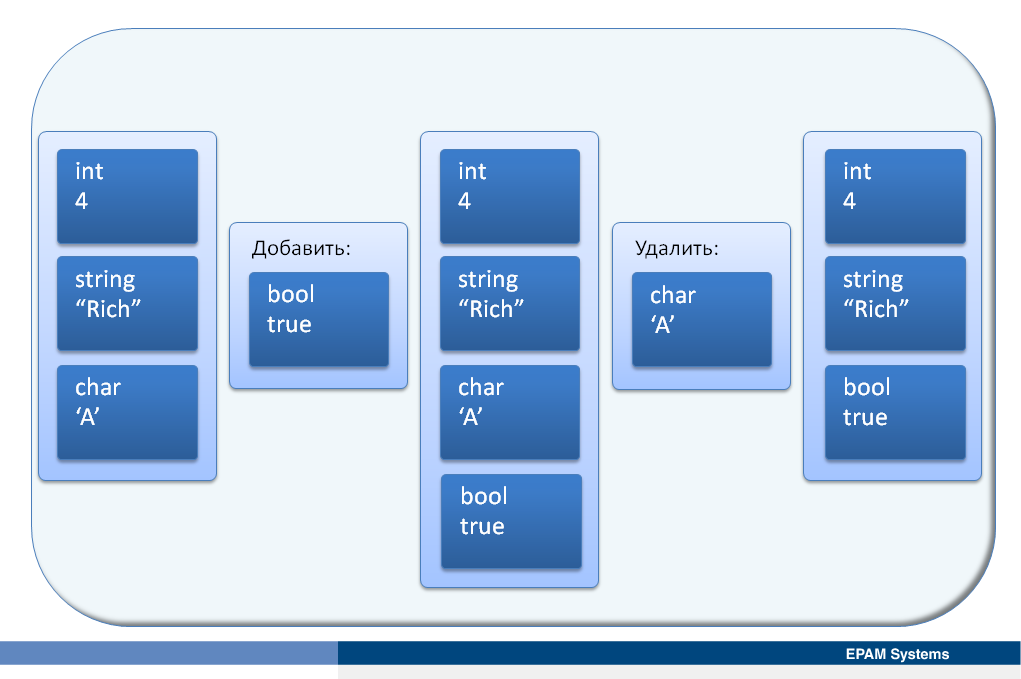
**Использование коллекций и generics**

Понятие коллекции



Коллекцией является тип, агрегирующий объекты. Коллекция ведет себя как контейнер для набора объектов. Можно создавать экземпляр класса коллекции и добавлять объекты в коллекцию. Можно получить доступ к этим элементам с помощью методов, предоставляемых классом коллекцией. Классы коллекции могут показаться похожим на массивы, но, по сути, класс коллекция служит другой цели, отличной от массива, и в большинстве случаев является гораздо более гибкой.

При определении массива необходимо указать его размер и тип хранимых в нем данных. Массив типо-безопасен, однако имеет ряд ограничения, в частности, при определении массива нужно указать количество элементов, которые он может содержать. Часто это значение не возможно определить заранее. При этом, если указать слишком большой размер массива, будет использоваться слишком много памяти, если указать слишком маленький – памяти может не хватить. Массивы работают хорошо, когда точно известно, сколько значений необходимо в нем хранить.

Коллекции гораздо более гибкие. При создании экземпляра класса коллекции не нужно указывать размер коллекции. Коллекция растет и уменьшается динамично в зависимости от объема хранимых в ней данных. Такое динамическое управление пространством делает коллекции мощными по сравнению с массивами, необходимость изменения размера которых влечет за собой написание множества дополнительного кода.

При использовании массива для хранения объектов обязательно явно указывается тип его данных, например, массив целых чисел или массив строк. При этом, операции сохранения значений в массиве или их извлечения из него являются типобезопасными. При попытке сохранить или извлечь данные неправильного типа компилятор обнаруживает проблему и выполнение завершается с ошибкой.

При использовании коллекции тип данных для хранения не указывается. Классы коллекций хранят ссылки на другие объекты с помощью типа System.Object. Эта возможность позволяет строить коллекции, хранящие смешанные типы. Например, можно хранить объекты типа string и значения типа int в одной и той же коллекции. При извлечении элемента из коллекции нужно привести элемент к соответствующему типу.

При определении массива можно указать более чем одну размерность. Коллекции не имеют размерностей. Однако, можно имитировать многомерную коллекцию с помощью хранения коллекции в коллекции.

Обсуждение существующих коллекций.



.NET Framework определяет набор базовых классов коллекций в пространстве имен System.Collections. Большинство коллекций имеют некоторое сходство, все они реализуют ряд интерфейсов. В следующей таблице перечислены основные интерфейсы, поддерживаемые классами System.Collections и их назначение.

|  |  |
| --- | --- |
| **Интерфейс** | **Назначение** |
| **ICollection** | Определяет общие характеристики (т.е. размер, перечисление и безопастность к потокам) всех необобщенных типов коллекций |
| **ICloneable** | Позволяет реализующему объекту возвращать копию самого себя вызывающему коду |
| **IDictionary** | Позволяет объекту необобщенной коллекции представлять свое содержимое в виде пары «имя/значение» |
| **IEnumerable** | Возвращает объект, реализующий интерфейс IEnumerator |
| **IEnumerator** | Позволяет итерацию в стиле foreach по элементам коллекции |
| **IList** | Обеспечивает поведение добавления, удаления и индексирования элементов в списке объектов |
| **IComparer** | Определяет метод Compare() для сравнения объектов, хранящихся в коллекции |
| **IDictionaryEnumerator** | Определяет перечислитель для коллекции, реализующей интерфейс IDictionary |

Все коллекции имплементируют интерфейс IСollection, определяющий небольшое число методов и свойств, в том числе:

* CopyTo. Этот метод позволяет копировать содержимое коллекции в массив.
* GetEnumerator. Этот метод возвращает объект, называемый Enumerator, используемый для перебора элементов коллекции.
* Count. Это свойство, которое показывает текущее количество элементов в коллекции.

Некоторые коллекции реализуют интерфейс IList, который определяет членов, позволяющих получить доступ к элементам в коллекции с использованием массиво-подобной нотации, в дополнение к добавлению и удалению членов, с использованием методов Add и Remove.

В дополнение к членам, которые требуют интерфейсы ICollection и IList, большинство классов коллекций предоставляют конкретные методы и члены, лежащие в основе их функциональности. Названия для этих методов, как правило, тесно связаны с фукциональностью коллекции. Например, класс Queue (очередь) предоставляет методы Enqueue и Dequeue для добавления и удаления элементов из очереди FIFO (first-in, first-out) образом. Класс Stack (стек) предоставляет методы Push и Pop, позволяющие добавлять элементы FILO (first-in, last-out) образом.

Классы коллекции пространства имен System.Collections хранят объекты System.Object, а не объекты конкретного типа[[1]](#footnote-2). В класс коллекцию можно добавить объект любого типа, но при извлечения объекта из коллекции, необходимо привести его к коректному типу, прежде чем можно будет использовать все члены, которые он предоставляетт. При попытке привести объект коллекци к неправильному типу, приложение будет генерировать исключение InvalidCastException.

В следующем примере показано использование класса ArrayList[[2]](#footnote-3). Класс ArrayList реализует интерфейс IList. Он обеспечивает метод Add, который можно использовать для добавления объекта в конец коллекции, и метод RemoveAt, используемый для удаления элемента из коллекции в указанной позиции. Можно также использовать метод Remove для поиска и удаления первого вхождения заданного элемента коллекции. По мере добавления элементов коллекция растет автоматически.

// Create a new ArrayList object.

ArrayList list = new ArrayList();

// Add items to the ArrayList collection.

list.Add(3);

list.Add(4);

list.Add(6);

// Add a string to the ArrayList collection.

// The ArrayList stores objects, not specific

// types so you can add any type to the collection.

list.Add("String Object");

// Remove an object from the ArrayList collection

// by specifying the object to remove.

list.Remove(6);

// Remove an object from the ArrayList collection

// by specifying the index from which to remove the item.

list.RemoveAt(1);

// Use an indexer to access a specific item in the

// collection. Cast the object to its correct type.

int temp = ((int)list[0]) \* 5;

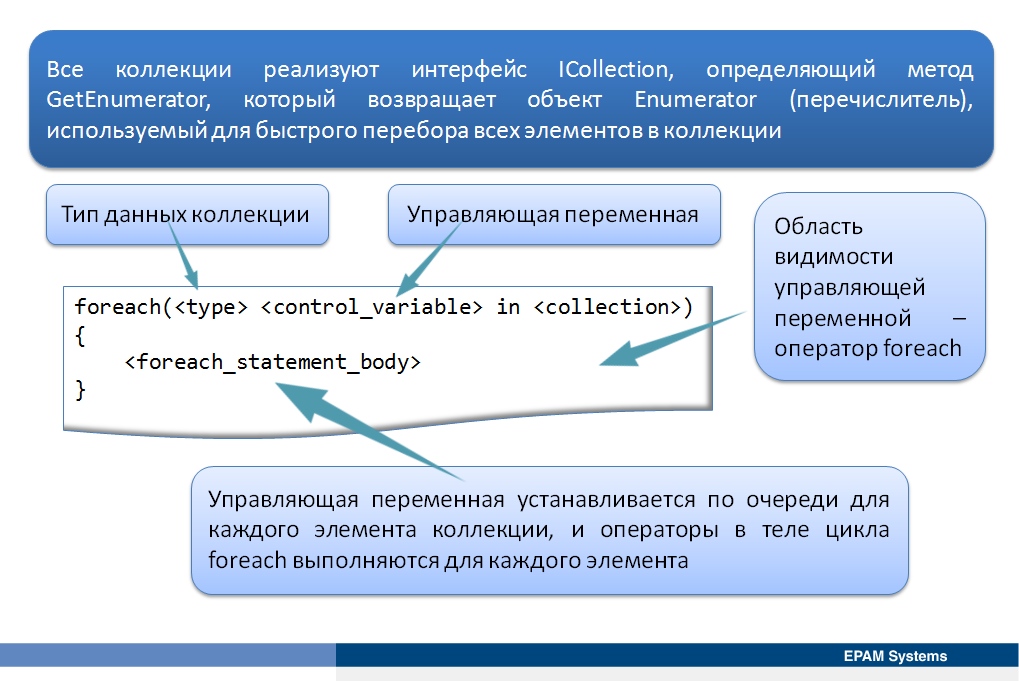
# Выбор класса коллекции

Внимательно относитесь к выбору класса коллекции. Неправильный тип может ограничить возможности использования коллекции.

Ответьте на следующие вопросы.

* Нужен ли вам последовательный список, элемент которого обычно удаляется сразу после извлечения его значения?
  + Если да, то рассмотрите возможность использования класса [Queue](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.queue) или универсального класса [Queue<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.queue-1), если требуется обработка по принципу "первым поступил — первым обслужен" (FIFO). Рассмотрите возможность использования класса [Stack](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.stack) или универсального класса [Stack<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.stack-1), если требуется обработка по принципу "последним поступил — первым обслужен" (LIFO). Для безопасного доступа из нескольких потоков используйте параллельные версии классов [ConcurrentQueue<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.concurrent.concurrentqueue-1) и [ConcurrentStack<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.concurrent.concurrentstack-1).
  + Если нет, то следует использовать другие коллекции.
* Нужен ли доступ к элементам в определенном порядке (FIFO, LIFO) или в произвольным порядке?
  + Класс [Queue](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.queue) и универсальный класс [Queue<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.queue-1) или [ConcurrentQueue<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.concurrent.concurrentqueue-1) предоставляют доступ в порядке FIFO. Дополнительные сведения см. в разделе [Преимущества использования потокобезопасных коллекций](https://docs.microsoft.com/ru-ru/dotnet/standard/collections/thread-safe/when-to-use-a-thread-safe-collection).
  + Класс [Stack](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.stack) и универсальный класс [Stack<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.stack-1) или [ConcurrentStack<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.concurrent.concurrentstack-1) предоставляют доступ в порядке LIFO. Дополнительные сведения см. в разделе [Преимущества использования потокобезопасных коллекций](https://docs.microsoft.com/ru-ru/dotnet/standard/collections/thread-safe/when-to-use-a-thread-safe-collection).
  + Универсальный класс [LinkedList<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.linkedlist-1) предоставляет последовательный доступ от начала к концу списка или наоборот.
* Требуется ли доступ к каждому элементу по индексу?
  + Классы [ArrayList](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.arraylist) и [StringCollection](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.specialized.stringcollection) и универсальный класс [List<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.list-1) предоставляют доступ к элементам по отсчитываемому от нуля индексу элемента.
  + Классы [Hashtable](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.hashtable), [SortedList](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.sortedlist), [ListDictionary](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.specialized.listdictionary) и [StringDictionary](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.specialized.stringdictionary) и универсальные классы [Dictionary<TKey,TValue>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.dictionary-2) и [SortedDictionary<TKey,TValue>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.sorteddictionary-2) предоставляют доступ к элементам по ключу элемента.
  + Классы [NameObjectCollectionBase](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.specialized.nameobjectcollectionbase) и [NameValueCollection](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.specialized.namevaluecollection) и универсальные классы [KeyedCollection<TKey,TItem>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.objectmodel.keyedcollection-2) и [SortedList<TKey,TValue>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.sortedlist-2) предоставляют доступ к элементам по отсчитываемому от нуля индексу или по ключу элемента.
* Будет ли каждый элемент содержать только одно значение, сочетание из одного ключа и одного значения или сочетание из одного ключа и нескольких значений?
  + Одно значение: используйте любую коллекцию, основанную на интерфейсе [IList](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.ilist) или на универсальном интерфейсе [IList<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.ilist-1).
  + Один ключ и одно значение: используйте любую коллекцию, основанную на интерфейсе [IDictionary](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.idictionary) или на универсальном интерфейсе [IDictionary<TKey,TValue>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.idictionary-2).
  + Одно значение с внедренным ключом: используйте универсальный класс [KeyedCollection<TKey,TItem>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.objectmodel.keyedcollection-2).
  + Один ключ и несколько значений: используйте класс [NameValueCollection](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.specialized.namevaluecollection).
* Требуется ли сортировать элементы в порядке, отличном от порядка их поступления?
  + Класс [Hashtable](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.hashtable) сортирует элементы по хэш-коду.
  + В классе [SortedList](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.sortedlist) и универсальных классах [SortedList<TKey,TValue>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.sortedlist-2) и [SortedDictionary<TKey,TValue>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.sorteddictionary-2) элементы сортируются по ключу. Порядок сортировки зависит от реализации интерфейса [IComparer](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.icomparer) для класса [SortedList](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.sortedlist) и от реализации универсального интерфейса [IComparer<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.icomparer-1) для универсальных классов [SortedList<TKey,TValue>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.sortedlist-2) и [SortedDictionary<TKey,TValue>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.sorteddictionary-2). Из двух универсальных типов [SortedDictionary<TKey,TValue>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.sorteddictionary-2) обеспечивает более высокую производительность, чем [SortedList<TKey,TValue>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.sortedlist-2), но [SortedList<TKey,TValue>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.sortedlist-2) потребляет меньше памяти.
  + Класс [ArrayList](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.arraylist) предоставляет метод [Sort](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.arraylist.sort), который принимает реализацию [IComparer](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.icomparer) в качестве параметра. Его универсальный аналог, универсальный класс [List<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.list-1), предоставляет метод [Sort](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.list-1.sort), который принимает реализацию универсального интерфейса [IComparer<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.icomparer-1) в качестве параметра.
* Требуются ли быстрый поиск и получение данных?
  + [ListDictionary](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.specialized.listdictionary) быстрее, чем [Hashtable](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.hashtable), для небольших коллекций (10 элементов или меньше). Универсальный класс [Dictionary<TKey,TValue>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.dictionary-2) обеспечивает более быстрый поиск, чем универсальный класс [SortedDictionary<TKey,TValue>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.sorteddictionary-2). Многопоточной реализацией является [ConcurrentDictionary<TKey,TValue>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.concurrent.concurrentdictionary-2). [ConcurrentBag<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.concurrent.concurrentbag-1) обеспечивает быструю многопоточную вставку для неупорядоченных данных. Дополнительные сведения об обоих многопоточных типах см. в разделе [Преимущества использования потокобезопасных коллекций](https://docs.microsoft.com/ru-ru/dotnet/standard/collections/thread-safe/when-to-use-a-thread-safe-collection).
* Требуются ли вам коллекции, принимающие только строки?
  + Классы [StringCollection](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.specialized.stringcollection) (на основе [IList](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.ilist)) и [StringDictionary](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.specialized.stringdictionary) (на основе [IDictionary](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.idictionary)) находятся в пространстве имен [System.Collections.Specialized](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.specialized).
  + Кроме того, можно использовать любой из универсальных классов коллекций в пространстве имен [System.Collections.Generic](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic) как строго типизированную строковую коллекцию, указав класс [String](https://docs.microsoft.com/ru-ru/dotnet/api/system.string) в качестве аргумента универсального типа. Например, можно объявить переменную с типом [List<String>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.list-1) или [Dictionary<String,String>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.dictionary-2).

Итерация по коллекции



Все коллекции реализуют интерфейс ICollection, определяющий метод GetEnumerator. Этот метод возвращает объект, называемый Enumerator (перечислитель) и используемый для быстрого перебора всех элементов в коллекции. Перечислитель можно рассматривать как подвижный указатель на любой элемент в коллекции. Для передвижения по коллекции C# предоставляет оператор foreach, который автоматически получает перечислитель коллекции и использует его для извлечения по очереди каждого элемента коллекции.

ArrayList list = new ArrayList();

list.Add(99);

list.Add(10001);

list.Add(25);

//...

foreach (int i in list)

{

Console.WriteLine(i);

}

В приведенном выше примере создается и заполняется объект ArrayList, содержащий коллекцию целых значений. Оператор foreach отображает каждое из значений коллекции по очереди в порядке, в котором они встречаются в объекте ArrayList. Синтаксис оператора foreach имеет следующий вид.

foreach(<type> <control\_variable> in <collection>)

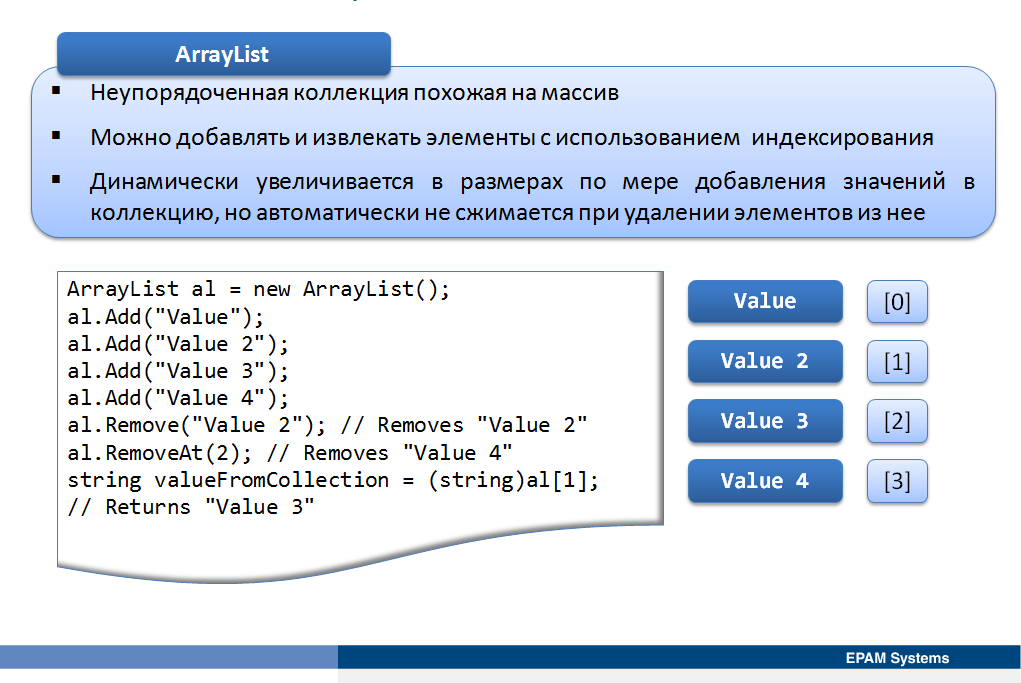
{

<foreach\_statement\_body>

}

В конструкции foreach определяется управляющая переменная и указывается тип данных коллекции. Управляющая переменная устанавливается по очереди для каждого элемента коллекции, и операторы в теле цикла foreach выполняются для каждого элемента. Область видимости управляющей переменной – оператор foreach. Для управляющей переменной важно указать тот же тип, что и тип данных в коллекции; компилятор автоматически генерирует код для приведения к этому типу данных, извлекаемых из коллекции,. Если для управляющей переменной указать неправильный тип, код будет генерировать исключение времени выполнения InvalidCastException.

