|  |
| --- |
| , RD Dep. |
| Конспект и раздаточный материал  NET.C#.11 Коллекции и обобщенные типы |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial version | Анжелика Кравчук |  |  |  |
| 1.1 | Review and corrections. | Владимир Тихон |  |  |  |

Contents

[1. Урок 1: Использование коллекций 3](#_Toc301440989)

[1.1. Что такое коллекция? 3](#_Toc301440990)

[1.2. Использование классов коллекций 4](#_Toc301440991)

[1.3. Итерация по коллекции 6](#_Toc301440992)

[1.4. Базовые классы коллекции 7](#_Toc301440993)

[1.5. Использование инициализаторов коллекции 9](#_Toc301440994)

[1.6. Демонстрация: Использование коллекций 10](#_Toc301440995)

[2. Урок 2: Создание и использование обобщенных типов 10](#_Toc301440996)

[2.1. Что такое обобщенный тип? 11](#_Toc301440997)

[2.2. Компиляция обобщенных типов и безопасность типов 12](#_Toc301440998)

[2.3. Определение пользовательских обобщенных типов 13](#_Toc301440999)

[2.4. Добавление ограничений для обощенных типов 14](#_Toc301441000)

[2.5. Демонстрация: Определение обобщенного типа 16](#_Toc301441001)

[3. Урок 3: Определение обобщенных интерфейсов, вариантность 16](#_Toc301441002)

[3.1. Определение обобщенных интерфейсов 17](#_Toc301441003)

[3.2. Что такое инвариантность? 18](#_Toc301441004)

[3.3. Определение и реализация ковариантного интерфейса 19](#_Toc301441005)

[3.4. Определение и реализация контравариантного интерфейса 21](#_Toc301441006)

[4. Урок 4: Использование обобщенных методов и делегатов 22](#_Toc301441007)

[4.1. Определение обобщенного метода 23](#_Toc301441008)

[4.2. Использование обобщенных методов 24](#_Toc301441009)

[4.3. Использование обобщенных делегатов .NET Framework 25](#_Toc301441010)

[4.4. Демонстрация: Обобщенные методы и делегаты 26](#_Toc301441011)

[5. Урок 5: Реализация пользовательского класса коллекции 26](#_Toc301441012)

[5.1. Пользовательские классы коллекции 27](#_Toc301441013)

[5.2. Обобщенные интерфейсы коллекций .NET Framework 28](#_Toc301441014)

[5.3. Обобщенные классы коллекции .NET Framework 30](#_Toc301441015)

[5.4. Демонстрация: Обобщенные классы коллекции .NET Framework 37](#_Toc301441016)

[5.5. Реализация класса коллекции DoubleEndedQueue 37](#_Toc301441017)

[5.6. Реализация класса коллекции Dictionary 40](#_Toc301441018)

[5.7. Демонстрация: Реализация пользовательского класса коллекции 43](#_Toc301441019)

[6. Урок 6: Добавление перечислителя пользовательскому классу коллекции 43](#_Toc301441020)

[6.1. Что такое перечислитель? 44](#_Toc301441021)

[6.2. Что такое интерфейс IEnumerable<T>? 45](#_Toc301441022)

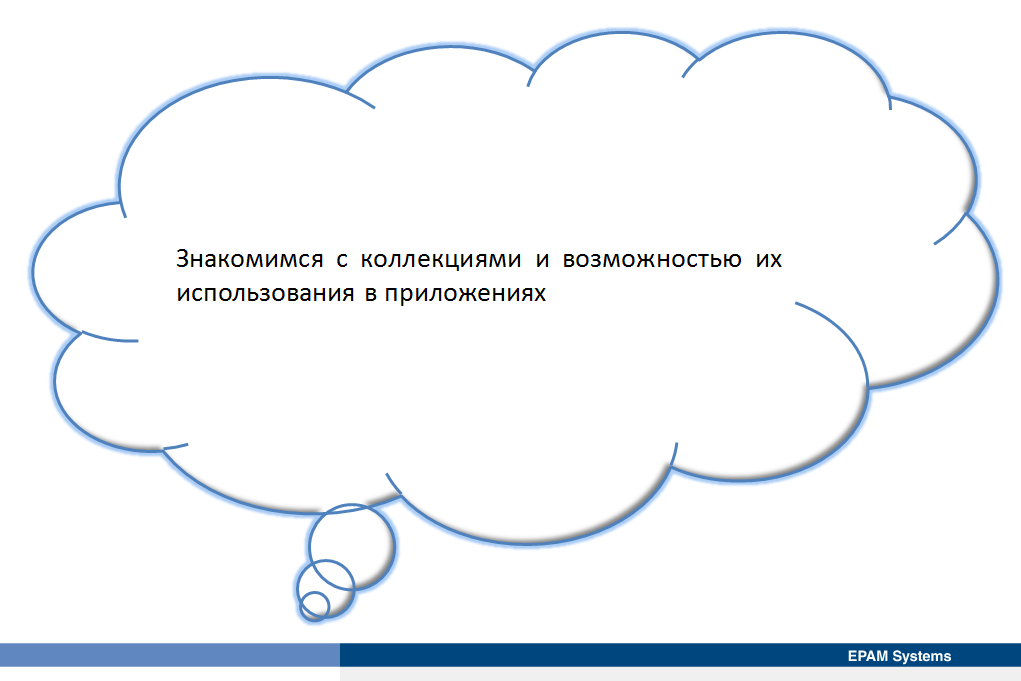
[6.3. Что такое интерфейс IEnumerator<T>? 46](#_Toc301441023)

[6.4. Реализация перечислителя вручную 48](#_Toc301441024)

[6.5. Реализация перечислителя с помощью итератора 49](#_Toc301441025)

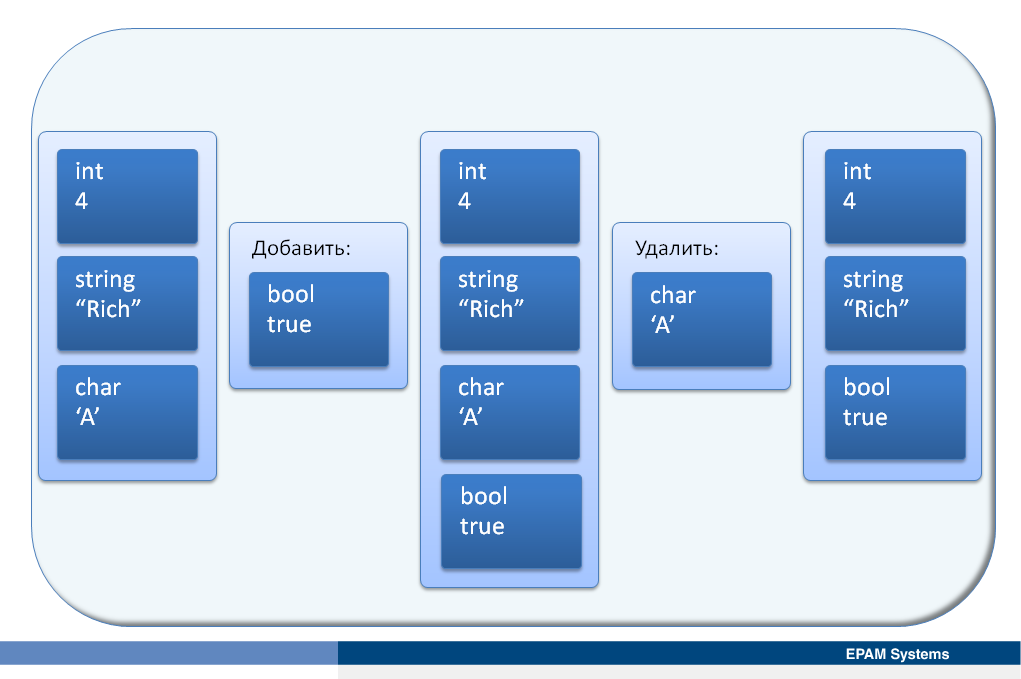
[6.6. Демонстрация: Добавление перечислителя пользовательскому классу коллекции 52](#_Toc301441026)

# Урок 1: Использование коллекций



Массивы можно использовать для группирования данных и хранения нескольких связанных экземпляров объектов. Массивы полезны, но имеют свои ограничения. Например, трудно изменить их размеры после создания, а получение доступа к элементам массива часто не отражает работу в реальном мире. Коллекции являются более гибкими. Они могут динамически изменять размер по мере добавления или удаления в них данных и обеспечивают ряд стратегий, используемых для доступа к их элементам.

## Что такое коллекция?



Коллекцией является тип, агригирующий 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 размеррассива, 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.

## Использование классов 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 |

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-1). 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-2). 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.

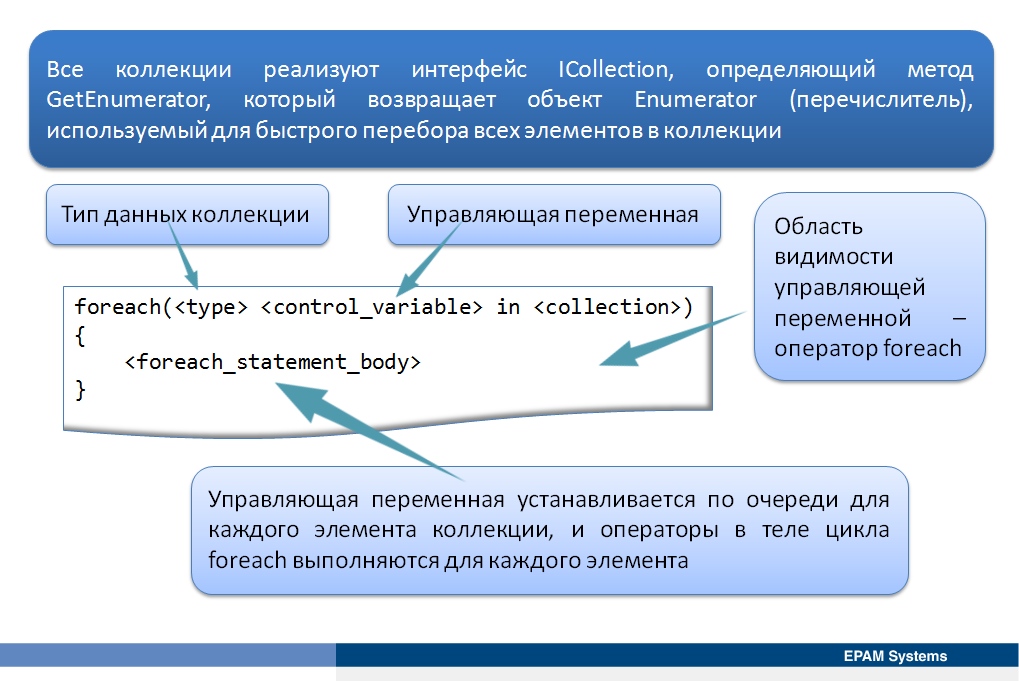
list.RemoveAt(1);

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

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

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

## 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>)

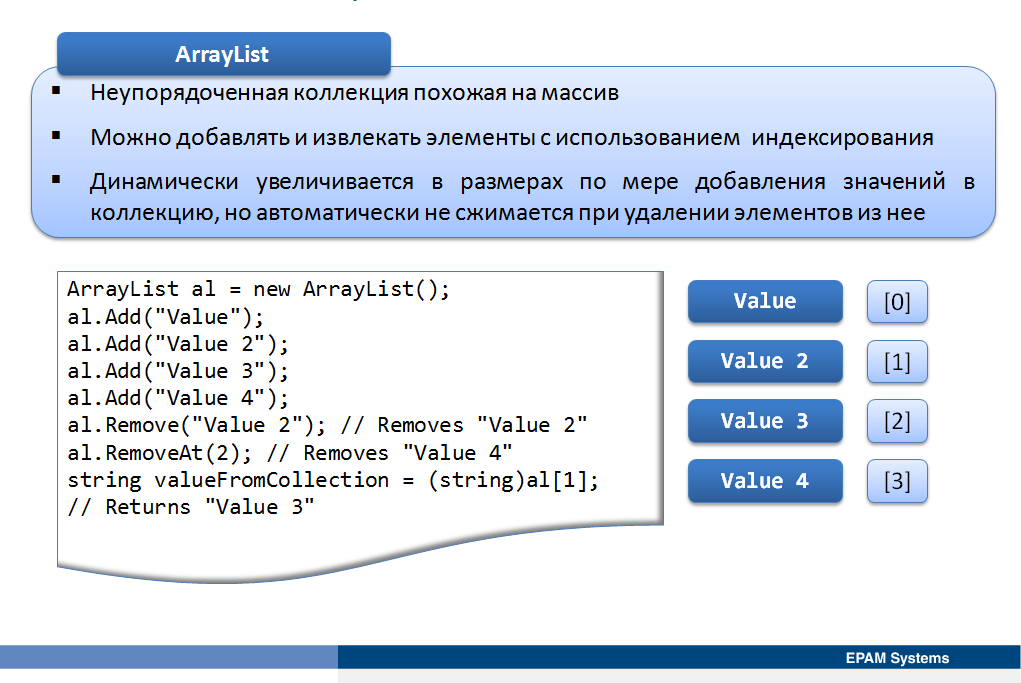
{

<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 предосталения 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

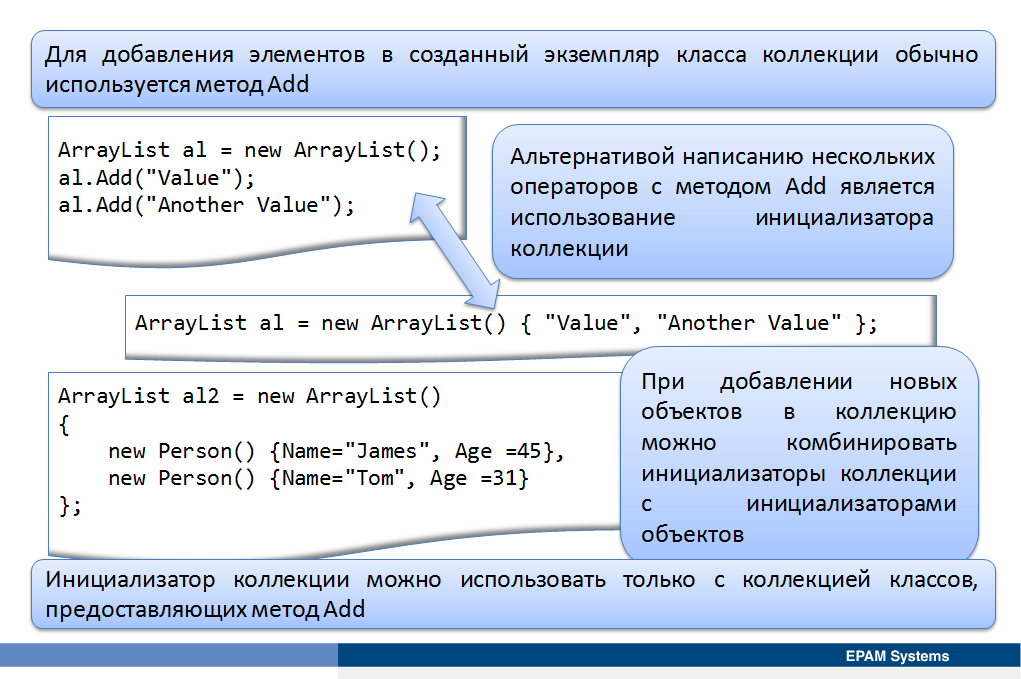
// by specifying the index.

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

// Returns "Value"

<http://go.microsoft.com/fwlink/?LinkId=192963>

## 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},

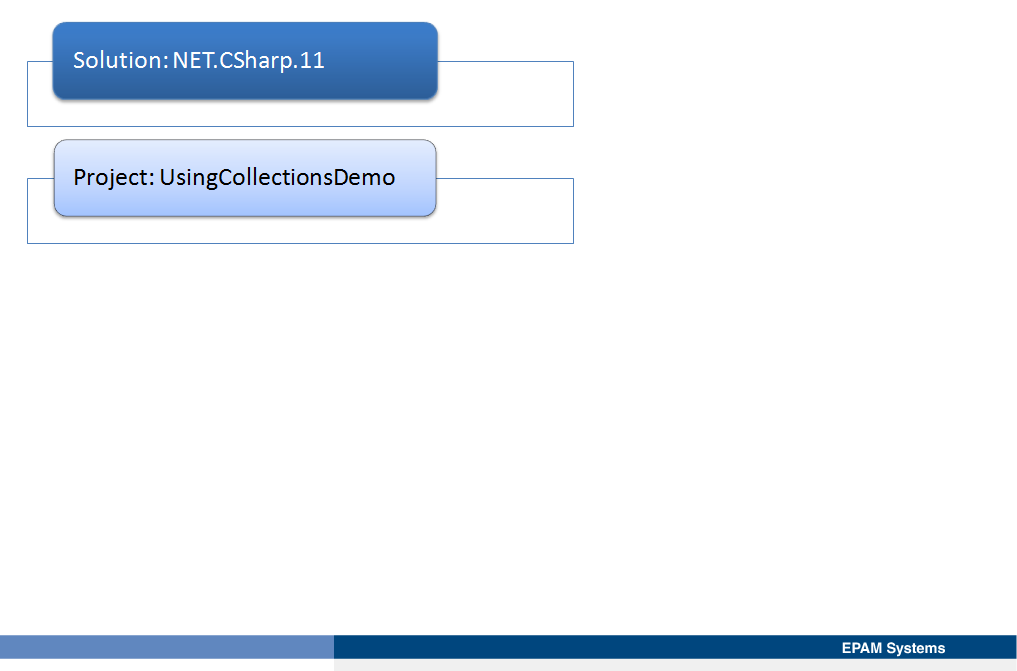
new Person() {Name="Tom", Age =31}

};

