

{

char[] chrs = { 'A', 'B', 'C', 'D' };

//this operator returns symbols from chrs array

public IEnumerator GetEnumerator()

{

foreach (var ch in chrs)

yield return ch;

}

}

class ItrDemo

{

static void Main()

{

MyClass me = new MyClass();

//show content of me

foreach (char ch in me)

Console.Write(ch + " ");

Console.WriteLine();

}

}

Рассмотрим эту программу более подробно. Во-первых, обратите внимание на то,

что в классе MyClass не указывается IEnumerator в качестве реализуемого интерфейса.

При создании итератора компилятор реализует этот интерфейс автоматически.

public IEnumerator GetEnumerator()

{

foreach (var ch in chrs)

yield return ch;

}

Это и есть итератор для объектов класса MyClass. Как видите, в нем явно реализуется

метод GetEnumerator(), определенный в интерфейсе IEnumerable. А теперь

перейдем непосредственно к телу данного метода. Оно состоит из цикла foreach,

в котором возвращаются элементы из массива chrs. И делается это с помощью оператора

yield return. Этот оператор возвращает следующий объект в коллекции,

которым в данном случае оказывается очередной символ в массиве chrs. Благодаря

этому средству обращение к объекту mc типа MyClass организуется в цикле foreach

внутри метода Main().

Обозначение yield служит в языке C# в качестве *контекстного ключевого слова.* Это

означает, что оно имеет специальное назначение только в блоке итератора. А вне этого

блока оно может быть использовано аналогично любому другому идентификатору.

Следует особо подчеркнуть, что итератор не обязательно должен опираться на массив

или коллекцию другого типа. Он должен просто возвращать следующий элемент

из совокупности элементов. Это означает, что элементы могут быть построены динамически

с помощью соответствующего алгоритма. В качестве примера ниже приведена

версия предыдущей программы, в которой возвращаются все буквы английского

алфавита, набранные в верхнем регистре. Вместо массива буквы формируются в цикле

for.

(***glava25\_31***)

class MyClass

{

char ch = 'A';

//this operator returns symbols of ABC

//in Upper register

public IEnumerator GetEnumerator()

{

for (int i = 0; i < 26; i++)

yield return (char)(ch + i);

}

}

class ItrDemo

{

static void Main()

{

MyClass me = new MyClass();

//show content of me

foreach (char ch in me)

Console.Write(ch + " ");

Console.WriteLine();

}

}

**Прерывание итератора**

Для преждевременного прерывания итератора служит следующая форма оператора

yield.

yield break;

Когда этот оператор выполняется, итератор уведомляет о том, что достигнут конец

коллекции. А это, по существу, останавливает сам итератор.

Приведенная ниже программа является версией предыдущей программы, измененной

с целью отобразить только первые десять букв английского алфавита.

(***glava25\_32***)

class MyClass

{

char ch = 'A';

//this operator returns symbols of ABC

//in Upper register

public IEnumerator GetEnumerator()

{

for (int i = 0; i < 26; i++)

{

if (i == 10) yield break;

yield return (char)(ch + i);

}

}

}

class ItrDemo

{

static void Main()

{

MyClass me = new MyClass();

//show content of me

foreach (char ch in me)

Console.Write(ch + " ");

Console.WriteLine();

}

}

**Применение нескольких операторов yield**

В итераторе допускается применение нескольких операторов yield. Но каждый

такой оператор должен возвращать следующий элемент в коллекции. В качестве примера

рассмотрим следующую программу.

(***glava25\_33***)

class MyClass

{

//this operator returns letter A, B, C, D, E

public IEnumerator GetEnumerator()

{

yield return 'A';

yield return 'B';

yield return 'C';

yield return 'D';

yield return 'E';

}

}

class ItrDemo5

{

static void Main()

{

MyClass mc = new MyClass();

//show content of me

foreach (char ch in mc)

Console.Write(ch + " ");

Console.WriteLine();

}

}

В данной программе внутри метода GetEnumerator() выполняются пять операторов

yield. Следует особо подчеркнуть, что они выполняются по очереди и каждый

раз, когда из коллекции получается очередной элемент. Таким образом, на каждом

шаге цикла foreach в методе Main() возвращается только один символ.

**Создание именованного итератора**

В приведенных выше примерах был продемонстрирован простейший способ реализации

итератора. Но ему имеется альтернатива в виде именованного итератора.

В данном случае создается метод, оператор или аксессор, возвращающий ссылку на

объект типа IEnumerable. Именно этот объект используется в коде для предоставления

итератора. Именованный итератор представляет собой метод, общая форма которого

приведена ниже:

public IEnumerable имя\_итератора(список\_параметров)

{

// ...

yield return obj;

}

где *имя\_итератора* обозначает конкретное имя метода; *список\_параметров* — от

нуля до нескольких параметров, передаваемых методу итератора; *obj —* следующий

объект, возвращаемый итератором. Как только именованный итератор будет создан,

его можно использовать везде, где он требуется, например для управления циклом

foreach.