Базовые классы коллекции



Пространство имен System.Collections содержит несколько классов коллекций общего назначения. Каждый из этих классов коллекций оптимизирован для реализации конкретного механизма для объединения и доступа к данным.

**Класс коллекция ArrayList.** Класс коллекция ArrayList похож на массив. В массив можно добавлять и извлекать элементы с использованием индексирования с нуля. Класс ArrayList динамически увеличивается в размерах по мере добавления значений в коллекцию. Чтобы получить или установить текущий размер коллекции можно использовать свойство Capacity. Класс ArrayList автоматически не сжимается при удалении элементов из коллекции. При удалении значительного числа элементов из коллекции можно использовать метод TrimToSize для уменьшения размера коллекции, или установить свойство Capacity в меньшее значение. В следующем примере показано простое использование класса ArrayList.

// Create a new ArrayList object.ё

ArrayList al = new ArrayList();

// Add values to the ArrayList collection.

al.Add("Value");

al.Add("Value 2");

al.Add("Value 3");

al.Add("Value 4");

// Remove a specific object from the ArrayList collection.

al.Remove("Value 2"); // Removes "Value 2"

// Remove an object from a specified index.

al.RemoveAt(2); // Removes "Value 4"

// Retrieve an object from a specified index.

string valueFromCollection = (string)al[1];

// Returns "Value 3"

**Класс коллекция Queue.** Класс Queue это FIFO структура данных. Вместо того предоставления методов Add и Remove, класс Queue обеспечивает методы Enqueue и Dequeue. При использовании для объекта метода Enqueue он автоматически добавляется в конец коллекции; при для объекта использовании метода Dequeue, он автоматически удаляется из начала коллекции. Для получения первого элемента очереди, при этом не удаляя его, используется метод Peek. Класс Queue растет автоматически при добавлении объектов в коллекцию. Для восстановления памяти для объекта Queue,за счет уменьшения размера коллекции можно использовать метод TrimToSize. В следующем примере показано использование класса Queue.

// Create a new Queue object.

Queue queue = new Queue();

// Add values to the Queue collection.

queue.Enqueue("Value");

queue.Enqueue("Value 2");

queue.Enqueue("Value 3");

queue.Enqueue("Value 4");

// Retrieve an object from the Queue collection.

string valueFromCollection = (string)queue.Dequeue();

// Returns "Value"

**Класс коллекция Stack.** Класс Stack это FILO структура данных. Для добавления и удаления элементов класс Stack предоставляет методы Push и Pop. При использовании метода Push для добавления объекта в коллекцию Stack объект добавляется в начало коллекции, при использовании метода Pop – автоматически извлекается из начала коллекции. Как класс Queue класс Stack предоставляют метод Peek для извлечения элемента из начала коллекции Stack, не удаляя его. Класс Stack растет автоматически, по мере того как объекты добавляются в коллекцию. Если нужно восстановить память для объекта Stack за счет снижения размера коллекции можно использовать метод TrimToSize. В следующем примере кода показано использование класса Stack.

// Create a new Stack object.

Stack stack = new Stack();

// Add values to the Stack collection.

stack.Push("Value");

stack.Push("Value 2");

stack.Push("Value 3");

stack.Push("Value 4");

// Retrieve a value from the Stack collection without

// removing it from the Stack collection.

string peekValueFromCollection = (string)stack.Peek();

// Returns "Value 4"

// Retrieve an object from the Stack collection.

string valueFromCollection = (string)stack.Pop();

// Returns "Value 4"

// Retrieve another object from the Stack collection.

string valueFromCollection2 = (string)stack.Pop();

// Returns "Value 3"

**Клас коллекция Hashtable.** Класс Hashtable позволяет хранить пары ключ-значение в коллекции быстрого доступа. При добавлении элемента в класс Hashtable с помощью метода Add предоставляются как ключ, так и значение. Ключ коллекции должен быть уникальным, а значение может существовать в двух и более экземплярах. Класс Hashtable хранит объекты, основанные на хэш-значениях ключа.

Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Получающееся хеш-значение играет роль индекса. Из класса Hashtable извлекается значение с помощью индексации и указав ключ значения, которое нужно получить. Класс Hashtable хэширует ключ для определения местоположения требуемого значения. Класс Hashtable значительно быстрее, чем другие коллекции для извлечения элемента из большой коллекции, потому что для нахождения правильного значения просматривает меньшее число элементов. Однако, для небольших коллекций накладные расходы при генерации хэш-кода каждый раз при добавлении значения к коллекции могут фактически снизить производительность приложения. Таким образом для небольших коллекций следует рассмотреть использование другого класса коллекции.

Класс Hashtable полагается на создания хэш-значений, добавленных к коллекции, по этой причине нужно добавить ключ в коллекцию, где тип ключа реализует метод GetHashCode. Каждый объект включает в себя реализацию по умолчанию метода GetHashCode, унаследованный от класса System.Object. Однако, часто необходимо добавить более сложный алгоритм хэширования для любых разрабатываемых типов. При разработке алгоритма хэширования для использования с классом коллекцией Hashtable, необходимо использовать (case-insensitive) алгоритм без учета регистра. В следующем примере показано использование класса Hashtable.

// Create a new Hashtable object.

Hashtable hashtable = new Hashtable();

// Add values to the Hashtable collection.

hashtable.Add("Key A", "Value");

hashtable.Add("Key B", "Value 2");

hashtable.Add("Key C", "Value 3");

hashtable.Add("Key D", "Value 4");

// Remove an item from the Hashtable collection by specifying the key.

hashtable.Remove("Key C");

// Retrieve an item from the Hashtable collection

// by specifying the key.

string valueFromCollection = (string)hashtable["Key B"];

// Returns "Value 2"

**Класс коллекция SortedList.** Как и класс Hashtable, класс коллекция SortedList хранит коллекцию пар ключ/значение объекта. Однако, значения в коллекции отсортированы посредством ключа. При переборе данные в коллекции SortedList будут представлены в порядке ключей. В следующем примере показано использование класса SortedList.

// Create a new SortedList object.

SortedList sortedList = new SortedList();

// Add values to the SortedList collection.

sortedList.Add("Key A", "Value");

sortedList.Add("Key B", "Value 2");

sortedList.Add("Key C", "Value 3");

sortedList.Add("Key D", "Value 4");

// Remove an item from the SortedList by specifying the key.

sortedList.Remove("Key C");

// Retrieve an item from the SortedList collection

// by specifying the key.

string valueFromCollection = (string)sortedList["Key B"];

// Returns "Value 2".

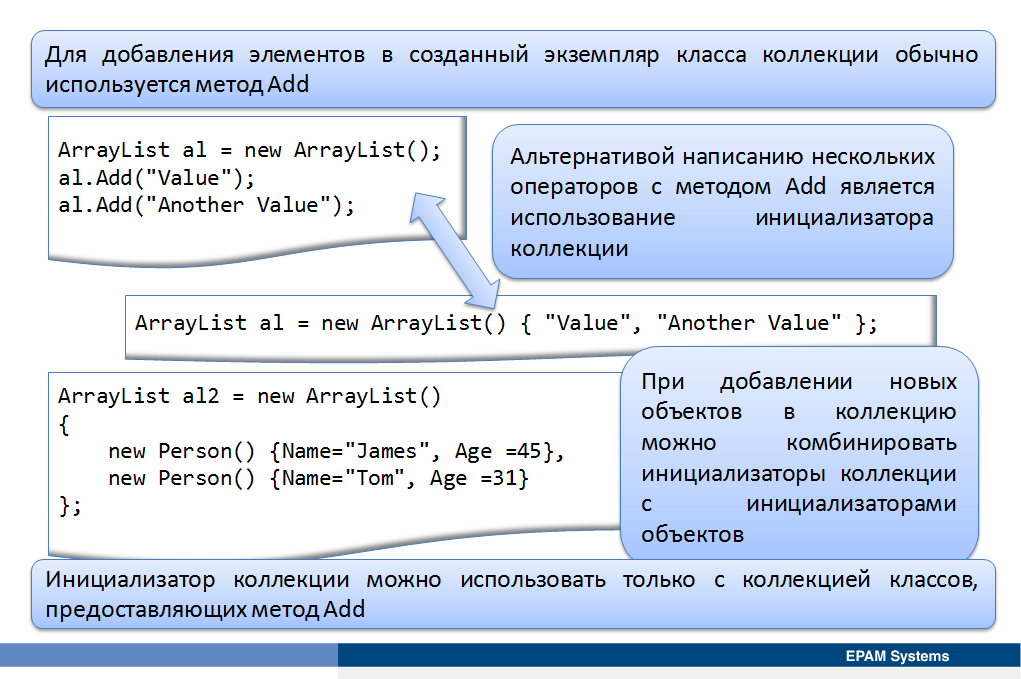
// Retrieve an item from the SortedList collection

// by specifying the index.

string valueFromCollection2 = (string)sortedList.GetByIndex(0);

// Returns "Value"

Использование инициализаторов коллекции



Для добавления элементов в созданный экземпляр класса коллекции обычно используется метод Add, поэтому создание коллекции часто оканчивается кодом, напоминающим следующий.

ArrayList al = new ArrayList();

al.Add("Value");

al.Add("Another Value");

Альтернативой написанию нескольких операторов с методом Add является использование инициализатора коллекции. Инициализация коллекции имеет синтаксис аналогичный инициализации объекта; определяется тип коллекции, а затем в коллекцию добавляются значения в фигурных скобках через запятую до точки с запятой. В следующем примере показано использование инициализатора коллекции вместо использования метода Add.

ArrayList al = new ArrayList() { "Value", "Another Value" };

При добавлении новых объектов в коллекцию можно комбинировать инициализаторы коллекции с инициализаторами объектов. Например,

ArrayList al2 = new ArrayList()

{

new Person() {Name="James", Age =45},

new Person() {Name="Tom", Age =31}

};

Инициализатор коллекции можно использовать только с коллекцией классов, предоставляющих метод Add. Компилятор использует метод Add для добавления в коллекцию объектов, указаных в инициализаторе коллекции. Классы коллекции, такие как класс Queue, не предоставляющие метод Add, не поддерживают инициализаторы коллекции.

**Примеры использования классов коллекций для хранения стандартных и пользовательских типов.**

Для типа элементов в [List<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.list-1) можно также определить собственный класс. В приведенном ниже примере класс Galaxy, который используется объектом [List<T>](https://docs.microsoft.com/ru-ru/dotnet/api/system.collections.generic.list-1), определен в коде.

private static void IterateThroughList()

{

var theGalaxies = new List<Galaxy>

{

new Galaxy() { Name="Tadpole", MegaLightYears=400},

new Galaxy() { Name="Pinwheel", MegaLightYears=25},

new Galaxy() { Name="Milky Way", MegaLightYears=0},

new Galaxy() { Name="Andromeda", MegaLightYears=3}

};

foreach (Galaxy theGalaxy in theGalaxies)

{

Console.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears);

}

// Output:

// Tadpole 400

// Pinwheel 25

// Milky Way 0

// Andromeda 3

}

public class Galaxy

{

public string Name { get; set; }

public int MegaLightYears { get; set; }

}

Использование словарей:

using System;

using System.Collections.Generic;

namespace ConsoleApplication1

{

class UserInfo

{

// Метод, реализующий словарь

public static Dictionary<int, string> MyDic(int i)

{

Dictionary<int, string> dic = new Dictionary<int,string>();

Console.WriteLine("Введите имя сотрудника: \n");

string s;

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

{

Console.Write("Name{0} --> ",j);

s = Console.ReadLine();

dic.Add(j, s);

Console.Clear();

}

return dic;

}

}

class Program

{

static void Main()

{

Console.Write("Сколько сотрудников добавить? ");

try

{

int i = int.Parse(Console.ReadLine());

Dictionary<int, string> dic = UserInfo.MyDic(i);

// Получить коллекцию ключей

ICollection<int> keys = dic.Keys;

Console.WriteLine("База данных содержит: ");

foreach (int j in keys)

Console.WriteLine("ID -> {0} Name -> {1}",j,dic[j]);

}

catch (FormatException)

{

Console.WriteLine("Неверный ввод");

}

Console.ReadLine();

}

}

}

Реализация интерфейса IComparer

Для сортировки объектов определяемых пользователем классов зачастую проще всего реализовать в этих классах интерфейс IComparable. Тем не менее данную задачу можно решить и с помощью интерфейса IComparer. Для этой цели необходимо сначала создать класс, реализующий интерфейс IComparer, а затем указать объект этого класса, когда потребуется сравнение.

Интерфейс IComparer существует в двух формах: обобщенной и необобщенной. Несмотря на сходство применения обеих форм данного интерфейса, между ними имеются некоторые, хотя и небольшие, отличия.

В необобщенном интерфейсе IComparer определяется только один метод Compare():

int Compare(object x, object у)

В методе Compare() сравниваются объекты х и у. Для сортировки объектов по нарастающей конкретная реализация данного метода должна возвращать нулевое значение, если значения сравниваемых объектов равны; положительное — если значение объекта х больше, чем у объекта у; и отрицательное — если значение объекта х меньше, чем у объекта у. А для сортировки по убывающей можно обратить результат сравнения объектов. Если же тип объекта x не подходит для сравнения с объектом y, то в методе Compare() может быть сгенерировано исключение ArgumentException.

Объект типа IComparer может быть указан при конструировании объекта класса SortedList, при вызове метода ArrayList.Sort(IComparer), а также в ряде других мест в классах коллекций. Главное преимущество применения интерфейса IComparer заключается в том, что сортировке подлежат объекты тех классов, в которых интерфейс IComparable не реализуется.

Интерфейс IComparer<T> является обобщенным вариантом интерфейса IComparer. В нем определяется приведенный ниже обобщенный вариант метода Compare():

int Compare(Т х, T у)

В этом методе сравниваются объекты х и у и возвращается нулевое значение, если значения сравниваемых объектов равны; положительное — если значение объекта х больше, чем у объекта у; и отрицательное — если значение объекта х меньше, чем у объекта у.

Рассмотрим пример:

using System;

using System.Collections.Generic;

namespace ConsoleApplication1

{

class AutoShop

{

public string CarName { set; get; }

public int MaxSpeed { get; set; }

public double Cost { get; set; }

public byte Discount { get; set; }

public int ID { get; set; }

public AutoShop() { }

public AutoShop(string CarName, int MaxSpeed, double Cost, byte Discount, int ID)

{

this.CarName = CarName;

this.MaxSpeed = MaxSpeed;

this.Cost = Cost;

this.Discount = Discount;

this.ID = ID;

}

public override string ToString()

{

return String.Format("{4}\tМарка: {0}\tМакс. скорость: {1}\tЦена: {2:C}\tСкидка: {3}%",

this.CarName,this.MaxSpeed,this.Cost,this.Discount,this.ID);

}

}

class CompInv<T> : IComparer<T>

where T : AutoShop

{

// Реализуем интерфейс IComparer<T>

public int Compare(T x, T y)

{

if (x.Cost < y.Cost)

return 1;

if (x.Cost > y.Cost)

return -1;

else return 0;

}

}

class Program

{

static void Main()

{

CompInv<AutoShop> cp = new CompInv<AutoShop>();

List<AutoShop> dic = new List<AutoShop>();

// Создадим множество автомобилей

dic.Add(new AutoShop("Toyota Corolla", 180, 300000, 5, 1));

dic.Add(new AutoShop("VAZ 2114i", 160, 220000, 0, 2));

dic.Add(new AutoShop("Daewoo Nexia", 140, 260000, 5, 3));

dic.Add(new AutoShop("Honda Torneo", 220, 400000, 7, 4));

dic.Add(new AutoShop("Audi R8 Best", 360, 4200000, 3, 5));

Console.WriteLine("Исходный каталог автомобилей: \n");

Console.WindowWidth = 100;

foreach (AutoShop a in dic)

Console.WriteLine(a);

Console.WriteLine("\nТеперь автомобили отсортированны по стоимости: \n");

dic.Sort(cp);

foreach (AutoShop a in dic)

Console.WriteLine(a);

Console.ReadLine();

}

}

}

# Перечислители

К элементам коллекции нередко приходится обращаться циклически, например, для отображения каждого элемента коллекции. С этой целью можно, с одной стороны, организовать цикл foreach, а с другой — воспользоваться перечислителем. ***Перечислитель*** — это объект, который реализует необобщенный интерфейс IEnumerator или обобщенный интерфейс IEnumerator<T>.

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:

* Получить перечислитель, устанавливаемый в начало коллекции, вызвав для этой коллекции метод GetEnumerator().
* Организовать цикл, в котором вызывается метод MoveNext(). Повторять цикл до тех пор, пока метод MoveNext() возвращает логическое значение true.
* Получить в цикле каждый элемент коллекции с помощью свойства Current.

Давайте рассмотрим пример:

using System;

using System.Collections.Generic;

namespace ConsoleApplication1

{

class Program

{

static void Main()

{

List<int> arr = new List<int>();

Random ran = new Random();

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

arr.Add(ran.Next(1, 20));

// Используем перечислитель

IEnumerator<int> etr = arr.GetEnumerator();

while (etr.MoveNext())

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

Console.WriteLine("\nПовторный вызов перечислителя: \n");

// Сбросим значение и вновь используем перечислитель

// для доступа к коллекции

etr.Reset();

while (etr.MoveNext())

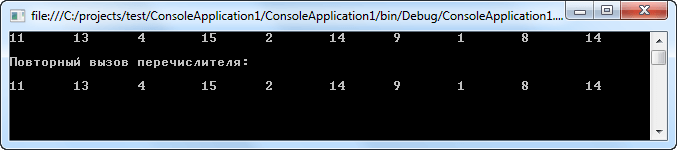
Console.Write(etr.Current + "\t");

Console.ReadLine();

}

}

}



Вообще говоря, для циклического обращения к элементам коллекции цикл fоreach 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.

# Реализация интерфейсов IEnumerable и IEnumerator

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.