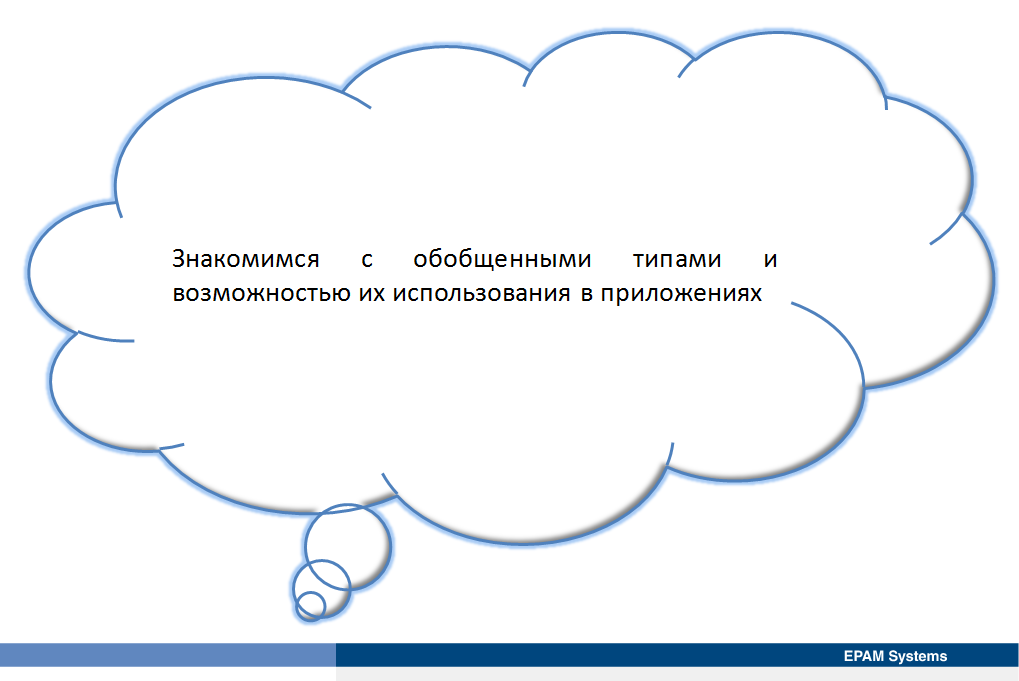
Инициализатор коллекции можно использовать только с коллекцией классов, предоставляющих метод Add. Компилятор использует метод Add для добавления в коллекцию объектов, указаных в инициализаторе коллекции. Классы коллекции, такие как класс Queue, не предоставляющие метод Add, не поддерживают инициализаторы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

<http://go.microsoft.com/fwlink/?LinkId=192964>

## Демонстрация: Использование коллекций

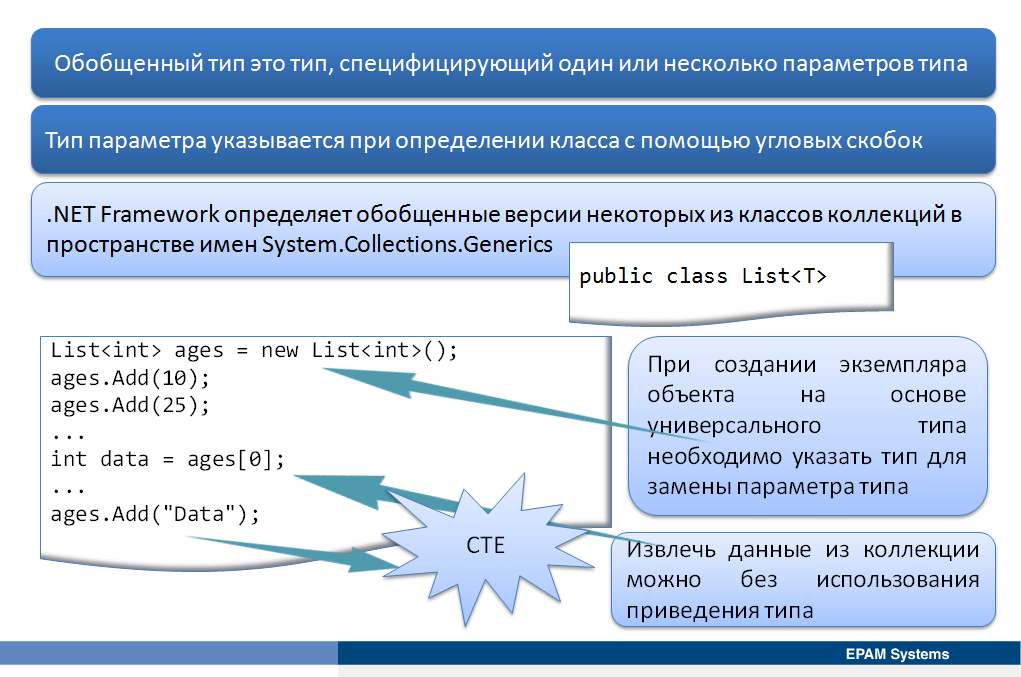


# Урок 2: Создание и использование обобщенных типов



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.

## Что такое обобщенный тип?



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);

...

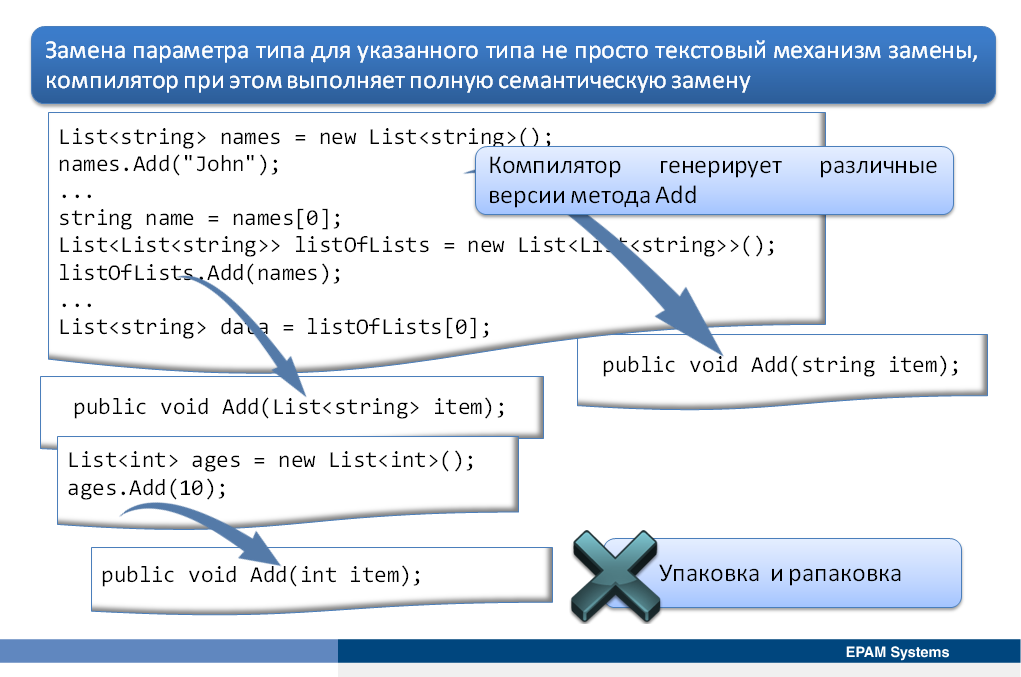
int data = ages[0]; // No cast necessary

...

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

http://go.microsoft.com/fwlink/?LinkId=192965

## Компиляция обобщенных типов и безопасность типов



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.

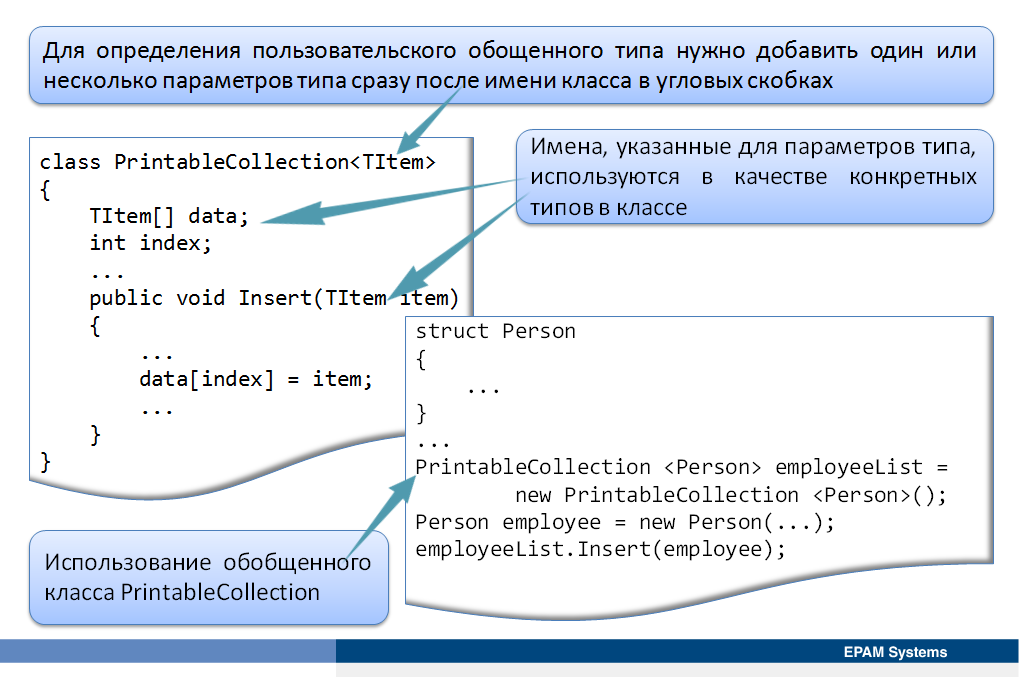
List<int> ages = new List<int>();

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 рапаковки этого значения.

## Определение пользовательских обобщенных типов



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-3).

struct Person

{

...

}

...

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

Person employee = new Person(...);

employeeList.Insert(employee);

Хорошей практикой является всегда инициализиовать 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);

...

}

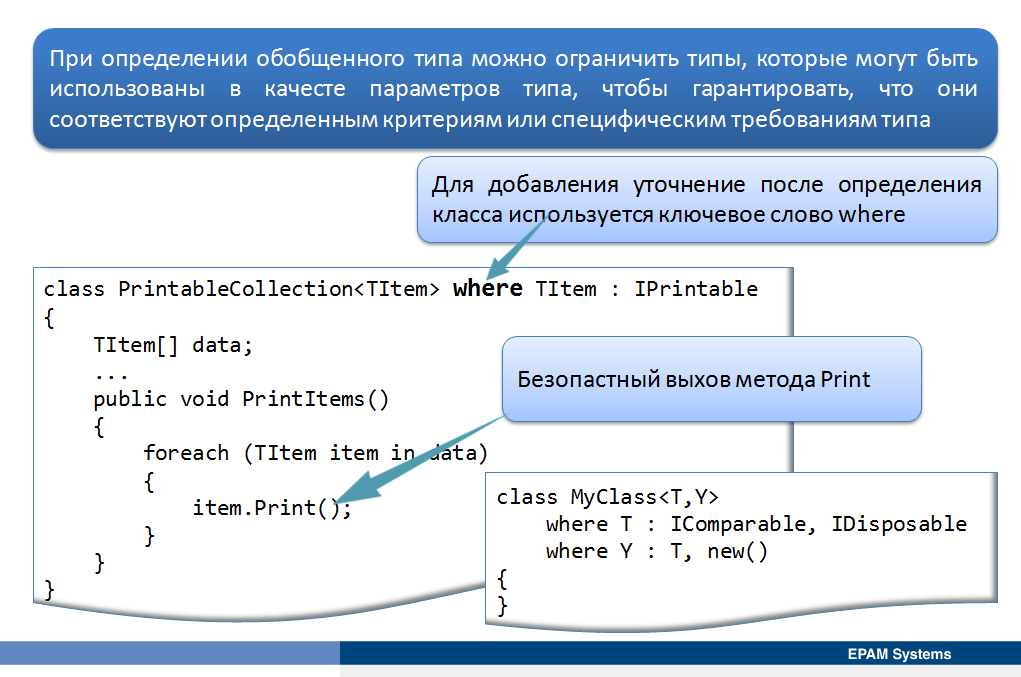
...

}

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.

## Добавление ограничений для обощенных типов



При определении обобщенного типа может потребоваться ограничить типы, которые могут быть использованы в качесте параметров типа, чтобы гарантировать, что они соответствуют определенным критериям или специфическим требованиям типа. Например, при определении класса PrintableCollection можно указать, что он может быть использован только с типами, которые знают, как форматировать и печатать себя, реализацией интерфейса IPrintable, определяющего метод Print.

В C# можно использовать уточнения для ограничения параметров типа, которые можно использовать в обощенных классах и других типах. Для добавления уточнение после определения класса используется ключевое слово where. В следующем примере указано, что параметр типа для типа PrintableCollection должен реализовывать интерфейс IPrintable.

class PrintableCollection<TItem> where TItem : IPrintable

{

TItem[] data;

...

public void PrintItems()

{

foreach (TItem item in data)

{

item.Print(); // item implements IPrintable, so it is safe

// to call the Print method.

}

}

}

Ключевое слово where можно добавить для каждого параметра типа, который определяется, и, более того, можно использовать несколько ограничений для каждого типа параметра. Несколько ограничений определяются, разделяя их уточнения запятой, как класс, реализующий несколько интерфейсов. В следующей таблице приведены уточнения, используемые для ограничения общего типа.

|  |  |
| --- | --- |
| **Ограничение** | **Описание** |
| **where T: struct** | Аргумент типа должен быть типом значения. Могут быть указаны любые типы значения, кроме Nullable. |
| **where T : class** | Аргумент типа должны быть ссылочным типом; это относится к любому классу, интерфейсу, делегату или типу массив. |
| **where T : new()** | Аргумент типа должен иметь public конструктор по умолчанию. Когда ограничение **new()**  используется вместе с другими ограничениями, оно должно быть указано последним. |
| **where T : <base class name>** | Аргумент типа должны быть наследником указанного базового класса. |
| **where T : <interface name>** | Аргумент типа должны имплементировать указанный интерфейс. Могут быть указаны несколько ограничений интерфейса. Уточняющий интерфейс также может быть универсальным. |
| **where T : U** | Тип аргумента, который поставляется для T должен вытекать из аргумента, который поставляется для U. |

В следующем примере показано ограничение двух параметров типа: параметр типа T должен реализовывать интерфейсы Icomparable и Idisposable, параметр типа Y должен прямо или косвенно вытекать из T (или быть тем же, что и T) и обеспечивать конструктор по умолчанию.

class MyClass<T,Y>

where T : IComparable, IDisposable

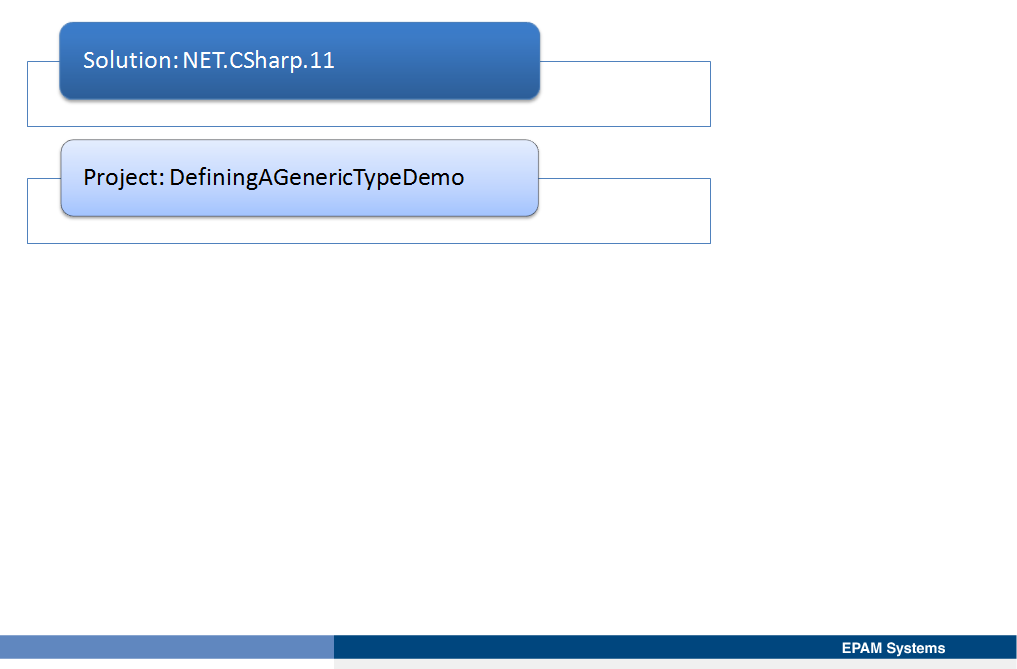
where Y : T, new()

{

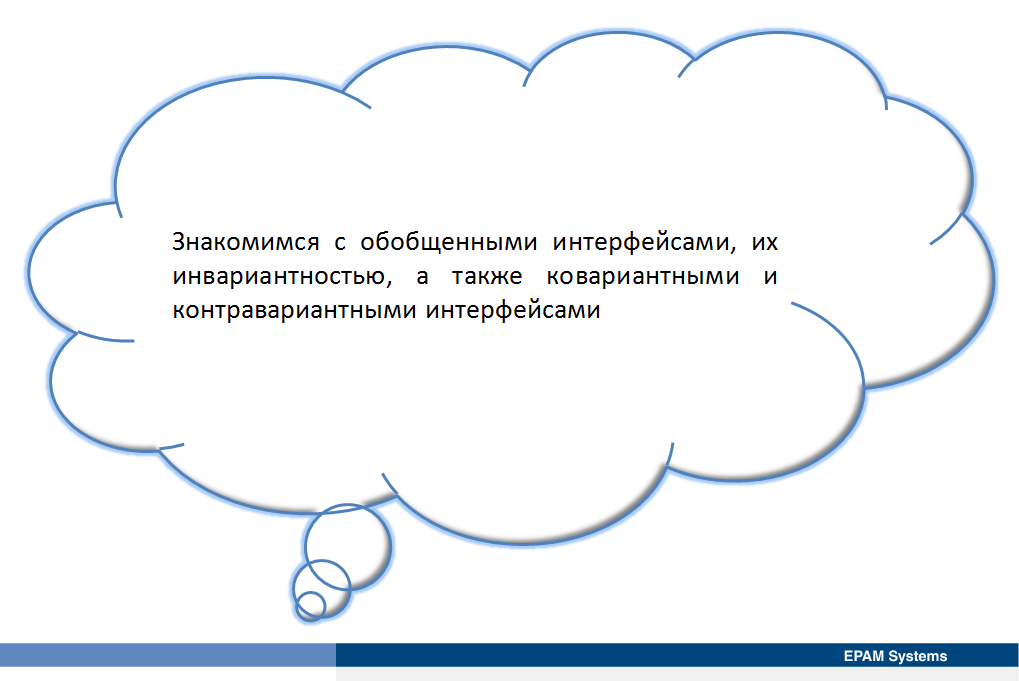
}

<http://go.microsoft.com/fwlink/?LinkId=192966>

## Демонстрация: Определение обобщенного типа

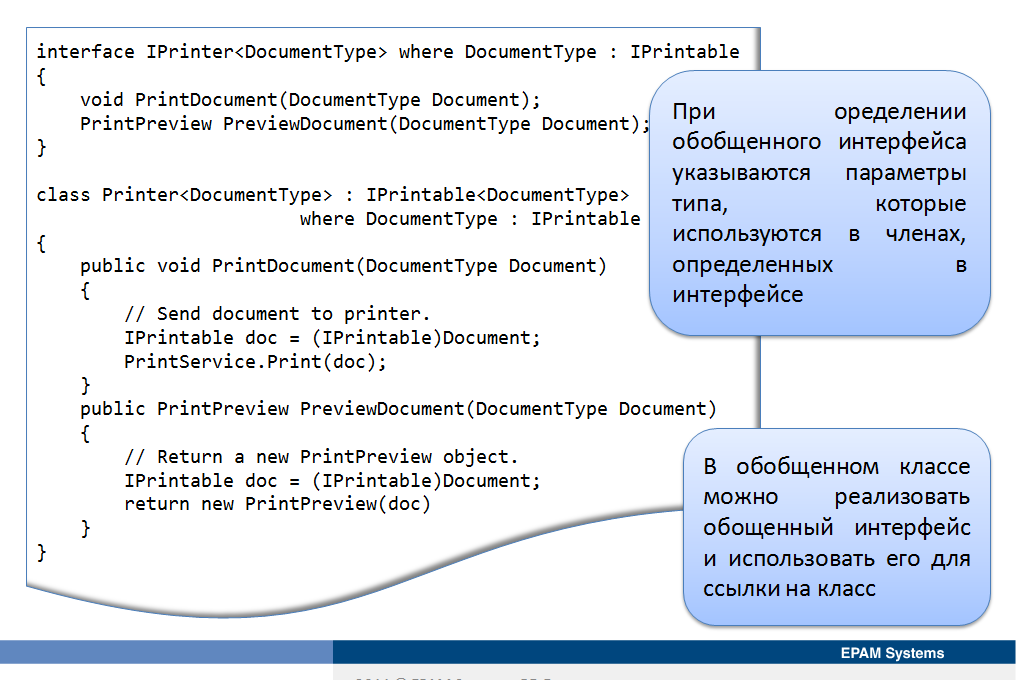


# Урок 3: Определение обобщенных интерфейсов, вариантность



В дополнение к классам и структурам обощения можно использовать для определения интерфейсов. Однако следует помнить, что .NET Framework использует строгую типизацию. При разработке обобщенного интерфейса строгая типизация может быть сдерживающей, предотвращая приведение между казалось бы, совместимыми типами, реализующими эти интерфейсы. Это ограничение называется инвариантностью. Урок описывает проблемы, которые может вызвать инвариантность обобщенных интерфейсов, и как можно определить обобщенные интерфейсы, преодолевающие некоторые из этих ограничений с помощью ковариантности и контравариантности.

