



GENERICS AND COLLECTIONS

.NET & JS LAB, RD BELARUS

Anzhelika Kravchuk

Почему необходимы обобщения

Достаточно часто возникает необходимость создать класс или метод, которые одинаково хорошо работают с данными любых типов. Наследование и полиморфизм способны помочь в этом, но до определенной границы. Методы, которые должны быть полностью универсальны, принимают параметры типа `object`, что влечет за собой две основные проблемы обобщенного программирования **до версии .NET 2.0**

- отсутствие безопасности типов

```
public class ObjectStack
{
    int position;
    object[] data = new object[10];
    public void Push (object obj) { data[position++] = obj; }
    public object Pop() { return data[--position]; }
}
```

```
ObjectStack stack = new ObjectStack();
stack.Push ("s"); // Wrong type, but no error!
int i = (int)stack.Pop(); // Downcast - runtime error
```



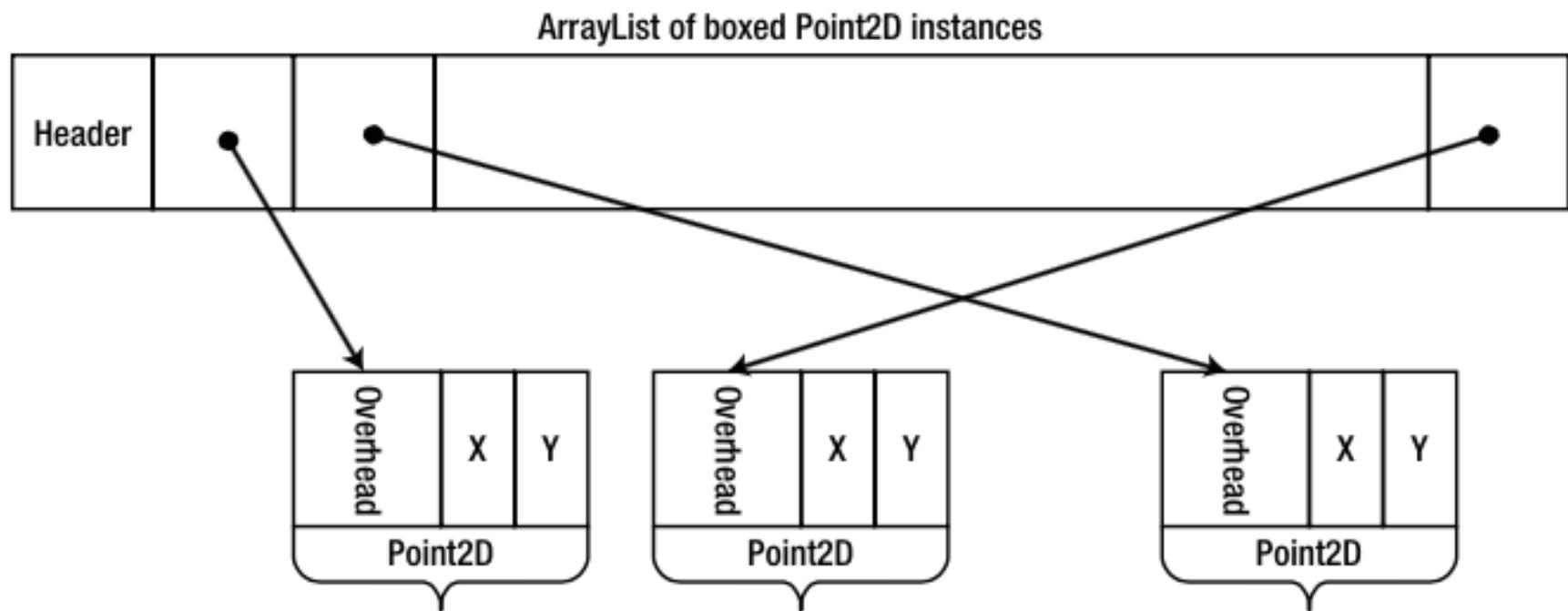
RTE

Почему необходимы обобщения

- упаковка

```
ArrayList line = new ArrayList(1000000);
for (int i = 0; i < 1000000; ++i)
{
    line.Add(new Point2D(i, i));
}
```

20 000 000 bytes vs 8 000 000 bytes



Почему необходимы обобщения

Обобщенные классы и методы позволяют писать по-настоящему **обобщенный** код без применения object с одной стороны и без специализации типа – с другой

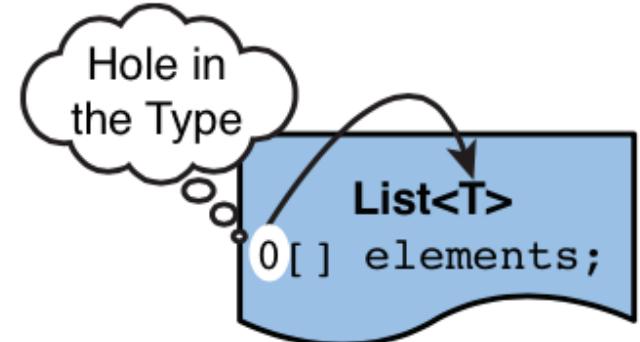
```
List<int> ages = new List<int>();  
ages.Add(10);  
ages.Add(25);  
...  
int data = ages[0];  
...  
ages.Add("Data");
```

```
public class List<T>
```

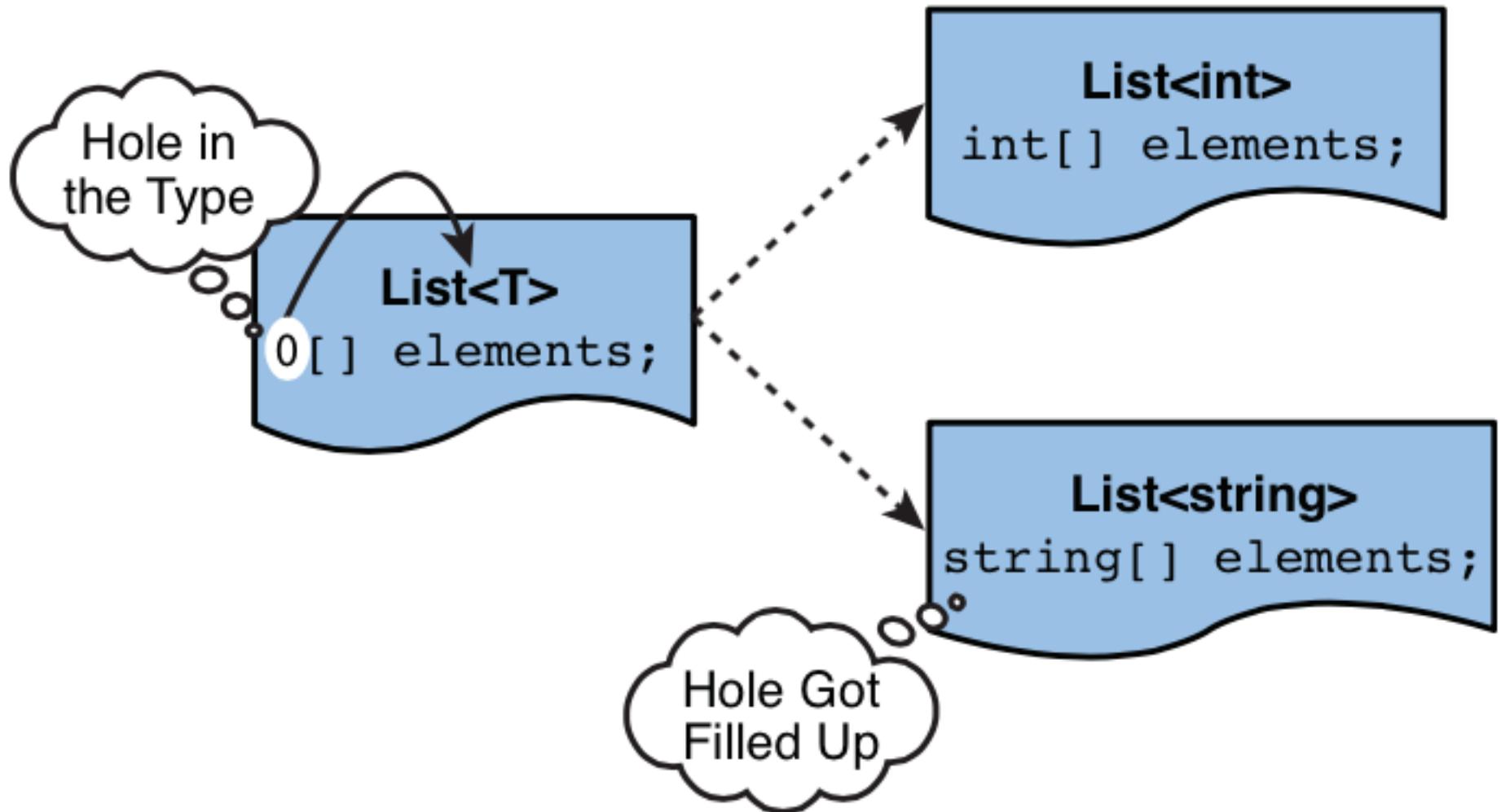
При создании экземпляра объекта на основе универсального типа необходимо указать тип для замены параметра типа

Извлечь данные из коллекции можно без использования приведения типа

СТЕ



Почему необходимы обобщения



Почему необходимы обобщения

Замена параметра типа для указанного типа не просто текстовый механизм замены, компилятор при этом выполняет полную семантическую замену

```
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];
```

Компилятор генерирует различные версии метода Add

```
public void Add(List<string> item);
```

```
public void Add(string item);
```

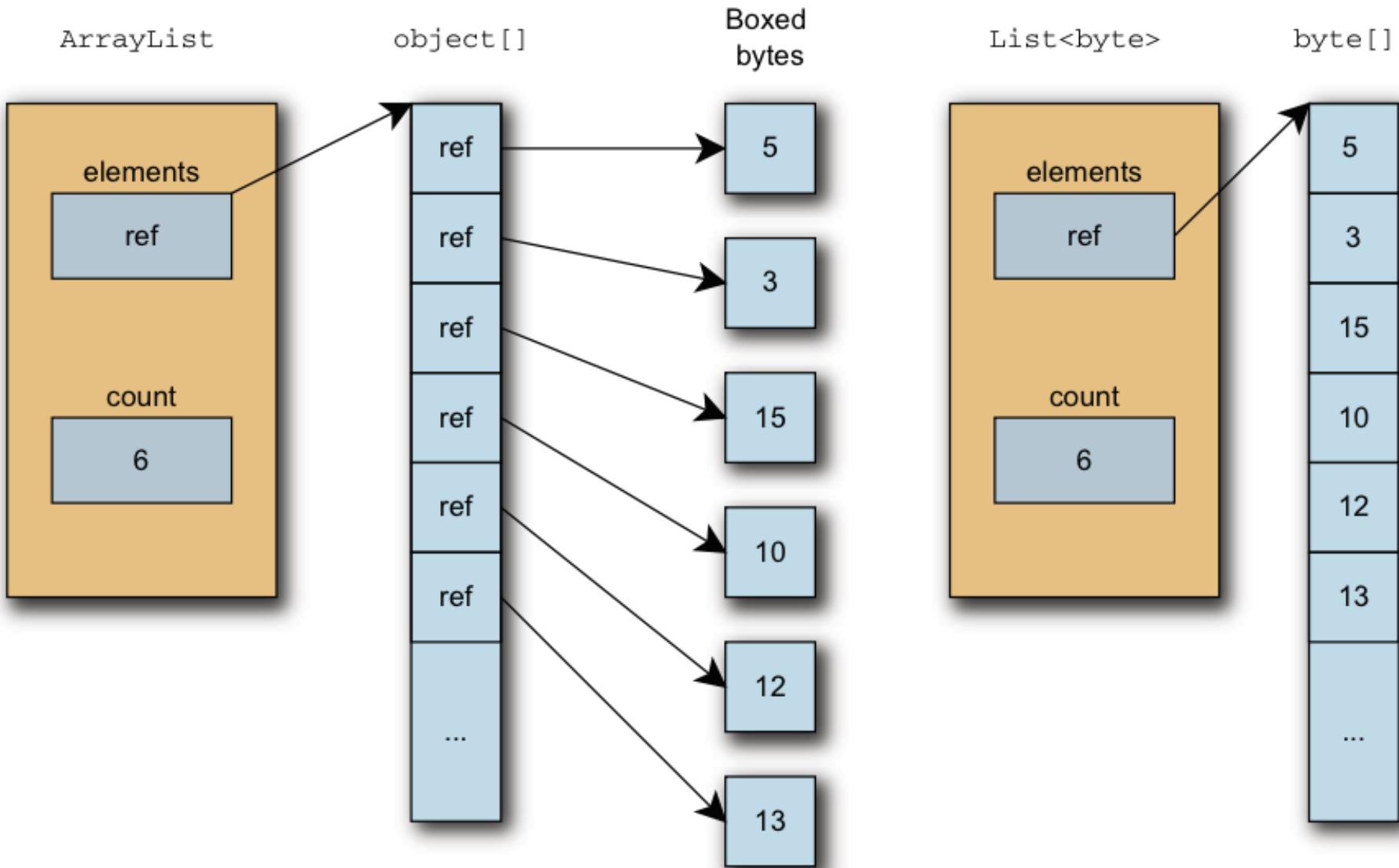
```
List<int> ages = new List<int>();  
ages.Add(10);
```

```
public void Add(int item);
```