Ниже приведен пример, в котором класс MyInt реализует данные интерфейсы:

using System;

using System.Collections;

namespace ConsoleApplication1

{

class MyInt : IEnumerable, IEnumerator

{

int[] ints = { 12, 13, 1, 4 };

int index = -1;

// Реализуем интерфейс IEnumerable

public IEnumerator GetEnumerator()

{

return this;

}

// Реализуем интерфейс IEnumerator

public bool MoveNext()

{

if (index == ints.Length - 1)

{

Reset();

return false;

}

index++;

return true;

}

public void Reset()

{

index = -1;

}

public object Current

{

get

{

return ints[index];

}

}

}

class Program

{

static void Main()

{

MyInt mi = new MyInt();

foreach (int i in mi)

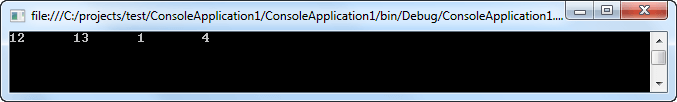
Console.Write(i+"\t");

Console.ReadLine();

}

}

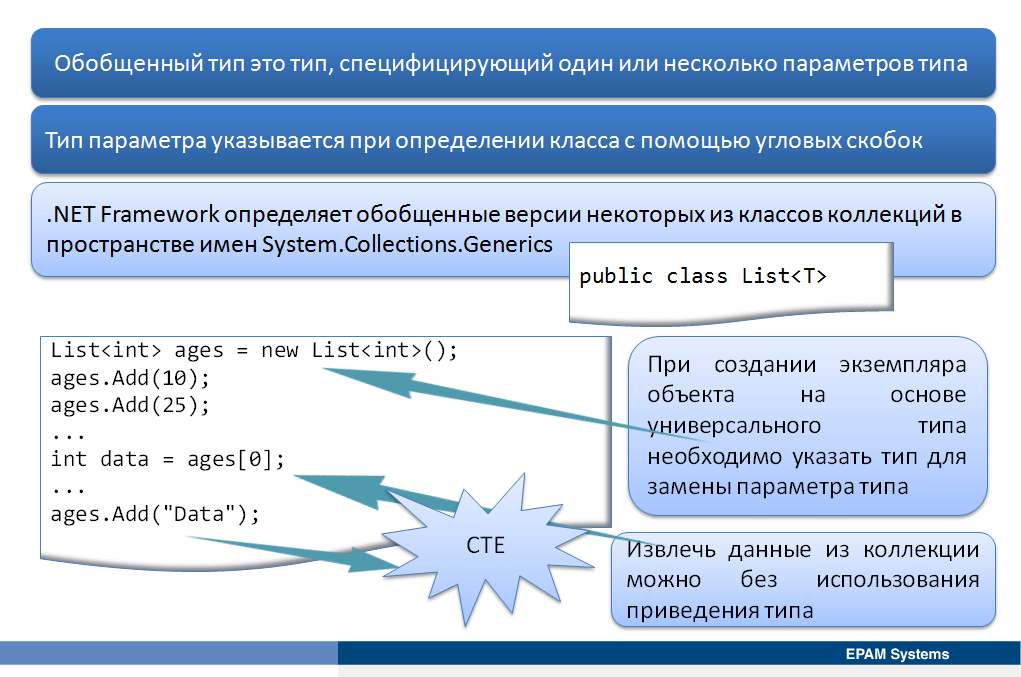
}



**Generics**

У классов коллекций пространства имен System.Collections существует одна серьезная проблема: они не типизированны. Они хранят ссылки на объекты System.Object, а не экземпляры определенного типа. Это создает опасность появления ошибок в приложениях, поскольку неправильное приведение типов может стать причиной генерации исключений во время выполнения приложения. C# включает в себя обобщенные типы, позволяющие указать, какой тип будет использован с классом. Используя обобщенные классы, можно обеспечить безопасность типов, разрабатывая в то же время гибкое приложение. Урок знакомит с обобщенными типами и возможностью их использования в приложениях.

Необходимость использования generics.



Базовые классы коллекции не позволяют указать тип данных, которые может содержать коллекция, что может привести ко многим проблемам. Код в следующем примере кода будет компилироваться, но при запуске будет генерироваться исключение InvalidCastException.

ArrayList names = new ArrayList();

names.Add("Alice");

...

int data = (int)names[0];

Также возникает вопрос, связанный с производительностью. Встроенные коллекции все содержат элементы типа System.Object. Для ссылки на любой ссылочный тип с небольшими накладными расходами можно использовать тип Object, но если в коллекции хранятся значимые типы, компилятор генерирует код упаковки данных. При извлечении значимого типа из коллекции компилятор генерирует код распаковки данных. Упаковка и распаковка налагать значительные накладные расходы. Чтобы обойти эти ограничения, желательно определить классы коллекции, определяющие тип хранимых элементов. Сделать это можно с помощью обобщений.

Обобщенный тип это тип, специфицирующий один или несколько параметров типа. Параметр типа похож на параметр метода за исключением того, что он определяет тип, а не значение. Тип параметра указывается при определении класса с помощью угловых скобок. При создании экземпляра объекта на основе универсального типа необходимо указать тип для замены параметра типа. .NET Framework определяет обобщенные версии некоторых из классов коллекций в пространстве имен System.Collections.Generics, в том числе тип List.

public class List<T>

Следующий пример показывает использование обобщенной коллекции List. Тип List представляет строго типизированный список объектов, доступ к которым можно получить с использованием индекса. Параметром типа для коллекции ages является int, и, следовательно, в этой коллекции можно хранить только значения int. Если попытаться сохранить в ней другой тип, компилятор выдаст ошибку. Кроме того, извлечь данные из коллекции можно без использования приведения типа.

List<int> ages = new List<int>();

ages.Add(10);

ages.Add(25);

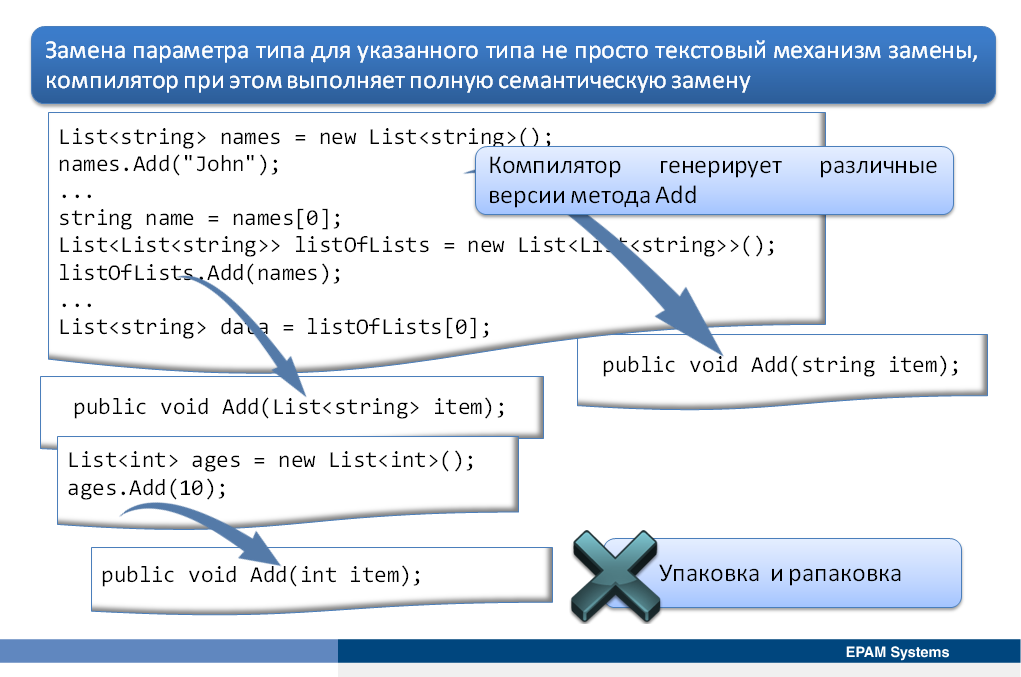
...

int data = ages[0]; // No cast necessary

...

ages.Add("Data"); // Compiler error

Создание generic классов.



Замена параметра типа для указанного типа не просто текстовый механизм замены, компилятор при этом выполняет полную семантическую замену. В случае типа List<T> можно указать любой допустимый тип параметра T.

List<string> names = new List<string>();

names.Add("John");

...

string name = names[0];

List<List<string>> listOfLists = new List<List<string>>();

listOfLists.Add(names);

...

List<string> data = listOfLists[0];

В первом примере создается коллекция-список строковых объектов, во втором – коллекция-список колекций строковых объектов. Для переменных компилятор генерирует различные версии метода Add: следующую версию метода для переменной names.

public void Add(string item);

А для переменной listOfLists компилятор генерирует другую версию метода Add.

public void Add(List<string> item);

Это означает, что нельзя вызвать метод Add с другим объектом типа List и передать параметр, имеющий неправильный тип без генерации ошибки компилятора. То же логическое обоснование относится ко всем другим методам и свойствам, которые реализует обобщенный класс List. При определении переменной с помощью обобщенного типа создается совершенно новый тип, и компилятор генерирует строго типизированные методы и свойства для данного типа.

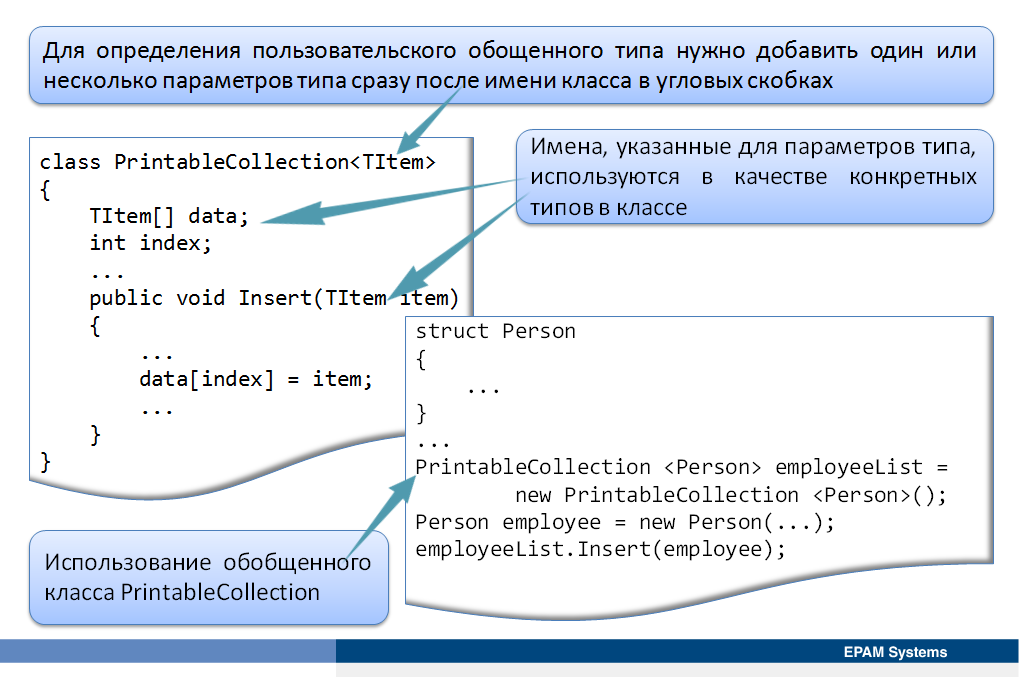
Использование значимого типа в качестве параметра типа положительно влияет на производительность. В следующем примере создается коллекция-список целых чисел, компилятор создает еще один вариант метода Add, принимающий параметр int.

List<int> ages = new List<int>();

ages.Add(10);

// The compiler generates the following method: public void Add(int item);

При вызове метода Add для переменной ages ему передается целое значение, компилятору нет необходимости генерации кода упаковки и распаковки этого значения.



Для определения пользовательского обощенного типа нужно добавить один или несколько параметров типа. Для определения параметров типа сразу после имени класса добавляются угловые скобки. В угловых скобках можно указать имена для каждого из параметров типа, при этом, можно использовать столько параметров типа, сколько необходимо, перечисляя их через запятую. Имена, указанные для параметров типа, используются в качестве конкретных типов в классе.

В следующем примере показан пользовательский обобщенный тип PrintableCollection, использующий параметр типа TItem. Предполагается, что тип PrintableCollection определяет коллекцию элементов, которые могут быть легко отформатированы для печати и отображения. Класс PrintableCollection определяет метод Insert, принимающий параметр типа TItem. Внутри класс использует TItem как и любой другой тип. В примере создается массив объектов TItem, а метод Insert вставляет в массив значение, определяемое его индексом в этом массиве.

class PrintableCollection<TItem>

{

TItem[] data;

int index;

...

public void Insert(TItem item)

{

...

data[index] = item;

...

}

}

Пользовательские обощенные типы используются путем предоставления типа для параметра типа. В следующем примере показано использование обощенного класса PrintableCollection для создания и использования печати коллекции структур Person[[3]](#footnote-4).

struct Person

{

...

}

...

PrintableCollection <Person> employeeList = new PrintableCollection <Person>();

Person employee = new Person(...);

employeeList.Insert(employee);

Хорошей практикой является всегда инициализировать члены типа, то же верно и при определении обобщенного типа. Как правило, добиться этого можно путем предоставления одного или нескольких конструкторов. Однако, если нужно инициализировать члены класса на основе параметра типа значением по умолчанию, не всегда легко определить, что это должно быть за значение по умолчанию. C# предоставляет ключевое слово default, используемое для решения этой проблемы.

class PrintableCollection<TItem>

{

TItem[] data;

int index;

TItem tempData;

...

public PrintableCollection()

{

this.tempData = default(TItem);

...

}

...

}

Не следует включать параметр типа в название конструктора для управляемого типа.

Ключевое слово default порождает значимые значение по умолчанию для типа, указанного в качестве параметра типа. Когда компилятор генерирует код для обощенного типа, он заменяет конструкцию default значением по умолчанию, зависящим от конкретного типа. Значение для ссылочных типов по умолчанию будет null и нуль для числовых значимых типов. Для структуры ключевое слово default будет инициализировать каждый член структуры нулем или null в зависимости от того, являются они значимым типом или ссылкой.

**Сравнительный анализ generic классов и классов коллекций (и их интерфейсов).**

**Основные преимущества использования обобщений:**

**Производительность**

Одним из основных преимуществ обобщений является производительность. Использование типов значений с необобщенными классами коллекций вызывает **упаковку (boxing)** и **распаковку (unboxing)** при преобразовании в ссылочный тип и обратно.

Типы значений сохраняются в стеке, а типы ссылок — в куче. Классы C# являются ссылочными типами, а структуры — типами значений. .NET позволяет легко преобразовывать типы значений в ссылочные, поэтому их можно использовать там, где ожидаются объекты (т.е. ссылочные типы). Например, объекту можно присвоить значение типа int.

Преобразование типа значений в ссылочный тип называется упаковкой (boxing). Упаковка происходит автоматически, когда метод ожидает параметр ссылочного типа, а ему передается тип значений. С другой стороны, упакованный тип значений может быть обратно преобразован к простому типу значений с помощью распаковки (unboxing). При распаковке требуется операция приведения.

**Безопасность**

Другим свойством обобщений является безопасность типов. Обобщения автоматически обеспечивают типовую безопасность всех операций. В ходе выполнения этих операций обобщения исключают необходимость обращаться к приведению типов и проверять соответствие типов в коде вручную.

**Повторное использование двоичного кода**

Обобщения повышают степень повторного использования двоичного кода. Обобщенный класс может быть определен однажды, и на его основе могут быть созданы экземпляры многих типов. При этом не нужно иметь доступ к исходным текстам, как это необходимо в случае шаблонов С++.

**"Разбухание" кода**

Насколько много кода генерируется при создании экземпляров конкретных типов из обобщений? Поскольку определение обобщенного класса включается в сборку, создание на его основе конкретных классов специфических типов не приводит к дублированию кода в IL.

Однако когда обобщенные классы компилируются [JIT-компилятором](https://professorweb.ru/my/glossariy/glossy/files/102.php#j1) в родной машинный код, для каждого конкретного типа значения создается новый класс. Ссылочные типы при этом разделяют общую реализацию одного родного класса. Причина в том, что в случае ссылочных типов каждый элемент представлен в памяти 4-байтным адресом (на 32-разрядных системах) и машинные реализации обобщенного класса с различными ссылочными типами-параметрами не отличаются друг от друга. В отличие от этого, типы значений содержатся в памяти целиком, и поскольку каждый из них требует разного объема памяти, то для каждого из них создаются свои экземпляры классов на основе обобщенного.

В случае использования классов обобщенных коллекций исчезают все описанные выше проблемы, включая затраты на упаковку/распаковку и недостаток безопасности типов. Кроме того, потребность в создании специального класса (обобщенной) коллекции становится довольно редкой. Вместо построения специальных коллекций, которые могут хранить людей, автомобили и целые числа, можно обратиться к обобщенному классу коллекции и указать тип хранимых элементов.

В показанном ниже методе класс List (из пространства имен System.Collection. Generic) используется для хранения различных типов данных в строго типизированной манере (пока не обращайте внимания на детали синтаксиса обобщений):

static void UseGenericList()

{

Console.WriteLine("\*\*\*\*\* Fun with Generics \*\*\*\*\*\n");

// Этот List<> может хранить только объекты Person.

List morePeople = new List();

morePeople.Add(new Person ("Frank", "Black", 50));

Console.WriteLine(morePeople[0]);

// Этот List<> может хранить только целые числа.

List moreInts = new List();

moreInts.Add(10); moreInts.Add(2);

int sum = moreInts[0] + moreInts[1];

// Ошибка компиляции! Объект Person не может быть добавлен в список int!

// moreInts.Add(new Person());

}

Первая коллекция List может содержать только объекты Person. Поэтому выполнять приведение при извлечении элементов из контейнера не требуется, что делает этот подход более безопасным в отношении типов. Вторая коллекция List может хранить только целые числа, и все они размещены в стеке; другими словами, здесь не происходит никакой скрытой упаковки/распаковки, как это имеет место в необобщенном ArrayList.

Вложенные типы внутрь generic класса.

Типы, объявленные внутри generic-типов, являются, по сути, тоже параметризованными. Даже если во вложенном типе не используются параметры внешнего типа, компилятор все равно учитывает их наличие. Так, в следующем описании:

class OuterGenericClass<T>

{

class Inner { }

}

Класс Inner является параметризованным, хотя и не содержит явно заданного списка параметров. В приведенном ниже примере конструируемые типы будут различны, несмотря на то, что сам вложенный класс не имеет собственных параметров:

OuterGenericClass<int>.Inner

OuterGenericClass<float>.Inner

Если же объявить вложенный тип с собственным набором параметров, то он будет параметризован одновременно собственными параметрами и параметрами внешнего типа:

class Outer<T>

{

class Inner<U> { }

}

Здесь реальный тип класса Inner будет Outer<T>.Inner<U>.

Если создать вложенный тип с параметрами, имена которых будут совпадать с именами параметров внешнего типа, то произойдет перекрытие области видимости. При этом внутри вложенного типа не будут доступны перекрытые параметры внешнего, а компилятор будет выдавать предупреждение. Так, следующее объявление:

class Outer<T>

{

public class Inner<T> { }

}

порождает класс:

Outer<T1>.Inner<T2>

где T1 и T2 – совершенно независимые типы. Можно изменить описание следующим образом:

class Outer<T>

{

T \_outerField;

public class Inner<T>

{

T \_innerField;

}

}

Тогда в следующем объявлении:

Outer<int>.Inner<string> variable;

поле \_outerField будет иметь тип int, а поле \_innerField – string.