## Определение обобщенных 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 оределении 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 обощенный интерфейс и использовать его для ссылки на класс, как и при работе с обычными классом и интерфейсом. В следующем примере показан обощенный интерфейс, который реализуется в обощенном 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)

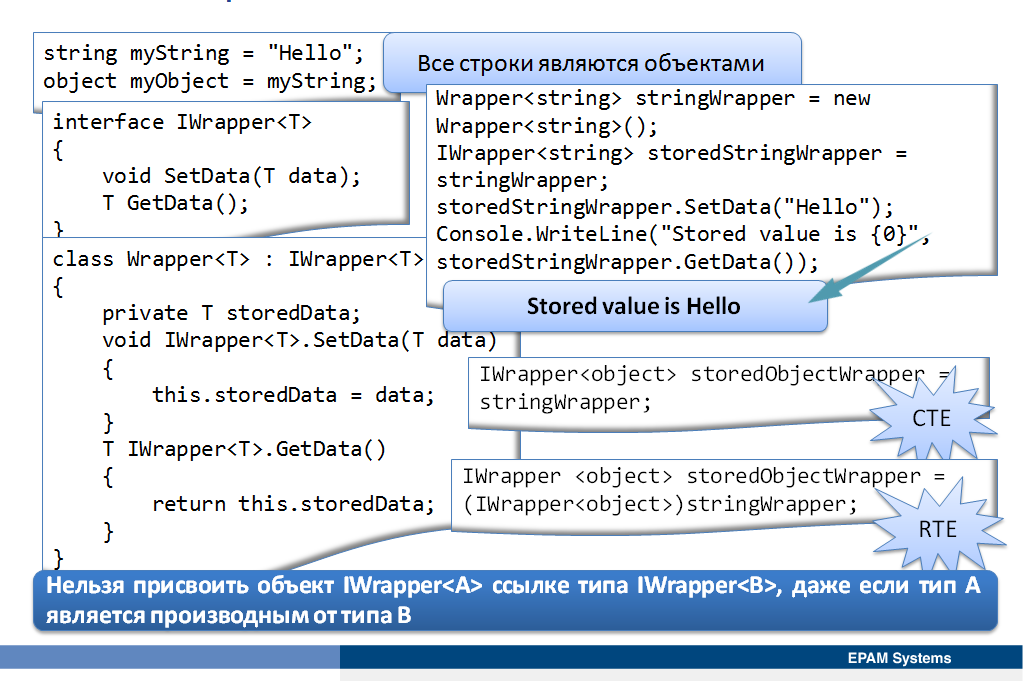
}

}

<http://go.microsoft.com/fwlink/?LinkId=192967>

<http://go.microsoft.com/fwlink/?LinkId=192968>

## Что такое инвариантность?



Тип object можно использовать для хранения значения или ссылки на любой другой тип. Например, следующий код вполне законен.

string myString = "Hello";

object myObject = myString;

В наследуемых терминах класс String является производным от класса Object, поэтому все строки являются объектами.

Следующий обобщенный класс Wrapper<T> является оберткой вокруг типа T, а обобщенный интерфейс IWrapper определяет методы SetData и GetData, которые класс Wrapper<T> реализует для хранения и извлечения данных соответственно.

interface IWrapper<T>

{

void SetData(T data);

T GetData();

}

class Wrapper<T> : IWrapper<T>

{

private T storedData;

void IWrapper<T>.SetData(T data)

{

this.storedData = data;

}

T IWrapper<T>.GetData()

{

return this.storedData;

}

}

Можно создать экземпляр данного класса и использовать его для того чтобы обернуть строку.

Wrapper<string> stringWrapper = new Wrapper<string>();

IWrapper<string> storedStringWrapper = stringWrapper;

storedStringWrapper.SetData("Hello");

Console.WriteLine("Stored value is {0}", storedStringWrapper.GetData());

В коде создается экземпляр типа Wrapper<string>, который для вызова метода SetData ссылается на объект через интерфейс IWrapper<string>. (Тип Wrapper<T> реализует интерфейс явно, поэтому необходимо вызывать методы через соответствующие ссылки на интерфейс). Код также вызывает метод GetData через интерфейс IWrapper<string>. Приведенный выше код генерирует сообщение «Stored value is Hello». Следующий оператор

IWrapper<object> storedObjectWrapper = stringWrapper;

похож на тот, который создает ссылку IWrapper<string> в предыдущем примере, с той лишь разницей, что параметр имеет тип object, а не string. Будет ли этот код компилируемым? Все строки являются объектами (можно присвоить значение строки ссылке на объект). Однако, если попробовать выполнить данную строку, компилятор выдаст сообщение об ошибке «Cannot implicitly convert type ‘Wrapper<string>’ to ‘IWrapper<object>’.». Явное приведение вида

IWrapper <object> storedObjectWrapper = (IWrapper<object>)stringWrapper;

дает компилируемый код, который однако генерирует исключение InvalidCastException. Проблема заключется в том, что, хотя все строки являются объектами, обратное неверно: не все объекты являются строками. Если приведенный выше оператор был бы разрешен, можно бы написать следующий код, в котором выполняется попытка сохранить объект Window в поле string (Window - класс, используемый для создания окна в WPF приложениях).

IWrapper <object> storedObjectWrapper = (IWrapper <object>)stringWrapper;

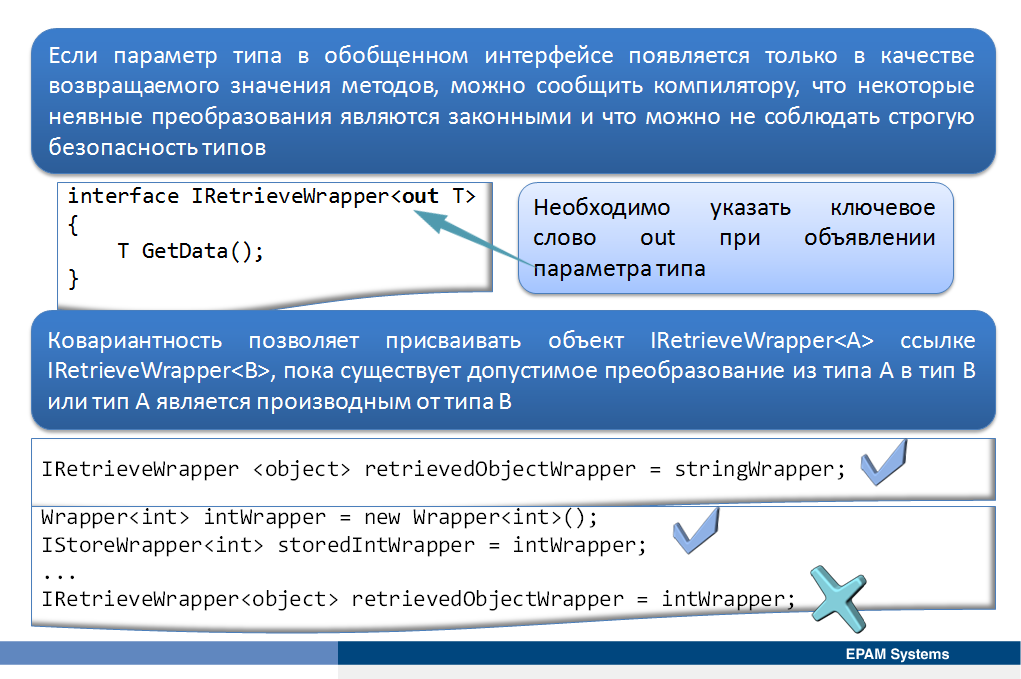
Window myWindow = new Window ();

storedObjectWrapper.SetData(myWindow);

Интерфейс IWrapper<T> является инвариантным: нельзя присвоить объект IWrapper<A> ссылке типа IWrapper<B>, даже если тип A является производным от типа B. По умолчанию C# реализует это ограничение для обеспечения безопасности типов.

Универсальные классы и структуры инварианты.

## Определение и реализация ковариантного интерфейса



Следующие интерфейсы IStoreWrapper<T> и IRetrieveWrapper<T> определены вместо интерфейса IWrapper<T> и реализуются в классе Wrapper<T>.

interface IStoreWrapper<T>

{

void SetData(T data);

}

interface IRetrieveWrapper<T>

{

T GetData();

}

class Wrapper<T> : IStoreWrapper<T>, IRetrieveWrapper<T>

{

private T storedData;

void IStoreWrapper<T>.SetData(T data)

{

this.storedData = data;

}

T IRetrieveWrapper<T>.GetData()

{

return this.storedData;

}

}

Функционально класс Wrapper<T> такой же, как раньше, только доступ к методам SetData и GetData осуществляется с помощью различных интерфейсов.

Wrapper<string> stringWrapper = new Wrapper<string>();

IStoreWrapper<string> storedStringWrapper = stringWrapper;

storedStringWrapper.SetData("Hello");

IRetrieveWrapper<string> retrievedStringWrapper = stringWrapper;

Console.WriteLine("Stored value is {0}", retrievedStringWrapper.GetData());

Является ли теперь следующий код действительным?

IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;

Нет, компиляция потерпит неудачу с той же ошибкой. Однако, несмотря на то, что компилятор C# считает такой оператор не типобезопасным, причины предполагать что это верно больше не действительны. Интерфейс IRetrieveWrapper<T> позволяет только читать данные, которые хранятся в объекте IWrapper<T> с помощью метода GetData, и он не предоставляет какой-либо способ изменения данных. В таких ситуациях как эта, где параметр типа появляется в обобщенном интерфейсе только в качестве возвращаемого значения методов, можно сообщить компилятору, что некоторые неявные преобразования являются законными и что можно не соблюдать строгую безопасность типов. Это можно сделать, указав ключевое слово out при объявлении параметра типа.

interface IRetrieveWrapper<out T>

{

T GetData();

}

Такая возможность называется ковариантностью. Ковариантность позволяет присваивать объект IRetrieveWrapper<A> ссылке IRetrieveWrapper<B>, пока существует допустимое преобразование из типа A в тип B или тип A является производным от типа B. Следующий код компилируется и работает правильно.

// string derives from object, so this is now legal

IRetrieveWrapper <object> retrievedObjectWrapper = stringWrapper;

Спецификатор out с параметром типа можно указать только тогда, когда параметр типа возникает в качестве возвращаемого типа методов. Если параметр тип используется, чтобы указать тип любых параметров метода, спецификатор out является недопутимым, код не будет компилироваться. Кроме того, ковариантность работает только с ссылочными типами потому, что типы значения не могут образовывать иерархию наследования. Следующий код не скомпилируется, поскольку int является значимым типом.

Wrapper<int> intWrapper = new Wrapper<int>();

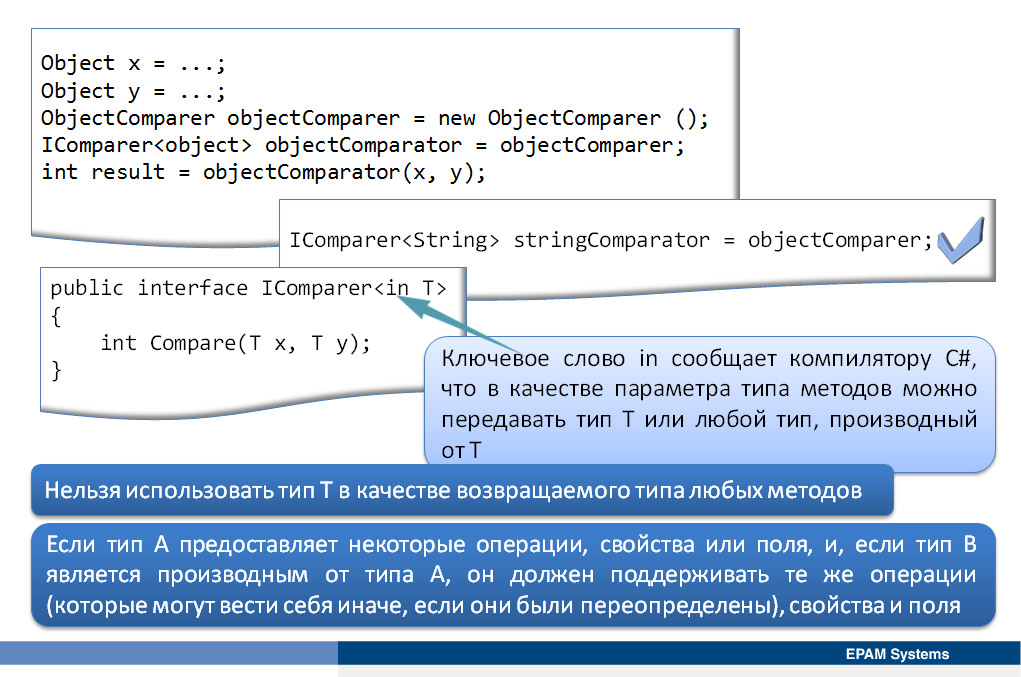
IStoreWrapper<int> storedIntWrapper = intWrapper; // this is legal

...

// the following statement is not legal – ints are not objects

IRetrieveWrapper<object> retrievedObjectWrapper = intWrapper;

## Определение и реализация контравариантного интерфейса



Контравариантность является следствием ковариантности и позволяет использовать обобщающий интерфейс для ссылки на объект типа B через ссылку на тип A, пока типа B является производным от типа A.

Пространство имен System.Collections.Generic в .NET Framework предоставляет интерфейс IComparer, определенный следующим образом.

public interface IComparer<in T>

{

int Compare(T x, T y);

}

Класс, реализующий этот интерфейс определяет метод Compare, используемый для сравнения двух объектов типа, которые специфицирует параметр типа T. Метод Compare возвращает целое значение: нуль, если параметры x и y имеют одинаковое значение, отрицательное число, если x меньше y, и положительное, если x больше y. В следующем примере показано сравнение объектов в соответствии с их хэш-кодом[[4]](#footnote-4).

class ObjectComparer : IComparer<object>

{

int Comparer<object>.Compare(Object x, Object y)

{

int xHash = x.GetHashCode();

int yHash = y.GetHashCode();

if (xHash == yHash)

return 0;

if (xHash < yHash)

return -1;

return 1;

}

}

Можно создать объект ObjectComparer и вызвать метод Compare через интерфейс IComparer<Object> для сравнения двух объектов следующим образом.

Object x = ...;

Object y = ...;

ObjectComparer objectComparer = new ObjectComparer ();

IComparer<object> objectComparator = objectComparer;

int result = objectComparator(x, y);

На этот же объект можно также ссылаться через версию интерфейса IComparer, сравнивающую строки.

IComparer<String> stringComparator = objectComparer;

Как же компилятор C# узнает, что в коде метода Compare не будут выполняться любые операции конкретного типа, которые могут завершиться неудачно, если будет вызыван метод через интерфейс, основанный на другом типе? Если вернуться к определению интерфейса IComparer, можно увидеть спецификатор in параметра типа. Ключевое слово in сообщает компилятору С#, что в качестве параметра типа методов можно передавать тип T или любой тип, производный от T. Нельзя использовать тип T в качестве возвращаемого типа любых методов. Эта функция позволяет обращаться к объекту либо через обобщеный интерфейс, основанный на типе объекта, либо через обобщеный интерфейс, основанный на типе, производном от типа объекта. Таким образом, если тип A предоставляет некоторые операции, свойства или поля, и, если тип B является производным от типа A, он должен поддерживать те же операции (которые могут вести себя иначе, если они были переопределены), свойства и поля.

Ковариантность и контравариантность могут показаться мишурой тем в мире обобщений, но они полезны. Например, обощенный класс коллекция List<T> использует объекты IComparer<T> для реализации методов Sort и BinarySearch. Объект List<Object> может содержать коллекцию объектов любого типа, следовательно методы Sort и BinarySearch должны иметь возможность сортировать и искать объекты любого типа. Без использования контравариантности, методы Sort и BinarySearch должны будут включать логику, определяющую реальные типы элементов, которые будут сортироваться или находится, а затем реализовать конкретный тип сортировки или механизма поиска.