Упаковка и разпаковка

Почему необходимы обобщения



Определение пользовательских обобщенных типов

В языке C# существует два вида обобщений: обобщенные типы (классы, интерфейсы, делегаты, структуры) и обобщенные методы

Для определения пользовательского обобщенного типа (метода) нужно добавить один или несколько параметров типа сразу после имени класса (метода) в угловых скобках

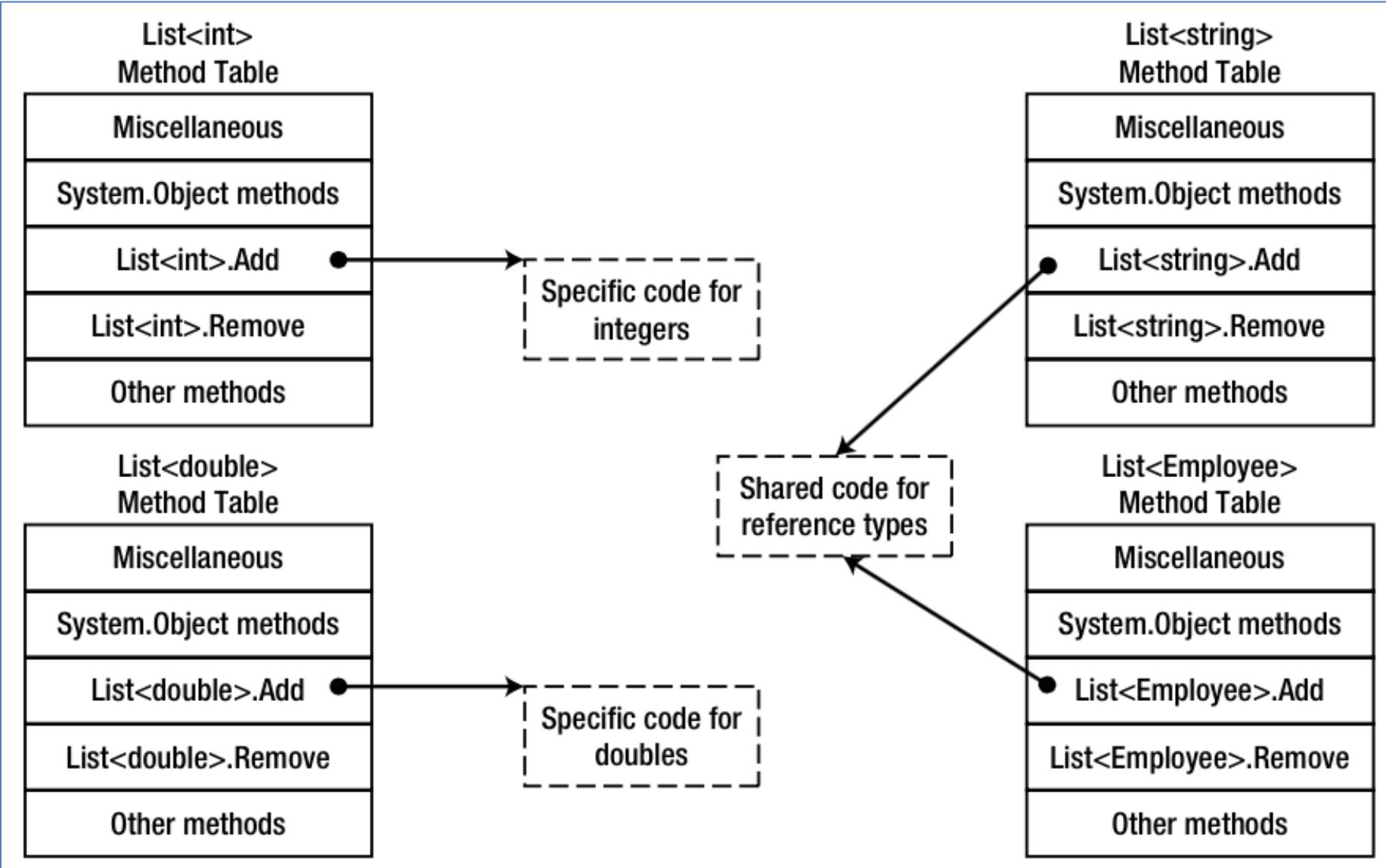
```
class PrintableCollection<TItem>
{
    TItem[] data;
    int index;
    ...
    public void Insert(TItem item)
    {
        ...
        data[index] = item;
        ...
    }
}
```

Использование обобщенного класса PrintableCollection

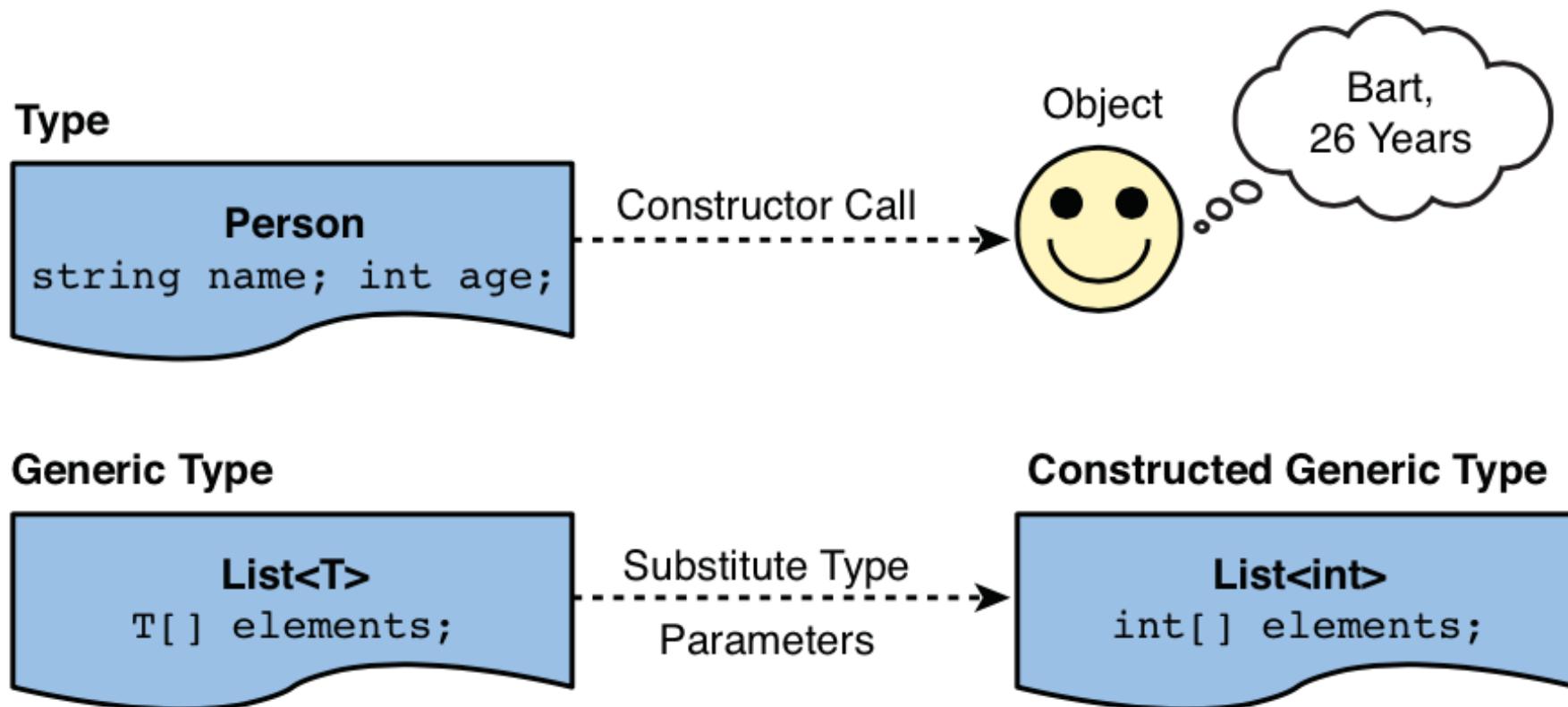
```
struct Person
{
    ...
}
...
PrintableCollection <Person> employeeList =
    new PrintableCollection <Person>();
Person employee = new Person(...);
employeeList.Insert(employee);
```

Имена, указанные для параметров типа, используются в качестве конкретных типов в классом

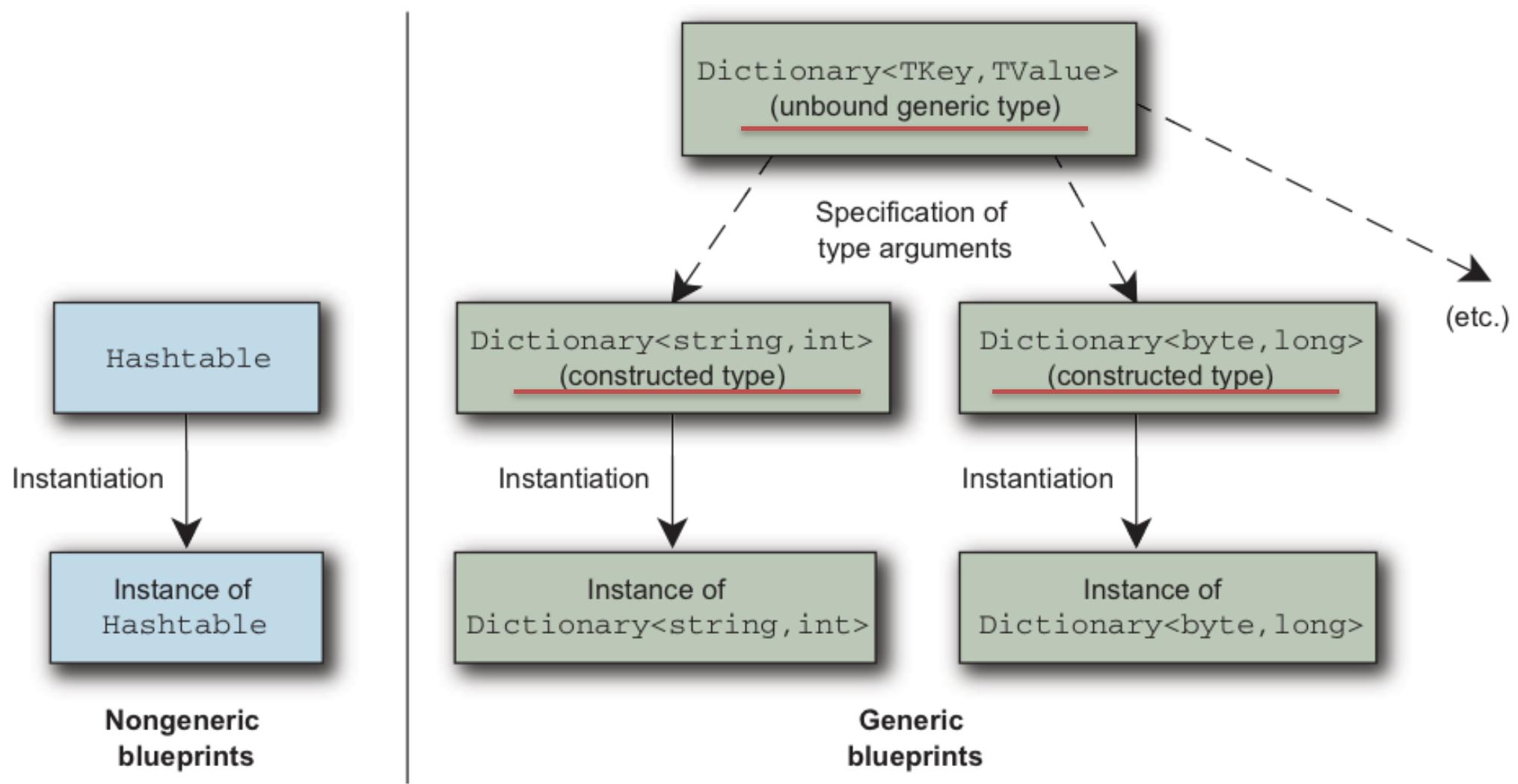
Внутреннее устройство обобщенных типов



Внутреннее устройство обобщенных типов



Внутреннее устройство обобщенных типов



В действительности весь код выполняется в
контексте закрытого сконструированного типа