В общем, можно сказать, что каждая специализация порождает отдельный тип. Это хорошо демонстрирует следующий пример:

using System;

class Outer<T>

{

public class Inner

{

public static int \_count = 0;

public Inner() { \_count++; }

}

}

class App

{

static void Main()

{

Outer<int>.Inner x1 = new Outer<int>.Inner();

Console.WriteLine(Outer<int>.Inner.\_count); // Выведет 1

Outer<double>.Inner x2 = new Outer<double>.Inner();

Console.WriteLine(Outer<int>.Inner.\_count); // Выведет 1

Outer<int>.Inner x3 = new Outer<int>.Inner();

Console.WriteLine(Outer<int>.Inner.\_count); // Выведет 2

Console.WriteLine("---");

Console.WriteLine(typeof(Outer<int>.Inner).FullName);

Console.WriteLine();

Console.WriteLine(typeof(Outer<double>.Inner).FullName);

Console.ReadLine();

}

}

Если выполнить его, то в консоль будет выведено:

1

1

2

---

Outer+Inner[[System.Int32, mscorlib, Version=1.2.3400.0, Culture=neutral, Public KeyToken=b77a5c561934e089]]

Outer+Inner[[System.Double, mscorlib, Version=1.2.3400.0, Culture=neutral, Publi cKeyToken=b77a5c561934e089]]

Обратите внимание, что для каждого конструируемого типа создан свой набор статических переменных. Причем это никак не связано с тем, что класс вложенный. Такое поведение характерно для параметризованных типов в целом.

Использование ограничений.

Допустим, что требуется создать метод, оперирующий содержимым потока, включая объекты типа FileStream или MemoryStream. На первый взгляд, такая ситуация идеально подходит для применения обобщений, но при этом нужно каким-то образом гарантировать, что в качестве аргументов типа будут использованы только типы потоков, но не int или любой другой тип. Кроме того, необходимо как-то уведомить компилятор о том, что методы, определяемые в классе потока, будут доступны для применения. Так, в обобщенном коде должно быть каким-то образом известно, что в нем может быть вызван метод Read().

Для выхода из подобных ситуаций в C# предусмотрены ограниченные типы. Указывая параметр типа, можно наложить определенное ограничение на этот параметр. Это делается с помощью **оператора where** при указании параметра типа:

*class имя\_класса<параметр\_типа> where параметр\_типа : ограничения { // ...*

где ограничения указываются списком через запятую.

В C# предусмотрен ряд ограничений на типы данных:

**Ограничение на базовый класс**

Требует наличия определенного базового класса в аргументе типа. Это ограничение накладывается указанием имени требуемого базового класса. Разновидностью этого ограничения является неприкрытое ограничение типа, при котором на базовый класс указывает параметр типа, а не конкретный тип. Благодаря этому устанавливается взаимосвязь между двумя параметрами типа.

**Ограничение на интерфейс**

Требует реализации одного или нескольких интерфейсов аргументом типа. Это ограничение накладывается указанием имени требуемого интерфейса.

**Ограничение на конструктор**

Требует предоставить конструктор без параметров в аргументе типа. Это ограничение накладывается с помощью оператора new().

**Ограничение ссылочного типа**

Требует указывать аргумент ссылочного типа с помощью оператора class.

**Ограничение типа значения**

Требует указывать аргумент типа значения с помощью оператора struct.

Среди всех этих ограничений чаще всего применяются ограничения на базовый класс и интерфейс, хотя все они важны в равной степени.

## Связь между параметрами типа с помощью ограничений

Существует разновидность ограничения на базовый класс, позволяющая установить связь между двумя параметрами типа. В качестве примера рассмотрим следующее объявление обобщенного класса:

class MyObj<T, V> where V : T { ...

В этом объявлении оператор where уведомляет компилятор о том, что аргумент типа, привязанный к параметру типа V, должен быть таким же, как и аргумент типа, привязанный к параметру типа Т, или же наследовать от него. Если подобная связь отсутствует при объявлении объекта типа MyObj, то во время компиляции возникнет ошибка. Такое ограничение на параметр типа называется неприкрытым ограничением типа.

С параметром типа может быть связано несколько ограничений. В этом случае ограничения указываются списком через запятую. В этом списке первым должно быть указано ограничение class либо struct, если оно присутствует, или же ограничение на базовый класс, если оно накладывается. Указывать ограничения class или struct одновременно с ограничением на базовый класс не разрешается. Далее по списку должно следовать ограничение на интерфейс, а последним по порядку — ограничение new().

Например, следующее объявление считается вполне допустимым:

class MyObj<T> where T : MyClass, IMyInterface, new() {

// ...

В данном случае параметр типа T должен быть заменен аргументом типа, наследующим от класса MyClass, реализующим интерфейс IMyInterface и использующим конструктор без параметра.

Если же в обобщении используются два или более параметра типа, то ограничения на каждый из них накладываются с помощью отдельного оператора where.

# Ограниченные классы

**Ограничение на базовый класс** позволяет указывать базовый класс, который должен наследоваться аргументом типа. Ограничение на базовый класс служит двум главным целям:

* Оно позволяет использовать в обобщенном классе те члены базового класса, на которые указывает данное ограничение. Это дает, например, возможность вызвать метод или обратиться к свойству базового класса. В отсутствие ограничения на базовый класс компилятору ничего не известно о типе членов, которые может иметь аргумент типа. Накладывая ограничение на базовый класс, вы тем самым даете компилятору знать, что все аргументы типа будут иметь члены, определенные в этом базовом классе.
* Ограничение на базовый класс гарантирует использование только тех аргументов типа, которые поддерживают указанный базовый класс. Это означает, что для любого ограничения, накладываемого на базовый класс, аргумент типа должен обозначать сам базовый класс или производный от него класс. Если же попытаться использовать аргумент типа, не соответствующий указанному базовому классу или не наследующий его, то в результате возникнет ошибка во время компиляции.

Ниже приведена общая форма наложения ограничения на базовый класс, в которой используется оператор where:

*where T : имя\_базового\_класса*

где T обозначает имя параметра типа, а имя\_\_базового\_класса — конкретное имя ограничиваемого базового класса. Одновременно в этой форме ограничения может быть указан только один базовый класс.

Давайте рассмотрим пример:

using System;

namespace ConsoleApplication1

{

class UserInfo

{

public UserInfo(string Name, int Age)

{

this.Name = Name;

this.Age = Age;

}

public string Name { get; set; }

public int Age { get; set; }

}

// Создадим класс, унаследованный от UserInfo

class AllInfoUser : UserInfo

{

public AllInfoUser(string Family, string Name, int Age)

: base(Name, Age)

{

this.Family = Family;

}

public string Family { get; set; }

public override string ToString()

{

string s = String.Format("Информация о пользователе: \n{0} {1} {2}\n",this.Name,Family,this.Age);

return s;

}

}

// Обобщенный класс использующий ограничение на базовый класс

class Info<T> where T : UserInfo

{

T[] UserList;

int i;

public Info()

{

UserList = new T[3];

i = 0;

}

public void Add(T obj)

{

if (i == 3) return;

UserList[i] = obj;

i++;

return;

}

public void ReWrite()

{

foreach (T t in UserList)

Console.WriteLine(t.ToString());

}

}

class Program

{

static void Main()

{

Info<AllInfoUser> database1 = new Info<AllInfoUser>();

database1.Add(new AllInfoUser(Name: "Alex", Family: "Erohin", Age: 26));

database1.Add(new AllInfoUser(Name: "Alexey", Family: "Volkov", Age: 28));

database1.Add(new AllInfoUser(Name: "Dmitryi", Family: "Medvedev", Age: 50));

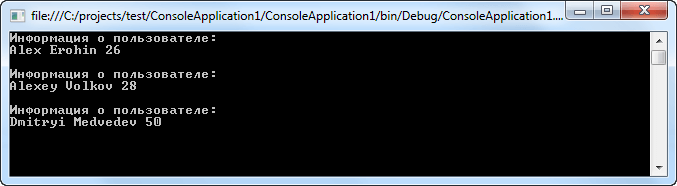
database1.ReWrite();

Console.ReadLine();

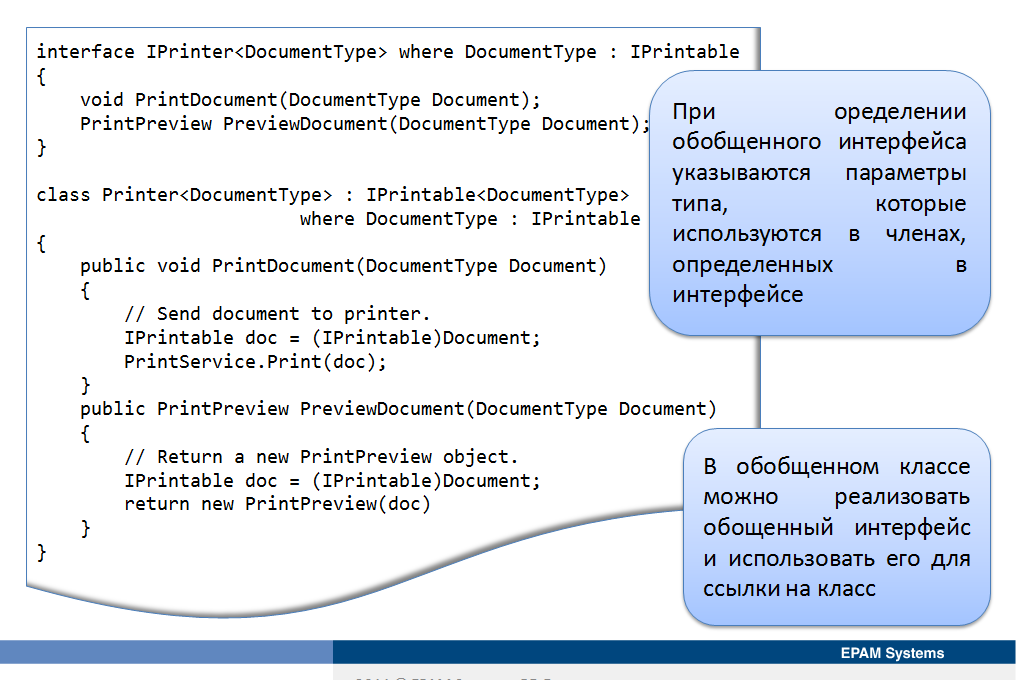
}

}

}



Создание generic интерфейсов



В дополнение к определению обобщенных классов и структур можно определить обобщенные интерфейсы. Обобщенный интерфейс похож на любой другой интерфейс за исключением того, что при его определении указываются параметры типа, которые используются в членах, определенных в интерфейсе. Как и обобщенные классы и структуры, обобщенные интерфейсы могут определить ограничения на параметры типа. В обобщенном классе можно реализовать обобщенный интерфейс и использовать его для ссылки на класс, как и при работе с обычными классом и интерфейсом. В следующем примере показан обобщенный интерфейс, который реализуется в обобщенном классе.

interface IPrinter<DocumentType> where DocumentType : IPrintable

{

void PrintDocument(DocumentType Document);

PrintPreview PreviewDocument(DocumentType Document);

}

class Printer<DocumentType> : IPrintable<DocumentType> where DocumentType : IPrintable

{

public void PrintDocument(DocumentType Document)

{

// Send document to printer.

IPrintable doc = (IPrintable)Document;

PrintService.Print(doc);

}

public PrintPreview PreviewDocument(DocumentType Document)

{

// Return a new PrintPreview object.

IPrintable doc = (IPrintable)Document;

return new PrintPreview(doc)

}

}

## Применение ограничения на интерфейс

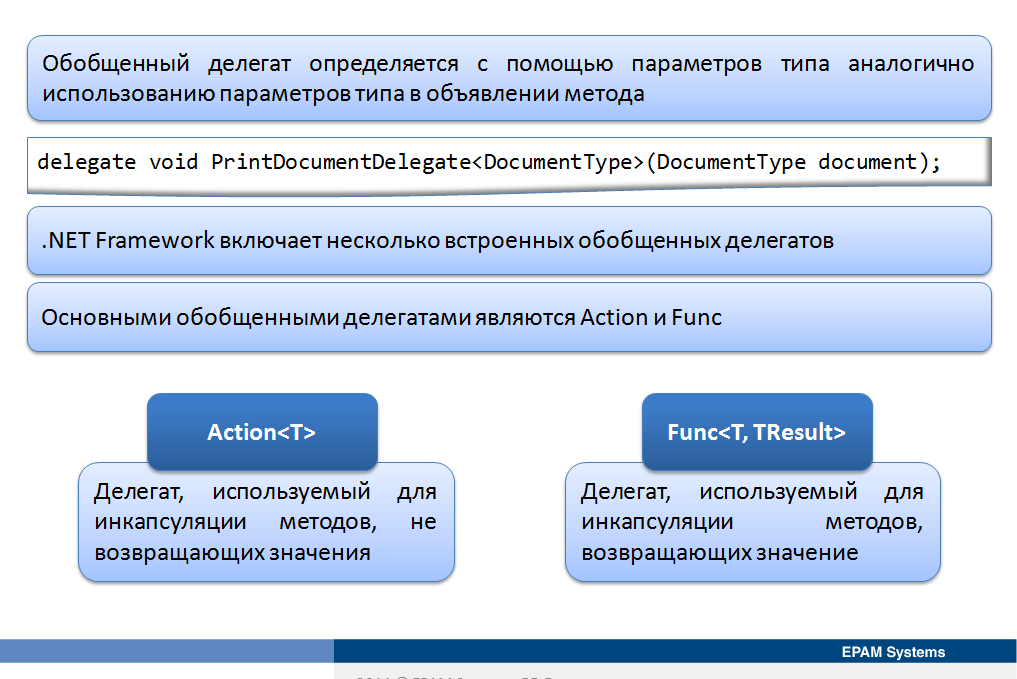
Ограничение на интерфейс позволяет указывать интерфейс, который должен быть реализован аргументом типа. Это ограничение служит тем же основным целям, что и ограничение на базовый класс. Во-первых, оно позволяет использовать члены интерфейса в обобщенном классе. И во-вторых, оно гарантирует использование только тех аргументов типа, которые реализуют указанный интерфейс. Это означает, что для любого ограничения, накладываемого на интерфейс, аргумент типа должен обозначать сам интерфейс или же тип, реализующий этот интерфейс.

Ниже приведена общая форма наложения ограничения на интерфейс, в которой используется оператор where:

*where T : имя\_интерфейса*

где T — это имя параметра типа, а имя\_интерфейса — конкретное имя ограничиваемого интерфейса. В этой форме ограничения может быть указан список интерфейсов через запятую. Если ограничение накладывается одновременно на базовый класс и интерфейс, то первым в списке должен быть указан базовый класс.

Создание generic делегатов



Обобщенный делегат определяется с помощью параметров типа аналогично использованию параметров типа в объявлении метода.

delegate void PrintDocumentDelegate<DocumentType>(DocumentType document);

.NET Framework включает несколько встроенных обобщенных делегатов. В ряде случаев можно использовать встроенные делегаты вместо определения пользовательских. Основными обобщенными делегатами являются Action и Func.

Делегат Action это делегат, используемый для инкапсуляции методов, не возвращающих значения (возвращаемым типом является void). Можно использовать делегат Action вместо объявления пользовательских делегатов для методов без возвращаемого значения. .NET Framework включает в себя несколько перегруженных делегатов Action, которые позволяют указать типы параметров, передаваемых делегату; с каждой перегрузкой, увеличивается число параметров, которые можно указать. Делегат Action имеет перегрузки для поддержки от нуля и 16 параметров (в предыдущих версиях .NET Framework, делегат Action поддерживал только до четырех параметра). В следующем примере кода показано использование делегата Action с простыми лямбда-выражением.

// Define a delegate by using the generic Action delegate.

Action<string, int> myDelegate = null;

// Add a handler for the delegate by using a lambda expression.

myDelegate += ((param1, param2) =>

{

Console.WriteLine("{0} : {1}", param1, param2.ToString());

});

// Invoke the delegate.

if (myDelegate != null)

{

myDelegate("Value", 5);

}

Делегат Func очень похож на делегат Action, но с одним основным отличием: делегат Func возвращает значение. При использовании делегата Func всегда указывается тип возвращаемого значения в дополнение к типам параметров, последний параметр типа всегда тип возвращаемого значения. Как и делегат Action делегат Func поддерживает от нуля до 16 параметров. В следующем примере кода показано использование делегата Func с простыми лямбда-выражением.

// Define a delegate by using the generic Func generic delegate.

Func<string, int, string> myDelegate = null;

// Add a handler for the delegate by using a lambda expression.

myDelegate += ((param1, param2) =>

{

return String.Format("{0} : {1}", param1, param2.ToString());

});

// Invoke the delegate.

if (myDelegate != null)

{

string returnedValue;

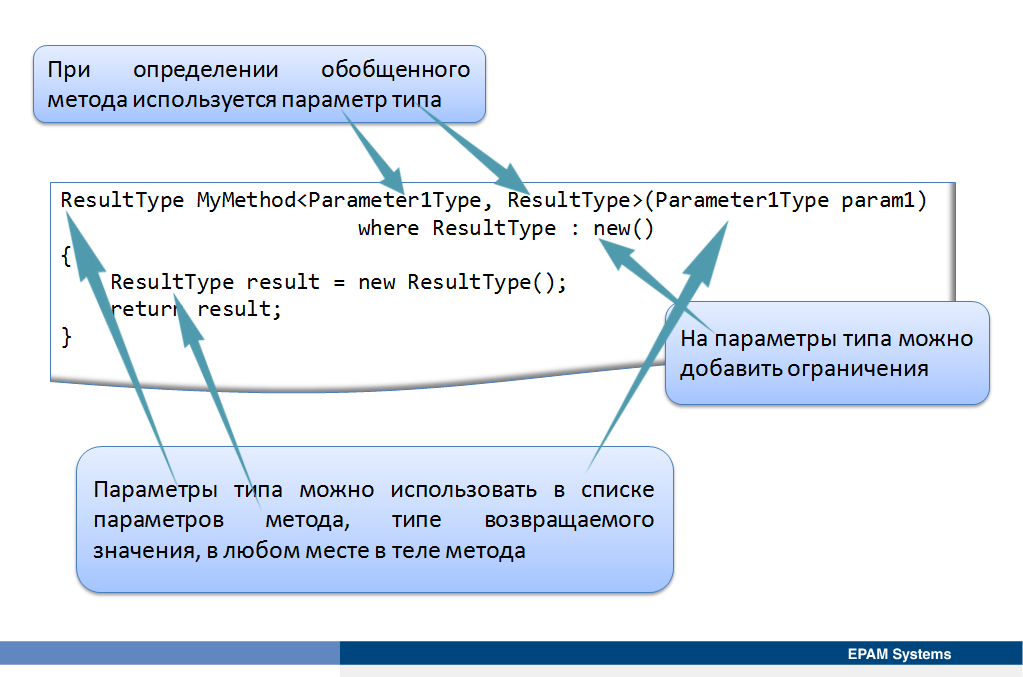
returnedValue = myDelegate("Value", 5);

Console.WriteLine(returnedValue);

}

Делегаты Action и Func используют контрвариантность типов входных параметров. Для результирующего выходного типа делегат Func использует ковариантность. Вариантность в делегатах Action и Func новое в .NET Framework 4.