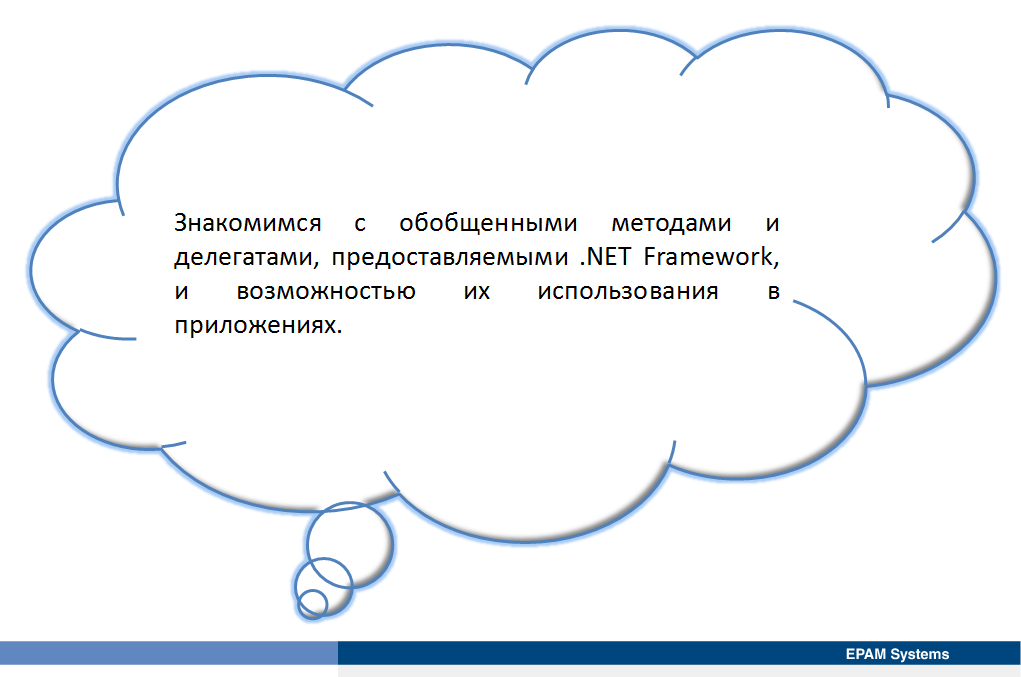
**Подводя итоги ковариации и контрвариации.** На основе примеров данного урока можно просуммировать понятия ковариантности и контравариантности, а именно:

*Ковариантность*. Если методы в обобщенном интерфейсе могут возвращать строки, они также могут возвращать объекты. (Все строки являются объектами)

*Контрвариантность*. Если методы в обобщенном интерфейсе могут принимать параметры object, они могут принимать параметры string. (Если можно выполнять операции с использованием объекта, значит можно выполнять ту же операцию с использованием строки, потому что все строки являются объектами.)

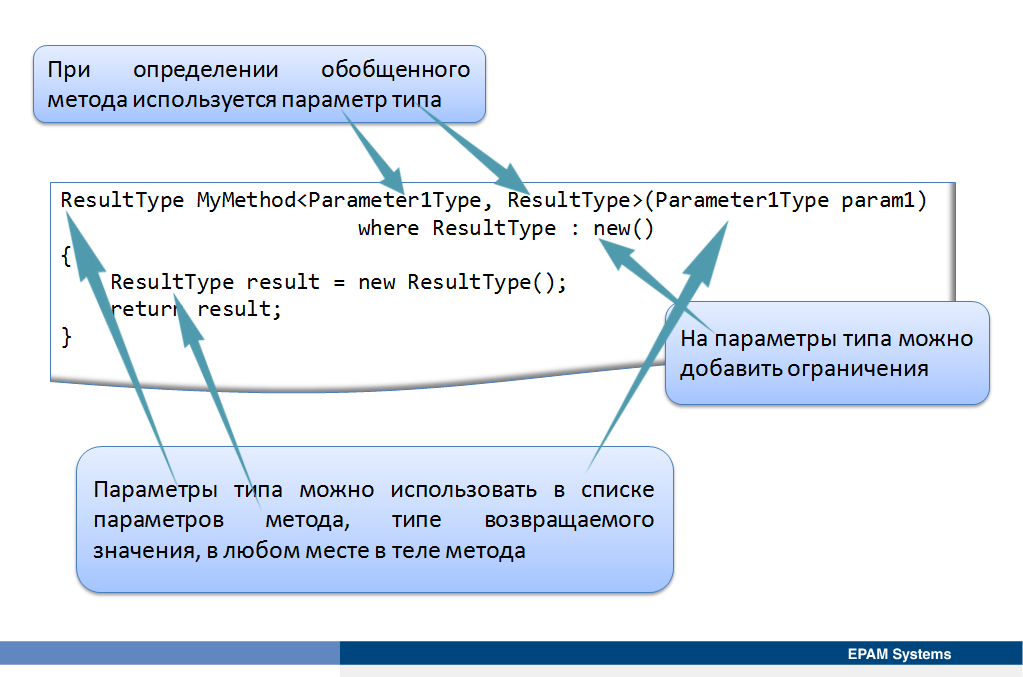
http://go.microsoft.com/fwlink/?LinkId=192969

# Урок 4: Использование обобщенных методов и делегатов



Обобщенные классы, используюемые другими классами типобезопасным образом являются мощным особенностью .NET Framework. Однако иногда необходимо разрабатывать обобщенные методы и делегаты. Обобщенные методы и делегаты похожи на обобщенные классы в том, что при использовании метода или делегата указывается параметр типа. Урок знакомит с обобщенными методами и делегатами, предоставляемыми .NET Framework, и возможностью их использования в приложениях.

## Определение обобщенного метода



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 списоком 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()

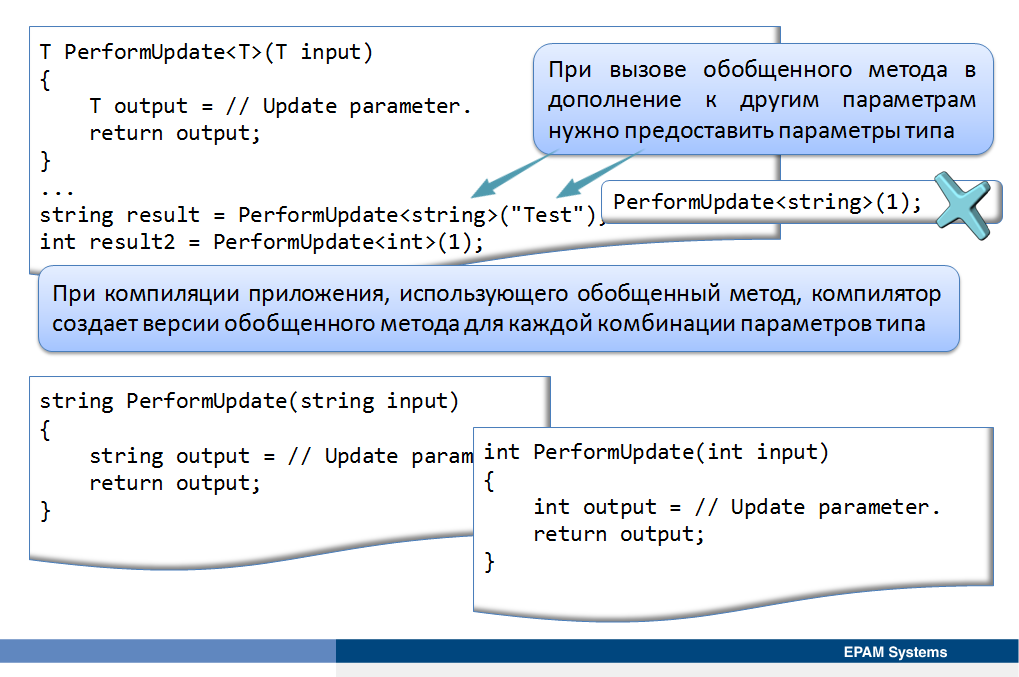
{

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)

{

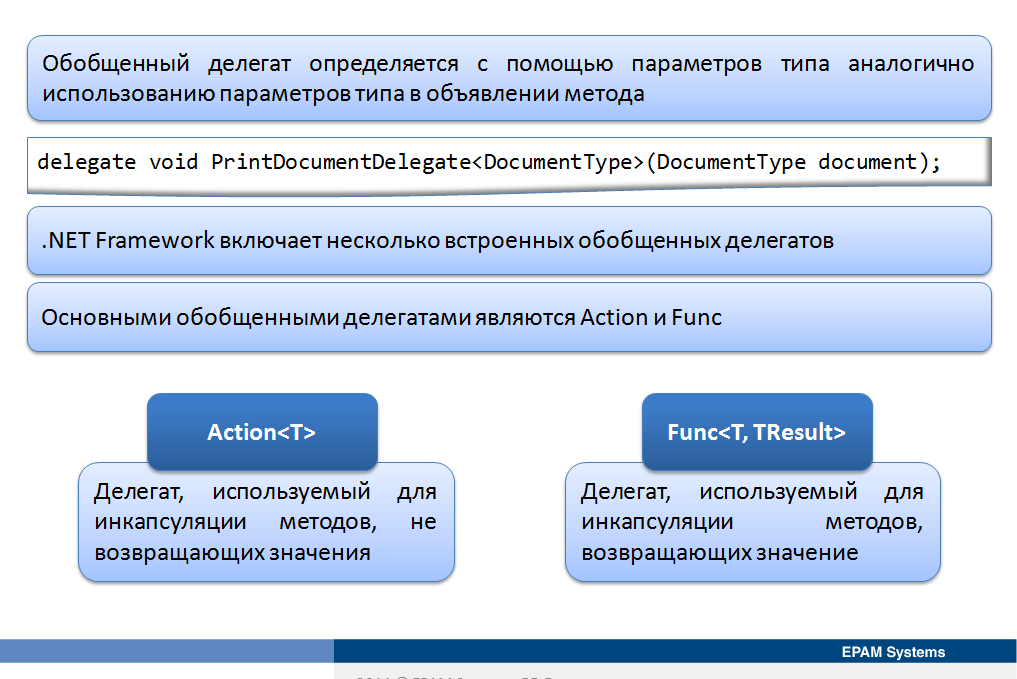
int output = // Update parameter.

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.

## Использование обобщенных делегатов .NET Framework



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;

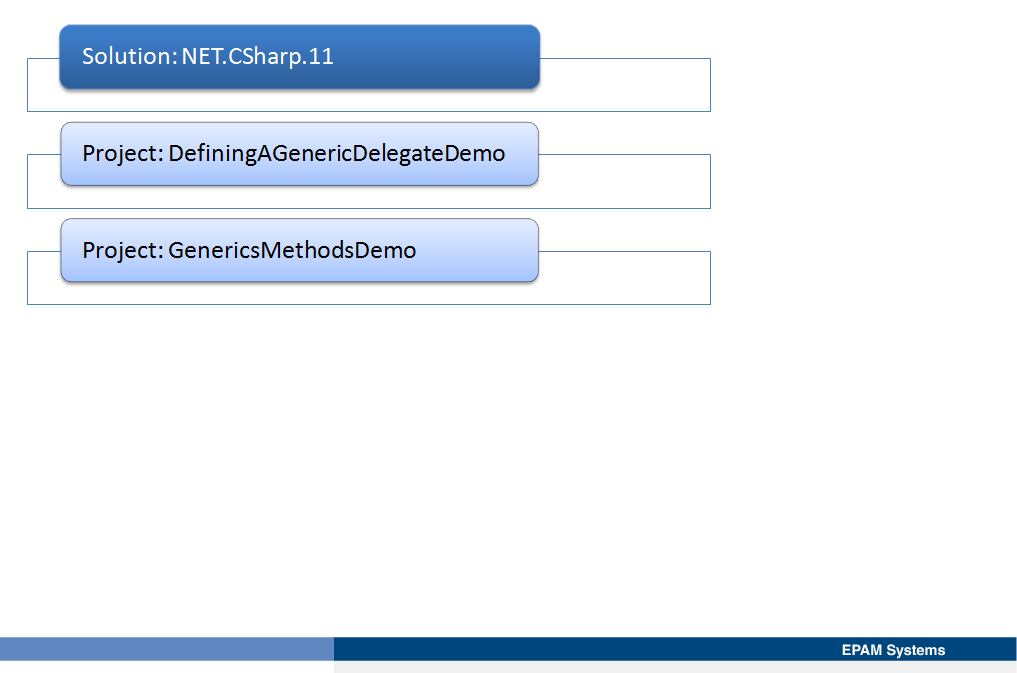
returnedValue = myDelegate("Value", 5);

Console.WriteLine(returnedValue);

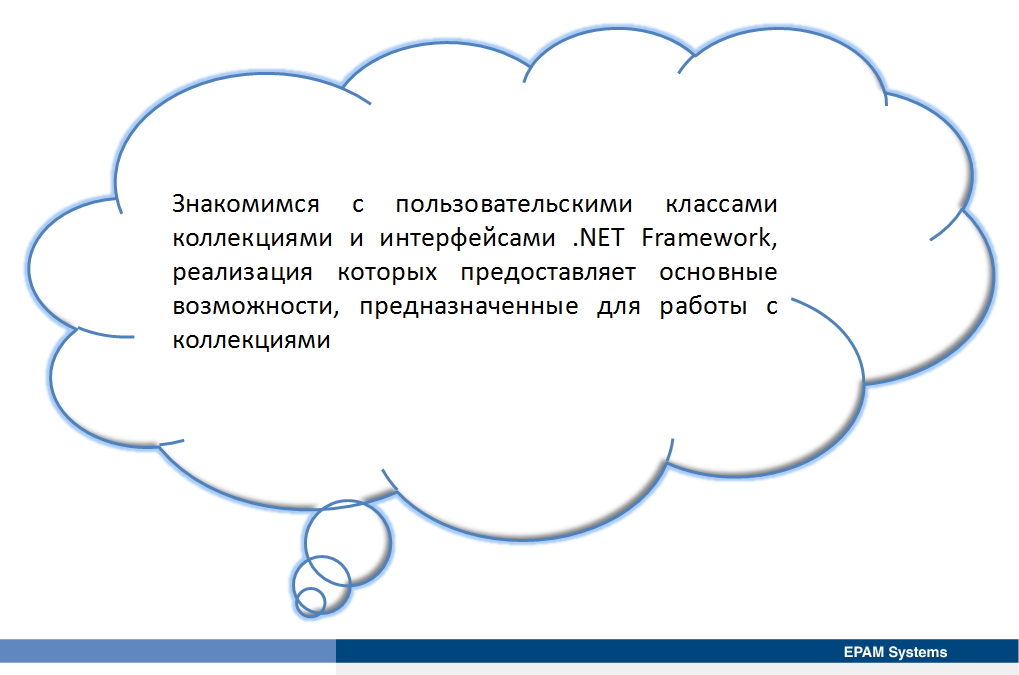
}

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.

## Демонстрация: Обобщенные методы и делегаты



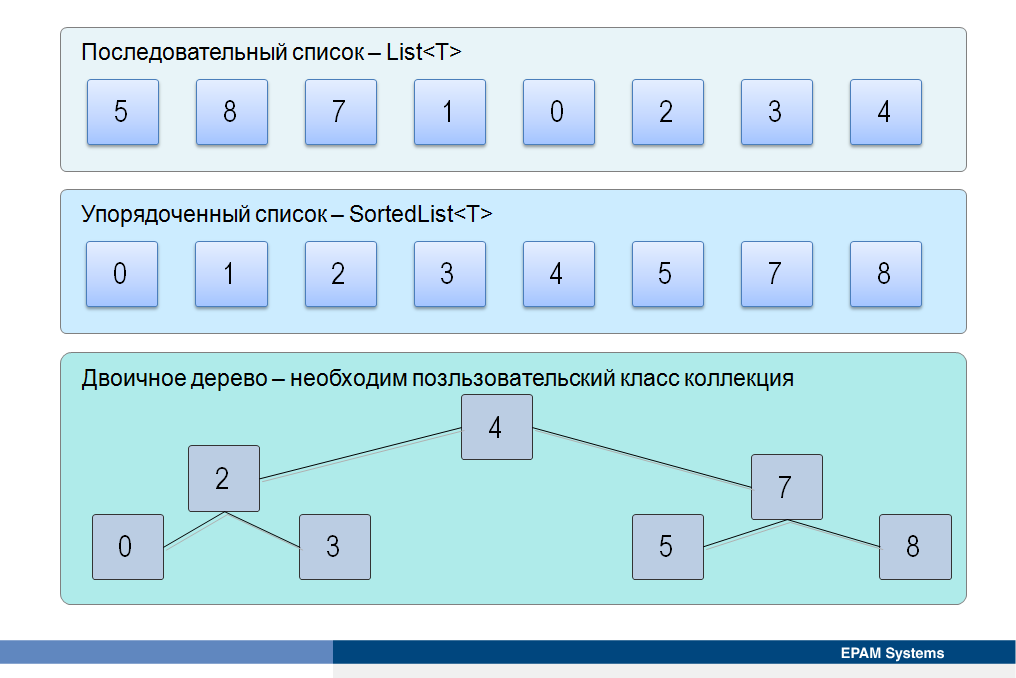
# Урок 5: Реализация пользовательского класса коллекции



.NET Framework предоставляет ряд классов коллекций, позволяющих хранить и извлекать данные, используя различную семантику. Например, класс Queue позволяет хранить и извлекать данные по принципу first-in, first-out (FIFO), а класс Stack реализует механизм first-in, last-out (FILO). Другие классы коллекций, такие как класс Hashtable, позволяют хранить данные и получать к ним доступ с помощью ключа в форме словаря. Если ни один из классов коллекций библиотеки классов .NET Framework не обеспечивает конкретные потребности приложения, можно реализовать свой собственный класс коллекцию.

Язык C# предоставляет несколько возможностей, предназначенные для работы с коллекциями. При разработке пользовательского класса коллекции следует убедиться, что для поддержки этих возможностей реализованы необходимые функциональности: инициализаторы коллекции или возможность перечислять ее элементы. Урок знакомит с пользовательскими классами коллекциями, а также с интерфейсами .NET Framework, реализация которых обеспечивает эту поддержку.