Внутреннее устройство обобщенных типов

| Method signature in generic type | Method signature after type parameter substitution |
|-------------------------------------|--|
| void Add(TKey key, TValue value) | void Add(string key, int value) |
| TValue this[TKey key] { get; set; } | int this[string key] { get; set; } |
| bool ContainsValue(TValue value) | bool ContainsValue(int value) |
| bool ContainsKey(TKey key) | bool ContainsKey(string key) |

Внутреннее устройство обобщенных типов

```
namespace System.Collections.Generic
{
    public class Dictionary<TKey, TValue>
        : IEnumerable<KeyValuePair<TKey, TValue>>

    {
        public Dictionary() { ... }

        public void Add(TKey key, TValue value) { ... }

        public TValue this[TKey key]
        {
            get { ... }
            set { ... }
        }

        public bool ContainsValue(TValue value) { ... }

        public bool ContainsKey(TKey key) { ... }

        [... other members ...]
    }
}
```

Declares
method
using type
parameters

Declares
generic
class

Implements
generic
interface

Declares
parameterless
constructor

Members only declare type parameters when they're introducing new ones—and only methods can do that

Определение пользовательских обобщенных типов

Для инициализации членов класса на основе параметра типа значением по умолчанию C# предоставляет ключевое слово `default`

```
class PrintableCollection<TItem>
{
    TItem[] data;
    int index;
    TItem tempData;
    ...
    public PrintableCollection()
    {
        this.tempData = default(TItem);
        ...
    }
    ...
}
```

При компиляции генерируется код, в котором конструкция `default` заменяется значением по умолчанию, зависящим от конкретного типа

Добавление ограничений для обобщенных типов

| 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 реализовывать указанный интерфейс. Могут быть указаны несколько ограничений интерфейса. Уточняющий интерфейс может быть универсальным |
| where T : U | Тип аргумента, который поставляется для T должен вытекать из аргумента, который поставляется для U |

Добавление ограничений для обобщенных типов. Ограничение ссылочного типа

```
struct RefSample<T> where T : class
```

RefSample<IDisposable>
RefSample<string>
RefSample<int[]>

RefSample<Guid>
RefSample<int>



When a type parameter is constrained this way, you can compare references (including null) with == and !=, but be aware that unless there are any other constraints, only references will be compared, even if the type in question overloads those operators (as string does, for example).

Добавление ограничений для обобщенных типов. Ограничение значимого типа

```
class ValSample<T> where T : struct
```

ValSample<int>

ValSample< FileMode >

ValSample<object>

ValSample<StringBuilder>



When a type parameter is constrained to be a value type, comparisons using == and != are prohibited

Добавление ограничений для обобщенных типов. Ограничение конструктора

```
public T CreateInstance<T>() where T : new()
{
    return new T();
}
```

CreateInstance<int>()
CreateInstance<object>()

CreateInstance<string>()



Добавление ограничений для обобщенных типов. Ограничение преобразования типа

| Declaration | Constructed type examples |
|--|--|
| class Sample<T> where T : Stream | <i>Valid:</i> Sample<Stream> (identity conversion) <i>Invalid:</i> Sample<string> |
| struct Sample<T> where T : IDisposable | <i>Valid:</i> Sample<SqlConnection> (reference conversion) <i>Invalid:</i> Sample<StringBuilder> |
| class Sample<T> where T : IComparable<T> | <i>Valid:</i> Sample<int> (boxing conversion) <i>Invalid:</i> Sample<FileInfo> |
| class Sample<T,U> where T : U | <i>Valid:</i> Sample<Stream, IDisposable> (reference conversion) <i>Invalid:</i> Sample<string, IDisposable> |

Добавление ограничений для обобщенных типов

```
class Sample<T> where T : Stream, IEnumerable<string>, IComparable<int>
```

```
class Sample<T> where T : Stream, ArrayList, IComparable<int>
```

Значимые типы

Запечатанные классы

System.Object

System.Enum

System.ValueType

System.Delegate

Добавление ограничений для обобщенных типов

```
class Sample<T> where T : class, IDisposable, new()
class Sample<T> where T : struct, IDisposable
class Sample<T,U> where T : class where U : struct, T
class Sample<T,U> where T : Stream where U : IDisposable
```

```
class Sample<T> where T : class, struct
class Sample<T> where T : Stream, class
class Sample<T> where T : new(), Stream
class Sample<T> where T : IDisposable, Stream
class Sample<T> where T : XmlReader, IComparable, IComparable
class Sample<T,U> where T : struct where U : class, T
```



Обобщенные методы

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

```
void AddToQueue(Report report)
{
    printQueue.Add(report);
}
```

```
void AddToQueue(ReferenceGuide referenceGuide)
{
    printQueue.Add(referenceGuide);
}
```

```
void AddToQueue<DocumentType>(DocumentType document)
{
    printQueue.Add(document);
}
```

Использование обобщенного метода с параметром типа для параметра метода снимает дублирование кода при сохранении безопасности типов

Определение обобщенного метода

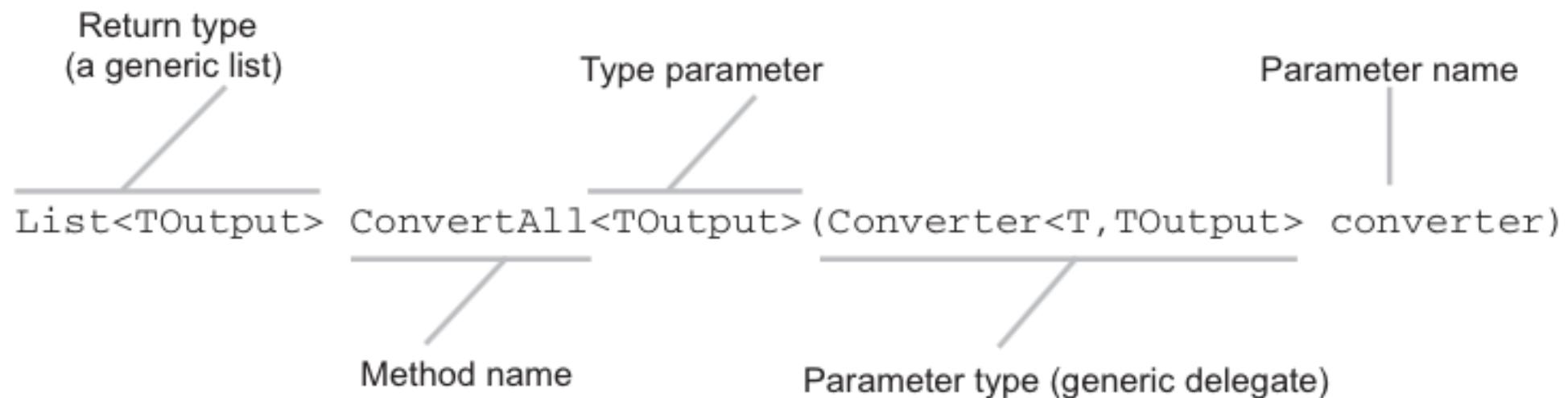
При определении обобщенного метода используется параметр типа

```
ResultType MyMethod<Parameter1Type, ResultType>(Parameter1Type param1)
    where ResultType : new()
{
    ResultType result = new ResultType();
    return result;
}
```

На параметры типа можно добавить ограничения

Параметры типа можно использовать в списке параметров метода, типе возвращаемого значения, в любом месте в теле метода

Определение обобщенного метода



```
List<TOutput> ConvertAll<TOutput>(Converter<string, TOutput> converter)
```

```
List<Guid> ConvertAll(Converter<string, Guid> converter)
```

Использование обобщенных методов

```
T PerformUpdate<T>(T input)
{
    T output = // Update parameter.
    return output;
}
...
string result = PerformUpdate<string>("Test");
int result2 = PerformUpdate<int>(1);
```

При вызове обобщенного метода в дополнение к другим параметрам нужно предоставить параметры типа

PerformUpdate<string>(1);



При компиляции приложения, использующего обобщенный метод, компилятор создает версии обобщенного метода для каждой комбинации параметров типа

```
string PerformUpdate(string input)
{
    string output = // Update parameter.
    return output;
}
```

```
int PerformUpdate(int input)
{
    int output = // Update parameter.
    return output;
}
```

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

Основными обобщенными делегатами являются Action и Func

Action<T>

Делегат, используемый для инкапсуляции методов, не возвращающих значения

Func<T, TResult>

Делегат, используемый для инкапсуляции методов, возвращающих значение

Использование обобщенных делегатов .NET Framework

```
Action<string, int> myDelegate = null;
myDelegate += ((param1, param2) =>
{
    Console.WriteLine("{0} : {1}", param1, param2.ToString());
});
if (myDelegate != null)
{
    myDelegate("Value", 5);
}
```

Примеры использования
делегатов Action и Func

```
Func<string, int, string> myDelegate = null;
myDelegate += ((param1, param2) =>
{
    return String.Format("{0} : {1}", param1, param2.ToString());
});
if (myDelegate != null)
{
    string returnedValue;
    returnedValue = myDelegate("Value", 5);
    Console.WriteLine(returnedValue);
}
```

Определение обобщенных интерфейсов

```
interface IPrinter<DocumentType> where DocumentType : IPrintable
{
    void PrintDocument(DocumentType Document);
    PrintPreview PreviewDocument(DocumentType Document);
}

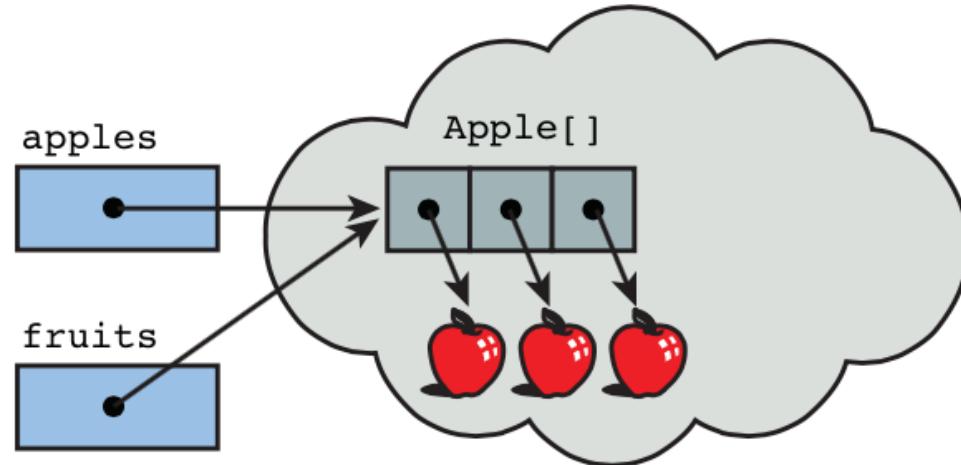
class Printer<DocumentType> : IPrintable<DocumentType>
    where DocumentType : IPrintable
{
    public void PrintDocument(DocumentType Document)
    {
        // Send document to printer.
        IPrintable doc = (IPrintable)Document;
        PrintService.Print(doc);
    }
    public PrintPreview PreviewDocument(DocumentType Document)
    {
        // Return a new PrintPreview object.
        IPrintable doc = (IPrintable)Document;
        return new PrintPreview(doc)
    }
}
```