Создание generic методов.



Как обобщенные типы, так и обобщенные методы и делегаты содержат параметр типа, который можно использовать в списке параметров и возвращаемом типе для метода или делегата. Обобщенные методы позволяют определить методы, выполняющие действия над объектом при сохранении безопасности типов.

В следующем примере показан простой метод, добавляющий отчет в очередь на печать.

void AddToQueue(Report report)

{

printQueue.Add(report);

}

Если потребуется метод для добавления в очередь на печать другого типа документа, можно определить новый метод вида.

void AddToQueue(ReferenceGuide referenceGuide)

{

printQueue.Add(referenceGuide);

}

Хотя такой подход является типизированным, он добавляет значительную дополнительную работу для разработчиков и риски внедрения новых ошибок, или дублирование любых существующих ошибок. Использование обобщенного метода с параметром типа для параметра метода снимает дублирование кода при сохранении безопасности типов; если определить тип для метода при разработке приложения, приложение не будет компилироваться при попытке использования несовместимого с методом типа.

В следующем примере показан обобщенный метод, обеспечивающий типобезопастную функциональность при одновременном сокращении количества разработки и дублирования кода, а также связанной с ними возможностью появления ошибок.

void AddToQueue<DocumentType>(DocumentType document)

{

printQueue.Add(document);

}

Для определения обобщенного метода используется параметр типа (или несколько параметров), заключенный в угловые скобки, определенный перед списком параметров. При использовании параметров типа необходимо указать идентификатор для каждого параметра типа, требуемого методу. Эти идентификаторы можно использовать в списке параметров метода, типе возвращаемого значения, а также в любом месте в теле метода.

На параметры типа можно добавить ограничения, используя тот же синтаксис, который применяется для ограничения типов для обобщенного класса.

В следующем примере показан обобщенный метод, имеющий два параметра типа, и включающий в себя ограничение на тип параметра ResultType.

ResultType MyMethod<Parameter1Type, ResultType>(Parameter1Type param1)

where ResultType : new()

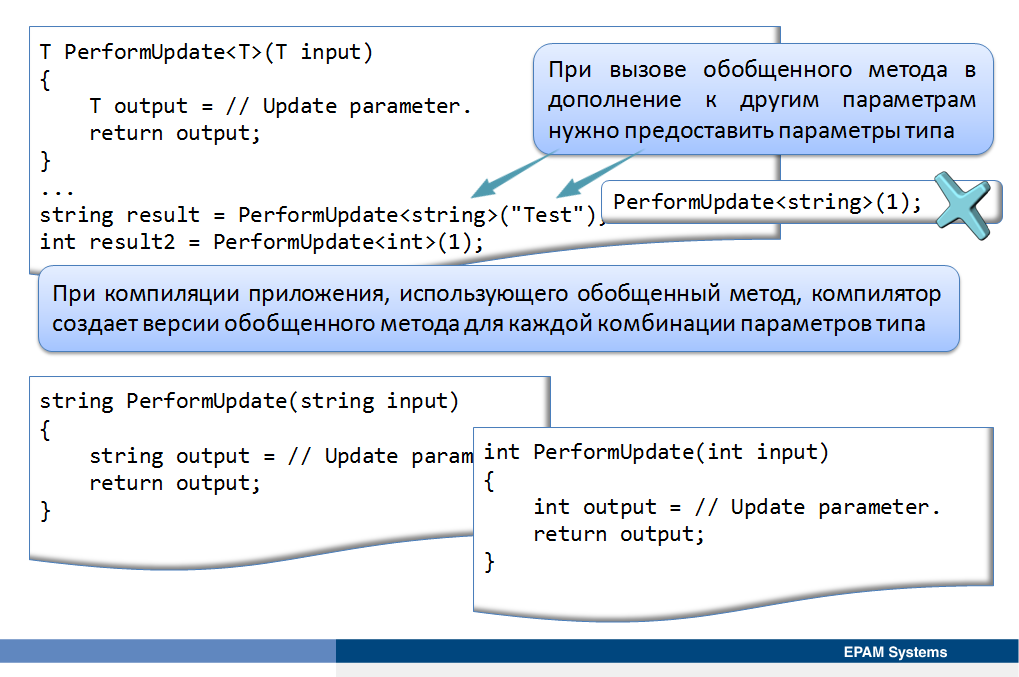
{

ResultType result = new ResultType();

return result;

}

Использование обобщенных методов



Обобщенный метод вызывается как и любой другой метод, кроме того, что в дополнение к другим параметрам нужно предоставить параметры типа. При использовании обобщенного метода, компилятор проверяет типы, используемые методом, для обеспечения типобезопасности. Например, если указать, что используется тип string и передать методу целое число, компилятор сообщит об ошибке, как если бы передавался неправильный параметр в обычном методе в списке параметров. В следующем примере показано, как вызывать обобщенный метод, указав параметр типа.

// Generic method.

T PerformUpdate<T>(T input)

{

T output = // Update parameter.

return output;

}

...

// Generic method usage.

...

string result = PerformUpdate<string>("Test");

int result2 = PerformUpdate<int>(1);

При компиляции приложения, использующего обобщенный метод, компилятор создает версии обобщенного метода для каждой комбинации параметров типа. Следующий пример показывает, как компилятор преобразует обобщенный метод из предыдущего примера кода в неуниверсальный метод.

// Compile time generated methods.

string PerformUpdate(string input)

{

string output = // Update parameter.

return output;

}

int PerformUpdate(int input)

{

int output = // Update parameter.

return output;

}

При компиляции приложения порождаются эквиваленты этих методов в Microsoft Intermediate Language (MSIL); компилятор преобразует обобщенные методы вызов к вызову (calls to calls) в конкретные методы. Вызывать эти методы непосредственно нельзя.

**Итераторы**

Что такое итератор?

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.

Синтаксис и примеры использования итераторов.

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.

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. В итераторе допускается применение нескольких операторов yield. Но каждый такой оператор должен возвращать следующий элемент в коллекции.

***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.

Давайте рассмотрим пример использования итераторов:

using System;

using System.Collections;

namespace ConsoleApplication1

{

class Letter

{

char ch = 'А';

int end;

public Letter(int end)

{

this.end = end;

}

// Итератор, возвращающий end-букв

public IEnumerator GetEnumerator()

{

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

{

if (i == 33) yield break; // Выход из итератора, если закончится алфавит

yield return (char)(ch + i);

}

}

// Создание именованного итератора

public IEnumerable MyItr(int begin, int end)

{

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

{

yield return (char)(ch + i);

}

}

}

class Program

{

static void Main()

{

Console.Write("Сколько букв вывести? ");

int i = int.Parse(Console.ReadLine());

Letter lt = new Letter(i);

Console.WriteLine("\nРезультат: \n");

foreach (char c in lt)

Console.Write(c + " ");

Console.Write("\nВведите пределы\n\nMin: ");

int j = int.Parse(Console.ReadLine());

Console.Write("Max: ");

int y = int.Parse(Console.ReadLine());

Console.WriteLine("\nРезультат: \n");

foreach (char c in lt.MyItr(j,y))

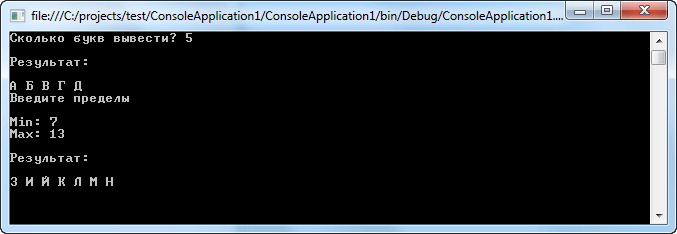
Console.Write(c + " ");

Console.ReadLine();

}

}

}



**Что такое LINQ?**

Большинство приложений выполняют какую-то обработку данных, которая может быть тривиальной. Например, получение списка параметров конфигурации приложения из файла. Однако, обработка данных может быть и сложной, например, если приложение выполняет массовое обновление до 5 миллионов записей в реляционной базе данных. Исторически сложилось так, что логика для выполнения таких операций была тесно связана с архитектурой приложения и структурой данных. Следовательно, если структура данных изменялась, то, как правило, необходимо было сделать значительные изменения в логике обработки данных.

Обработка различных данных, как правило, связана с построением запроса к ним. Запрос представляет собой выражение, получающее данные из источника данных. Запросы обычно выражаются на специальном языке запросов. К настоящему моменту разработаны различные языки для различных типов источников данных, например SQL для реляционных баз данных и XQuery для XML. Таким образом, разработчики вынуждены изучать новый язык запросов для каждого типа источника данных или формата данных, который они должны поддерживать. LINQ упрощает эту ситуацию, предлагая согласованную модель для работы с данными в различных видах источников данных и в различных форматах. В запросе LINQ работа всегда осуществляется с объектами. Для запросов и преобразований данных в XML-документах, базах данных SQL, наборах данных ADO.NET, коллекциях .NET и любых других форматах, для которых доступен поставщик LINQ, используются одинаковые базовые шаблоны кодирования.

Все запросы LINQ состоят из трех различных действий.

* Получение источника данных.
* Создание запроса.
* Выполнение запроса.

В следующем примере показано реализация этих трех частей операции запроса в исходном коде. В примере в качестве источника данных для удобства используется массив целых чисел; тем не менее, те же принципы применимы и к другим источникам данных.

class IntroToLINQ

{

static void Main()

{

// Три составляющих запроса LINQ:

// 1. Получение источника данных.

int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

// 2. Создание запроса.

// переменная запроса numQuery является IEnumerable<int>

var numQuery =

from num in numbers

where (num % 2) == 0

select num;

// 3. Выполнение запроса.

foreach (int num in numQuery)

{

Console.Write("{0,1} ", num);

}

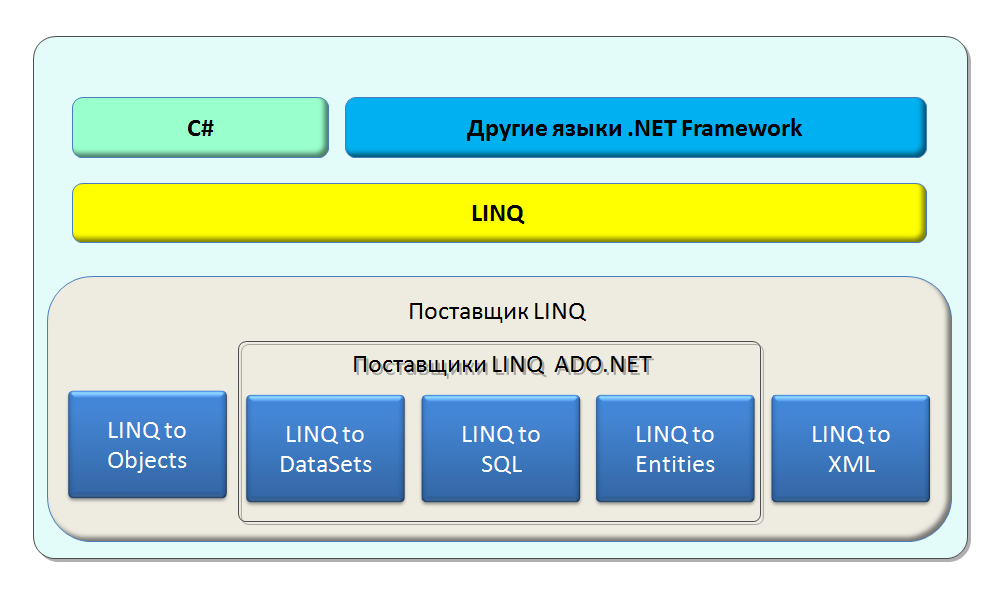
}

}

Запрос указывает, какую информацию нужно извлечь из источника или источников данных. При необходимости, запрос также указывает способ сортировки, группировки и формирования этих сведений перед возвращением. Запрос хранится в переменной запроса и инициализируется выражением запроса.

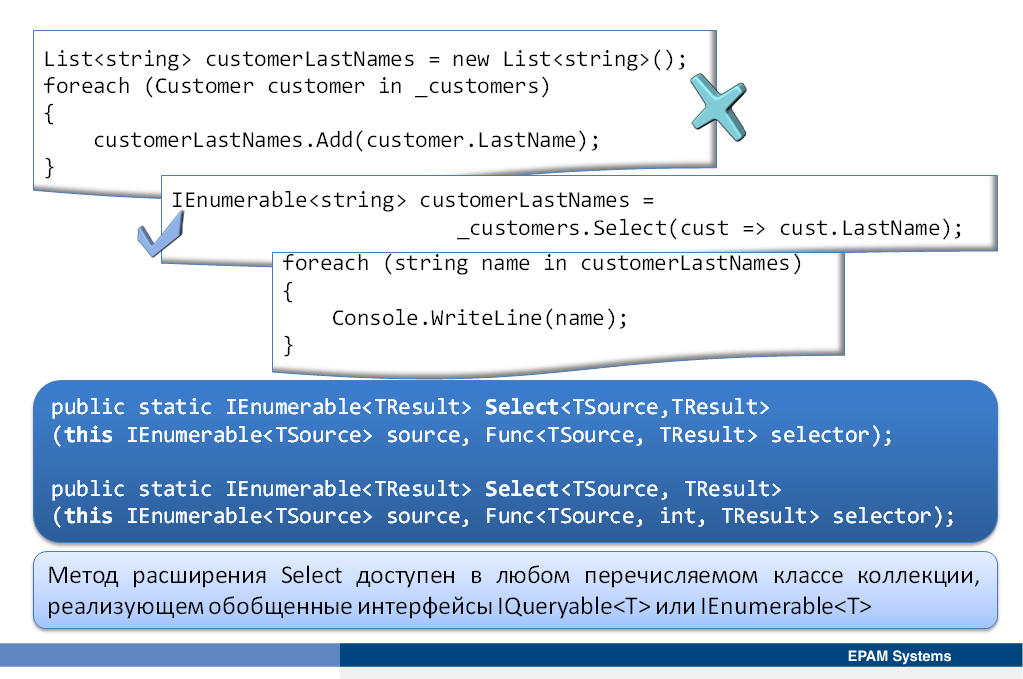
**Цели и задачи LINQ**

Использование LINQ обеспечивает высокоуровневое описание данных, которые должно получить приложение, но при этом точно не указыватся, как извлекаются данные. Поставщик LINQ принимает это описание и генерирует соответствующий код. Поставщик LINQ, по сути, является компонентом, реализующим необходимые интерфейсы, чтобы обеспечить коду возможности запросов. Каждый провайдер LINQ настроен для работы с определенным типом источника данных.



LINQ можно расширить путем создания собственных провайдеров.

Использование LINQ и коллекций



Технически LINQ to Objects – это набор классов, содержащих типичные методы обработки коллекций: поиск данных, сортировка, фильтрация и т.д, происходящие в памяти (in-memory data set). Ядром LINQ to Objects является статический класс Enumerable пространства имен System.Linq (Для использования System.Linq необходимо подключить сборку System.Core.dll). Этот класс содержит набор методов расширения интерфейса IEnumerable<T>.

namespace System.Linq

{

public static class Enumerable...

}

В этом разделе основное внимание уделяется тому, как использовать LINQ функциональность с помощью методов расширения, определеных в пространстве имен System.Linq. Альтернативный подход состоит в использовании выражений запросов LINQ.

Одной из самых основных функций LINQ является способность проектировать данные из коллекции. Например, для того, чтобы получить из списка объектов Customer второй список, содержащий только фамилию каждого клиента, можно использовать оператор foreach для перебора всей коллекции, извлекая необходимые данные, как показано в следующем примере кода.

IEnumerable<Customer> \_customers = new[]

{

new Customer{ FirstName = "Luka", LastName="Abrus", Age = 41},

new Customer{ FirstName = "Syed", LastName="Abbas", Age = 23},

new Customer{ FirstName = "Keith", LastName="Harris", Age = 59},

new Customer{ FirstName = "David", LastName="Pelton", Age = 25},

new Customer{ FirstName = "John", LastName="Peoples", Age = 37},

new Customer{ FirstName = "Toni", LastName="Poe", Age = 29},

new Customer{ FirstName = "Jeff", LastName="Price", Age = 74}

};

List<string> customerLastNames = new List<string>();

foreach (Customer customer in \_customers)

{

customerLastNames.Add(customer.LastName);

}

Гораздо более простым решением будет использование метода расширения Select, который обеспечивает LINQ.

Метод расширения Select доступен на любом перечисляемом классе коллекции, реализующем обобщенные интерфейсы IQueryable<T> или IEnumerable<T>. Простейшая форма метода Select принимает обобщенный делегат, который идентифицирует данные проекта. Самый простой способ для осуществления этого делегата – использовать лямбда-выражение.

public static IEnumerable<TResult> Select<TSource,TResult>(this IEnumerable<TSource>

source,Func<TSource, TResult> selector);

public static IEnumerable<TResult> Select<TSource, TResult>(this IEnumerable<TSource>

source,Func<TSource, int, TResult> selector);

В следующем примере показано использование метода расширения Select для получения списка фамилий из массива объектов Customer.

IEnumerable<string> customerLastNames = \_customers.Select(cust => cust.LastName);

Возвращаемое методом Select значение является ссылкой на перечислимую коллекцию, по которой можно итерироваться для выборки и обработки данных. В следующем примере перебирается коллекция, возвращенная вызовом предыдущего метода Select, а затем отображаются результаты.

foreach (string name in customerLastNames)

{

Console.WriteLine(name);

}

Если необходимо вернуть данные из более чем одного свойства, можно создать анонимный тип. Анонимный тип это тип без названия, которое компилятор генерирует автоматически. Компилятор неявно создает тип, не требующий явного объявления, которое ожидается от стандартного типа, такого как класс. для создания экземпляра анонимного типа необходимо использовать ключевое слово new, за которым следует пара фигурных скобок, в которых определяются имена полей и значения, которые будет содержать тип. Типы полей компилятор выводит из значений, предоставленых для них.