## Пользовательские классы коллекции



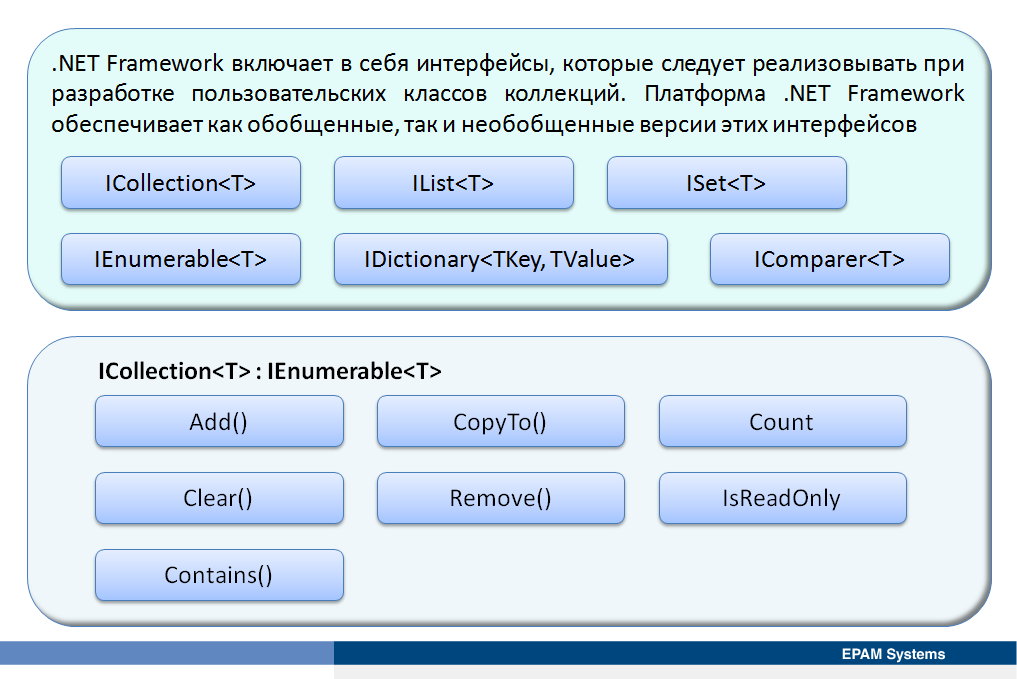
Пользовательский класс коллекция это класс, позволяющий хранить и извлекать объекты коллекции и реализующий различные интерфейсы, определяемые .NET Framework. Реализуя определенные интерфейсы .NET Framework, пользовательские классы коллекции обеспечиваются поддержкой языковых конструкций C#. Например, чтобы использовать класс коллекцию с конструкцией foreach, он должен предусмотреть перечислитель. Поэтому необходимо реализовывать интерфейс IEnumerable, определяющий метод GetEnumerator. Перечислитель должен предоставить методы, определяемые интерфейсом IEnumerator, которые приложение может использовать для перемещения по элементам коллекции в упорядоченном виде.

Если коллекция должна поддерживать LINQ (Language-Integrated Query) запросы, следует реализовать интерфейс ICollection. Этот интерфейс расширяет интерфейс IEnumerable методами, которые преобразовывают коллекцию в запрашиваемый объект.

В зависимости от возможностей, которые будет предоставлять пользовательский класс коллекция, можно реализовывать другие интерфейсы. Например, если необходимо обеспечить доступ к элементам коллекции с помощью индекса, нужно реализовать интерфейс IList.

.NET Framework включает в себя как обобщенные, так и необобщенные версии коллекций. В зависимости от требований типа, можно выбрать – разработать либо обобщенный, либо необобщенный пользовательский класс коллекцию. Необобщенные пользовательские классы коллекции, как правило, предназначены для работы с определенным типом, который может ограничить их использование в других приложениях. Можно создать коллекцию для работы с элементами System.Object, однако это может оказать влияние на производительность и безопасность типов. Если не существут особых причин для разработки необобщенного класса коллекции, как правило, следует реализовывать обобщенный класс коллекцию.

## Обобщенные интерфейсы коллекций .NET Framework



.NET Framework включает в себя несколько интерфейсов, которые следует реализовывать при разработке пользовательских классов коллекций. Платформа .NET Framework обеспечивает как обобщенные, так и необобщенные версии этих интерфейсов. Основными обобщенными интерфейсами .NET Framework, являются:

* ICollection<T>.
* IEnumerable<T>
* IList<T>
* IDictionary<TKey, TValue>
* ISet<T>
* IComparer<T>

**Интерфейсы ICollection<T> и IEnumerable<T>.** Интерфейс ICollection<T> это базовый интерфейс для всех обобщенных классов коллекций. Этот интерфейс нужно реализовывать в каждом разрабатываемом классе коллекции. Интерфейс ICollection<T> наследуется от интерфейса IEnumerable<T>; когда реализован интерфейс ICollection<T>, следует реализовать интерфейс IEnumerable<T>. Интерфейс ICollection<T> определяет следующие методы и свойства, описание которых представлены в следующей таблице.

|  |  |
| --- | --- |
| **Член** | **Описание** |
| Add  (метод) | Метод добавляет новый элемент в коллекцию. Для использования инициализатора коллекции необходимо предоставить public метод Add. Для поддержки инициализатора коллекции, тип также должен реализовывать интерфейс IEnumerable. |
| Clear  (метод) | Метод удаляет все элементы из коллекции. |
| Contains  (метод) | Метод возвращает значение, указывающее, существует ли конкретный элемент в коллекции. |
| CopyTo  (метод) | Метод копирует коллекцию в указанный массив. |
| Remove  (метод) | Метод удаляет один элемент из коллекции. |
| Count  (свойство) | Свойство указывает количество элементов в коллекции. |
| IsReadOnly  (свойство) | Свойство указывает, является ли коллекция коллекцией только для чтения. |

**Интерфейс IList<T>.** Интерфейс IList<T> это базовый интерфейс для линейных коллекций, позволяющих получить доступ к элементам с помощью индекса. Интерфейс IList<T> наследуется от интерфейса ICollection<T> и определяет следующие методы и свойство-индексатор:

|  |  |
| --- | --- |
| **Член** | **Описание** |
| IndexOf  (метод) | Метод осуществляет поиск указанного объекта и возвращает отсчитываемый от нуля индекс его первого вхождения. |
| Insert  (метод) | Метод вставляет новый элемент по указанному индексу в коллекцию. |
| RemoveAt  (метод) | Метод удаляет один элемент из коллекции по указанному индексу. |
| Item  (свойство) | Свойство является индексатором, который позволяет получить или установить значение по указанному индексу в коллекции. |

**Интерфейс IDictionary<TKey, TValue>.** Интерфейс IDictionary<TKey, TValue> это базовый интерфейс коллекции на основе словаря. Коллекция на основе словаря хранит пары комбинаций ключ/значение; когда в словарь-коллекцию добавляется элемент, необходимо указать ключ и значение. Ключ в коллекции должен быть уникальным. Интерфейс IDictionary<TKey, TValue> наследуется от интерфейса ICollection<T>. Однако следует помнить, что интерфейс ICollection<T> принимает один параметр типа, а интерфейс IDictionary<TKey, TValue> имеет два параметра типа. Чтобы помочь в решении этого несоответствия .NET Framework предоставляет обобщенный тип KeyValuePair<TKey, TValue>. При разработке обобщенного класса коллекции, каждый элемент коллекции может быть представлен как объект KeyValuePair<TKey, TValue>. Можно использовать этот тип в качестве параметра типа при реализации интерфейса ICollection<T> в классе словаре. Интерфейс IDictionary<TKey, TValue> определяет следующие методы и свойства.

|  |  |
| --- | --- |
| **Член** | **Описание** |
| Add  (метод) | перегруженный метод добавляет новый элемент в коллекцию с указанным ключом и значением. |
| ContainsKey  (метод) | Этот метод возвращает логическое значение, чтобы указать, содержит ли коллекция элемент с конкретным ключем. |
| GetEnumerator  (метод) | Этот перегруженный метод возвращает перечислитель объектов KeyValuePair<TKey, TValue>. |
| Remove  (метод) | Этот перегруженный метод удаляет элемент с указанным ключем из коллекции. |
| TryGetValue  (метод) | Этот метод возвращает логическое значение, чтобы указать, является ли метод успешным. Если метод успешен, он устанавливает значение выходного параметра в значение, связанное с указанным ключем. |
| Item  (свойство) | Это свойство индексатор, которое позволяет получить или установить значение в коллекции, на основе указанного ключа. |
| Keys  (свойство) | Свойство возвращает коллекцию ключей коллекции, используя ссылку на интерфейс ICollection<T>. |
| Values  (свойство) | Свойство возвращает коллекцию значений, используя ссылку на интерфейс ICollection<T>. |

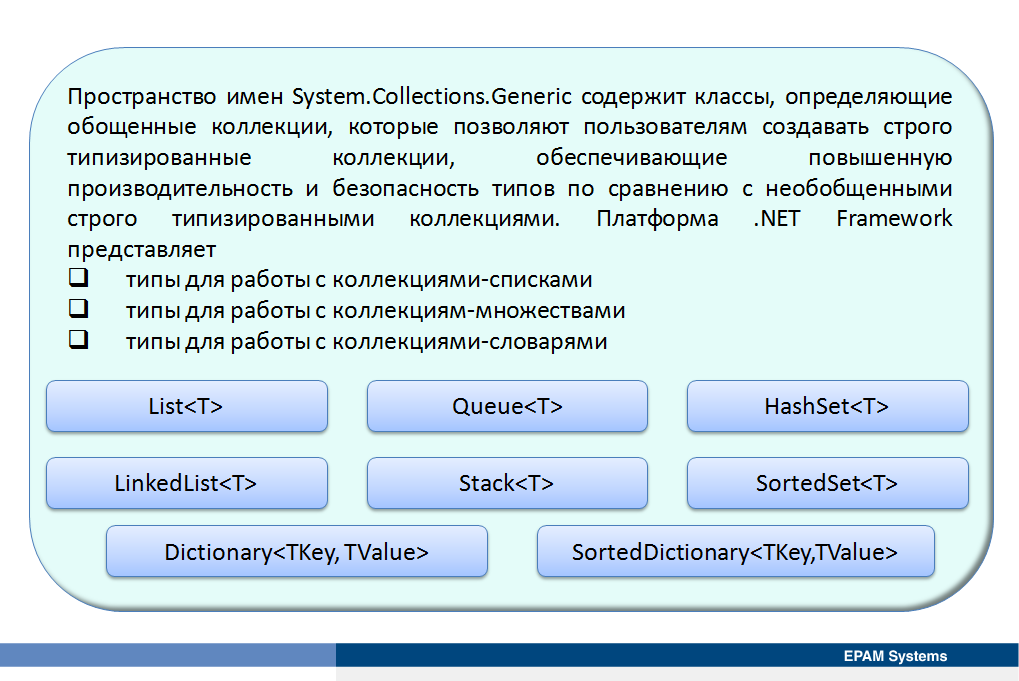
**Интерфейс ISet<T>.** Интерфейс ISet<T> предназначен для работы с коллекциями-множествами. Его набор методов отражает типичные операции для множеств.

http://go.microsoft.com/fwlink/?LinkId=192970

http://go.microsoft.com/fwlink/?LinkId=192971

http://go.microsoft.com/fwlink/?LinkId=192972

## Обобщенные классы коллекции .NET Framework



Пространство имен System.Collections.Generic содержит классы, определяющие обощенные коллекции, которые позволяют пользователям создавать строго типизированные коллекции, обеспечивающие повышенную производительность и безопасность типов по сравнению с неуниверсальными строго типизированными коллекциями. Платформа .NET Framework представляет

* типы для работы с коллекциями-списками;
* типы для работы с коллекциям-множествами;
* типы для работы с коллекциями-словарями.

**Типы для работы с коллекциями-списками.**

Класс List<T> пространства имён System.Collections.Generic является основным классом для представления наборов, допускающий динамическое добавление элементов. Для хранения данных набора используется внутренний массив. Класс List<T> реализует интерфейсы IList<T> и IList. Класс List<T> определяет следующие методы и свойства

|  |  |
| --- | --- |
| **Член** | **Описание** |
| Add  (метод) | Добавление одного элемента |
| AddRange  (метод) | Добавление набора элементов |
| Insert  (метод) | Вставка элемента в заданную позицию |
| InsertRange  (метод) | Вставка набора элементов |
| Remove  (метод) | Удаление элемента |
| RemoveAt  (метод) | Удаление элемента на указанной позиции со сдвигом остальных |
| RemoveRange  (метод) | Удаление диапазона элементов |
| RemoveAll  (метод) | Удаление всех элементов, удовлетворяющих заданному предикату |
| this[int index] | Основной индексатор |
| GetRange  (метод) | Получение подсписка |
| GetEnumerator  (метод) | Получение перечислителя |
| BinarySearch  (метод) | Поиск элемента в упорядоченном наборе |
| IndexOf  (метод) | Индекс первого вхождения своего аргумента в набор |
| LastIndexOf  (метод) | Индекс последнего вхождения своего аргумента в набор |
| Contains() | Проверка, содержится ли указанный элемент в наборе |
| Exists  (метод) | Проверка, содержит ли набор элемент, удовлетворяющий заданному предикату |
| Find  (метод) | Возвращает первый элемент, удовлетворяющий предикату, который задан как параметр метода |
| FindLast  (метод) | Возвращает первый элемент с конца набора, удовлетворяющий предикату, который задан как параметр метода |
| FindAll  (метод) | Возвращает все элементы набора, удовлетворяющие предикату |
| FindIndex  (метод) | Возвращает индекс первого вхождения элемента, удовлетворяющего предикату, который задан как параметр метода |
| FindLastIndex  (метод) | Возвращает индекс последнего вхождения элемента, удовлетворяющего предикату, который задан как параметр метода |
| TrueForAll  (метод) | Возвращает true, если заданный предикат верен для всех элементов набора |
| Sort  (метод) | Сортировка набора (возможно, с применением собственного объекта для сравнения элементов) |
| ToArray  (метод) | Преобразование набора в массив |
| CopyTo  (метод) | Копирование набора или его части в массив |
| AsReadOnly  (метод) | Преобразование набора в коллекцию только для чтения |
| ConvertAll  (метод) | Конвертирование набора одно типа в набора другого типа |
| Count  (свойство) | Количество элементов в наборе |
| Capacity  (свойство) | Емкость набора |
| TrimExcess  (метод) | Усечение размера внутреннего массива до необходимой минимальной величины |
| Clear  (метод) | Очистка списка ‑ удаление всех элементов |
| Reverse  (метод) | Изменение порядка элементов на противоположный |
| ForEach  (метод) | Выполняет указанное действие для всех элементов списка |

Класс List<T> имеет три конструктора: конструктор без параметров; конструктор, позволяющий создать набор на основе коллекции – производится копирование элементов коллекции в список; конструктор, принимающий в качестве параметра начальную емкость набора (емкость набора – это количество элементов набора, которое он способен содержать без увеличения размера внутреннего массива).

На примере коллекции List<T> следует отметить одну особенность, присущую всем коллекциям. Если в коллекции хранятся структуры, то части структуры нельзя изменить при помощи индексатора коллекции.

Следующий пример демонстрирует использование некоторых свойств и методов класса List<T>.

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

dinosaurs.Add("Tyrannosaurus");

dinosaurs.Add("Amargasaurus");

dinosaurs.Add("Mamenchisaurus");

dinosaurs.Add("Deinonychus");

dinosaurs.Add("Compsognathus");

foreach(string dinosaur in dinosaurs)

{

Console.WriteLine(dinosaur);

}

Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

Console.WriteLine("Count: {0}", dinosaurs.Count);

Console.WriteLine("\nContains(\"Deinonychus\"): {0}",

dinosaurs.Contains("Deinonychus"));

dinosaurs.Insert(2, "Compsognathus");

Console.WriteLine("\ndinosaurs[3]: {0}", dinosaurs[3]);

dinosaurs.Remove("Compsognathus");

dinosaurs.TrimExcess();

dinosaurs.Clear();

Делая выбор между классами List<T> и его необобщенным аналогом ArrayList, предлагающими сходные функциональные возможности, следует помнить, что класс List<T> в большинстве случаев обрабатывается быстрее и является потокобезопасным. Если в качестве типа T класса List<T> используется ссылочный тип, оба класса действуют идентичным образом. Однако если в качестве типа T используется тип значений, необходимо принять во внимание особенности, связанные с реализацией и упаковкой.

Если в качестве типа T используется тип значений, компилятор генерирует реализацию класса List<T> специально для этого типа значений. Это означает, что элемент списка List<T> не требует упаковки перед его использованием, и после создания примерно 500 элементов списка объем памяти, сэкономленной за счет отказа от их упаковки, превысит объем памяти, используемой для генерации реализации класса.

Необходимо убедиться, что значимый тип, используемый в качестве типа T, реализует обобщенный интерфейс IEquatable<T>. В противном случае такие методы, как Contains, должны будут вызывать метод Object.Equals(Object), который осуществляет упаковку соответствующего элемента списка. Если тип значений реализует интерфейс IСomparable, следует также реализовать универсальный интерфейс IComparable<T>, чтобы избежать упаковки элементов списка методами BinarySearch и Sort или передавать объект IComparer<T> в методы BinarySearch и Sort.

Использование реализации класса List<T>, определяемой конкретным типом, является предпочтительным по сравнению с использованием класса ArrayList или самостоятельным составлением коллекции со строго типизированной оболочкой. Дело в том, что собственная реализация должна выполнять все функции, которые уже автоматически предоставляются в .NET Framework, и к тому же общеязыковая среда выполнения может использовать общие метаданные и код MSIL, что невозможно обеспечить в собственной реализации.

Класс LinkedList<T> является обобщенным классом для работы с двусвязным списком. Такой список позволяет осуществлять вставку и удаление элемента без сдвига остальных элементов. Однако для доступа к элементу по индексу требуется проход по списку. LinkedList<T> реализует интерфейсы ICollection и ICollection<T>. Каждый элемент двусвязного списка представлен объектом LinkedListNode<T>, .

public sealed class LinkedListNode<T>

{

public LinkedListNode(T value);

public LinkedList<T> List { get; }

public LinkedListNode<T> Next { get; }

public LinkedListNode<T> Previous { get; }

public T Value { get; set; }

}

При добавлении элемента можно указать, чтобы он помещался в начало или конец списка, или относительно существующего в списке элемента. Для этого класс LinkedList<T> содержит специальные методы. Аналогичные методы существуют и для удаления элементов списка. Класс LinkedList<T> содержит свойства для числа элементов, для указания на первый и последний элемент. Имеются методы для поиска элементов. Некоторые из перечисленных методов и свойств представлены в следующей таблице.

|  |  |
| --- | --- |
| **Член** | **Описание** |
| AddAfter  (метод) | Добавление существующего нового узла после существующего указанного узла в LinkedList<T>  Добавление нового узла, содержащего указанное значение, после указанного существующего узла в LinkedList<T> (перегруженная версия) |
| AddBefore  (метод) | Добавление существующего нового узла перед существующим указанным узлом в LinkedList<T>  Добавление нового узла, содержащего указанное значение, перед указанным существующим узлом в LinkedList<T> (перегруженная версия) |
| AddFirst  (метод) | Добавление указанного нового узла в начало LinkedList<T>  Добавление нового узла, содержащего указанное значение, в началоLinkedList<T> (перегруженная версия) |
| AddLast  (метод) | Добавление указанного нового узла в конец LinkedList<T>  Добавление нового узла, содержащего указанное значение, в конец LinkedList<T> (перегруженная версия) |
| Clear  (метод) | Удаление всех узлов из LinkedList<T> |
| Contains  (метод) | Определение, принадлежит ли значение LinkedList<T> |
| CopyTo  (метод) | Копирует целый массив LinkedList<T> в совместимый одномерный массив Array, начиная с указанного индекса целевого массива |
| Remove  (метод) | Удаление первого вхождения указанного значения из LinkedList<T>  Удаление указанного узла из LinkedList<T> (перегруженная версия) |
| RemoveFirst  (метод) | Удаление узла в начале LinkedList<T> |
| RemoveLast  (метод) | Удаление узла в конце LinkedList<T> |
| Find  (метод) | Нахождение первого узла, содержащего указанное значение |
| FindLast  (метод) | Нахождение последнего узла, содержащего указанное значение |
| Count  (свойство) | Возвращение числа узлов, которое в действительности хранится в LinkedList<T> |
| First  (свойство) | Получение первого узла объекта LinkedList<T> |
| Last  (свойство) | Получение последнего узла объекта LinkedList<T> |

Пользователь может удалить узлы и повторно их вставить – как в тот же, так и в другой список. В результате в куче не размещается ни одного дополнительного объекта. Объект LinkedList<T> принимает значение null в качестве допустимого свойства Value для ссылочных типов и разрешает дублирование значений. Если объект LinkedList<T> пуст, свойства First и Last содержат значение null. Класс LinkedList<T> не поддерживает цепочки, разделение, циклы и другие возможности, которые могут оставить список в несогласованном состоянии.