При определении обобщенного интерфейса указываются параметры типа, которые используются в членах, определенных в интерфейсе

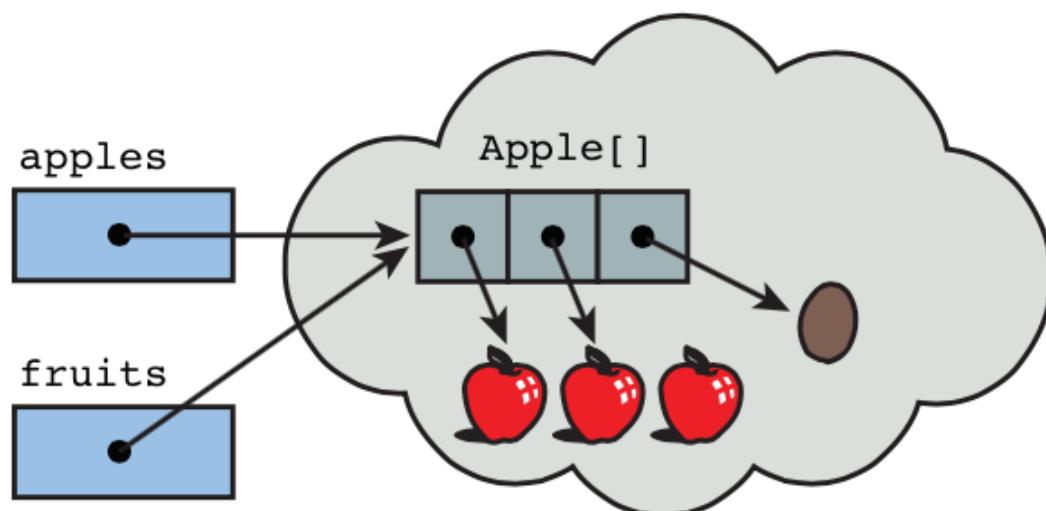
В обобщенном классе можно реализовать обобщенный интерфейс и использовать его для ссылки на класс

Что такое ковариантность

```
Apple[] apples = new Apple[] { apple1, apple2, apple3 };
Fruit[] fruits = apples;
```



```
fruits[2] = new Coconut();
```



Что такое инвариантность

```
string myString = "Hello";
object myObject = myString;
```

```
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;
    }
}
```

Все строки являются объектами

```
Wrapper<string> stringWrapper = new
Wrapper<string>();
IWrapper<string> storedStringWrapper =
stringWrapper;
storedStringWrapper.SetData("Hello");
Console.WriteLine("Stored value is {0}",
storedStringWrapper.GetData());
```

Stored value is Hello

```
IWrapper<object> storedObjectWrapper =
stringWrapper;
```

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

CTE

RTE

Нельзя присвоить объект `IWrapper<A>` ссылке типа `IWrapper`, даже если тип `A` является производным от типа `B`

Определение и реализация ковариантного интерфейса

```
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;  
    }  
}
```

```
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());
```

```
IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;
```

CTE

Определение и реализация ковариантного интерфейса

Если параметр типа в обобщенном интерфейсе появляется только в качестве возвращаемого значения методов, можно сообщить компилятору, что некоторые неявные преобразования являются законными и что можно не соблюдать строгую безопасность типов

```
interface IRetrieveWrapper<out T>
{
    T GetData();
}
```

Необходимо указать ключевое слово `out` при объявлении параметра типа

Ковариантность позволяет присваивать объект `IRetrieveWrapper<A>` ссылке `IRetrieveWrapper`, пока существует допустимое преобразование из типа А в тип В или тип А является производным от типа В

```
IRetrieveWrapper<object> retrievedObjectWrapper = stringWrapper;
```



```
Wrapper<int> intWrapper = new Wrapper<int>();
IStoreWrapper<int> storedIntWrapper = intWrapper;
...
IRetrieveWrapper<object> retrievedObjectWrapper = intWrapper;
```



Определение и реализация контравариантного интерфейса

Контравариантность позволяет использовать обобщающий интерфейс для ссылки на объект типа В через ссылку на тип А, пока тип В является производным от типа А

```
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

```
class ObjectComparer : IComparer<object>
{
    int IComparer<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;
    }
}
```

Определение и реализация контравариантного интерфейса

```
object x = ...;  
object y = ...;  
ObjectComparer objectComparer = new ObjectComparer();  
IComparer<object> objectComparator = objectComparer;  
int result = objectComparator(x, y);
```