В следующем примере показано использование метода расширения Select для извлечения свойств FirstName и LastName в анонимный тип и возвращения перечислимой коллекции этого типа. Затем выполняется итерация по коллекции для отображения результатов.

var customerNames = \_customers.Select(cust =>

new { FirsName = cust.FirstName, LastName = cust.LastName });

foreach (var customer in customerNames)

{

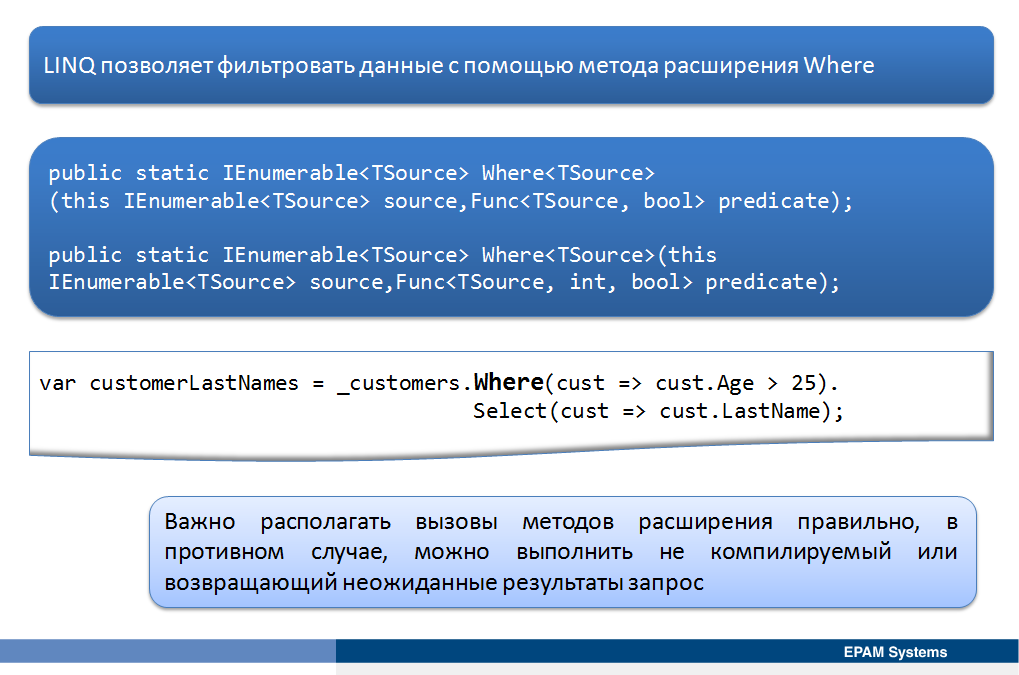
Console.WriteLine("{0} {1}", customer.FirsName, customer.LastName);

}

Именами полей в анонимном типе может быть любой допустимый идентификатор, они не обязаны быть такими же, как имена полей в базовом типе.

В этом примере заранее не известно название типа, который возвращает метод Select, таким образом переменные customerNamesAnonim и customer определены как var.

Фильтрация данных



С помощью LINQ запросов можно фильтровать строки данных и выбирать конкретные поля из типов, можно также возвращать данные в определенном порядке. LINQ предоставляет схожую с оператором OrderBySQL функциональность с помощью методов расширения OrderBy, OrderByDescending, ThenBy и ThenByDescending.

Большинство методов расширения, таких как Select, Where и OrderBy являются обобщенными методами. Это означает, что компилятор работает, определяя типы для использования на основе контекста, но когда существует потенциальная двусмысленность, может понадобиться указать соответствующие параметры типа.

Методы расширения OrderBy и OrderByDescending позволяют сортировать данные по конкретному полю в порядке возрастания или убывания. Как и в других методах расширения, эти два метода ожидают делегат, который определяет поле или выражение для сортировки данных.

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>

(this IEnumerable<TSource> source,

Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>

(this IEnumerable<TSource> source,

Func<TSource, TKey> keySelector,

IComparer<TKey> comparer);

public static IOrderedEnumerable<TSource> OrderByDescending<TSource, TKey>

(this IEnumerable<TSource> source,

Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> OrderByDescending<TSource, TKey>

(this IEnumerable<TSource> source,

Func<TSource, TKey> keySelector,

IComparer<TKey> comparer);

В следующем примере кода показано, как использовать метод расширения OrderBy для сортировки массива объектов Customer по полю FirstName в порядке возрастания.

var sortedCustomers = \_customers.OrderBy(cust => cust.FirstName);

В следующем примере показано использование метода расширения OrderByDescending для сортировки массива объектов по полю FirstName в порядке убывания.

var sortedCustomers = \_customers.OrderByDescending(cust => cust.FirstName);

Хотя методы расширения OrderBy и OrderByDescending позволяют выполнять основные сортировки, иногда можно выполнить дополнительную сортировку в том же операторе. Методы расширения OrderBy и OrderByDescending возвращают объект IOrderedEnumerable, который предоставляет два дополнительных метода расширения: ThenBy и ThenByDescending. Методы расширения ThenBy и ThenByDescending позволяют указать дополнительные ключи сортировки для данных.

public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>

(this IOrderedEnumerable<TSource> source,

Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>

(this IOrderedEnumerable<TSource> source,

Func<TSource, TKey> keySelector,

IComparer<TKey> comparer);

public static IOrderedEnumerable<TSource> ThenByDescending<TSource, TKey>

(this IOrderedEnumerable<TSource> source,

Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> ThenByDescending<TSource, TKey>

(this IOrderedEnumerable<TSource> source,

Func<TSource, TKey> keySelector,

IComparer<TKey> comparer);

В следующем примере кода показано использование методов расширения OrderBy и OrderByDescending. В примере используется метод OrderBy для сортировки результатов по полю FirstName, а затем используется метод ThenBy для сортировки записей по полю Age.

var sortedCustomers = \_customers.OrderBy(cust => cust.FirstName).

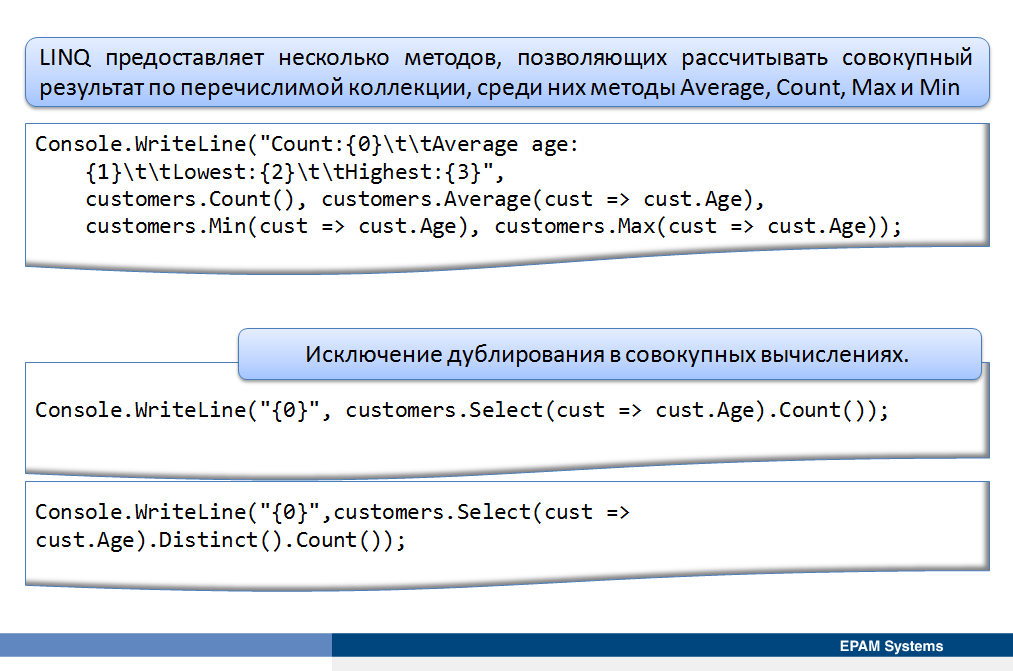
ThenBy(cust => cust.Age);

Метод расширения ThenByDescending позволяет применять дополнительные сортировки последовательности по убыванию.

var sortedCustomers = \_customers.OrderByDescending(cust => cust.FirstName).

ThenByDescending(cust => cust.Age);

Группировка данных и выполнение совокупных вычислений



LINQ предоставляет несколько методов, позволяющих рассчитывать совокупный результат по перечислимой коллекции, среди них методы Average, Count, Max и Min. Методы Average, Count, Max и Min. принимают делегат, указывающий поле, по которому вычисляется совокупное значение. Метод Count принимает делегат, определяющий предикат для вычисления и включает элемент в вычисление только, если предикат возвращает значение true. Как правило метод Count используют для определения общего количество строк в коллекции, и поэтому делегат часто опускается (все совокупные методы перегружены).

В следующем примере показано использование каждого из указанных методов с массивом объектов customer. Пример отображает общее число клиентов, их средний возраст, а также минимальный и максимальный возраст.

IEnumerable<Customer> customers = new[]

{

new Customer{ FirstName = "Luka", LastName="Abrus", Age = 41},

new Customer{ FirstName = "Syed", LastName="Abbas", Age = 23},

new Customer{ FirstName = "Keith", LastName="Harris", Age = 59},

new Customer{ FirstName = "David", LastName="Pelton", Age = 29},

new Customer{ FirstName = "John", LastName="Peoples", Age = 37},

new Customer{ FirstName = "Toni", LastName="Poe", Age = 29},

new Customer{ FirstName = "Jeff", LastName="Price", Age = 74}

};

Console.WriteLine("Count:{0}\t\tAverage age:{1}\t\tLowest:{2}\t\tHighest:{3}",

customers.Count(), customers.Average(cust => cust.Age),

customers.Min(cust => cust.Age), customers.Max(cust => cust.Age));

Совокупные вычисления можно выполнить непосредственно на фоне перечислимой коллекции, но часто требуется вычислить совокупное значение для разных групп данных; например, определить, сколько клиентов внутри определенных возрастных диапазонов. Можно разделить перечислимую коллекцию на группы, используя метод расширения GroupBy. Метод GroupBy ожидает делегат, показывающий, как группируются данные; делегированный метод возвращает селектор, и все элементы, имеющие значения, совпадающие со значением селектора, помещаются в одну группу. После чего можно выполнить совокупные методы по каждой группе.

Значение, возвращаемое методом GroupBy является перечислимой коллекцией объектов, реализующей интерфейс IGrouping. Интерфейс IGrouping представляет коллекцию элементов, имеющих общее значение ключа, и предоставляют доступ к этим ключам, используя свойство Key.

В следующем примере используется метод GroupBy, разделяющий клиентов на группы, определяемые их возрастом. Затем пример отображает общее число клиентов в каждой группе и селектор, определяющий группу. Следует отметить, что значение селектора доступно через свойство группы Key. Пример назначает группы с помощью этого ключа, а также отображает ключ как часть вывода.

var customersGroupedByAgeRange = customers.GroupBy(cust =>

{

if (cust.Age < 20)

return "age < 20";

if (cust.Age >= 20 && cust.Age < 40)

return "age >= 20 and < 40";

if (cust.Age >= 40 && cust.Age < 60)

return "age >= 40 and < 60";

if (cust.Age >= 60)

return "age >= 60";

return "Error";

});

foreach (var cust in customersGroupedByAgeRange.OrderBy(cust => cust.Key))

{

Console.WriteLine("{0}\t\t{1}", cust.Key, cust.Count());

}

Код генерирует следующий результат.

age >= 20 and < 40      4

age >= 40 and < 60      2

age >= 60      1

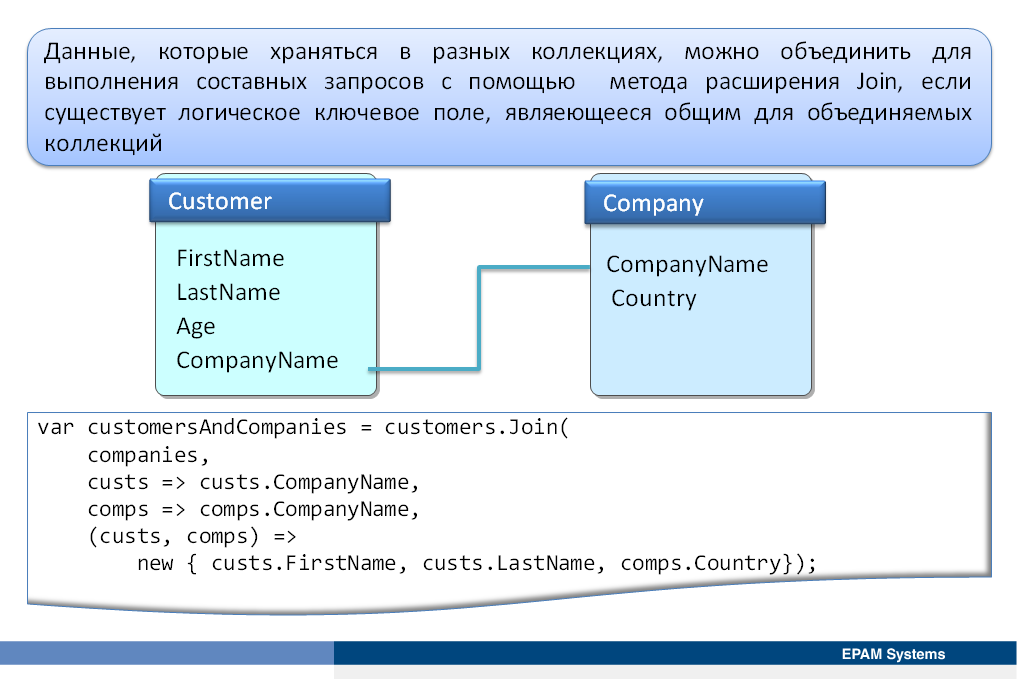
Совокупный методы, такие как Count или Average включает в себя все соответствующие элементы в своих вычислениях. Однако иногда нужно отбросить любые повторяющиеся значения из этих вычислениий. Запрос в следующем примере подсчитывается число возрастов, которые находятся в массиве customers.

Console.WriteLine("{0}", customers.Select(cust => cust.Age).Count());

Если два или более клиентов имеют однин и тот же возраст, этот возраст считается дважды. Исключить повторяющиеся значения можно с помощью метода расширения Distinct перед выполнением агрегации.

Console.WriteLine("{0}",customers.Select(cust => cust.Age).Distinct().Count());

Объединение данных из различных наборов данных



Данные, которые храняться в разных коллекциях, можно объединить для выполнения составных запросов, если существует логическое ключевое поле, являеющееся общим для обеих коллекций. Для этой цели LINQ предоставляет метод расширения Join.

В следующем примере объявляются два массива. Первый содержит информацию о клиентах, включая фамилию, имя, возраст и имя компании, в которой они работают. Второй массив содержит информацию о компаниях, включа название и страну, где находится компания.

IEnumerable<Customer> customers = new[]

{

new Customer{ FirstName = "Luka", LastName="Abrus", Age = 41, CompanyName = "Contoso"},

new Customer{ FirstName = "Syed", LastName="Abbas", Age = 23, CompanyName = "Fabrikam"},

new Customer{ FirstName = "Keith", LastName="Harris", Age = 59, CompanyName = "Contoso"},

new Customer{ FirstName = "David", LastName="Pelton", Age = 41, CompanyName = "Contoso"},

new Customer{ FirstName = "John", LastName="Peoples", Age = 23, CompanyName = "Contoso"},

new Customer{ FirstName = "Toni", LastName="Poe", Age = 29, CompanyName = "Fabrikam"},

new Customer{ FirstName = "Jeff", LastName="Price", Age = 23, CompanyName = "Fabrikam"}

};

IEnumerable<Company> companies = new[]

{

new Company{CompanyName = "Contoso", Country = "United Kingdom"},

new Company{CompanyName = "Fabrikam", Country = "United States"}

};

С помощью метода расширения Join можно получить перечислимую коллекцию, содержащую данные из обоих массивов customers и companies, объединяя их посредством поля CompanyName.

Чтобы использовать метод расширения Join, предоставляютя следующие параметры:

* Перечислимая коллекция, с которой происходит объединение.
* Метод, определяющий общие поля, идентифицирующие метод расширения Select.
* Метод, определяющий общие ключевые поля, на основании которых объединяются различные наборы данных.
* Метод, определяющий поля, требуемые от результирующего перечислимого набора, возвращающемого методом расширения Join.

В следующем примере показано использование метода расширения Join для объединения массивов customers и companies посредством поля CompanyName и отображения фамилии и имени клиента, а также страны, в которой он проживает.

var customersAndCompanies = customers.Join(companies,

custs => custs.CompanyName,

comps => comps.CompanyName,

(custs, comps) => new { custs.FirstName, custs.LastName, comps.Country });

foreach (var item in customersAndCompanies)

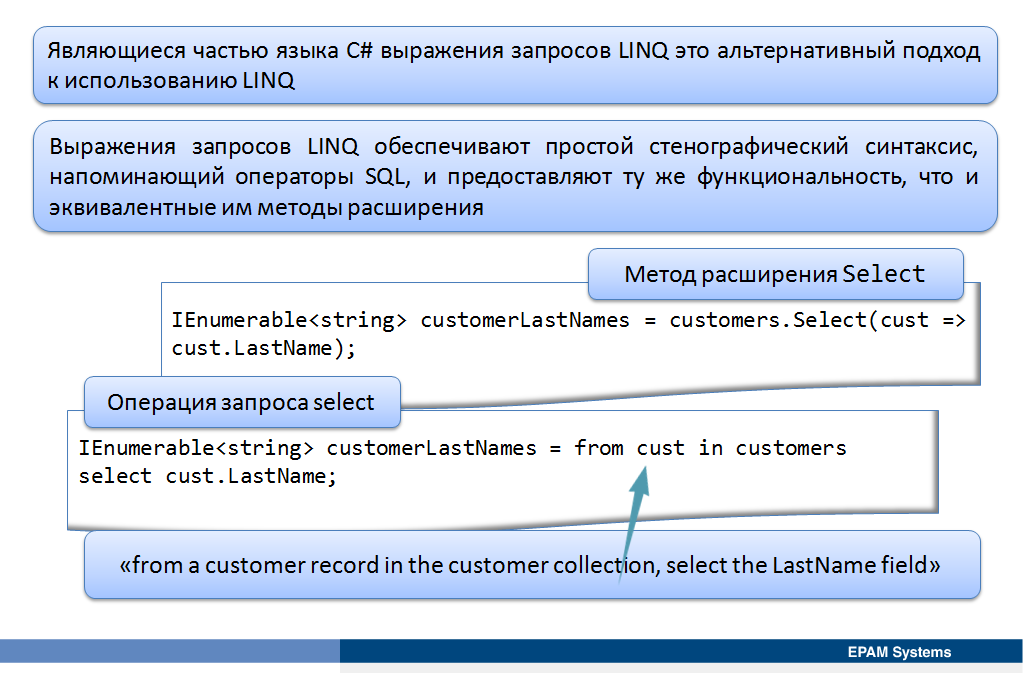
{

Console.WriteLine(item);

}

Метод Join возвращает коллекцию IQueryable, которую можно использовать в сочетании с другими методами расширения LINQ, такими как Select, Where и OrderBy.

Выражения запросов C#



Методы расширения LINQ и типы, определенные в пространстве имен System.Linq, являются очень мощным динамическим средством, однако синтаксис их может быть довольно громоздким, что может легко добавлять в код ошибки, которые трудно обнаружить и исправить.

Разработчики C# признали этот факт и предоставили альтернативный подход к использованию LINQ через выражения запросов LINQ, являющиеся частью языка C#. Выражения запросов LINQ обеспечивают простой стенографический синтаксис, напоминающий операторы SQL; и предоставляют ту же функциональность, что и эквивалентные им методы расширения. Во время компиляции, сокращенный синтаксис выражений запросов компилируется в соответствующий код, использующий методы расширения LINQ.

Например, для выборки списка фамилий из массива объектов Customer можно использовать метод расширения.

IEnumerable<string> customerLastNames = customers.Select(cust => cust.LastName);

Однако, можно использовать выражения запроса from, in и select для достижения того же результата, как показано в следующем примере.

IEnumerable<string> customerLastNames = from cust in customers select cust.LastName;

Подход выражений запросов значительно более интуитивен и прост. Предыдущую операцию можно прочитать следующим обазом «from a customer record in the customer collection, select the LastName field».