Ниже приведён пример использования некотрых методов и свойств класса LinkedList<T>.

string[] words = { "the", "fox", "jumped", "over", "the", "dog" };

LinkedList<string> sentence = new LinkedList<string>(words);

sentence.Contains("jumped"));

sentence.AddFirst("today");

LinkedListNode<string> mark1 = sentence.First;

sentence.RemoveFirst();

sentence.AddLast(mark1);

sentence.RemoveLast();

sentence.AddLast("yesterday");

mark1 = sentence.Last;

sentence.RemoveLast();

sentence.AddFirst(mark1);

sentence.RemoveFirst();

LinkedListNode<string> current = sentence.FindLast("the");

sentence.AddAfter(current, "old");

sentence.AddAfter(current, "lazy");

current = sentence.Find("fox");

sentence.AddBefore(current, "quick");

sentence.AddBefore(current, "brown");

mark1 = current;

LinkedListNode<string> mark2 = current.Previous;

current = sentence.Find("dog");

try

{

sentence.AddBefore(current, mark1);

}

catch (InvalidOperationException ex)

{

Console.WriteLine("Exception message: {0}", ex.Message);

}

sentence.Remove(mark1);

sentence.AddBefore(current, mark1);

var tune = new LinkedList<string>();

tune.AddFirst("do"); // do

tune.AddLast("so"); // do - so

tune.AddAfter(tune.First, "re"); // do - re- so

tune.AddAfter(tune.First.Next, "mi"); // do - re - mi- so

tune.AddBefore(tune.Last, "fa"); // do - re - mi - fa- so

tune.RemoveFirst(); // re - mi - fa - so

tune.RemoveLast(); // re - mi - fa

var miNode = tune.Find("mi");

tune.Remove(miNode); // re - fa

tune.AddFirst(miNode); // mi - re - fa

Классы Queue<T> и Stack<T> реализуют структуры данных «очередь» и «стек» на основе массива. Конструкторы данных классов, как и конструкторы класса List<T>, позволяют создать объект на основе другой коллекции, а также указать значение для емкости (но емкость не доступна в виде отдельного свойства). Элементы классов вполне предсказуемы и описаны ниже в таблицах.

|  |  |
| --- | --- |
| **Член** | **Описание** |
| Clear  (метод) | Очистка очереди ‑ удаление всех элементов |
| Contains  (метод) | Проверка, содержится ли указанный элемент в очереди |
| CopyTo  (метод) | Копирование очереди в массив |
| Count  (свойство) | Количество элементов (свойство только для чтения) |
| Dequeue  (метод) | Извлечение элемента из очереди |
| Enqueue  (метод) | Помещение элемента в очередь |
| GetEnumerator  (метод) | Получение перечислителя |
| Peek  (метод) | Чтение очередного элемента без его удаления из очереди |
| ToArray  (метод) | Преобразование очереди в массив |
| TrimExcess  (метод) | Усечение размера внутреннего массива до необходимой минимальной величины |

|  |  |
| --- | --- |
| Член | Описание |
| Clear  (метод) | Очистка стека ‑ удаление всех элементов |
| Contains  (метод) | Проверка, содержится ли указанный элемент в стеке |
| CopyTo  (метод) | Копирование стека в массив |
| Count  (свойство) | Количество элементов (свойство только для чтения) |
| GetEnumerator  (метод) | Получение перечислителя |
| Peek  (метод) | Чтение очередного элемента без его удаления из стека |
| Pop  (метод) | Извлечение элемента из стека |
| Push  (метод) | Помещение элемента в стек |
| ToArray  (метод) | Преобразование стека в массив |
| TrimExcess  (метод) | Усечение размера внутреннего массива до необходимой минимальной величины |

**Типы для работы с коллекциям-множествами**

В библиотеке базовых классов платформы .NET существуют два класса для представления множеств (коллекций, не содержащих повторяющихся элементов) – HashSet<T> и SortedSet<T>. Оба класса реализуют интерфейс ISet<T>.

Класс HashSet<T> описывает множество, в котором вхождение элемента проверяется на основе хэш-кода. Конструкторы класса HashSet<T> позволяют создать множество на основе коллекции, а также указать объект, реализующий интерфейс IEqualityComparer<T> для проверки равенства элементов множества.

Кроме реализации методов интерфейса ISet<T>, класс HashSet<T> содержит экземплярный метод RemoveWhere для удаления элементов, удовлетворяющих заданному предикату. Статический метод CreateSetComparer возвращает экземпляр класса, реализующего IEqualityComparer<HashSet<T>>. Следующий пример показывает использование HashSet<T>:

var setOne = new HashSet<char>("the quick brown fox");

var setTwo = new HashSet<char>("jumps over the lazy dog");

setOne.IntersectWith(setTwo); // the uro

Console.WriteLine(setOne.Contains('t')); // True

Console.WriteLine(setOne.Contains('j')); // False

setTwo.RemoveWhere(c => c < 'k');

Класс SortedSet<T> – это множество, поддерживающее набор элементов в отсортированном порядке. Конструкторы класса SortedSet<T> позволяют создать множество на основе коллекции, а также указать объект, реализующий интерфейс IComparer<T> для проверки порядка элементов множества. Набор методов SortedSet<T> схож с набором методов класса HashSet<T>. Экземплярный метод Reverse изменяет порядок элементов на противоположный, а метод GetViewBetween возвращает фрагмент («окно») исходного множества между двумя элементами.

var setOne = new SortedSet<char>("А роза упала на лапу Азора");

// setOne = Аазлнопру

var setTwo = setOne.GetViewBetween('б', 'р');

setTwo.Add('в'); // успешно, setTwo = взлнопр

setTwo.Add('ф'); // исключение, так как символ не принадлежит окну

**Типы для работы с коллекциями-словарями**

Словарь это коллекция, хранящая пары ключ/значение, с возможностью доступа к элементам по ключу. Базовая библиотека платформы .NET предлагает набор коллекций-словарей, как классических, так и с различными дополнительными возможностями.

Универсальный класс Dictionary<TKey, TValue> – это классический словарь с возможностью указать типы для ключа и значения. Данный класс является одним из наиболее часто используемых классов-коллекций (наряду с классом List<T>). Класс Dictionary<TKey, TValue> реализует обе версии интерфейса IDictionary. Ниже приведен пример использования класса.

// конструируем словарь и помещаем в него один элемент

var d = new Dictionary<string, int> {{"One", 1}};

d["Two"] = 2; // помещаем ещё один элемент, используя индексатор

d["Two"] = 22; // обновляем существующий элемент

d["Three"] = 3;

Console.WriteLine(d["Two"]); // печатает "22"

Console.WriteLine(d.ContainsKey("One")); // True (быстрая операция)

Console.WriteLine(d.ContainsValue(3)); // True (медленная операция)

int val;

if (!d.TryGetValue("onE", out val))

{

Console.WriteLine("No val");

}

// Различные способы перечисления словаря

foreach (var kv in d)

{

Console.WriteLine(kv.Key + "; " + kv.Value);

}

foreach (string s in d.Keys)

{

Console.WriteLine(s);

}

foreach (int i in d.Values)

{

Console.WriteLine(i);

}

Словарь может работать с элементами любого типа, так как у любого объекта можно получить хэш-код и сравнить объекты на равенство, используя методы GetHashCode и Equals. Пользовательские типы могут переопределять данные методы, предоставляя их эффективную реализацию. Кроме этого, конструктору словаря можно передать объект, реализующий интерфейс IEqualityComparer<TKey>. Типичным примером использования такого подхода является конструирование словаря, обеспечивающего сравнение строк-ключей независимо от их регистра:

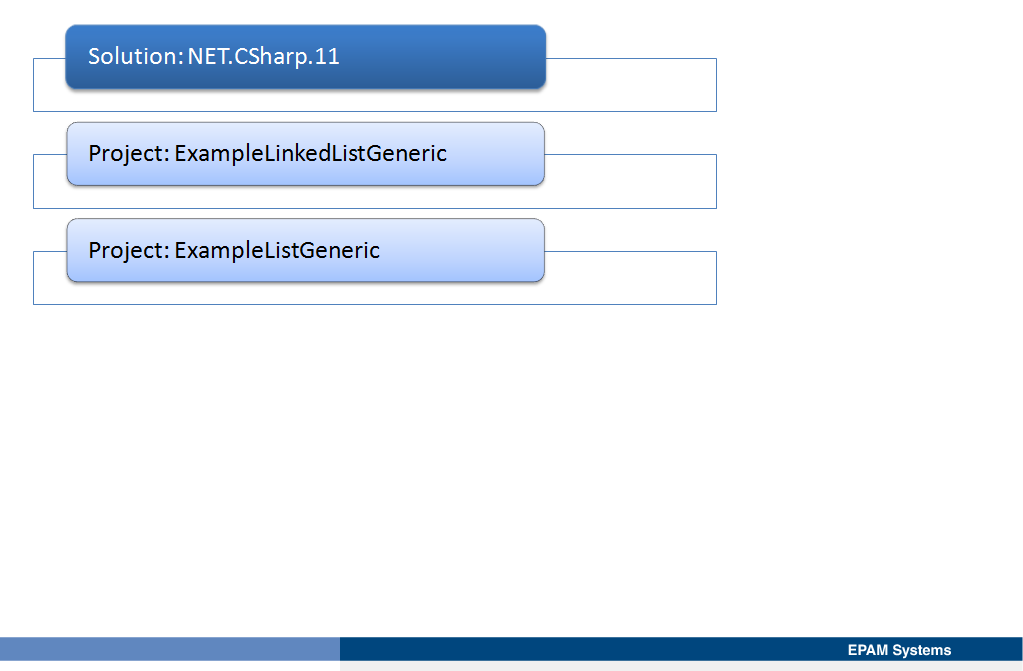
var d = new Dictionary<string, int>(StringComparer.OrdinalIgnoreCase);

Платформа .NET предоставляет два обобщенных класса-словаря, организованных так, что их элементы всегда отсортированы по ключу:

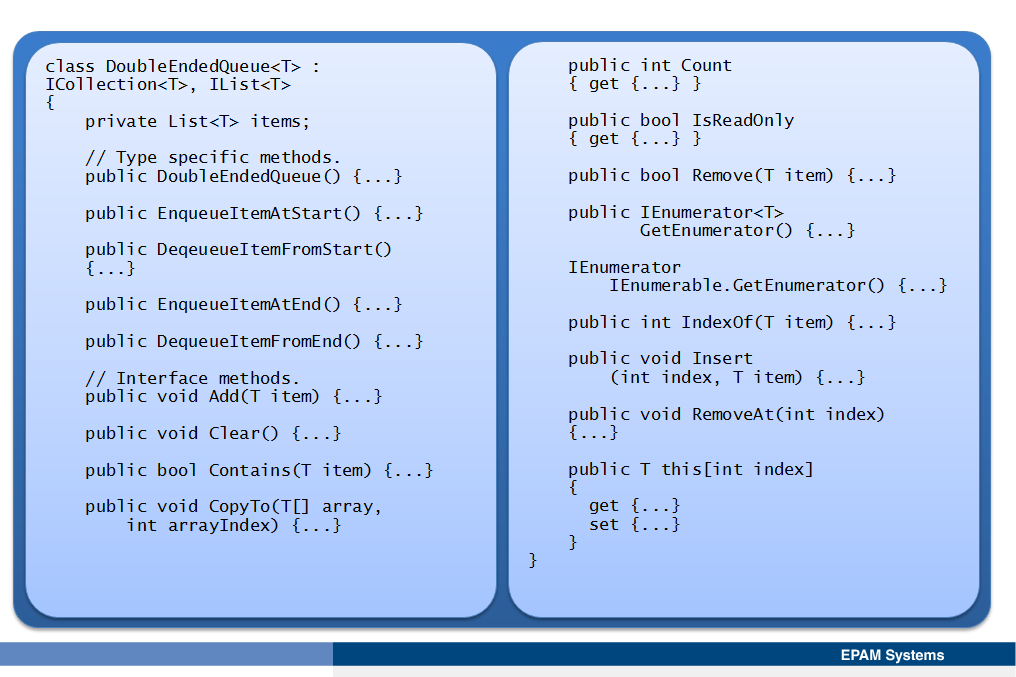
* SortedDictionary<TKey,TValue>;
* SortedList<TKey,TValue>.

Данные классы используют разные внутренние структуры для хранения элементов словаря. Класс SortedDictionary<TKey,TValue> быстрее классов SortedList и SortedList<TKey,TValue> при выполнении вставки элементов. Но классы SortedList и SortedList<TKey,TValue> могут предоставить возможность, которой нет у SortedDictionary<TKey,TValue> ‑ доступ к элементу не только по ключу, а и с использованием целочисленного индекса.

## Демонстрация: Обобщенные классы коллекции .NET Framework



## Реализация класса коллекции DoubleEndedQueue



Простой класс коллекция хранит одно значение. Для реализации такого класса коллекции можно определить обобщенный класс, обеспечивающий для хранения элементов соответствующую структуру. В некоторых ситуациях можно использовать типы коллекции, предлагаемые .NET Framework, внутри пользовательского типа; в других сценариях можно задать пользовательские типы для хранения элементов, которые добавляются в коллекцию. В пользовательском типе необходимо реализовать интерфейс ICollection<T>. Если строятся линейные коллекции, необходимо также реализовать интерфейс IList<T>.

В следующем примере кода показан класс коллекция, реализующий двусвязную очередь. Класс предоставляет методы, позволяющие вставлять в очередь и удалять из нее элементы с любого конца. Для хранения элементов коллекция использует объект List<T>.

class DoubleEndedQueue<T> : ICollection<T>, IList<T>

{

// Определение объекта List<T> для хранения элементов,

// добавляемых в коллекцию.

private List<T> items;

// Добавление в класс конструктора.

public DoubleEndedQueue()

{

items = new List<T>();

}

// Методы для добавления и извлечения элементов.

public void EnqueueItemAtStart(T item)

{

// Реализация деталей не показана.

throw new **NotImplementedException**();

}

public T DeQueueItemFromStart()

{

// Реализация деталей не показана.

throw new **NotImplementedException**();

}

public void EnqueueItemAtEnd(T item)

{

// Реализация деталей не показана.

throw new **NotImplementedException**();

}

public T DeQueueItemFromEnd()

{

// Реализация деталей не показана.

throw new **NotImplementedException**();

}

#region ICollection<T> Members

// Определение метода Add для добавления элемента в коллекцию.

public void Add(T item)

{

items.Add(item);

}

// Определение метода Clear для удаления всех элементов из коллекции.

public void Clear()

{

items.Clear();

}

// Определение метода Contains, возвращающего значения true или false

// в зависимости от того, содержит ли коллекции заданный элемент

public bool Contains(T item)

{

return items.Contains(item);

}

// Определение метода CopyTo для копирования элементов ICollection

// в массив, начиная с конкретного индекса Array.

public void CopyTo(T[] array, int arrayIndex)

{

items.CopyTo(array, arrayIndex);

}

// Определение метода Remove для удаления из коллекции заданного

// элемента. Значения true или false возвращаются

// в зависимости от того, была ли операция успешной

public bool Remove(T item)

{

return items.Remove(item);

}

// Определение свойства только для чтения Count, возвращающего

// количество элементов в коллекции

public int Count

{

get

{

return items.Count;

}

}

// Определение свойства только для чтения IsReadOnly,

// возвращающего значения true или false в зависимости

// от того, является ли коллекция коллекцией только для чтения.

public bool IsReadOnly

{

get

{

return false;

}

}

#endregion

#region IEnumerable<T> Members

// Определение метода GetEnumerator, который возвращает

// объект IEnumerator<T>

public IEnumerator<T> GetEnumerator()

{

// Не реализован. Следующий урок

throw new **NotImplementedException**();

}

#endregion

#region IEnumerable Members

// Добавление метода IEnumerable.GetEnumerator method.

// Это не универсальная версия метода GetEnumerator.

IEnumerator IEnumerable.GetEnumerator()

{

// Всегда возвращает результатом вызывая метод GetEnumerator

return GetEnumerator();

}

#endregion

#region IList<T> Members

// Добавление метода IndexOf, возвращающего индекс первого

// появляющегося в коллекции элемента.

public int IndexOf(T item)

{

return items.IndexOf(item);

}

// Добавление метода Insert, добавляющего в коллекцию

// элемент по индексу.

public void Insert(int index, T item)

{

items.Insert(index, item);

}

// Добавление метода Insert, удаляющего из коллекции

// элемент по индексу.

public void RemoveAt(int index)

{

items.RemoveAt(index);

}

// Добавление индексатора, позволяющего чтение/запись

// элементов коллекции, основанные на индексе.

public T this[int index]

{

get

{

return items[index];

}

set

{

items[index] = value;

}

}

#endregion

}

## Реализация класса коллекции Dictionary



При реализации класса коллекции словарь нужно имплементировать интерфейс IDictionary<TKey, TValue>, который в свою очередь наследуется от интерфейса ICollection<T>, поэтому необходимо также реализовать и этот интерфейс.

В следующем примере кода показан класс коллекция словарь, использующий объект Dictionary<TKey, TValue> для внутреннего хранения данных. Обычная коллекция Dictionary<TKey, TValue> требует, чтобы добавляемые в коллекцию элементы имели уникальное значение ключа, и метод Add интерфейса IDictionary выбрасывал исключение ArgumentException, если обнаружен дубликат ключа. Следующий класс коллекция IntelligentDictionary реализует дополнительный метод AddItem. Этот метод принимает те же параметры, что и метод Add интерфейса IDictionary, но, если обнаруживает дубликат ключа, генерирует новый случайный ключ и пытается снова добавить элемент в колекцию. Используемый ключ передается обратно в качестве возвращаемого значения.

class IntelligentDictionary<TKey, TValue> : IDictionary<TKey, TValue>

{

// Определение объекта Dictionary<TKey, TValue>

// для хранения добавляемых в коллекцию данных.

