# Паттерн Итератор (Iterator)

**Назначение**: представляет доступ ко всем элементам составного объекта, не раскрывая его внутреннего представления.

## Мотивация

Практически любое приложение в той или иной мере работает с коллекциями данных. Мы постоянно используем векторы, списки, деревья и хеш-таблицы. В некоторых случаях для обработки данных коллекции используется специфический интерфейс конкретных коллекций, но в большинстве случаев внутренний доступ осуществляется за счет специального абстрактного слоя – итераторов.

Итераторы предоставляют абстрактный интерфейс для доступа к содержимому составных объектов, не раскрывая клиентам их внутреннюю структуру. В результате получается четкое разделение ответственностей: клиенты получают возможность работать с разными коллекциями унифицированным образом, и классы-коллекций становятся проще за счет того, что ответственность за перебор ее элементов отводится отдельной сущности.

Итераторы настолько укоренились в большинстве языков и платформ, что их поддержка появилась даже на уровне языков программирования: foreach в C#, range-for в C++ 11, for в Java 5+ и даже в консервативном Eiffel появилась аналогичная конструкция – [across](http://bertrandmeyer.com/2010/01/26/more-expressive-loops-for-eiffel/). Более того, многие языки поддерживают не только «потребление итераторов» с помощью циклов foreach и им подобным, но еще и их создание за счет блоков итераторов ([Iterator Block](http://msdn.microsoft.com/en-us/library/dscyy5s0.aspx) в C# и VB) или так называемых конструкторов последовательностей (Sequence Comprehension), доступных в [F#](http://msdn.microsoft.com/en-us/library/dd233209.aspx), [Scala](http://www.scala-lang.org/old/node/111), [Phyton](http://python-3-patterns-idioms-test.readthedocs.org/en/latest/Comprehensions.html) и многих других языках).

Итераторы отлично подходят для чтения данных из некоторого источника. Так, например, вместо класса LogFileReader мы могли бы использовать класс LogFileSource, который бы реализовывал интерфейс IEnumerable<LogEntry>:

public class LogFileSource : IEnumerable<LogEntry>

{

private readonly string \_logFileName;

public LogFileSource(string logFileName)

{

\_logFileName = logFileName;

}

public IEnumerator<LogEntry> GetEnumerator()

{

foreach (var line in File.ReadAllLines(\_logFileName))

{

yield return LogEntry.Parse(line);

}

}

IEnumerator IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

}

Листинг 4.1 – Код класса LogFileSource

## Классическая диаграмма классов паттерна Итератор

Паттерн Итератор – это один из немногих паттернов, который пришел в .NET Framework из книги банды четырех практически в неизменном виде. Если взять исходную диаграмму классов из книги "Design Patterns", заменитьAggregate на IEnumerable<T>, Iterator на IEnumerator<T> и немного изменить методы класса Iterator, то мы получим очень похожую картину:

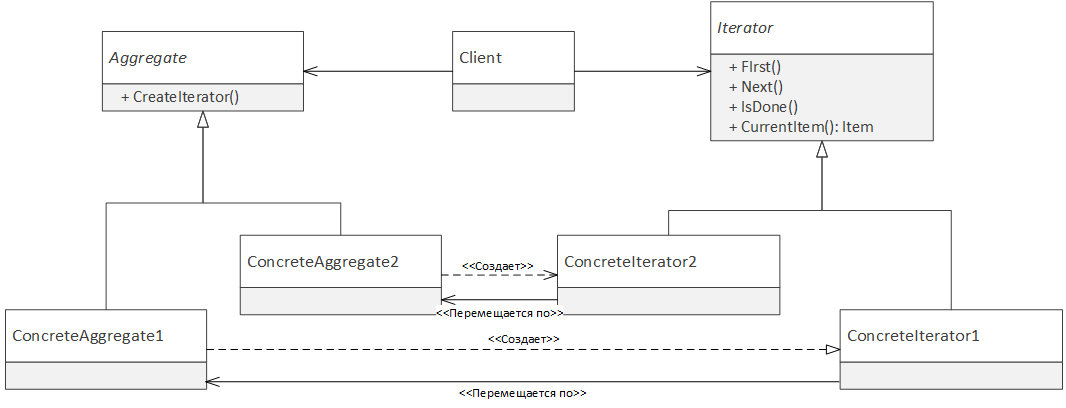


Рисунок 4.1 – Классическая диаграмма паттерна Итератор

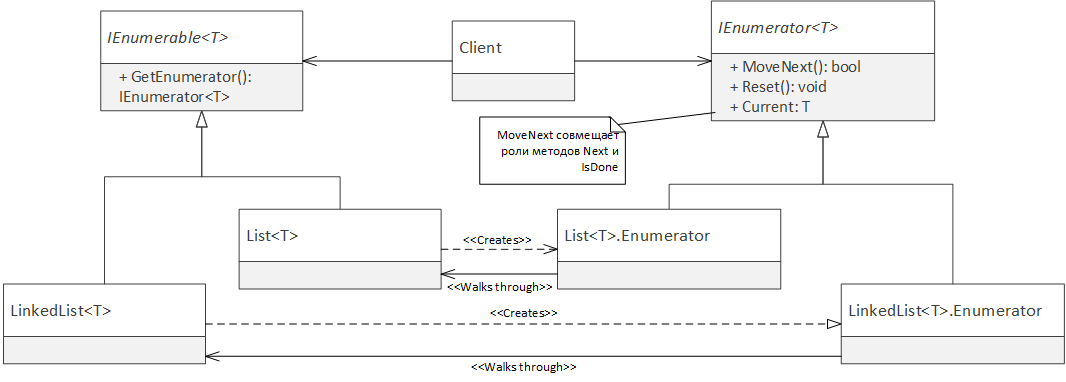


Рисунок 4.2 – Паттерн Итератор в .NET Framework

Участники

* Iterator (IEnumerator<T>) определяет интерфейс итератора.
* Aggregate (IEnumerable<T>) составной объект, по которому может перемещаться итератор.
* ConcreteAggregate1 (List<T>) конкретная реализация агрегата.
* ConcreteIterator (List.Enumerator<T>) конкретная реализация итератора для определенного агрегата.

## Обсуждение

Итераторы в .NET являются *однонаправленными итераторами только для чтения*. При этом для получения итератора используется метод GetEnumerator интерфейса IEnumerable, который каждый раз возвращает новый экземпляр итератора.

Интерфейс IEnumerator также довольно прост:

* MoveNext – переход на следующий элемент агрегата. Возвращает false, если достигнут конец последовательности.
* Current – возвращает текущий элемент.
* Reset – возвращает итератор к началу агрегата. Реализуется не всегда.

Сразу после создания, итератор указывает на - 1-й элемент, поэтому для перехода к первому элементу нужно вызвать MoveNext как минимум один раз.

Итераторы в .NET могут показаться довольно примитивными, особенно по сравнению с итераторами в С++. Стандарт С++ определяет несколько разных типов итераторов: Input <- Forward <- Bidirectional <- RandomAccess. В С++ есть даже OutputIterator, т.е. итератор для вывода данных.

Платформа .NET поддерживает два типа итераторов: необобщенные и обобщенные итераторы. Первые появились с первой версии платформы .NET, а вторые были добавлены со второй версии вместе с [обобщениями](http://msdn.microsoft.com/en-us/library/ms172192(v=vs.110).aspx) (generics) и [блоками итераторов](http://msdn.microsoft.com/en-us/library/dscyy5s0.aspx) (iterator blocks).