Выражение запроса должна подчиняться следующим правилам:

1. Выражение запроса должно начинаться с конструкции from, которая указывает на обрабатываемую коллекцию.
2. Затем выражение запроса может содержать ноль или более конструкции from, let или where. Конструкция let представляет переменную и присваивает ей значение. Конструкция where фильтрует элементы коллекции.
3. Затем выражение запроса может включать ноль или более конструкций orderby, с полями сортировки и необязательным указанием на направление упорядочивания. Направление может быть ascending или descending.
4. Затем следует конструкция select или group.
5. Наконец, в оставшейся части выражения запроса может следовать необязательная конструкция продолжения. Такой конструкцией является into.

В сравнении с SQL, порядок предложения в запросе LINQ отличается; оператор from всегда предшествует оператору select.

Как и в методах расширения в выражениях запроса можно воспользоваться анонимными типами и выводить новые объекты для хранения подмножества полей из оригинального набора данных. В следующем примере показано использование выражения запроса и анонимных типов.

var custs = from cust in customers select new { cust.FirstName, cust.LastName };

Почти для каждого метода расширения LINQ существует эквивалентное выражение запроса. В следующем списке сравнивается синтаксис методов расширения Where, OrderBy, GroupBy и Join и эквивалентных выражений запроса:

Следующие примеры иллюстрируют метод расширения Where и соответствующее выражение запроса where.

//Where extension method example

var customersOver25 = customers.Where(cust => cust.Age > 25);

//where query operator example

var customersOver25 = from cust in customers where cust.Age > 25 select cust;

Следующие примеры иллюстрируют метод расширения OrderBy и соответствующее выражение запроса orderby.

//OrderBy extension method example

var sortedCustomers = customers.OrderBy(cust => cust.FirstName);

//orderby query operator example

var sortedCustomers = from cust in customers orderby cust.FirstName select cust;

Следующие примеры иллюстрируют метод расширения GroupBy и соответствующее выражение запроса groupby.

//GroupBy extension method example

var customersGroupedByAge = customers.GroupBy(cust => cust.Age);

//groupby query operator example

var customersGroupedByAge = from cust in customers group cust by cust.Age;

Следующие примеры кода иллюстрируют метод расширения Join и соответствующее выражение запроса join

//Join extension method example

var customersAndCountries = customers.Join( companies,

cust => cust.CompanyName,

comp => comp.CompanyName,

cust, comp) =>

new {cust.FirstName,cust.LastName,comp.Country});

//join query operator example

var customersAndCountries1 = from cust in customers

join comp in companies on cust.CompanyName

equals comp.CompanyName

select new {cust.FirstName, cust.LastName, comp.Country};

Совокупные вычисления можно выполнять на результатах, которые возвращают LINQ запросы, использующие выражения запросов. Запрос, использующий выражение запроса LINQ, возвращает перечислимое результирующее множество. Для использования совокупных методов, таких, как Count, Max, Min и Distinct, необходимо обернуть запрос LINQ в скобки а затем применить соответствующий метод расширения. В следующем примере показано использование метода расширения Count с простым запросом, использующим выражение запроса.

var customerCount = (from cust in customers select cust).Count();

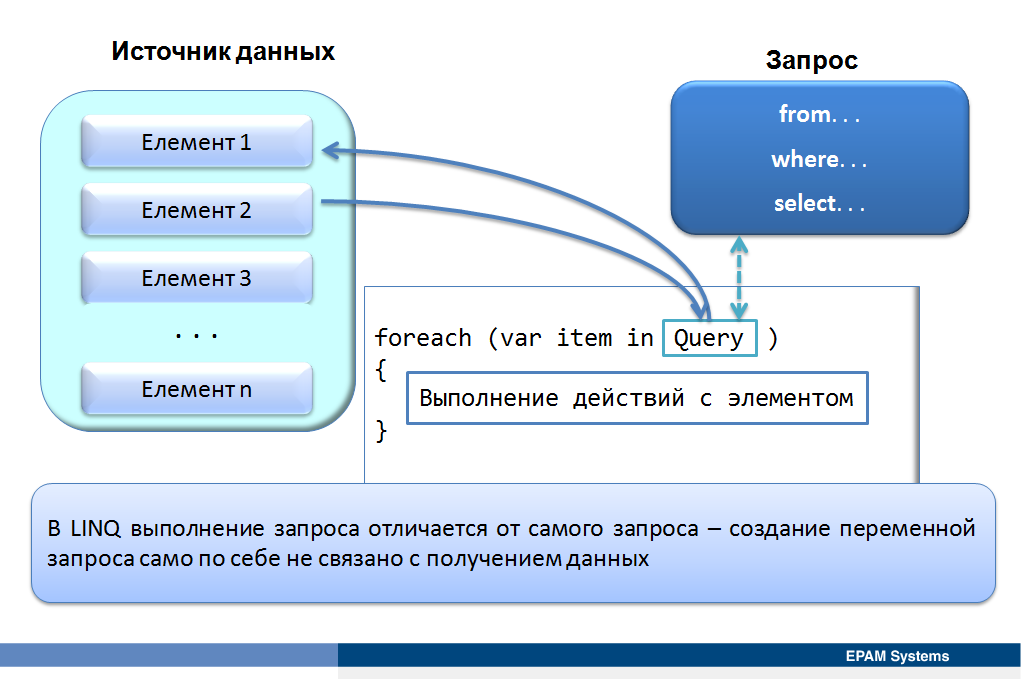
Такой же подход использовать с другими краткими методами расширения. Ниже приведены примеры использования методов расширения Max, Min и Distinct с выражениями запросов.

var maxAge = (from cust in customers select cust.Age).Max();

var minAge = (from cust in customers select cust.Age).Min();

var possibleAges = (from cust in customers select cust.Age).Distinct();

Отложенное и раннее вычисление запросов



При использовании LINQ определить перечислимую коллекцию можно либо с помощью методов расширения LINQ или с помощью операций запросов. Во время выполнения методов расширения LINQ и операций запросов приложение не строит коллекцию; данные получаются только тогда, когда происходит итерирование по коллекции. Это означает, что данные в оригинальной коллекции могут измениться после выполнения запросов LINQ, но до извлечения данных, определяемых запрос; поэтому данные всегда актуальны. Запрос в следующем примере определяет перечислимую коллекцию американских компаний.

var usCompanies = from a in companies where String.Equals(a.Country, "United States") select a.CompanyName;

Данные в массиве companies не будут найдены, а какие-либо условия, указанные в операторе where, не вычислены до тех пор, пока не будет перебирана коллекция usCompanies.

foreach (string name in usCompanies)

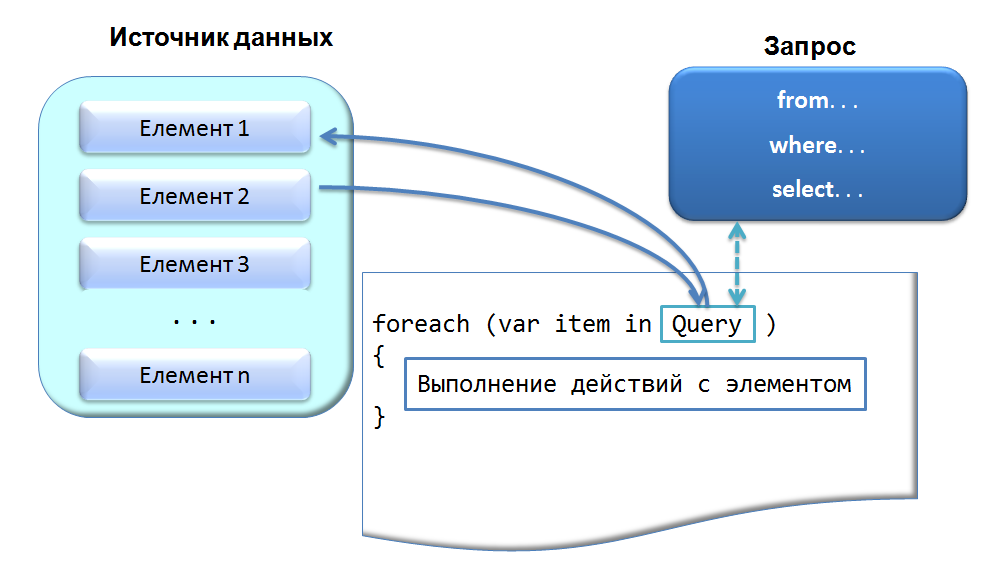
{

Console.WriteLine(name);

}

При изменении данных в массиве companies после определения коллекции usCompanies, но, прежде чем выполнить итерацию по коллекции (например, при добавлении новой компании), данные будут обновлены. Эта стратегия называется отложенным вычислением.

В LINQ выполнение запроса отличается от самого запроса; другими словами, создание переменной запроса само по себе не связано с получением данных. На рисунке показана завершенная операция запроса.



Можно форсировать вычисление запросов LINQ и генерировать статическую, кэшированную коллекцию. Эта коллекция будет копией исходных данных и не изменится при изменении данных в наборе. LINQ предоставляет метод ToList для создания статического объекта List, содержащего кэшированную копию данных.

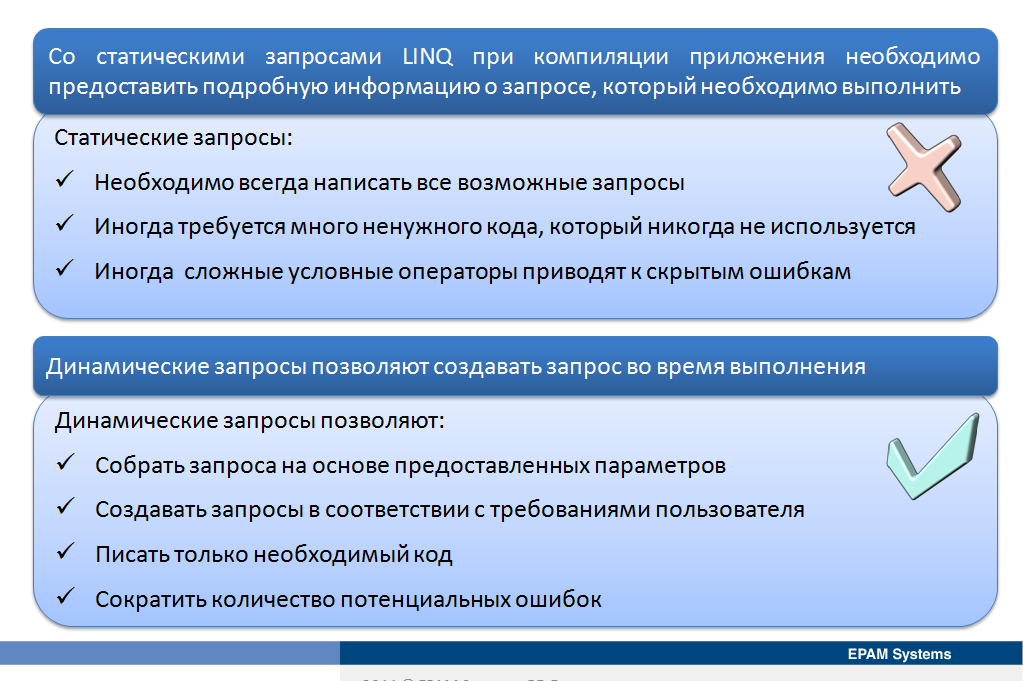
var usCompanies = from a in companies.ToList()

where String.Equals(a.Country, "United States")

select a.CompanyName;

На этот раз, список компаний, является фиксированным при определении запроса. Если добавить в массив companies больше американских компаний, они не появятся при переборе коллекции usCompanies. LINQ также предоставляет метод ToArray, который сохраняет в кэше коллекцию в виде массива.

Динамический запрос LINQ



LINQ предоставляет очень мощный механизм, позволяющий отделить бизнес-логику приложения от логики, необходимой для получения данных, которые использует приложение. В большинстве ситуаций можно использовать статические запросы LINQ, но могут быть и случаи, когда критерии запроса, порядок, в котором требуется данные, или даже получаемые данные не известны до времени выполнения. Например, форма запроса может зависеть от ввода пользователя или результатов некоторых других видов обработки.

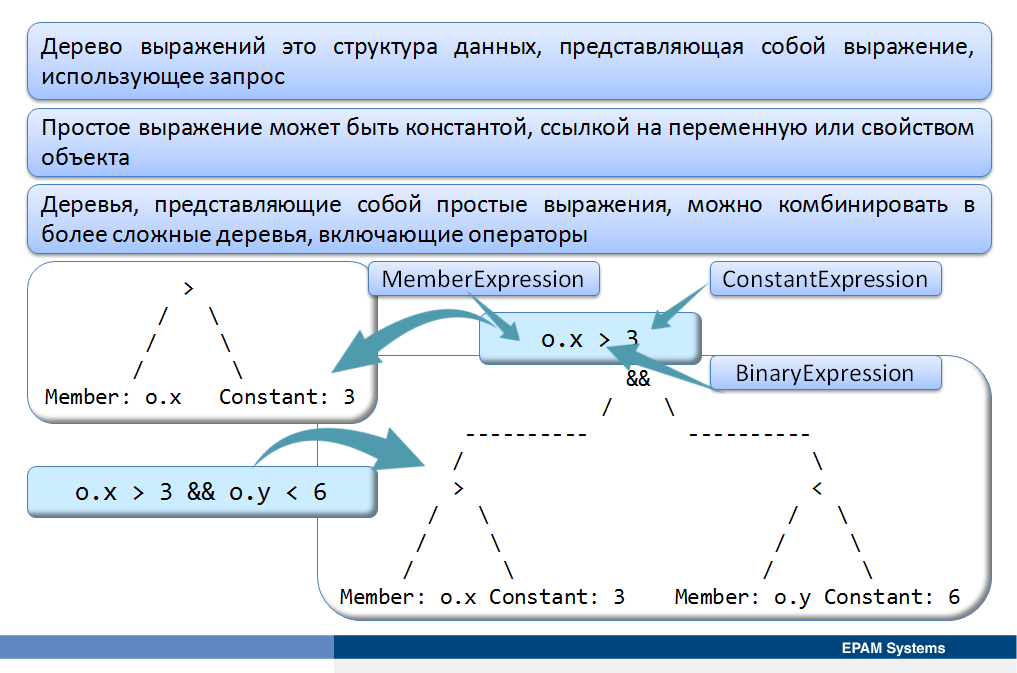
Например, может потребоваться запросить таблицы базы данных, в которой хранятся результаты тестов, выполненных студентами. Эта таблица может содержать поля, хранящие имя студента, название теста и оценку теста для этого студента. С помощью этой простой схемы, можно разрабатывать приложения для запроса базы данных и применять фильтры к данным. Простые запросы можно выполнять с помощью статического LINQ.

Однако предположим, что нужно сделать приложение более гибкими. Например, если нужно отфильтровать результаты для студентов, получивших оценки в определенном диапазоне для одного или нескольких тестов и отсортировать их по совокупности ключей (по возрастанию/убыванию названия теста, оценке, имени студента или любому другому сочетанию).

В некоторых случаях можно получить все три поля из базы данных, но в других случаях, может существовать заинтересованность только в подмножестве полей. Один из вариантов заключается в написании нескольких запросов, а затем использовании операторов if или switch для выбора запроса для использования, однако такой подход влечет за собой написание множества кода, что увеличивает риск появления ошибок. Другой вариант состоит в извлечении всех данных коллекции в память, а затем программном получении соответствующих данных из этой коллекции, но такой подход также потенциально подвержен ошибкам и не использует всю мощь LINQ.

Использование динамических запросов LINQ помогает решить эту проблему. При разработке динамических запросов LINQ представление запроса строится в виде дерева выражений (expression tree), которое компилируется во время выполнения, а затем выполняется. Тогда запросы точно представляют потребности пользователя без необходимости разработки нескольких запросов или обработки данных вручную в памяти.

Дерево выражений



Дерево выражений это структура данных, представляющая собой выражение, использующее запрос. Простое выражение может быть константой, ссылкой на переменную или свойством объекта. Можно комбинировать деревья, представляющие собой простые выражения, в более сложные деревья, включающие операторы.

Например, выражение o.x > 3 состоит из двух меньших выражений (переменной-члена x объекта o и постоянной 3), объединенных в большее выражение с помощью операции > (больше чем). Следующая диаграмма показывает, каким образом можно себе представить выражение.

           >

         /   \

        /     \

       /       \

Member: o.x   Constant: 3

Соответствующее дерево выражений для этого выражения состояит из объекта MemberExpression, который ссылается на поле x объекта o, и объекта ConstantExpression, представляющего постоянное значение 3; эти объекты объединены в объект BinaryExpression, сравнивающего их с помощью операции > (больше чем). Дерево выражений может представлять постоянную, член или комбинировать деревья выражений вместе с помощью операций. Так, например, дерево выражения в следующей диаграмме представляет выражение o.x > 3 && o.y < 6

                       &&

                     /    \

          ----------        ----------

         /                            \

         >                            <

       /   \                        /   \

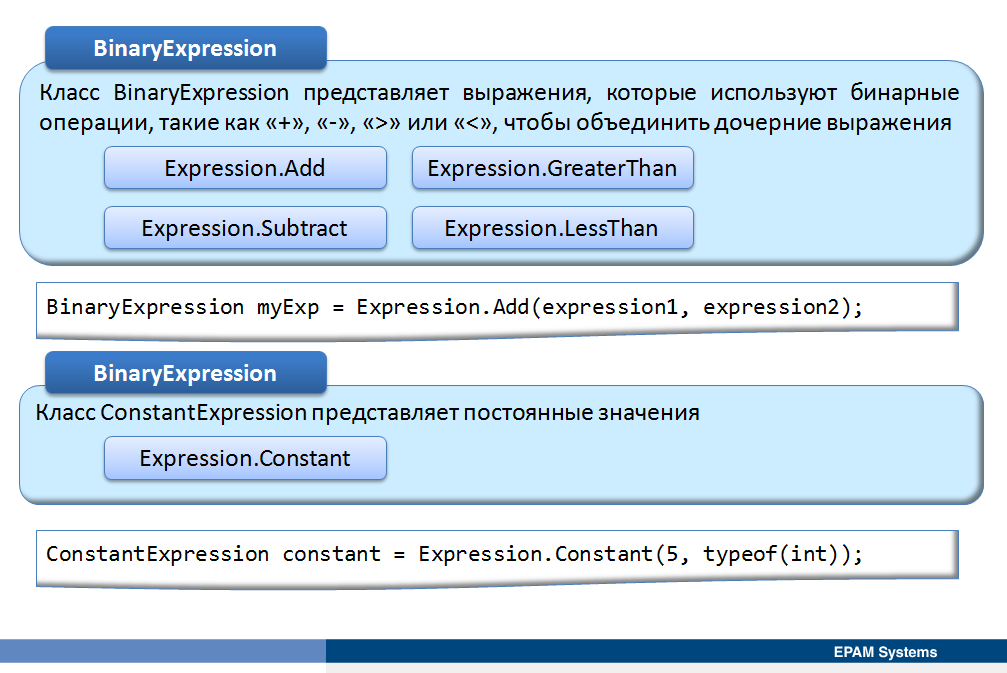
      /     \                      /     \

     /       \                    /       \

Member: o.x Constant: 3    Member: o.y Constant: 6

Деревья выражений обеспечивают очень гибкий механизм для разработки выражений, которые можно использовать как часть запроса LINQ. Можно строить деревья выражений, представляющие лямбда-выражения, на которые затем можно сослаться в запросе LINQ. Можно скомпилировать дерево выражения, представляющее лямбда-выражение во время выполнения, а затем вызывать его так же, как обычное лямбда-выражение.