```
IComparer<String> stringComparator = objectComparer;
```



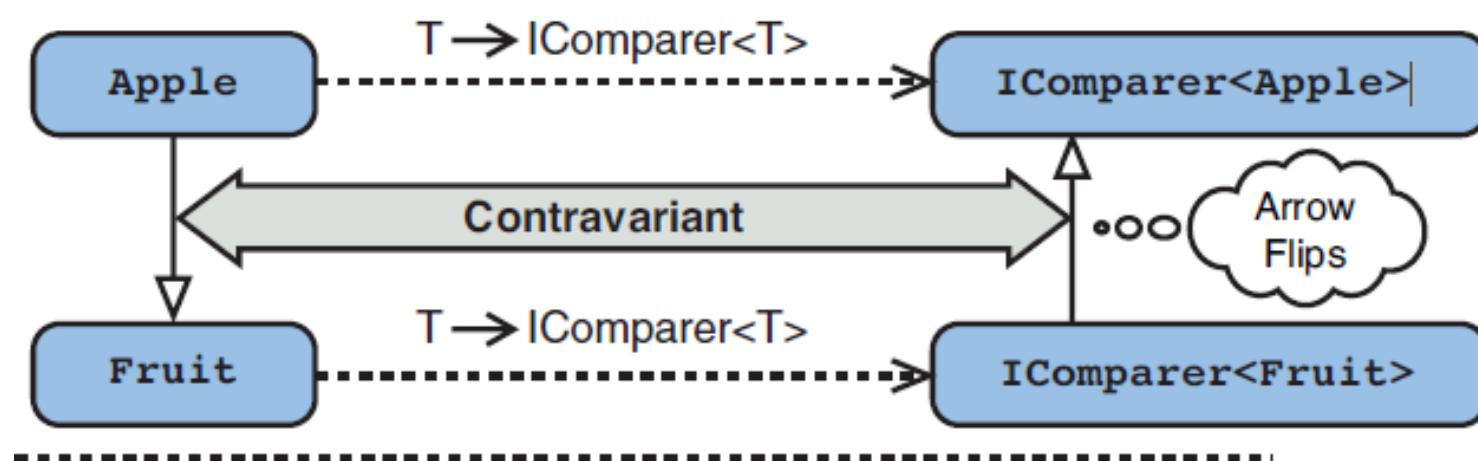
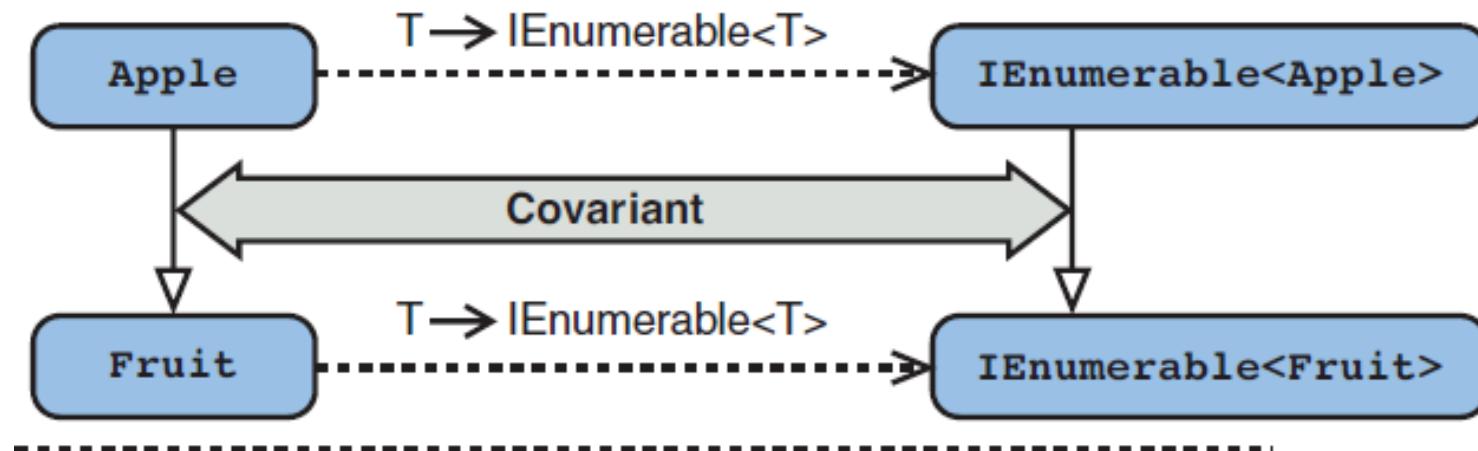
```
public interface IComparer<in T>  
{  
    int Compare(T x, T y);  
}
```

Ключевое слово `in` сообщает компилятору C#, что в качестве параметра типа методов можно передавать тип `T` или любой тип, производный от `T`

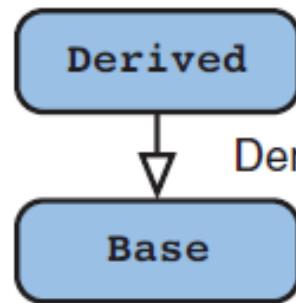
Нельзя использовать тип `T` в качестве возвращаемого типа любых методов

Если тип А предоставляет некоторые операции, свойства или поля, и, если тип В является производным от типа А, он должен поддерживать те же операции (которые могут вести себя иначе, если они были переопределены), свойства и поля

Определение и реализация контравариантного интерфейса



Legend



Derived can be used where Base is expected.

Определение и реализация контравариантного интерфейса

Подводя итоги ковариации и контравариации

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

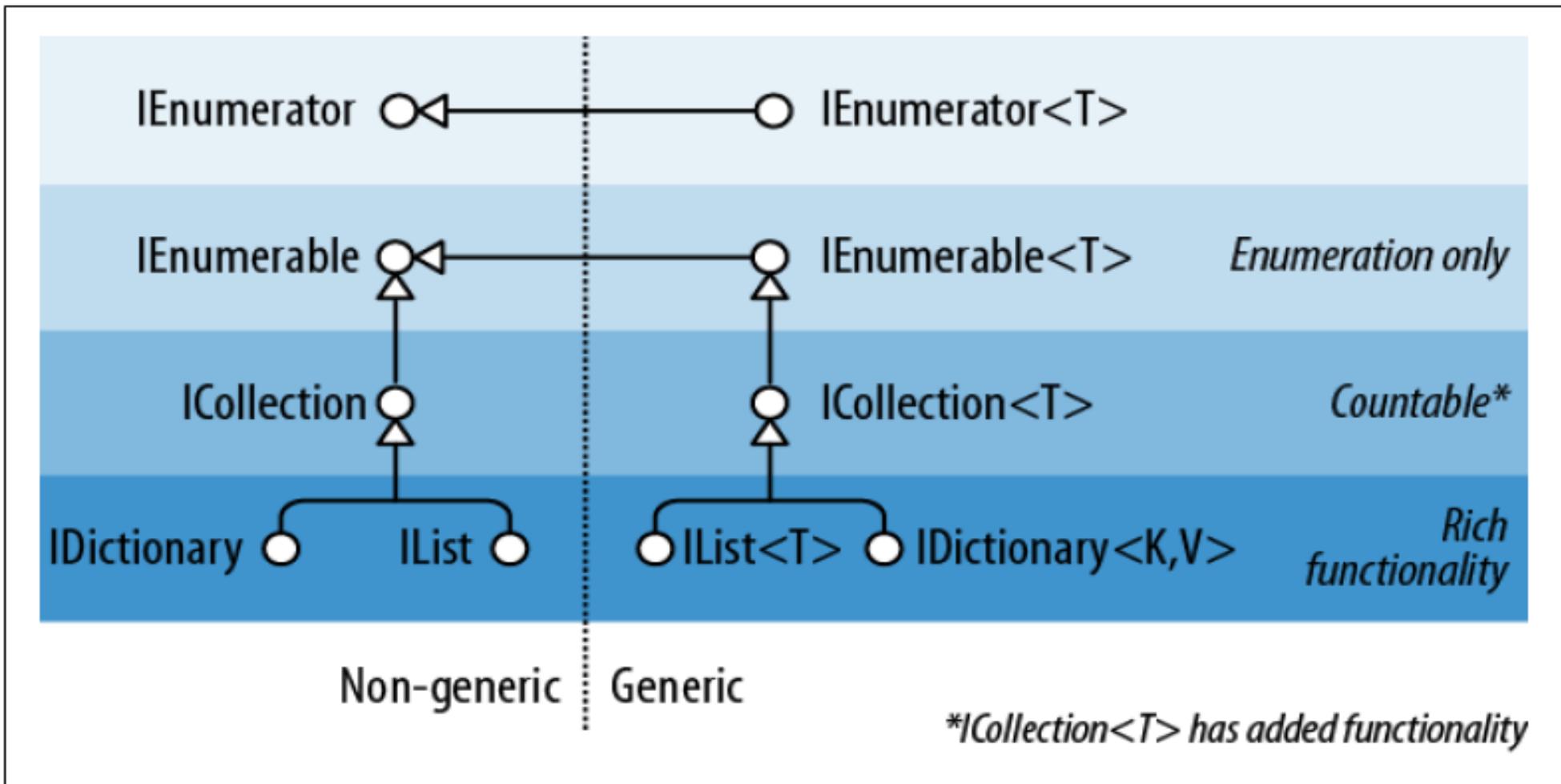
Контравариантность. Если методы в обобщенном интерфейсе могут принимать параметры `object`, они могут принимать параметры `string`. (Если можно выполнять операции с использованием объекта, значит можно выполнять ту же операцию с использованием строки, потому что все строки являются объектами.)

Коллекции

| Namespace | Contains |
|---|--|
| <code>System.Collections</code> | Nongeneric collection classes and interfaces |
| <code>System.Collections.Specialized</code> | Strongly typed nongeneric collection classes |
| <code>System.Collections.Generic</code> | Generic collection classes and interfaces |
| <code>System.Collections.ObjectModel</code> | Proxies and bases for custom collections |
| <code>System.Collections.Concurrent</code> | Thread-safe collections |

- Интерфейсы, которые определяют стандартные протоколы коллекций
- Готовые к использованию классы коллекций (списки, словари и т.д.)
- Базовые классы для написания коллекций, специфичных для приложений

Обобщенные интерфейсы коллекций .NET Framework



Шаблон проектирования «Итератор»

Шаблон (поведенческий) проектирования «Итератор» предназначен для последовательного доступа ко всем элементам коллекции (агрегата), не раскрывая ее внутренней структуры

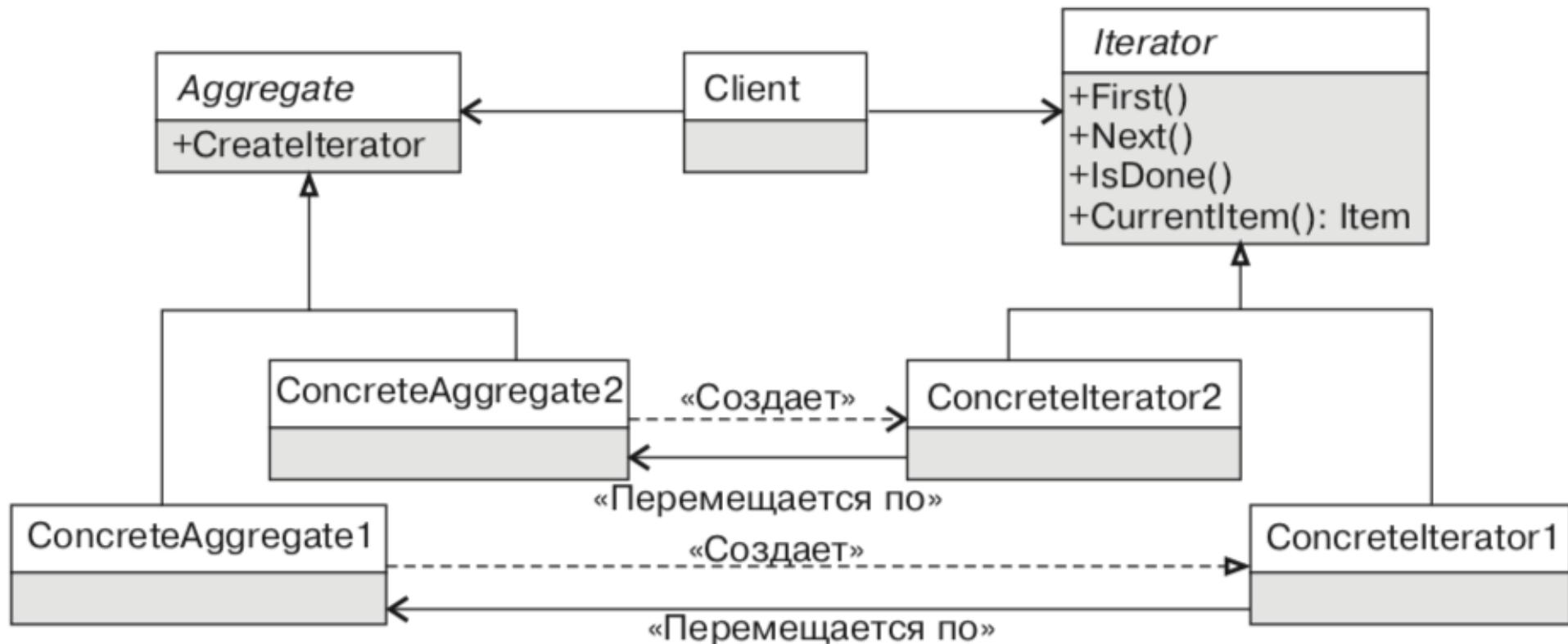
Данный шаблон применяется, если необходимо:

- ✓ обеспечить доступ к содержимому объекта без раскрытия его реализации
- ✓ (или) предоставить нескольких способов обхода коллекции
- ✓ (или) реализовать единый интерфейс обхода различных коллекций

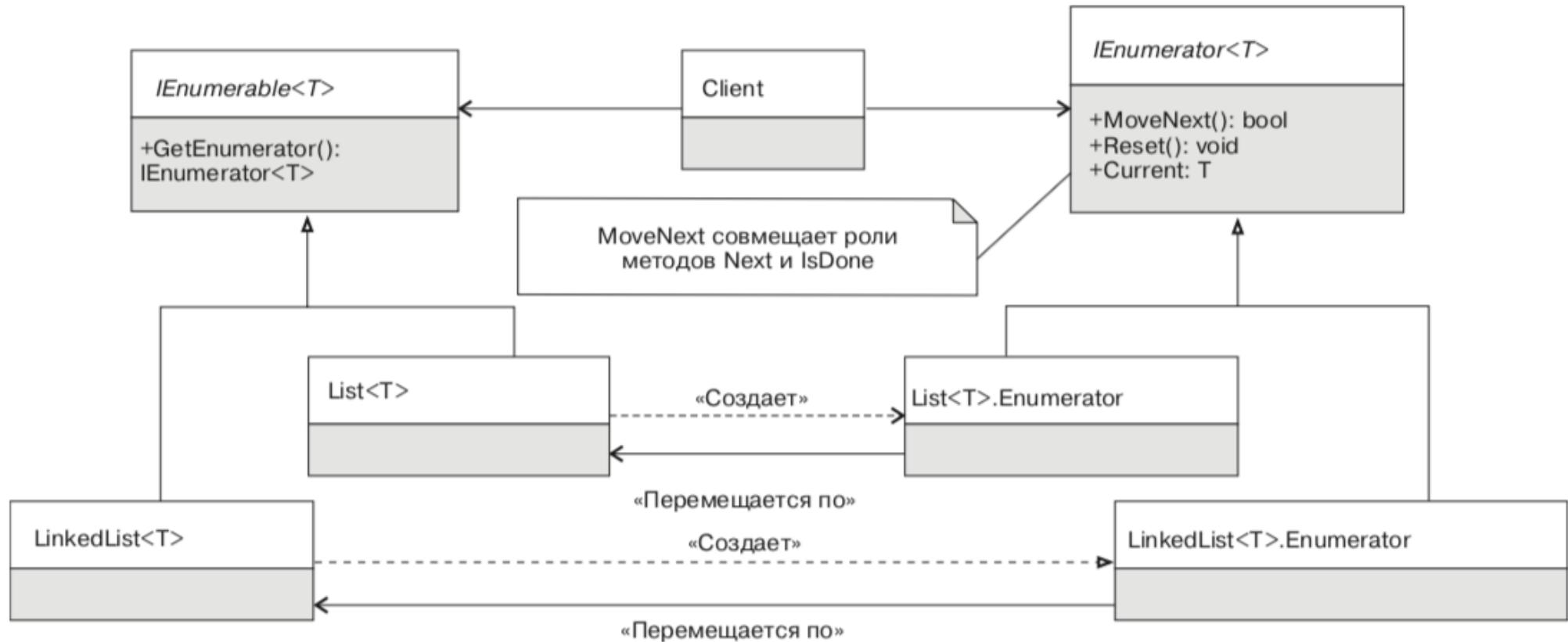
Итераторы предоставляют абстрактный интерфейс для доступа к содержимому составных объектов, не раскрывая клиентам их внутреннюю структуру. В результате получается четкое разделение ответственостей: клиенты получают возможность работать с разными коллекциями унифицированным образом, а классы коллекций становятся проще за счет того, что ответственность за перебор ее элементов возлагается на отдельную сущность.

Многие языки высокого уровня (C#, C++, F#, PHP) имеют собственную поддержку итераторов, но если стандартных возможностей не достаточно или нужна своя логика обхода элементов, то возможно реализовать данный шаблон самостоятельно (реализация потребуется и для пользовательских типов коллекций)

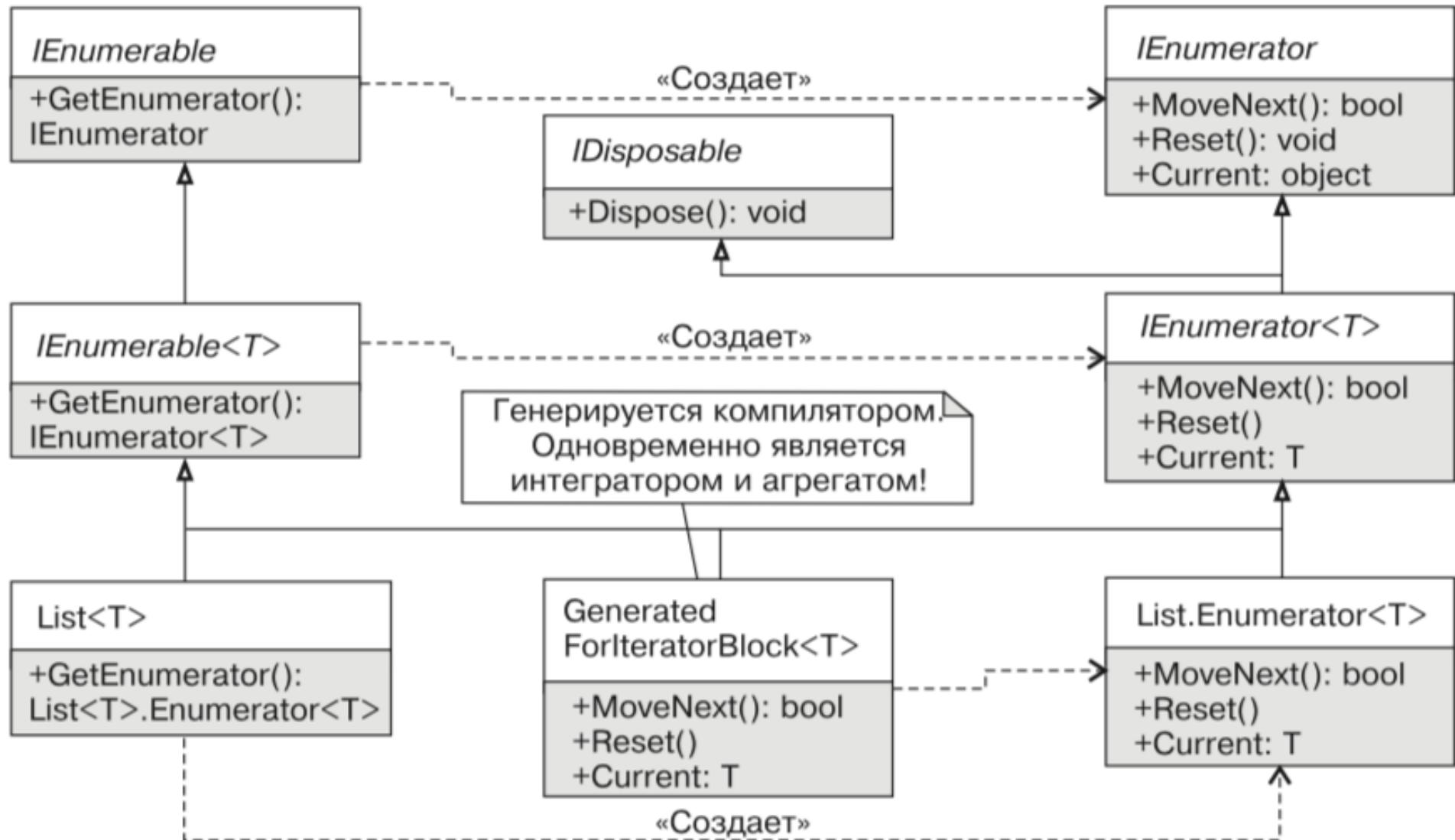
Шаблон проектирования «Итератор»



Шаблон проектирования «Итератор»



Шаблон проектирования «Итератор»



Шаблон проектирования «Итератор»

В качестве основного инструмента реализации паттерна проектирования «Однонаправленный итератор» используются интерфейсы `IEnumerable` (`System.Collections`) и обобщенная версия интерфейса `IEnumerable<T>` (`System.Collections.Generics`)

Во второй версии в языке программирования C# появилась новая возможность языка, под названием «итератор», которая реализуется в языке программирования C# с помощью блока итератора (Iterator Block), однако на самом деле эта возможность может быть использована как для реализации паттерна проектирования «Итератор», так и паттерна проектирования «Генератор».

Что такое итератор?

Реализация интерфейса `IEnumerable` и `IEnumerable <T>` может понадобится

- для поддержки оператора `foreach`
- для взаимодействия со всем, что ожидает стандартной коллекции
- для удовлетворения требований более развитого интерфейса пользовательской коллекции
- для поддержки инициализаторов коллекций

Для реализации интерфейса `IEnumerable/IEnumerable<T>` потребуется предоставить перечислитель, который можно реализовать

- вернув перечислитель внутренней коллекции, если класс является оболочкой другой коллекции
- через итератор с использование оператора `yield return`
- за счет создания экземпляра собственной реализации `IEnumerable/IEnumerable<T>`

Шаблон проектирования «Итератор»