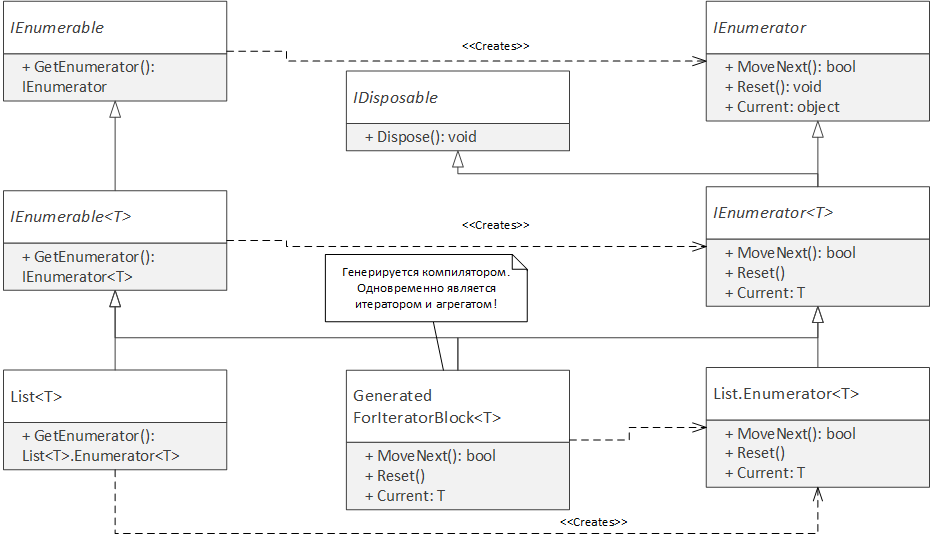


Рисунок 4.3 – Обобщенные и необобщенные итераторы.

### Особенности итераторов в C#/.NET

#### Контракт итераторов

У любого класса или интерфейса есть контракт. Иногда контракт бывает явным, и тогда он описывается с помощью соответствующих инструментов, таких как [Code Contracts](http://research.microsoft.com/en-us/projects/contracts/). Иногда он является неявным, тогда для понимания принципов использования, предусловий, постусловий и инвариантов, нужно обратиться к документации, внимательно прочитать секцию "remarks", а часто, еще и вникнуть в детали реализации.

При установке библиотеки Code Contracts в нашем распоряжении появляются контракты всех типов BCL. Однако встроенные контракты интерфейсов IEnumerable/IEnumerator не слишком хороши, поэтому давайте выведем их самостоятельно.

Нас будут интересовать контракты лишь метода MoveNext и свойства Current:

public T Current

{

get

{

Contract.Requires(!Disposed, "Iterator should not be disposed.");

Contract.Requires(IteratorPointsToCorrectValue,

"MoveNext() should be called and return 'true'.");

Contract.Ensures(true, "Returns current value from the Aggregate.");  
  
 // Возвращаем произвольное значение, чтобы удовлетоврить компилятор

return default(T);

}

}

public bool MoveNext()

{

Contract.Requires(!Disposed, "Iterator should not be disposed.");

Contract.Requires(Valid, "Iterator should be valid.");

// Если итератор еще не дошел до конца агрегата,

// то он будет перемещен на следующий элемент

Contract.Ensures(Finished() ||

InternalIndex == Contract.OldValue(InternalIndex) + 1);

Contract.Ensures(Contract.Result<bool>() ==

(InternalIndex < InternalLength));

return default(bool);

}

Листинг 4.2 – Неформальный контракт итераторов в .NET

**ПРИМЕЧАНИЕ**  
Обращаю внимание, что это не настоящие контракты итератора, это лишь мое представление того, **каким они могли бы быть!** Настоящий контракт итератора, определенный в mscorlib.Contracts.dll не содержит всех этих вещей.

**Контракт свойства Current**

**Предусловие**

* Итератор не должен быть освобожден с помощью вызова Dispose.
* Итератор должен указывать на корректное значение: пользовательский код должен вызвать метод MoveNext, который должен вернуть true.

**Постусловие**

* Свойство Current вернет значение, на которое «указывает» итератор. Это условие нельзя выразить в форме предиката, потому используется Contract.Ensures(true, “”). Итератор не налагает ограничений, вернет ли это свойство null или нет.

**Контракт метода MoveNext**

**Предусловие**

* Итератор не должен быть освобожден с помощью вызова Dispose.
* Итератор должен быть валидным (коллекция не должна быть изменена после получения текущего итератора).

**Постусловие**

* Если итератор не дошел до конца коллекции, то итератор перейдет на следующий элемент и метод вернетtrue, в противном случае, метод вернет false.

Тут есть несколько интересных моментов. Во-первых, предусловие свойства Current слабее предусловия метода MoveNext. На самом деле у свойства Current вообще нет предусловий: мы можем обратиться к свойству Current после вызова Dispose и до вызова MoveNext и не получим исключений! Я же добавил эти требования в контракт, поскольку никто в здравом уме не должен обращаться к свойству Current без выполнения этих условий.

И еще один момент, связанный с методом MoveNext: вам никто не запрещает вызывать MoveNext на завершенном итераторе. В этом случае метод MoveNext просто вернет false! Вот это более валидное требование, поскольку оно позволяет заново проходить по завершенному итератору, а также свободно использовать итератор пустой коллекции (который можно рассматривать как завершенный).