Типы выражений



Каждый узел в дереве выражений является объектом выражения (expression object). Пространство имен System.Linq.Expressions в библиотеке классов .NET Framework определяет типы выражений, которые можно использовать для представления любого допустимого выражения C#. Все эти типы наследуются от класса Expression пространства имен System.Linq.Expressions.

При разработке WPF приложений следует знать, что WPF определяет тип Expression. Поэтому, если необходимо использовать тип Expression из пространства имен System.Linq.Expressions, полезно создать псевдоним для пространства имен и использовать псевдоним, чтобы избежать двусмысленности.

В следующем примере кода показано, как добавить псевдоним для пространства имен System.Linq.Expressions имени Expressions.

using Expressions = System.Linq.Expressions;

При построении простых выражений используется соответствующий тип выражения. Простые выражения можно комбинировать в более сложные, используя класс Expressions. Класс Expressions это абстрактный тип, действующий как родитель всех типов выражения. Класс Expressions обеспечивает большое количество статических фабрик-методов для объединения выражений вместе в новые деревья выражений с помощью любых операций, доступных в языке C#. Например, метод-фабрика Expression.Add[[4]](#footnote-5) принимает два объекта Expression и, объединяя их с помощью оператора сложения (+), создает новое дерево выражений.

**Класс BinaryExpression.** Класс BinaryExpression представляет выражения, которые используют бинарные операции, такие как «+», «-», «>» или «<», чтобы объединить дочерние выражения. Такие методы, как Expression.Add, Expression.Subtract, Expression.GreaterThan и Expression.LessThan возвращают объект BinaryExpression. В следующем примере показано создание объекта BinaryExpression, складывающего два значения.

// Assume that expression1 and expression2 are expression tree objects.

BinaryExpression myExp = Expression.Add(expression1, expression2);

**Класс ConstantExpression.** Класс ConstantExpression представляет постоянные значения. Для создания объекта ConstantExpression используется статический метод Expression.Constant. При этом необходимо предоставить значение и, возможно, тип System.Type объекта в качестве аргументов для метода Constant. Следует обратить внимание, что можно использовать оператор typeof для возвращения типа. В следующем примере показано определение выражения ConstantExpression для целого значения 5.

ConstantExpression constant = Expression.Constant(5, typeof(int));

**Класс MemberExpression.** Класс MemberExpression позволяет обращаться к свойствам или полям объекта. Например, можно использовать объект MemberExpression для представления выражения x.member в бинарном выражении x.member > 20.

Для создания экземпляра класса MemberExpression используется статический метод MakeMemberAccess класса Expression. Метод MakeMemberAccess принимает два аргумента: ссылку на объект, который содержит член в форме объекта Expression и ссылку на себя в форме объекта System.Reflection.MemberInfo. Объект Expression, представляющий объект, может быть ссылкой на параметр, передаваемый в лямбда-выражение. или Constant выражением, если нужно сослаться на объект, который был создан за пределами лямбда-выражения.

Можно использовать отражение для создания объекта MemberInfo, представляющего член объекта.

В следующем примере кода показано создание объекта MemberExpression для доступа к свойству объекта myData типа MyType.

// Assume that the propertyInfo object is a valid instance of the

// MemberInfo class that references a field that the MyType type exposes.

MyType myData = ...;

MemberExpression member = Expression.MakeMemberAccess(Expression.Constant(myData),

propertyInfo);

Иногда может потребоваться доступ к статическому члену типа. В этом случае можно указать ссылку null на конкретный экземпляр объекта в качестве первого параметра метода MakeMemberAccess. В следующем примере кода показано создание объекта MemberExpression для доступа к статическому свойству.

// Assume that the propertyInfo object is a valid MemberInfo object

// that references a static property of a type.

MemberExpression staticProperty = Expression.MakeMemberAccess(null, propertyInfo);

**Класс UnaryExpression.** Класс UnaryExpression представляет выражения, основаные на унарной операции. Пример унарной операции – отрицание значения. Чтобы создать унарное выражение, инвертирующее значение, используется статический метод Negate класса Expression.

Expressions.UnaryExpression negation = Expressions.Expression.Negate(parameter);

Для создания объекта UnaryExpression также можно использовать следующие методы:

* Метод ArrayLength создает выражение, возвращающее длину одномерного массива, передаваемого как параметр.
* Метод Convert создает выражение, преобразующее объект, первый параметр которого указывает на тип, который задается с помощью объекта System.Type в качестве второго параметра.
* Метод ConvertChecked создает выражение, преобразуюшее объект, первый параметр которого указывает на тип, который задается с помощью объекта System.Type в качестве второго параметра, и выполняет проверку переполнения преобразования
* Метод Negate создает выражение, которое инвертирует значение выражения.
* Метод NegateChecked создает выражение, которое инвертирует значение выражения и проверяет числовое переполнение.
* Метод Not создает выражение, которое выполняет побитовую операцию NOT по параметру.
* Метод Quote создает выражение, которое возвращает постоянное значение типа параметра.
* Метод TypeAs создает выражение, выполняюшее прямую ссылку или упаковку преобразования и возвращает нуль, если преобразование не удалось.
* Метод UnaryPlus создает выражение, выполняюшее операцию унарный плюс.

**Класс Expression<TDelegate>.** Лямбда-выражения содержат два основных элемента: список параметров и тело, которое может вернуть значение, основаное на вычислении, включающем в себя эти параметры. В следующем примере показано лямбда-выражение, принимающее параметр x. Тело лямбда-выражения это бинарное выражение, основанное на этом параметре.

x => x > 2

Для построения дерева выражений используется обобщенный тип Expression<TDelegate>, представляющий лямбда-выражение. Параметр типа TDelegate должнен ссылаться на делегат, соответствующий сигнатуре лямбда-выражения. В приведенном выше примере можно использовать тип Func<int,bool>, представляющий делегат, который принимает один целочисленный параметр и возвращает логическое значение.

Для построения дерева выражений используется метод Expression.Parameter, представляющий параметр лямбда-выражений. Этот метод предполагает тип параметра и имя в качестве аргументов. Затем можно построить объект BinaryExpression, который ссылается этот параметр и выполняет указанное вычисление. Наконец, можно объединить эти два дерева выражения в лямбда-выражение с помощью обобщенного метода Expression.Lambda, указывая тип делегата, на который ссылается лямбда-выражение как на параметр типа. В следующем примере показано построкемк дерева выражения для лямбда-выражения предыдущего примера.

Expression<Func<int, bool>> lambda = null;

ParameterExpression param = Expression.Parameter(typeof(int), "x");

ConstantExpression two = Expression.Constant(2, typeof(int));

BinaryExpression body = Expression.GreaterThan(param, two);

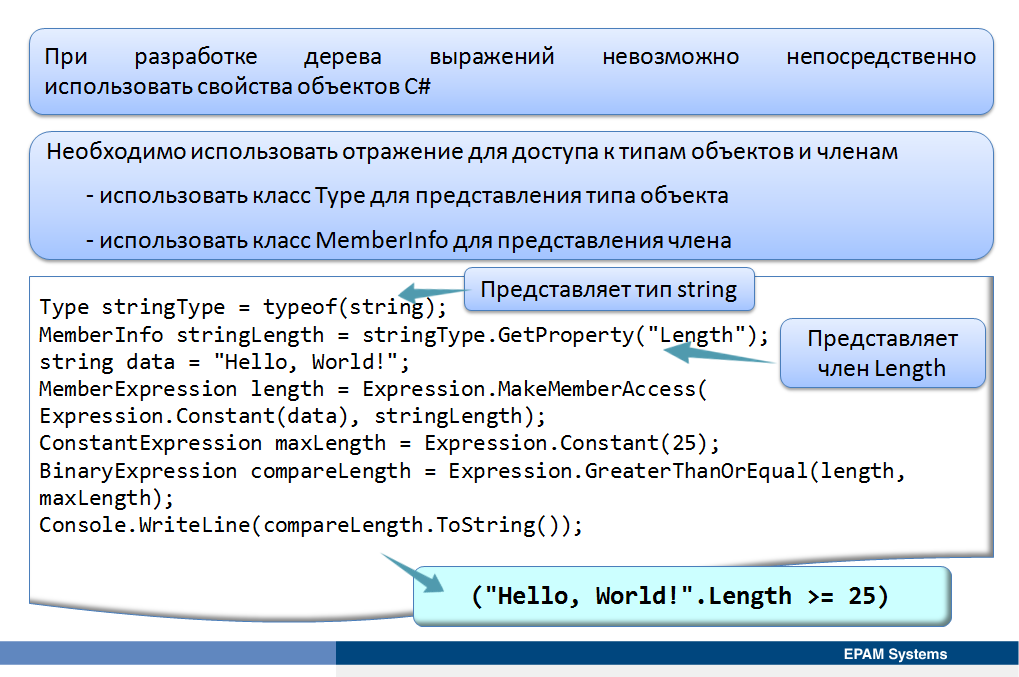
lambda = Expression.Lambda<Func<int, bool>>(body, param);

Console.WriteLine(lambda.ToString());

В следующем примере показан вывод, который генерирует этот код.

x => (x > 2)

Получение сведений о типах во время выполнения



C# компилятор обеспечивает безопасность типов и генерирует ошибки компиляции, если код не типизирован. При использовании деревьев выражений безопасность типа проверяется во время выполнения после построения выражения, но до вычисления. Некоторые методы Expression, такие как MakeMemberAccess, ожидают предоставления сведений о типе члена, на который ссылаются. Другие методы, такие как Expression.Constant и Expression.Parameter, ожидают предоставления информации о типе постоянной или параметре, участвующих в выражении. Для получения такого рода информации динамически можно использовать отражение.

Самым простым способом получения информации о типе является использование оператора typeof. Этот оператор ожидает тип в качестве аргумента и возвращает объект System.Type, содержащий подробные метаданные, описывающие этот тип. Общеязыковая среда выполнения (CLR) может использовать эти метаданные для выполнения проверки типов.

Использование отражения для получения информации о типе является вычислительно дорогостоящей операцией. Если нужно сослаться на тип только один раз, можо использовать ключевое слово typeof, однако, если нужно использовать тот же тип несколько раз, необходимо создать экземпляр типа Type и использовать этот экземпляр. В следующем примере показано создание экземпляра типа Type для представления типа string.

Type stringType = typeof(string);

Можно ссылаться на объект stringType, указав тип объекта ParameterExpression, как показано в следующем примере кода.

ParameterExpression param = Expression.Parameter(stringType, "data");

Класс Type позволяет ссылаться на тип, однако часто нужен доступ к членам объекта, а не к объекту. Для ссылки на член типа, можно использовать класс MemberInfo. Класс MemberInfo определен в пространстве имен System.Reflection. Чтобы создать объект MemberInfo можно использовать методы экземпляра, которые предоставляет класс Type. Есть большое количество методов, в том числе GetField и GetProperty, которые можно использовать для получения информации о полях и свойствах, предоставляемых типом. Многие из этих методов требуют указать имя члена в виде строки. Однако, если предоставляется неверное имя члена, метод будет возвращать null.

В следующем примере показано создание объекта MemberInfo для представления свойства Length объекта string. Можно использовать объект stringLength как аргумент MemberInfo метода MakeMemberAccess и построить дерево выражений, которое сравнивает длину строки с определенным значением.

Type stringType = typeof(string);

MemberInfo stringLength = stringType.GetProperty("Length");

string data = "Hello, World!";

MemberExpression length = Expression.MakeMemberAccess(

Expression.Constant(data), stringLength);

ConstantExpression maxLength = Expression.Constant(25);

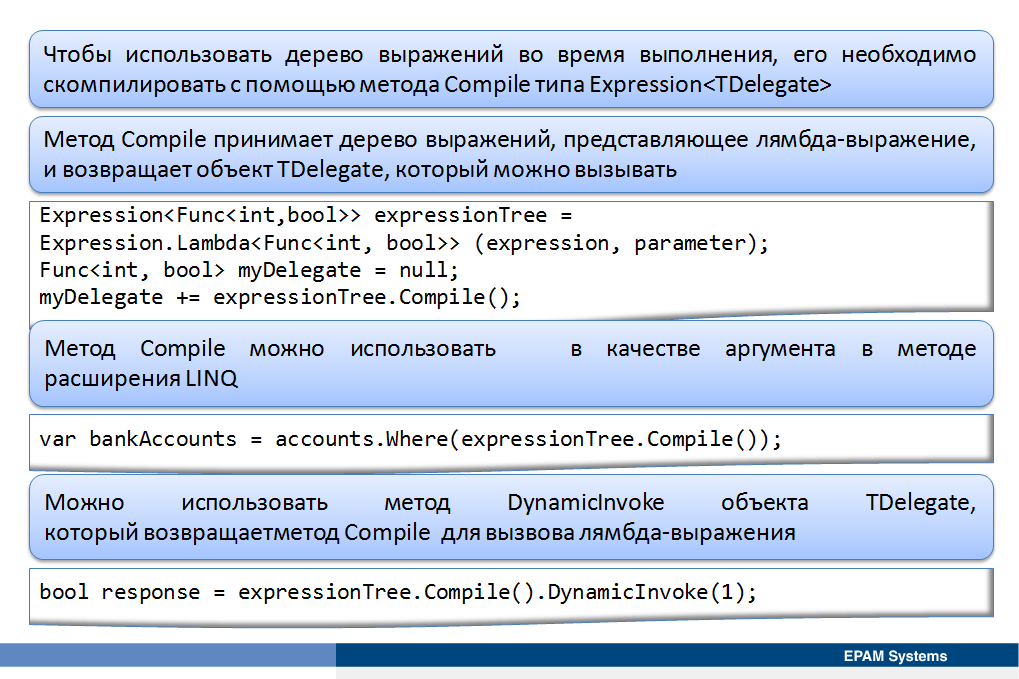
BinaryExpression compareLength = Expression.GreaterThanOrEqual(length, maxLength);

Console.WriteLine(compareLength.ToString());

Ниже показан вывод, который генерирует этот код.

("Hello, World!".Length >= 25)

Компиляция и выполнение динамических запросов LINQ



Дерево выражений является структурой данных. Чтобы использовать дерево выражений во время выполнения, его необходимо скомпилировать. Добиться этого можно с помощью метода Compile типа Expression<TDelegate>. Метод Compile принимает дерево выражений, представляющее лямбда-выражение, и возвращает объект TDelegate, который можно вызывать. Если лямбда-выражение определяет элемент LINQ запроса, можно просто перебирать результаты, которые запрос возвращает для выполнения выражения.

Использование динамических лямбда-выражений не ограничивается LINQ запросами, построенные лямбда-выражения можно использовать динамически везде, где возможно использовать обычные, статические лямбда-выражения. Для выполнения динамического лямбда-выражения в таких ситуациях, можно использовать метод DynamicInvoke объекта TDelegate, который возвращает метод Compile.

В следующем примере показано создание динамичного LINQ запроса и результаты, полученные при его выполнении. В примере перебираются объекты TestScore, которые хранятся в коллекции List<TestScore>, и извлекаются все результаты тестов с отметкой более чем 50 и отображает имена кандидатов с этими баллами в алфавитном порядке.

class TestScore

{

public int Score { get; set; }

public string Name { get; set; }

}

class Program

{

static void Main(string[] args)

{

List<TestScore> scores = new List<TestScore>()

{

new TestScore {Score = 90, Name = "Mike"},

new TestScore {Score = 60, Name = "Louisa"},

new TestScore {Score = 85, Name = "Antony"},

new TestScore {Score = 100, Name = "Richard"},

new TestScore {Score = 45, Name = "Jason"},

new TestScore {Score = 35, Name = "Tom"},

new TestScore {Score = 96, Name = "Chris"},

new TestScore {Score = 26, Name = "Adam"},

new TestScore {Score = 71, Name = "Charles"},

new TestScore {Score = 91, Name = "Alison"},

new TestScore {Score = 34, Name = "John"}

};

var passes1 = scores.Where(testResult => testResult.Score > 50).

OrderBy(testResult => testResult.Name).

Select(testResult => testResult.Name);

// Run the query and display the results.

foreach (var pass in passes1)

{

Console.WriteLine(pass);

}

Console.WriteLine("==========================================");

Console.WriteLine("Compiling and Running a Dynamic LINQ Query");

Console.WriteLine("==========================================");

/\* The following code generates a LINQ query that is equivalent

to the following:

var passes = scores.Where(testResult => testResult.Score > 50).

OrderBy(testResult => testResult.Name).

Select(testResult => testResult.Name);

**Note that this query involves three lambda expressions:**

1. The lambda expression for the Where method takes a TestScore

object and returns a Boolean value.

2. The lambda expression for the OrderBy method takes a TestScore

object and returns a string (the data to order by).

3. The lambda expression for the Select method takes a TestScore

object and returns a string (the candidate names).

\*/

// Build the lambda expression for the Where method:

// testResult => testResult.Score > 50

Type testScoreType = typeof(TestScore);

ParameterExpression testResultParam = Expression.Parameter(testScoreType, "testResult");

MemberInfo scoreProperty = testScoreType.GetProperty("Score");

MemberExpression valueInScoreProperty = Expression.MakeMemberAccess(testResultParam, scoreProperty);

ConstantExpression fifty = Expression.Constant(50, typeof(int));

BinaryExpression scoreGreaterThanFifty = Expression.GreaterThan(valueInScoreProperty, fifty);

Expression<Func<TestScore, bool>> whereExpression =

Expression<Func<TestScore, bool>>.Lambda<Func<TestScore, bool>>(scoreGreaterThanFifty, testResultParam);

// Build the lambda expression for the OrderBy method: testResult => testResult.Name

MemberInfo nameProperty = testScoreType.GetProperty("Name");

MemberExpression valueInNameProperty = Expression.MakeMemberAccess(testResultParam, nameProperty);

Expression<Func<TestScore, string>> orderByExpression =

Expression<Func<TestScore, string>>.Lambda<Func<TestScore, string>>(valueInNameProperty, testResultParam);

// Build the lambda expression for the Select method: testResult => testResult.Name

Expression<Func<TestScore, string>> selectExpression =

Expression<Func<TestScore, string>>.Lambda<Func<TestScore, string>>(valueInNameProperty, testResultParam);

// Compile the lambda expressions, starting with the Where expression.

IEnumerable<TestScore> passingScores = scores.Where(whereExpression.Compile());

// Now append the OrderBy expression.

passingScores = passingScores.OrderBy(orderByExpression.Compile());

// Finally, add the Select expression.

IEnumerable<string> passes = passingScores.Select(selectExpression.Compile());

// Run the query and display the results.

foreach (var pass in passes)

{

Console.WriteLine(pass);

}

Console.ReadLine();

}

}

1. При получении объектов из коллекции необходимо необходимо следить за правильным приведением объектов, и избегать, где это возможно, добавления объектов различных типов одной и той же коллекции. [↑](#footnote-ref-2)
2. Класс ArrayList это простой класс коллекция, реализующий массив, который может динамически изменять свой размер. [↑](#footnote-ref-3)
3. Можно определить обобщенные структуры в дополнение к обобщенным классам. [↑](#footnote-ref-4)
4. В модуле будут описаны некоторые из наиболее часто используемых типов выражений и статических методов-фабрик, которые используются для их построения. В пространстве имен System.Linq.Expressions существует множество других доступных типов выражений. [↑](#footnote-ref-5)