Именованные итераторы оказываются весьма полезными в некоторых ситуациях,

поскольку они позволяют передавать аргументы итератору, управляющему процессом

получения конкретных элементов из коллекции. Например, итератору можно

передать начальный и конечный пределы совокупности элементов, возвращаемых из

коллекции итератором. Эту форму итератора можно перегрузить, расширив ее функциональные

возможности. В приведенном ниже примере программы демонстрируются

два способа применения именованного итератора для получения элементов коллекции.

В одном случае элементы перечисляются в заданных начальном и конечном

пределах, а в другом — элементы перечисляются с начала последовательности и до

указанного конечного предела.

(***glava25\_34***)

class MyClass

{

char ch = 'A';

//this iterator returns letter ABC

//start from A until breakline

public IEnumerable MyItr(int end)

{

for (int i = 0; i < end; i++)

yield return (char)(ch + i);

}

//this iterator returns in borders

public IEnumerable MyItr(int begin, int end)

{

for (int i = begin; i < end; i++)

yield return (char)(ch + i);

}

}

class ItrDemo6

{

static void Main()

{

MyClass mc = new MyClass();

//return first 7 letter

Console.Write("First 7 letter: ");

foreach (char ch in mc.MyItr(7))

Console.Write(ch + " ");

Console.WriteLine();

//return from F to L

Console.Write("From F to L: ");

foreach (char ch in mc.MyItr(5, 12))

Console.Write(ch + " ");

Console.WriteLine();

}

}

**Создание обобщенного итератора**

В приведенных выше примерах применялись необобщенные итераторы, но, конечно,

ничто не мешает создать обобщенные итераторы. Для этого достаточно возвратить

объект обобщенного типа IEnumerator<T> или IEnumerable<T>. Ниже приведен

пример создания обобщенного итератора.

(***glava25\_35***)

class MyClass<T>

{

T[] array;

public MyClass(T[] a)

{

array = a;

}

//this iterator return symbols from char

public IEnumerator<T> GetEnumerator()

{

foreach (T obj in array)

yield return obj;

}

}

class GenericItrDemo

{

static void Main()

{

int[] nums = { 4, 3, 6, 4, 7, 9 };

MyClass<int> mc = new MyClass<int>(nums);

foreach (var x in mc)

Console.Write(x + " ");

Console.WriteLine();

bool[] bVals = { true, true, false, true };

MyClass<bool> mc2 = new MyClass<bool>(bVals);

foreach (var x in mc2)

Console.Write(x + " ");

Console.WriteLine();

}

}

В данном примере массив, состоящий из возвращаемых по очереди объектов, передается

конструктору класса MyClass. Тип этого массива указывает в качестве аргумента

типа в конструкторе класса MyClass.

Метод GetEnumerator() оперирует данными обобщенного типа т и возвращает

перечислитель типа IEnumerator<T>. Следовательно, итератор, определенный в

классе MyClass, способен перечислять данные любого типа.

**Инициализаторы коллекций**

В C# имеется специальное средство, называемое *инициализатором коллекции* и упрощающее

инициализацию некоторых коллекций. Вместо того чтобы явно вызывать метод

Add(), при создании коллекции можно указать список инициализаторов. После

этого компилятор организует автоматические вызовы метода Add(), используя значения

из этого списка. Синтаксис в данном случае ничем не отличается от инициализации

массива. Обратимся к следующему примеру, в котором создается коллекция типа

List<char>, инициализируемая символами С, А, Е, В, D и F.

List<char> lst = new List<char>() { 'С', 'А', 'Е', 'В', 'D', 'F' };

После выполнения этого оператора значение свойства lst.Count будет равно 6,

поскольку именно таково число инициализаторов.

Для инициализации коллекции типа LinkedList<TKey, TValue>, в которой хранятся

пары "ключ-значение", инициализаторы приходится предоставлять парами, как

показано ниже.

SortedList<int, string> lst =

new SortedList<int, string>() { { 1, "один" }, { 2, "два" }, { 3, "три" } };

Компилятор передаст каждую группу значений в качестве аргументов методу

Add(). Следовательно, первая пара инициализаторов преобразуется компилятором

в вызов Add(1, "один").

Компилятор вызывает метод Add() автоматически для ввода инициализаторов в

коллекцию, и поэтому инициализаторы коллекций можно использовать только в коллекциях,

поддерживающих открытую реализацию метода Add(). Это означает, что

инициализаторы коллекций нельзя использовать в коллекциях типа Stack, Stack<T>,

Queue или Queue<T>, поскольку в них метод Add() не поддерживается. Их нельзя

применять также в тех коллекциях типа LinkedList<T>, где метод Add() предоставляется

как результат явной реализации соответствующего интерфейса.