#### Блоки итераторов

Процесс создания итераторов вручную является достаточно утомительным занятием, которое включает управление состоянием, и перемещением по элементам коллекции при вызове MoveNext. С помощью блока итераторов реализовать итератор существенно проще (\*):

public static IEnumerator<int> CustomArrayIterator(this int[] array)

{

foreach (var n in array) { yield return n; }

}

Листинг 4.3 – Пример итератор массива

(\*) Cноска: Пример создания итератора вручную рассмотрен в заметке в моей статье «Итераторы в C#. Часть 1», расположенной по адресу http://sergeyteplyakov.blogspot.com/2010/06/c-1.html.

С помощью блока итераторов можно создавать итераторы для своих коллекций, существующих коллекций или вообще для внешних ресурсов, таких как файлы. Для этого достаточно открыть файл в начале метода и возвращать прочитанные блоки данных с помощью yield return. Кроме того, мы можем генерировать данные бесконечным образом, о чем мы поговорим в разделе «Итераторы и генераторы».

**ПРИМЕЧАНИЕ**  
Теперь должно быть понятно, почему метод Reset в контракте итератора является необязательным. Представьте себе, что итератор возвращает данные, пришедшие по сети. Как в этом случае мы сможем реализовать метод Reset?

Блок итераторов преобразуется компилятором языка C# в конечный автомат с несколькими состояниями, соответствующие начальному положению итератора (когда он указывает на -1-й элемент), конечному положению (когда итератор прошел все элементы) и «среднему» положению, при котором он указывает на определенный элемент. При этом блок итераторов представляет собой некую форму сопрограмм (corouting) (\*), которые продолжают исполнение с предыдущего места благодаря методу MoveNext.

(\*) Подробнее о сопрограммах можно прочитать в Сети. Например: https://ru.wikipedia.org/wiki/Сопрограмма

**«Ленивость» итераторов**

Итераторы, полученные с помощью блока итераторов являются ленивыми: их тело исполняется не в момент вызова метода, а при переборе элементов с помощью метода MoveNext. Это приводит к некоторым особенностям обработки ошибок, ведь даже валидация аргументов метода, возвращающего итератор, будет производиться уже в момент «потребления» итератора.

public static IEnumerable<string> ReadFromFile(string path)

{

if (path == null) throw new ArgumentNullException("path");

foreach(string line in File.ReadLines(path))

{

yield return line;

}

}

// Где будет ошибка?

var result = ReadFromFile(null); //1

foreach (var l in result)

{

Console.WriteLine(l); //2

}

Листинг 4.4 – Пример блока итераторов

На этом же принципе построена большая часть методов LINQ (Language Integrated Query), что позволяет получать сложные запросы без лишних накладных расходов.

Подробнее об итераторах в языке C#, а также о деталях реализации блоков итераторов смотрите в статьях: «Итераторы в C#», доступной по адресу <http://bit.ly/IteratorsInCSharp>. Подробнее об предусловиях в блоке итераторов и асинхронных методах можно прочитать в моей статье «[Когда предусловия не являются предусловиями](http://sergeyteplyakov.blogspot.com/2014/11/when-preconditions-are-not-preconditions.html)».

**Использование итераторов в цикле foreach**

Цикл foreach является универсальным инструментом для обработки коллекций/последовательностей. Способ его преобразования компилятором зависит от типа перебираемой коллекции (обобщенная/необобщенная) и представляет простой цикл while. Пример обхода необобщенной коллекции выглядит таким образом:

public static void ForEachIEnumerable(IEnumerable sequence)

{

// foreach(var e in sequence) {Console.WriteLine(e);}

IEnumerator enumerator = sequence.GetEnumerator();

object current = null;

try

{

while (enumerator.MoveNext())

{

current = enumerator.Current;

Console.WriteLine(current);

}

}

finally

{

IDisposable disposable = enumerator as IDisposable;

if (disposable != null)

{

disposable.Dispose();