```
public class CustomIterator<T>
{
    private readonly T[] container;
    private int currentIndex;

    public CustomIterator() { }

    public CustomIterator(T[] list)
    {
        currentIndex = -1;
        container = list;
    }

    public T Current
    {
        get
        {
            if (currentIndex < 0 || currentIndex > container.Length)
                throw new InvalidOperationException();
            return container[currentIndex];
        }
    }
}
```

Шаблон проектирования «Итератор»

```
public void Reset()
{
    currentIndex = -1;
}

public bool MoveNext()
{
    if (_currentIndex < _container.Length)
    {
        currentIndex++;
        return (_currentIndex < _container.Length);
    }
    return false;
}
```

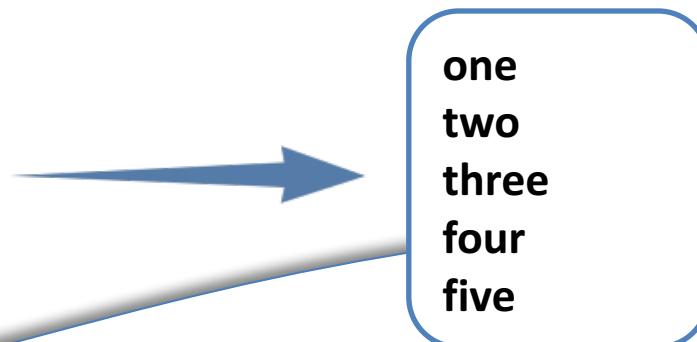
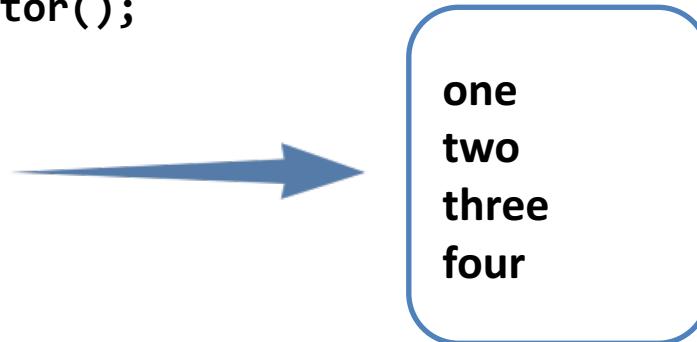
Шаблон проектирования «Итератор»

```
var array = new CustomContainer<string>
    (new string[] { "one", "two", "three", "four" });
```

```
CustomIterator<string> iterator = array.Iterator();
while (iterator.MoveNext())
{
    Console.WriteLine(iterator.Current);
}
iterator.Reset();
```

```
array = new CustomContainer<string>
    (new string[] { "one", "two", "three", "four", "five" });
```

```
iterator = array.Iterator();
while (iterator.MoveNext())
{
    Console.WriteLine(iterator.Current);
}
iterator.Reset();
```



Шаблон проектирования «Итератор»

Использование паттерна «Итератор» в языке C# - оператор **foreach** упрощает работу с итераторами, самостоятельно вызывая **MoveNext** до тех пор, пока эта функция не вернет **false**:

```
public CustomIterator<T> GetEnumerator()
{
    return new CustomIterator<T>(_container);
}
```

```
var array = new CustomContainer<string>
    (new string[] { "one", "two", "three", "four", "five" });
```

```
foreach (var temp in array)
{
    Console.WriteLine(temp);
```



```
one  
two  
three  
four  
five
```

Итерация по коллекции

Все коллекции реализуют интерфейс `ICollection`, определяющий метод `GetEnumerator`, который возвращает объект `Enumerator` (перечислитель), используемый для быстрого перебора всех элементов в коллекции

Тип данных коллекции

Управляющая переменная

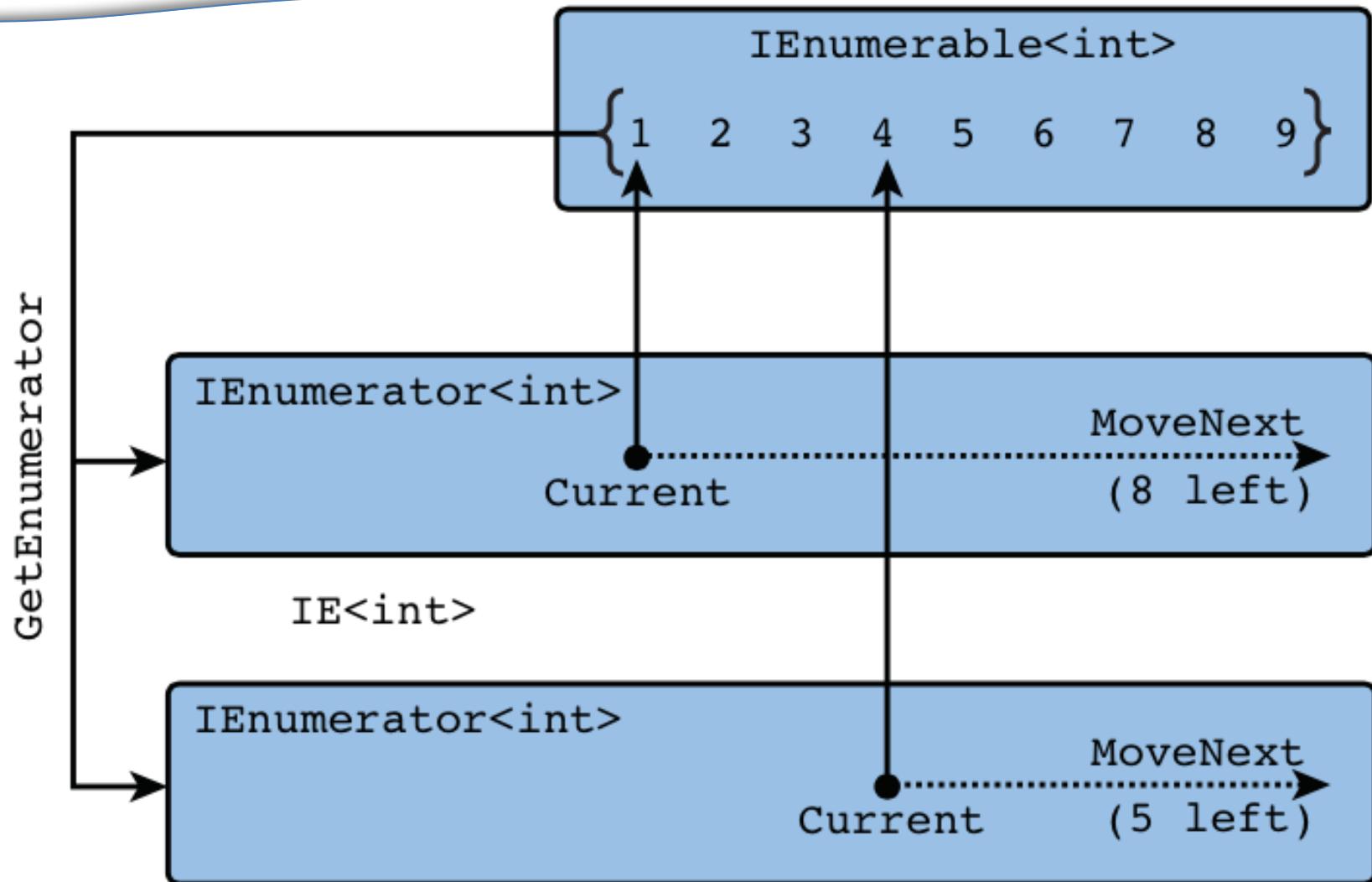
Область видимости
управляющей
переменной –
оператор `foreach`