Dictionary<TKey, TValue> items;

// Определение консруктора класса.

public IntelligentDictionary()

{

items = new Dictionary<TKey, TValue>();

}

// Добавление в словарь пар ключ/значение.

// Если существует дубликат ключа, генерируется новый случайный ключ,

// и осуществляется попытка опять добавить элемент в коллекцию.

// Возвращается сгенерированный ключ

public TKey AddItem(TKey key, TValue value)

{

throw new NotImplementedException();

}

#region Implementation of IEnumerable<out KeyValuePair<TKey,TValue>>

public IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator()

{

throw new NotImplementedException();

}

#endregion

#region Implementation of ICollection<KeyValuePair<TKey,TValue>>

// Определение перегруженного метода Add для реализации

// интерфейса ICollection<T>. В качестве параметра типа

// используется универсальный тип KeyValuePair<TKey, TValue>.

public void Add(KeyValuePair<TKey, TValue> item)

{

// Вызов уже определенного метода Add, использующего

// свойства параметра item в качестве параметров.

Add(item.Key, item.Value);// AddItem

}

// Определение метода Clear для удаления всех элементов из коллекции

public void Clear()

{

items.Clear();

}

// Определение перегруженного метода Contains реализующего интерфейс

// ICollection<T>. В качестве параметра типа

// используется универсальный тип KeyValuePair<TKey, TValue>.

public bool Contains(KeyValuePair<TKey, TValue> item)

{

return items.Contains(item);

}

// Определение метода CopyTo, копирующего элементы коллекции в массив.

// Массив является одномерным и хранит экземпляры

// KeyValuePair<TKey, TValue> типа.

public void CopyTo(KeyValuePair<TKey, TValue>[] array, int arrayIndex)

{

((ICollection<KeyValuePair<TKey, TValue>>)items).CopyTo(array, arrayIndex);

}

// Определение свойства только для чтения Count,

// возвращающего число элементов коллекции.

public int Count

{

get { return items.Count; }

}

// Определение свойства только для чтения IsReadOnly,

// возвращающего значения true или false в зависимости

// от того, является ли коллекция коллекцией только для чтения.

public bool IsReadOnly

{

get { return false; }

}

// Добавление метода Remove, который удаляет элемент из коллекции,

public bool Remove(KeyValuePair<TKey, TValue> item)

{

return items.Remove(item.Key);

}

#endregion

#region Implementation of IDictionary<TKey,TValue>

// Определение метода Add для добавления в коллекцию нового элемента,

// определенному парой ключ/значение.

public void Add(TKey key, TValue value)

{

items.Add(key, value);

}

// Определение метода ContainsKey method, возвращающего значения

// true или false в зависимости от того, содержит ли коллекция заданный

// ключем элемент.

public bool ContainsKey(TKey key)

{

return items.ContainsKey(key);

}

// Определение свойства Keys, возвращающего ключи, которые

// хранятся в коллекции.

public ICollection<TKey> Keys

{

get { return items.Keys; }

}

// Определение метода Remove, удаляющего из коллекции элемент

// с указанным ключем и возвращающий значения

// true или false в зависимости от успешно выполненого методы

public bool Remove(TKey key)

{

return items.Remove(key);

}

// Определение метода TryGetValue, который возвращает булевское значение,

// чтобы указать успешно ли выполнен метод и, если это так, установить значение

// выходного параметра в значение соответствующее заданному ключу

public bool TryGetValue(TKey key, out TValue value)

{

return items.TryGetValue(key, out value);

}

// Определение свойства Values, возвращающего объект ICollection

// значение которого хранится в коллекции.

public ICollection<TValue> Values

{

get

{

return items.Values;

}

}

// Определение индексатора для чтения/записи значения

// коллекции, основанном на значении ключа.

public TValue this[TKey key]

{

get

{

return items[key];

}

set

{

items[key] = value;

}

}

#endregion

#region Implementation of IEnumerable

IEnumerator IEnumerable.GetEnumerator()

{

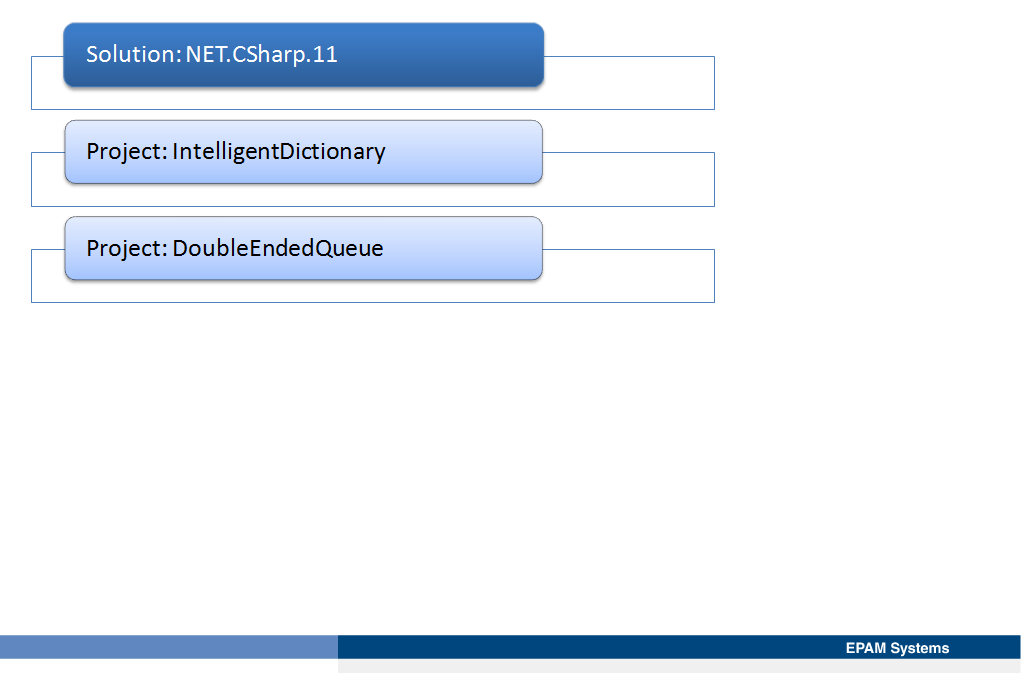
return GetEnumerator();

}

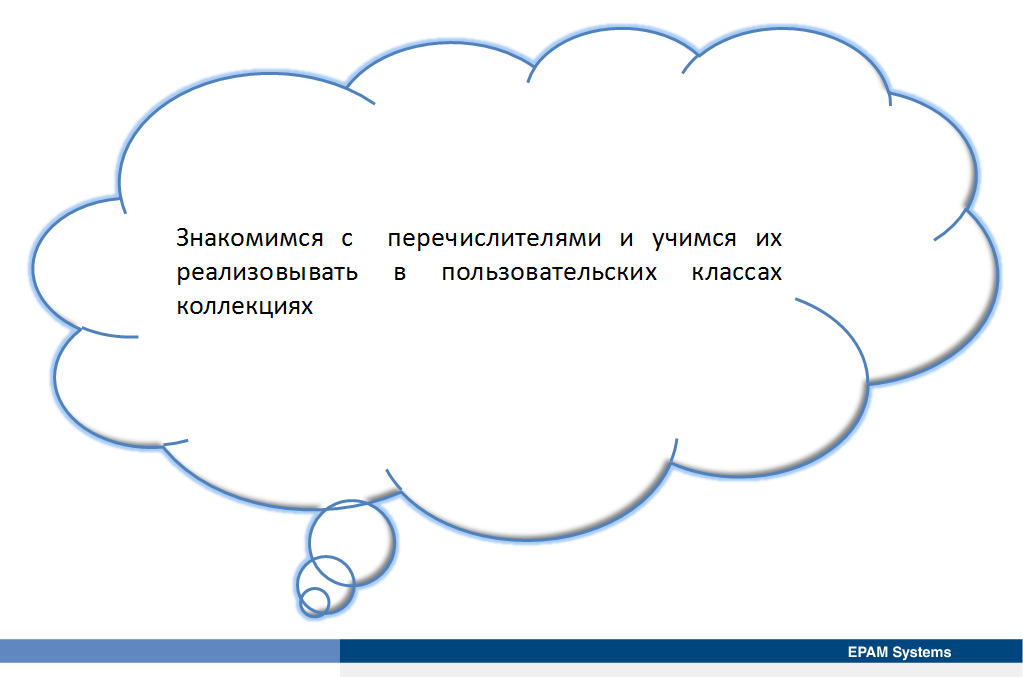
#endregion

}

## Демонстрация: Реализация пользовательского класса коллекции

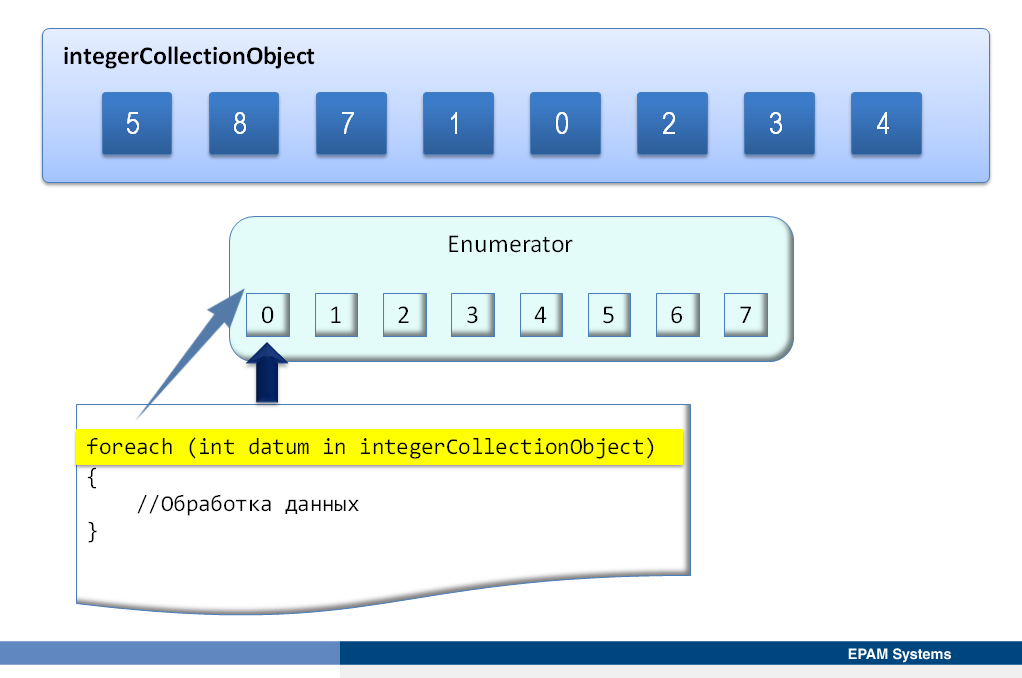


# Урок 6: Добавление перечислителя пользовательскому классу коллекции



C# предоставляет оператор foreach для перебора содержимого коллекции. Оператор foreach создает объект перечислитель коллекции, предоставляющий средства извлечения из коллекции по очереди каждого элемента. При определении пользовательского класса коллекции необходимо реализовать механизм, который оператор foreach может использовать для создания этого перечислителя. Урок знакомит с перечислителями и объясняет, как их реализовать в пользовательских классах коллекциях.

## Что такое перечислитель?



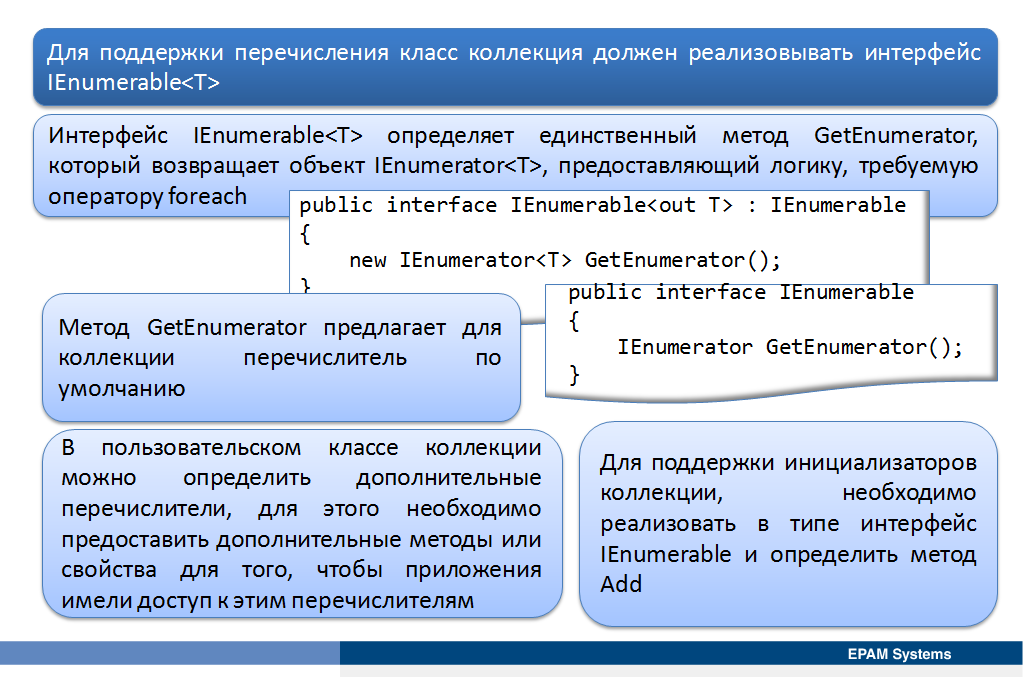
Перечислитель это тип, обеспечивающий метод, возвращающий каждый элемент коллекции в порядке, соответствуеющему коллекции. В некоторых коллекциях порядок не важен. Например, в классе список порядок определяет последовательность, в которой элементы добавляются в него. В других классах коллекциях порядок более значимый. Например, в классе SortedList элементы будут возвращены в порядке возрастания ключа, который указывается при добавлении элементов к коллекции.

О перечислителе можно думать как об указателе на элементы в коллекции. При создании экземпляра перечислителя можно его инициализировать, обратившись к первому элементу в коллекции. Указанное значение в коллекции извлекается по этой ссылке, а затем указатель перемещается на ссылку на следующий элемент в последовательности.

При использовании оператора foreach в пользовательском приложении, оператор foreach использует перечислитель, который создает пользовательский класс для получения каждого из элементов коллекции. Можно реализовать перечислитель для возвращения элементов из пользовательского класса коллекции в порядке наиболее подходящем для этого класса коллекции.

Перечислитель обеспечивает доступ только для чтения коллекции в одном направлении. Можно позволить приложениям перебирать элементы в классе коллекции в более чем одном порядке. Например, поддерживать прямое и обратное итерирование. Для этого в классе коллекции можно реализовать несколько перечислителей. Когда класс коллекция перебирается с помощью foreach, то автоматически используется перечислитель пользовательского класса по умолчанию, который по соглашению выполняет итерацию вперед. Однако, вручную можно вызвать именованный перечислитель, позволяющий приложению перебирать коллекцию в другом порядке.

## Что такое интерфейс IEnumerable<T>?



Для поддержки перечисления класс коллекция должен реализовывать интерфейс IEnumerable<T>. Интерфейс IEnumerable<T> определяет единственный метод GetEnumerator, который возвращает объект IEnumerator<T>, предоставляющий логику, требуемую оператору foreach.

public interface IEnumerable<out T> : IEnumerable

{

new IEnumerator<T> GetEnumerator();

}

При реализации интерфейса IEnumerable<T> метод GetEnumerator предлагает для коллекции перечислитель по умолчанию. В пользовательском классе коллекции можно определить дополнительные перечислители, однако необходимо предоставить дополнительные методы или свойства для того, чтобы приложения имели доступ к этим перечислителям.

Интерфейс IEnumerable<T> наследуется от интерфейса IEnumerable. Интерфейс IEnumerable необобщенная версия интерфейса IEnumerable<T> с собственным необобщенным методом GetEnumerator. Эти интерфейс и метод обеспечивают обратную совместимость со старыми приложениями, которые не используют обобщения. Если нет необходимости обеспечивать этот уровень поддержки, обычной практикой является выбросить исключение NotImplementedException в методе GetEnumerator интерфейса IEnumerable для обеспечения использования в приложении обобщенной типобезопастной версии метода GetEnumerator.

Для поддержки инициализаторов коллекции, необходимо реализовать в типе интерфейс IEnumerable и определить метод Add. В следующем примере показана реализация интерфейса IEnumerable<T> в классе коллекции. Класс коллекция также реализует метод Backwards, генерирующий перечислитель, который можно использовать для перебора коллекции в обратном порядке. Следует отметить, что при реализации дополнительного перечислителя возвращаетеся объект IEnumerable<T>. Этот объект ссылается на данные коллекции, но реализовывает метод GetEnumerator для создания перечислителя, возвращающего данные в соответствующей последовательности.

class CustomCollectionClass<T> : IEnumerable<T>

{

public IEnumerable<T> Backwards()

{

// Детальная реализация не приводится.

...

}

#region IEnumerable<T> Members

public IEnumerator<T> GetEnumerator()

{

// Детальная реализация не приводится.

...

}

#endregion

#region IEnumerable Members

IEnumerator IEnumerable.GetEnumerator()

{

throw new **NotImplementedException**();

}

#endregion

//...

}

В следующем примере показано использование перечислителя по умолчанию и перечислителя Backwards.

// Класс CustomCollectionClass реализует интерфейс IEnumerable<T>

// Класс CustomCollectionClass также предлагает метод Backwards,

// который возвращает объект IEnumerable<T>.

CustomCollectionClass<int> intCollection = new CustomCollectionClass<int>();

intCollection.Add(3);

intCollection.Add(5);

intCollection.Add(8);

intCollection.Add(2);

intCollection.Add(9);

intCollection.Add(1);

intCollection.Add(0);

// Конструкция foreach использует перечислитель по умолчанию.

foreach (int temp in intCollection)

{

...