}

}

}

Листинг 4.5 – Внутренняя реализация цикла foreach

**ПРИМЕЧАНИЕ**  
Для поддержки цикла foreach не обязательно наличие интерфейса IEnumerable/IEnumerable<T>. Достаточно, чтобы класс коллекции содержал метод GetEnumerator, который будет возвращать тип, с методом bool MoveNext() и свойством Current.  
Подробнее об этом можно почитать в статье – [Duck typing или так ли прост foreach](http://sergeyteplyakov.blogspot.com/2012/08/duck-typing-foreach.html).   
Также стоит обратить внимание, что реализация блока foreach изменилась в C# 5.0, начиная с которого переменная "current" внесена во внутреннюю область видимости. Подробности в статье: [Замыкания на переменных цикла в C# 5.0](http://sergeyteplyakov.blogspot.com/2012/04/c-50.html)

Любой типизированный итератор реализует интерфейс IDisposable, поскольку сам интерфейс IEnumerator<T>наследует IDisposable. Причина этого в том, что итераторы, полученные с помощью блока итераторов легко могут содержать ресурсы, которые освобождаются в блоке finally, вызов которого как раз и осуществляется путем вызова Dispose итератора. Но дело все в том, что блок итераторов может возвращать не только типизированный итератор, но и его предшественников: IEnumerable/IEnumerator, которые не реализуют интерфейс IDisposable.

#### Итераторы или генераторы

Блок итераторов может использоваться для создания итераторов, т.е. для обхода некоторого агрегата в памяти (коллекции) или за ее пределами (итератор содержимого файла). Но помимо этого блок итераторов может использоваться для создания генераторов.

Вот пример простого бесконечного генератора чисел Фибоначчи:

public static IEnumerable<int> GenerateFibonacci()

{

int prev = 0;

int current = 1;

while (true)

{

yield return current;

int tmp = current;

current = prev + current;

prev = tmp;

}

}

Листинг 4.6 – Использование блока итераторов для генерации бесконечной последовательности

В этом плане, блок итераторов в C# напоминает более общие концепции из других языков программирования под названием «списковое включение» (list comprehension), которые предназначены для создания последовательностей и коллекций.

#### Валидность итераторов

В некоторых языках, таких как С++, понятие инвалидации итераторов (когда итератор коллекции становится недействительным) определено в спецификации языка, в разделе, посвященном конкретной коллекции. Так, например, не для всех контейнеров операция добавления элемента делает итератор недействительным: добавление элемента в двусвязный список вполне допустима, а добавление элемента в вектор – нет.

Подобные правила, хотя и не столь формальные, существуют и для коллекций .NET Framework. Точнее, есть лишь одно правило и оно не привязано к конкретному типу коллекции: **при изменении коллекции все ранее полученные итераторы становятся недействительными**. Так, в обоих случаях ниже будет сгенерировано InvalidOperationException:

var list = new List<int> { 42, 12 };

var listIter = list.GetEnumerator();

listIter.MoveNext();

list.RemoveAt(1); // Удаляем 2-й элемент

Console.WriteLine(listIter.Current); // Ok

listIter.MoveNext(); // InvalidOperationException

var linked = new LinkedList<int>();

linked.AddLast(42);

var linkedIter = linked.GetEnumerator();

linkedIter.MoveNext();

linked.AddLast(12);

Console.WriteLine(linkedIter.Current); // Ok

linkedIter.MoveNext(); // InvalidOperationException

Листниг 4.7 – Примеры инвалидации итераторов в .NET

Это поведение коренным образом отличается от правил коллекций языка С++, поскольку в случае std::list обе приведенные операции были бы допустимыми.

#### Итераторы и структуры

Итераторы всех коллекций .NET Framework являются изменяемыми структурами. Это избавляет от дополнительного выделения памяти в управляемой куче при проходе по коллекции, а с другой стороны, может привести к неожиданному результату. Попробуйте предугадать поведение следующего кода, а потом запустите его, чтобы проверить свою догадку:

var x = new {Items = new List<int> {1, 2, 3}.GetEnumerator()};

while (x.Items.MoveNext())

{

Console.WriteLine(x.Items);

}

Листинг 4.8 – Неочевидное поведение итераторов

Но несмотря на потенциальную опасность, итераторы любой широко используемой коллекции должен быть структурой. Более того, в некоторых случаях есть правила, запрещающие использовать коллекции с классами-итераторами. Хорошим примером является правило участия в проекте Roslyn, которое запрещает использовать классы-итераторы в критических участках кода! ([Roslyn. How to Contribute](http://roslyn.codeplex.com/wikipage?title=How%20to%20Contribute), раздел CodingConventions)

**ПРИМЕЧАНИЕ**  
Подробнее о проблемах с изменяемыми значимыми типами читайте в заметках: ["О вреде изменяемых значимых типов"](http://sergeyteplyakov.blogspot.com/2011/07/blog-post.html) и ["О вреде изменяемых значимых типов. Часть 2"](http://sergeyteplyakov.blogspot.com/2012/12/2.html), а еще один пример проблемы изменяемых итераторов рассмотрен в заметке: ["Observable.Generate и перечисление списков"](http://sergeyteplyakov.blogspot.com/2011/11/observablegenerate.html).