```
foreach(<type> <control_variable> in <collection>)
{
    <foreach_statement_body>
}
```

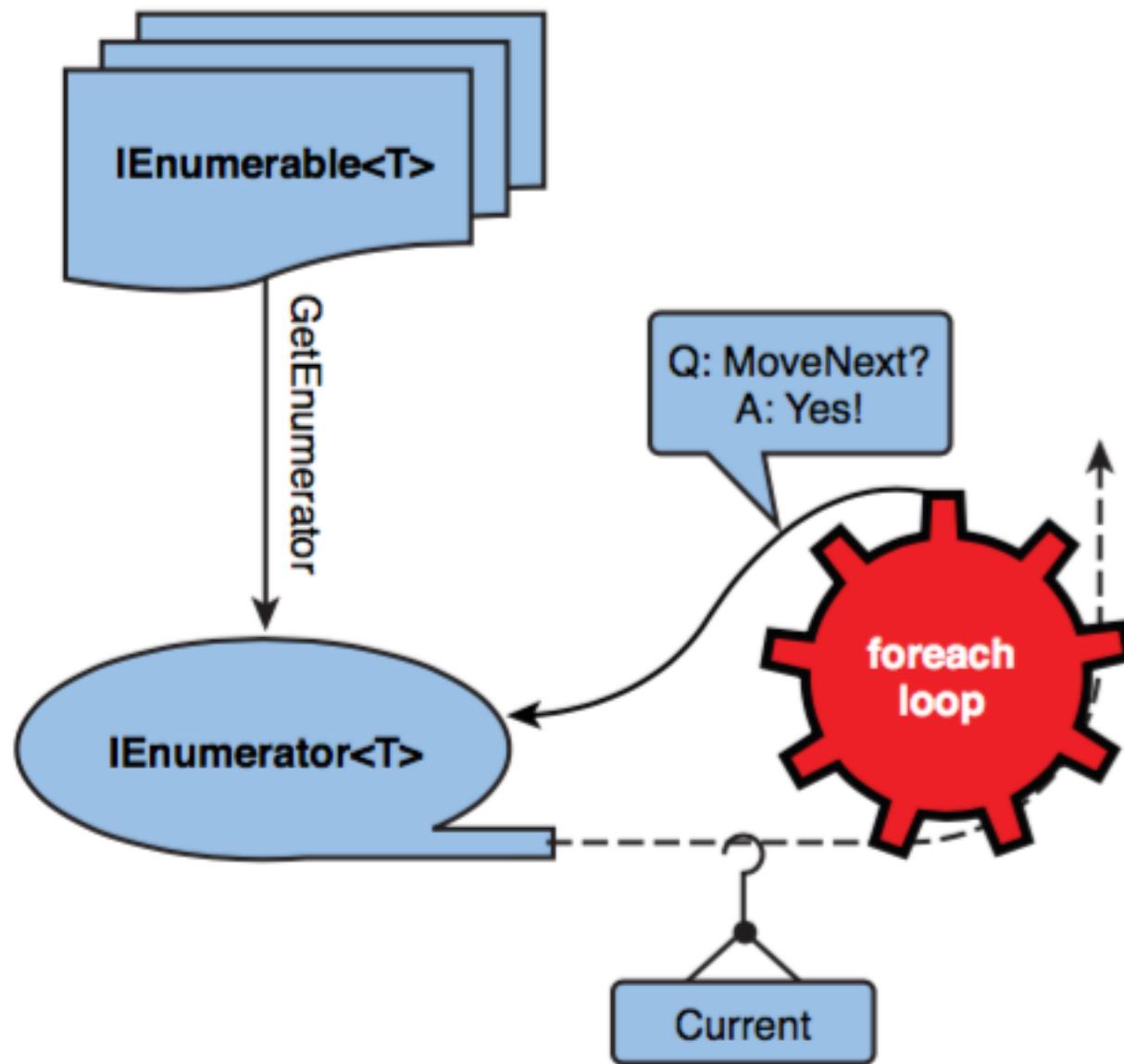
Управляющая переменная устанавливается по очереди для каждого элемента коллекции, и операторы в теле цикла `foreach` выполняются для каждого элемента

Что такое перечислитель?

```
foreach (int num in nums)
    // Do something
```



Что такое перечислитель?



Что такое перечислитель?

```
foreach (int num in nums) // Do something
```

```
using (var e = nums.GetEnumerator())  
{  
    while (e.MoveNext())  
    {  
        int num = e.Current;  
        // Do something  
    }  
}
```

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
{
    IEnumerator<T> GetEnumerator();
}
```

Метод `GetEnumerator` предлагает для коллекции перечислитель по умолчанию

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

В пользовательском классе коллекции можно определить дополнительные перечислители, для этого необходимо предоставить дополнительные методы или свойства для того, чтобы приложения имели доступ к этим перечислителям

Для поддержки инициализаторов коллекции, необходимо реализовать в типе интерфейс `IEnumerable` и определить метод `Add`

Что такое интерфейс `IEnumerable<T>`?

```
class CustomCollectionClass<T>
    : IEnumerable<T>
{
    public IEnumerator<T> GetEnumerator()
    {
        ...
    }

    IEnumrator IEnumerable.GetEnumerator()
    {
        throw new NotImplementedException();
    }

    // Additional enumerators return
    // instances of the IEnumerable<T>
    // interface.
    public IEnumerable<T> Backwards()
    {
        ...
    }

    foreach (int temp in intCollection.Backwards())
    {
        ...
    }
}
```

```
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);
```

```
foreach (int temp in intCollection)
{
    ...
}
```

Конструкция `foreach` использует
перечислитель по умолчанию

Конструкция `foreach` использует дополнительный
перечислитель `Backwards`

Что такое интерфейс `IEnumerator<T>`?

```
public interface IEnumerator<out T> : IDisposable, IEnumerator
{
    T Current { get; }
```

Типизированное свойство; возвращает из коллекции текущий элемент

```
public interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}
```

Извлекает текущий элемент из коллекции

Перемещает перечислитель к следующему элементу коллекции

Сбрасывает внутреннее состояние перечислителя так, что последующий вызов метода `MoveNext` располагает перечислитель снова перед первым элементом в коллекции

```
foreach (int datum in integerCollectionObject)
{
    ...
    datum = datum / 2;
```

СТЕ

Реализация перечислителя вручную на основе интерфейсов IEnumerable или IEnumerable<T>

При реализации интерфейса IEnumerable<T> вручную, необходимо предоставить собственные реализации для свойства Current и методов MoveNext и Reset

```
class CustomCollectionClass<T> : IEnumerable<T>
{
    T[] values = new T[10];
    int pointer = -1;
    public T Current
    {
        get
        {
            if (pointer != -1)
            {
                return values[pointer];
            }
            else
            {
                throw new InvalidOperationException();
            }
        }
    }
}
```

Данные хранятся в массиве

Хранит текущую позицию перечислителя

Определение свойства только для чтения Current для возвращения текущего элемента

Должна быть включена проверка того, что текущее значение верно

Реализация интерфейса IEnumerable<T>

Реализация перечислителя вручную на основе интерфейсов IEnumable или IEnumable<T>

```
object IEnumator.Current  
{  
    get { return (object)Current; }  
}  
public bool MoveNext()  
{  
    if (pointer < (vals.Length - 1))  
    {  
        pointer++;  
        return true;  
    }  
    else  
    {  
        return false;  
    }  
}  
public void Reset()  
{  
    pointer = -1;  
}  
public void Dispose() { }  
}
```

Возвращает свойство Current типобезопастного обобщенного интерфейса IEnumator<T>

Продвигает перечислитель к следующему элементу коллекции

Реализация интерфейса IEnumator

Сбрасывает внутреннее состояние перечислителя

Реализация интерфейса IDisposable

Реализация перечислителя с помощью итератора

Итератор является блоком кода, возвращающим (yields) последовательность значений коллекции

Итератор — метод доступа get или оператор, выполняющий настраиваемую итерацию класса массива или коллекции с помощью ключевого слова yield

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:

- yield return <выражение> возвращает следующее значение последовательности
- yield break прекращает генерацию последовательности

При создании итератора для класса или структуры реализация всего интерфейса IEnumarator не требуется – когда компилятор обнаруживает итератор, он автоматически создает методы Current, MoveNext и Dispose интерфейса IEnumarator или IEnumarator<T>

Итератор вызывается из клиентского кода с помощью оператора foreach

Реализация перечислителя с помощью итератора

```
public class CustomContainer<T>
{
    private T[] _container;
    public CustomContainer(. . .)
    {
        . . .
    }

    public IEnumerator<T> GetEnumerator()
    {
        for (var i = 0; i < _container.Length; i++)
            yield return _container[i];
    }
    . . .
}
```

Реализация перечислителя с помощью итератора

```
private bool MoveNext()
{
    switch (this.<>1_state)
    {
        case 0:
            this.<>1_state = -1;
            this.<i>5_1 = 0;
            while (this.<i>5_1 < this.<>4_this._container.Length)
            {
                this.<>2_current = this.<>4_this._container[this.<i>5_1];
                this.<>1_state = 1;
                return true;
            Label_0055:
                this.<>1_state = -1;
                this.<i>5_1++;
            }
            break;

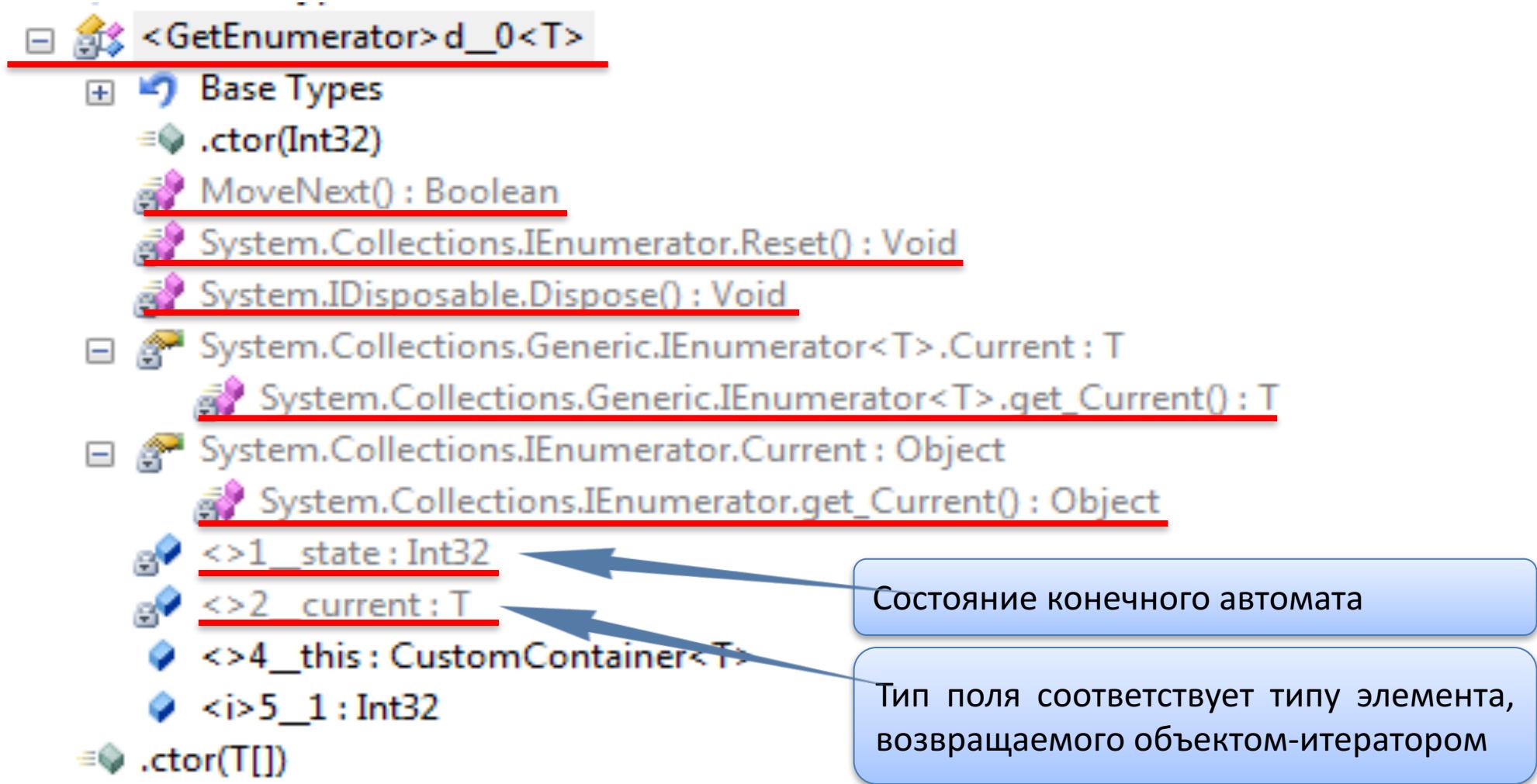
        case 1:
            goto Label_0055;
    }
    return false;
}
```

```
[DebuggerHidden]
void IEnumarator.Reset()
{
    throw new NotSupportedException();
}
```

```
[DebuggerHidden]
T IEnumarator<T>.get_Current()
{
    return this.<>2_current;
}
```

```
[DebuggerHidden]
object IEnumarator.get_Current()
{
    return this.<>2_current;
}
```