}

foreach (int temp in intCollection.Backwards())

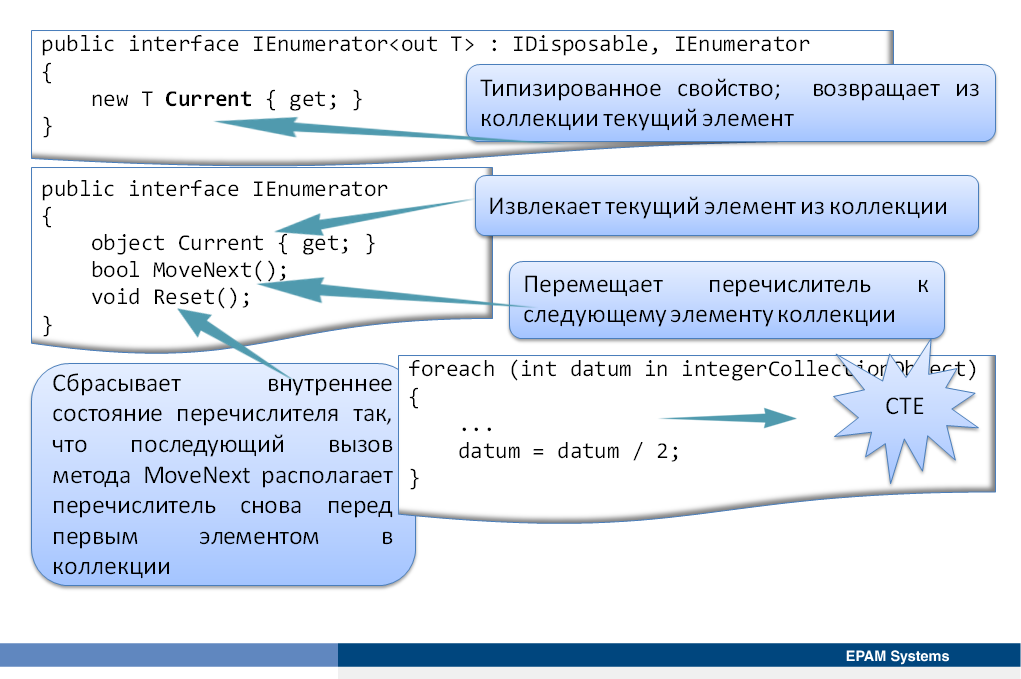
{

...

}

<http://go.microsoft.com/fwlink/?LinkId=192973>

## Что такое интерфейс IEnumerator<T>?



Объект перечислитель, возвращаемый методом GetEnumerator интерфейса IEnumerable<T>, должен реализовать интерфейс IEnumerator<T>, который определяет следующие члены:

* bool MoveNext();
* T Current { get; }
* void Reset();

public interface IEnumerator<out T> : IDisposable, IEnumerator

{

new T Current { get; }

}

public interface IEnumerator

{

object Current { get; }

bool MoveNext();

void Reset();

}

public interface IDisposable

{

void Dispose();

}

При создании экземпляра перечислителя он изначально не ссылается на элементы в коллекции. Чтобы перейти к первому элементу в коллекции, нужно использовать метод MoveNext. Этот метод возвращает логическое значение true, если найдена ссылка на элемент, в противном случае возвращает false. Затем можно извлечь элемент из коллекции с помощью свойства Current. Current это свойство только для чтения, и не может использоваться для обновления данных. Потом можно перейти к следующему элементу, вызвав метод MoveNext и использовать свойство Current для доступа к этому элементу. Получить доступ к свойству Current можно неоднократно без вызова метода MoveNext, результатом будем получение нескольких ссылок на один и тот же элемент. Метод Reset сбрасывает внутреннее состояние перечислителя так, что последующий вызов метода MoveNext располагает перечислитель снова перед первым элементом в коллекции.

Тот факт, что свойство Current является свойством только для чтения имеет влияние на управляющую переменную в цикле foreach. В следующем примере кода, переменная datum заполняется с помощью свойства Current. Переменная datum также является переменной только для чтения, и при попытке изменить ее в явном виде код не будет компилироваться.

foreach (int datum in integerCollectionObject)

{

...

datum = datum / 2; // Compilation error

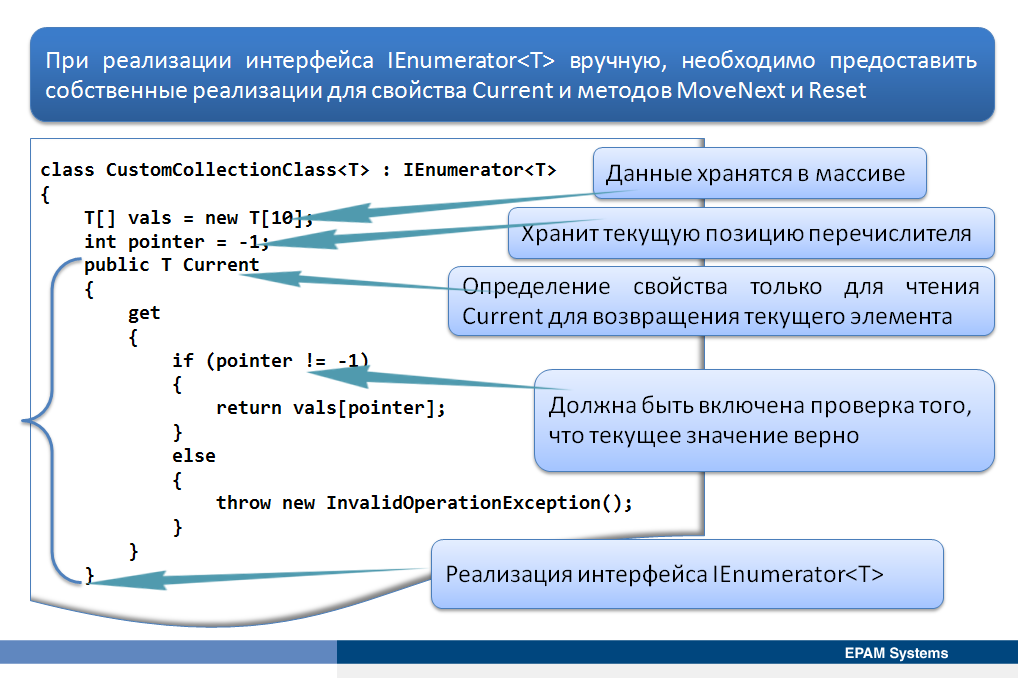
...

}

Ошибкой также будет попытка получить доступ к свойству Current до того, как метода MoveNext вызван по крайней мере один раз после создания перечислителя или после вызова метода Reset. В этой ситуации, перечислитель должен выбросить исключение InvalidOperationException. Использование свойства Current также должно выбросить исключение, если последний вызов метода MoveNext возвращает false, что означает, что указатель передвинулся за конец коллекции. Оператор foreach мешает мешает выполнять такие действия случайно, но поскольку можно непосредственно получить доступ к методам перечислителя, нужно быть готовым ловить и обрабатывать эти исключения.

http://go.microsoft.com/fwlink/?LinkId=192974

## Реализация перечислителя вручную



При реализации интерфейса IEnumerator<T> вручную, необходимо предоставить собственные реализации для свойства Current и методов MoveNext и Reset. В следующем примере показано, как можно реализовать интерфейс IEnumerator<T> вручную в пользовательском классе коллекции. Класс коллекция хранит данные в массиве. Метод MoveNext предъявляет индекс в массиве, а свойство Current возвращает элемент, который расположен согласно этому индексу.

class CustomCollectionClass<T> : IEnumerator<T>

{

// Массив хранит данные.

T[] vals = new T[10];

//...

public CustomCollectionClass() { }

// Целочисленная переменная pointer хранит текущую позицию перечислителя.

int pointer = -1;

#region IEnumerator<T> Members

// Определение свойства Current для возвращения текущего элемента.

// Должна быть включена проверка того, что текущее значение верно.

public T Current

{

get

{

// Check that the state is valid.

if (pointer != -1)

{

return vals[pointer];

}

// Генерируется исключение, если текущее значение не верно.

else

{

throw new InvalidOperationException();

}

}

}

#endregion

#region IDisposable Members

// Реализация интерфейса IDisposable.

public void Dispose() { }

#endregion

#region IEnumerator Members

// Определение свойства Current интерфейса IEnumerable.

// Возвращает свойство Current типобезопастного обобщенного

// интерфейса IEnumerator<T>.

object IEnumerator.Current

{

get { return (object)Current; }

}

// Определение метода MoveNext, который продвигает перечислитель

// к следующему элементу коллекции.

public bool MoveNext()

{

// Следует проверить, не продвинулся ли перечислитель

// за конец коллекции.

if (pointer < (vals.Length - 1))

{

pointer++;

// Возвращается значение true, если перечислитель продвинулся успешно.

return true;

}

else

{

// Возвращается значение false, если перечислитель

// продвинулся за конец коллекции.

return false;

}

}

// Определяется метод Reset, сбрасывающий внутреннее состояние перечислтеля так,

// что следующий вызов метода MoveNext располагает перечислитель

// снова перед первым элементо коллекции.

public void Reset()

{

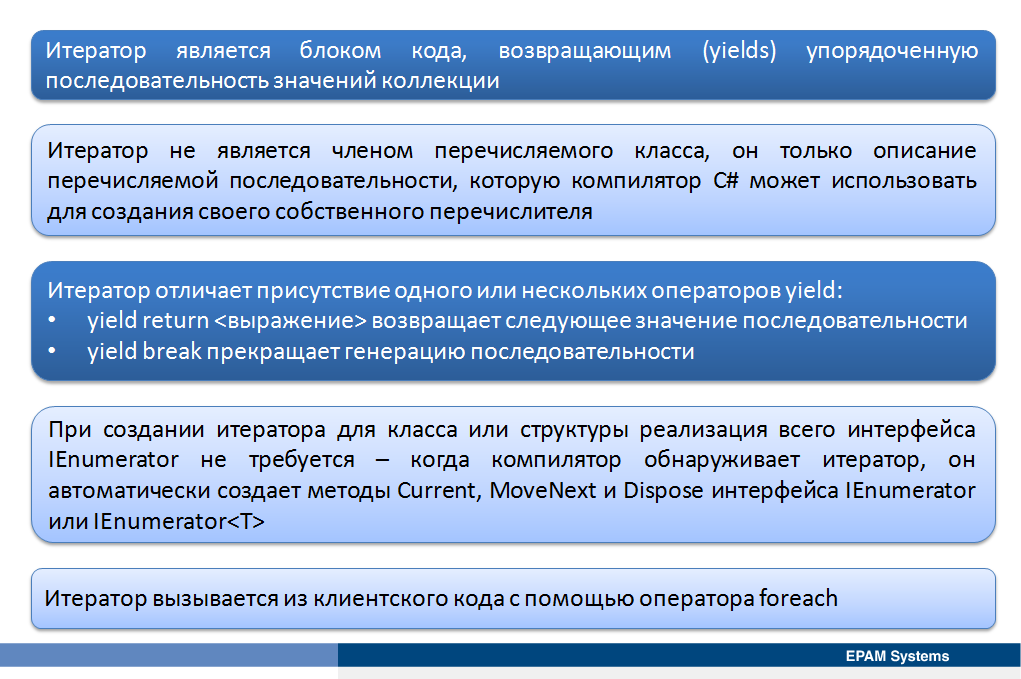
pointer = -1;

}

#endregion

}

## Реализация перечислителя с помощью итератора



В качестве реализации перечислителя вручную возможна альтернативы с помощью итератора. Итератор является блоком кода, возвращающим (yields) упорядоченную последовательность значений коллекции. Итератор не является членом перечисляемого класса, он только описание перечисляемой последовательности, которую компилятор C# может использовать для создания своего собственного перечислителя.

Итератор отличает присутствие одного или нескольких операторов yield. Оператор yield return <выражение> возвращает следующее значение последовательности, а оператор yield break прекращает генерацию последовательности. Итераторы могут использоваться в качестве тела метода, если тип метода – один из интерфейсов IEnumerator, IEnumerator<T>, IEnumerable, IEnumerable<T>.

При использовании оператором return ключевого слова yield элемент в исходной последовательности немедленно возвращается вызывающему объекту до того, как будет получен доступ к следующему элементу. Хотя итератор создается как метод, компилятор создает для него вложенный класс. Данный класс отслеживает положения итератора, пока в клиентском коде выполняется цикл foreach. Итератор вызывается из клиентского кода с помощью оператора foreach.

Можно создать итератор для класса, возвращающего элементы в обратном порядке или выполняющего операцию над каждым элементом перед тем, как итератор возвратит его. При создании итератора для класса или структуры реализация всего интерфейса IEnumerator не требуется. Когда компилятор обнаруживает итератор, он автоматически создает методы Current, MoveNext и Dispose интерфейса IEnumerator или IEnumerator<T>.

Класс BasicCollection<T> в следующем примере кода иллюстрирует реализацию итератора. Класс использует объект List<T> для хранения данных коллекции и метод FillList для их заполнения. Метод GetEnumerator реализуется с использованием итератора.

class BasicCollection<T>

{

private List<T> data = new List<T>();

public void FillList(params T[] items)

{

foreach (var datum in items)

{

data.Add(datum);

}

}

#region Implementation of IEnumerable

public IEnumerator<T> GetEnumerator()

{

foreach (var datum in data)

{

yield return datum;

}

}

#endregion

. . .

}

В методе GetEnumerator нигде явно не возвращается тип IEnumerator<T>. Ключевым моментом является использование ключевого слова yield, указывающее на значение, которое должна вернуть каждая итерация. Можно думать об операторе yield как о призыве к временному прекращению метода, передавая значение обратно вызывающей стороне. Когда в точке вызова требуется следующее значение, метод GetEnumerator продолжается с точки, в которой он был прерван, проходит цикл и возвращает следующее значение. Когда данные исчерпаны, цикл заканчивается, метод GetEnumerator завершается.

Код в методе GetEnumerator определяет итератор. Компилятор использует этот код для реализации интерфейса IEnumerator<T>, который содержит свойство Current и методы MoveNext и Reset. Такая реализация полностью соответствует функциональности, определяемой методом GetEnumerator. Увидеть этот сгенерированный код нельзя, но можно использовать оператор yield для уменьшить количество кода, который нужно писать, снивив при этом и возможность ошибок. Перечислитель, который порождает итератор, вызывается обычным образом.

BasicCollection<string> bc = new BasicCollection<string>();

bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");

foreach (string word in bc)

{

Console.WriteLine(word);

}

Данный код просто отображается содержимое объекта bc в следующем порядке.

**Twas, brillig, and, the, slithy, toves**

Возможности итерирования данных в другой последовательности можно реализовать с помощью дополнительных свойств, реализующих интерфейс IEnumerable, и использовать итератор для возврата данных. Например, в следующем примере показано свойство Reverse класса BasicCollection<T>, которое возвращает данные списка в обратном порядке.

class BasicCollection<T> : IEnumerable<T>

{

...

public IEnumerable<T> Reverse

{

get

{

for (int i = data.Count - 1; i >= 0; i--)

{

yield return data[i];

}

}

}

}

В следующем примере показано, как ссылаться на это свойство.

BasicCollection<string> bc = new BasicCollection<string>();

bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");

foreach (string word in bc.Reverse)

{

Console.WriteLine(word);

}

Этот код отображает содержимое объекта **bc** в обратном порядке, как показано в следующем примере.

**toves, slithy, the, and, brillig, Twas**

Итераторы реализуют концепцию отложенных вычислений. Каждое выполнение оператора yield return ведет к выходу из метода и возврату значения. Но состояние метода, его внутренние переменные и позиция yield return запоминаются, чтобы быть восстановленными при следующем вызове.

В следующем примере кажется, что вызов GetNumbers() приведет к «зацикливанию» программы. Однако использование итераторов обеспечивает этому методу следующее поведение. При первом вызове метод GetNumbers() возвращает значение i = 0, и состояние метода (значение переменной i) будет зафиксировано. При следующем вызове метод возвращает значение i = 1 и снова фиксирует состояние, и так далее.

public static class Helper

{

public static IEnumerable<int> GetNumbers()

{

int i = 0;

while (true) yield return i++;

}

}

Таким образом, следующий код успешно выводит на экран три числа:

foreach (var n in Helper.GetNumbers())

{

Console.WriteLine(n);

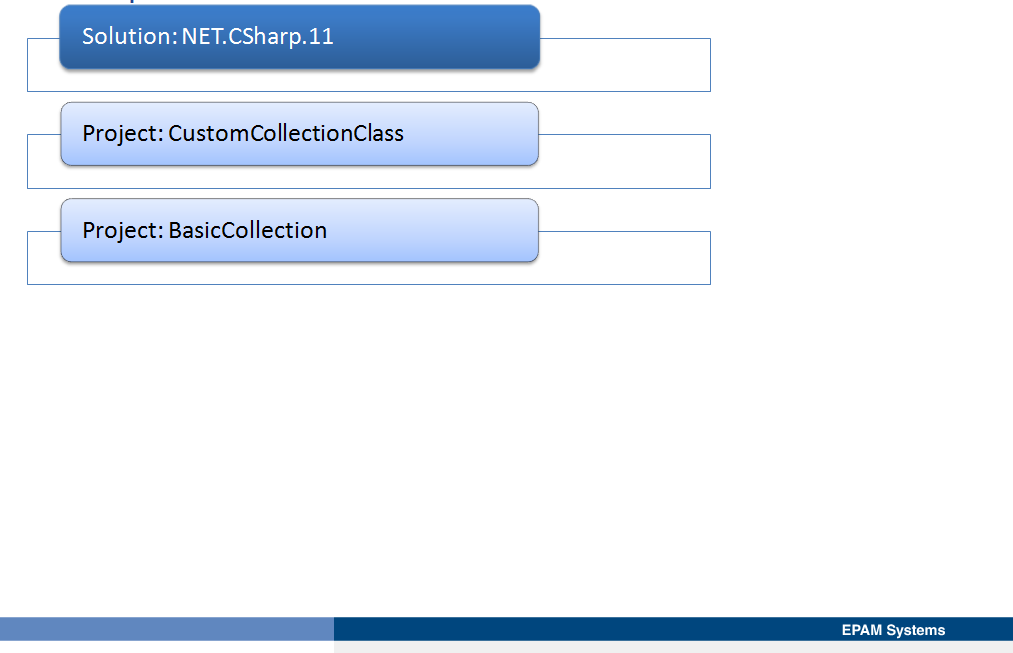
if (n == 2) break;

}

<http://go.microsoft.com/fwlink/?LinkId=192975>

<http://go.microsoft.com/fwlink/?LinkId=192976>

## Демонстрация: Добавление перечислителя пользовательскому классу коллекции



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-1)
2. Класс ArrayList это простой класс коллекция, реализующий массив, который может динамически изменять свой размер. [↑](#footnote-ref-2)
3. Можно определить обобщенные структуры в дополнение к обобщенным классам. [↑](#footnote-ref-3)
4. Класс Object реализует метод GetHashCode. Он возвращает целое число, которое идентифицирует объект. Все ссылочные типы наследуют этот метод и могут изменить его в собственной реализации. [↑](#footnote-ref-4)