### Push-based итераторы

Мало кто обратил внимание, что в книге Банды четырех определены два вида итераторов: *внешний* и *внутренний*, в зависимости от того, кто управляет итерацией – клиент или сам итератор.

*Внешний итератор* – это классический (pull-based) итератор, когда процессом обхода явно управляет клиент путем вызова метода Next или ему *подобного*. *Внутренний итератор* – это push-based итератор, которому передается метод обратного вызова и он сам уведомляет клиента о «посещении» следующего элемента.

*Несложно* догадаться, что ранее мы рассмотрели *внешний* итератор, а *внутренний* итератор в .NET представлен библиотекой [Reactive Extensions](https://rx.codeplex.com/) и парой интерфейсов: IObserver<T>/IObservable<T>. Да, эта пара интерфейсов больше напоминают наблюдатель, а не итератор, но пример все расставит по местам:

var list = new List<int> {1, 2, 3};

IObservable<int> observable = list.ToObservable();

observable.Subscribe(

onNext: n => Console.WriteLine("Processing: {0}", n),

onCompleted: () => Console.WriteLine("Sequece finished"));

Листинг 4.9 – Пример использования IObservable

*Данный* пример не имеет особого смысла, но, например, преобразование в «наблюдаемую» коллекцию объекта SqlDataReader, который также реализует IEnumerable вполне имело бы смысл.

**ПРИМЕЧАНИЕ**  
Подробнее познакомиться с «реактивными расширениями» можно в серии статей Ли Кэмпбелла (LeeCampbell) – [Introduction to Rx](http://www.introtorx.com/content/v1.0.10621.0/00_Foreword.html). В контексте задачи импорта логов, реактивные последовательности будут рассмотрены в следующей главе, при рассмотрении паттерна Наблюдатель.

## Применимость

По своему определению, итератор применяется для доступа к содержимому составных объектов, типичным примером которых являются коллекции. Но стоит ли делать «итерируемыми» бизнес объекты?

Какой подход более разумный: использовать LogFileReader, с методом IEnumerable<LogEntry Read(), или использовать класс LogFileSource, который будет реализовывать IEnumerable<LogEntry>?

Для меня первый вариант является более предпочтительным, поскольку более четко отражает производимые действия. LogFileSource прячет информацию о том, что происходит чтение записей из файла, и то, когда выполняется это действие. Происходит ли чтение файла в конструкторе? Проверяется ли наличие файла в конструкторе, а чтение выполняется при первом вызове метода MoveNext? Всех этих вопросов можно избежать при использовании класса LogFileReader.

**ПРИМЕЧАНИЕ**  
Недавно Эрик Липперт дал похожий совет на StackOverflow.com в ответе на вопрос: ["Why not inherit from List"](http://stackoverflow.com/questions/21692193/why-not-inherit-from-listt/21694054#21694054), поясняя, должен ли класс FootballTeam наследовать от List<Player>. Футбольная команда НЕ ЯВЛЯЕТСЯ списком игроков, поэтому класс FootballTeam не должен наследовать List<Player>. В этом случае гораздо лучше подходит отношение ИМЕЕТ, а значит команда должна содержать список игроков.

## Примеры в .NET Framework

В .NET Framework итераторы представлены парами интерфейсов:

* IEnumerable/IEnumerator для работы с необобщенными коллекциями (составными объектами);
* IEnumerable<T>/IEnumerator<T> для работы с обобщенными коллекциями (составными объектами)
* IObservable<T>/IObserver<T> для работы с «реактивными» (или push-based) коллекциями.

Использование итераторов в языке C# осуществляется с помощью цикла foreach, а создание итераторов упрощается за счет блоков итераторов.