Реализация перечислителя с помощью итератора



Каждый сгенерированный класс реализует конечный автомат, который отслеживает текущее состояние объекта-итератора и переходит к выполнению очередного блока кода внутри блока итератора после вызова метода `MoveNext`

Реализация перечислителя с помощью итератора

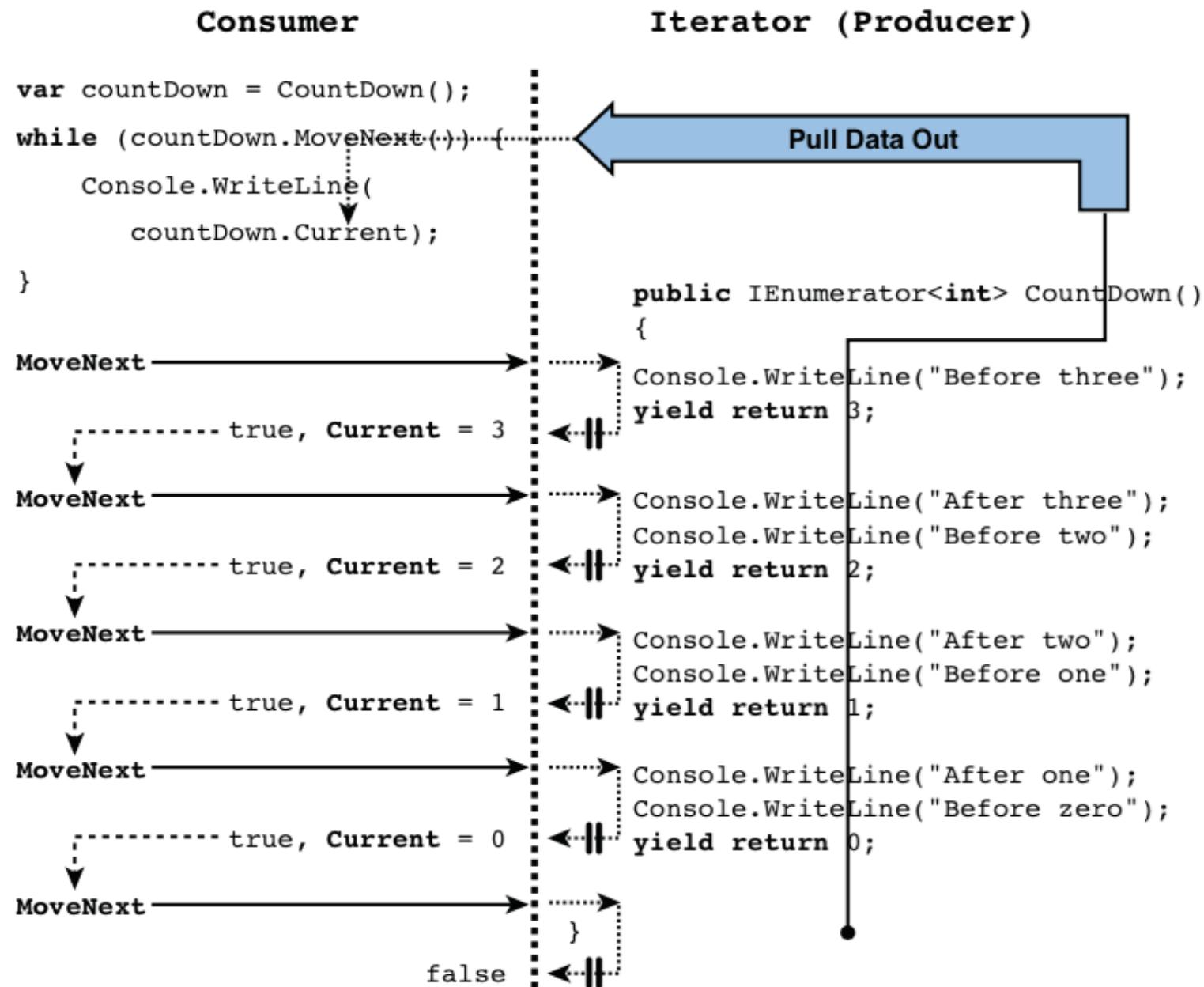


Что такое перечислитель?

```
public IEnumerator<int> CountDown()
{
    Console.WriteLine("Before three");
    yield return 3;
    Console.WriteLine("After three");
    Console.WriteLine("Before two");
    yield return 2;
    Console.WriteLine("After two");
    Console.WriteLine("Before one");
    yield return 1;
    Console.WriteLine("After one");
    Console.WriteLine("Before zero");
    yield return 0;
}
```

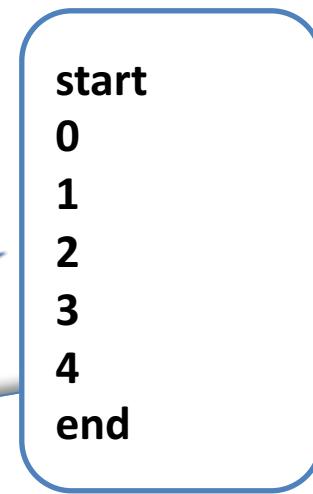
```
var countDown = CountDown();
while (countDown.MoveNext())
    Console.WriteLine(countDown.Current);
```

Что такое перечислитель?



Реализация перечислителя с помощью итератора

```
. . .
static IEnumerable<string> GetDemoEnumerable()
{
    yield return "start";
    for (int i=0; i < 5; i++)
    {
        yield return i.ToString();
    }
    yield return "end";
}
. . .
foreach (string x in GetDemoEnumerable())
{
    Console.WriteLine(x);
}
```



Поскольку весь код метода `GetEnumerator` расположен в методе `MoveNext` сгенерированного класса, то этот код вызовется не сразу после создания объекта итератора, а лишь после вызова метода `MoveNext`. При этом даже при вызове метода `MoveNext` этот код не будет вызван целиком, как привычно думать о коде обычного метода, вместо этого он будет вызываться по частям

Реализация перечислителя с помощью 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);
        }
    }
    public IEnumerator<T> GetEnumerator()
    {
        foreach (var datum in data)
        {
            yield return datum;
        }
    }
    ...
}
```

Компилятор использует этот код для реализации интерфейса `IEnumerator<T>`, который содержит свойство `Current` и методы `MoveNext` и `Reset`

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");
```

```
foreach (string word in bc)
```

```
{  
    Console.WriteLine(word);  
}
```

Twas, brillig, and, the, slithy, toves

Реализация перечислителя с помощью итератора

```
class BasicCollection<T>
{
    private List<T> data = new List<T>();
    . .
    public IEnumerable<T> Reverse
    {
        get
        {
            for (int i = data.Count - 1; i >= 0; i--)
            {
                yield return data[i];
            }
        }
    }
    . .
}
```

Для итерирования данных в другой последовательности, можно реализовать дополнительные свойства, реализующие интерфейс `IEnumerable`, и использовать итератор для возврата данных

```
BasicCollection<string> bc = new BasicCollection<string>();
bc.FillList("Twas", "brillig", "and", "the", "slithy", "toves");
```

```
foreach (string word in bc.Reverse)
{
    Console.WriteLine(word);
}
```

toves, slithy, the, and, brillig, Twas

Реализация перечислителя с помощью итератора

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

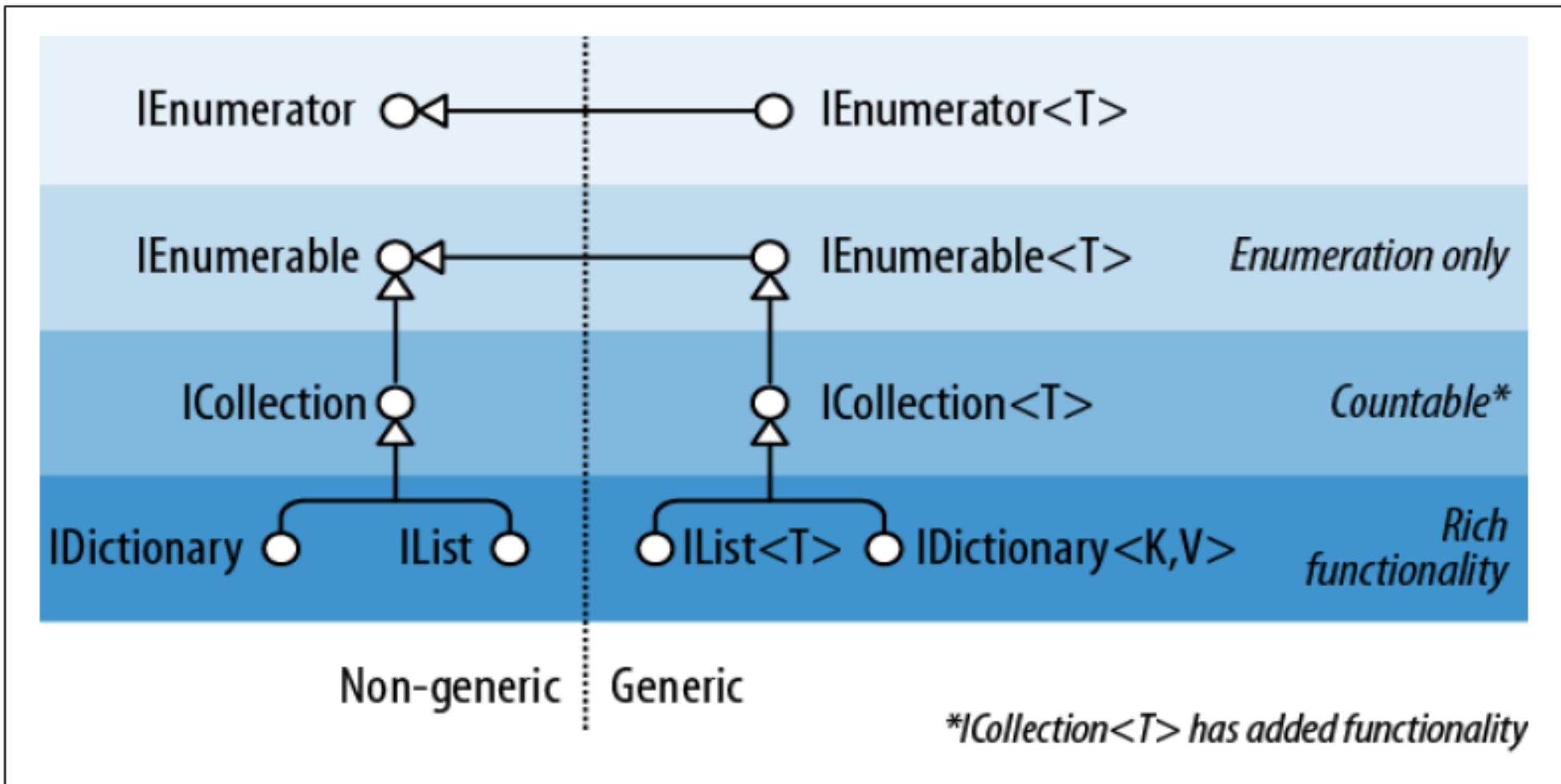
```
public static class Helper
{
    public static IEnumerable<int> GetNumbers()
    {
        int i = 0;
        while (true) yield return i++;
    }
}
```

```
foreach (var n in Helper.GetNumbers())
{
    Console.WriteLine(n);
    if (n == 2) break;
}
```

Блок finally внутри блока итераторов

```
public static IEnumerable<int> GetNumbers()
{
    try
    {
        yield return 7; // 1
        // 2: обработка первого элемента внешним кодом
        yield return 42; // 3
        // 4: обработка второго элемента внешним кодом
    }
    finally
    {
        Console.WriteLine("Внутри блока finally метода GetNumbers");
    }
}
```

Обобщенные интерфейсы коллекций .NET Framework



| Collection | Details | Insertion Time | Deletion Time | Lookup Time | Sorted | Index Access |
|-----------------------|--------------------------------------|--------------------|---------------|-------------|------------|--------------|
| List<T> | Automatically resizable array | Amortized $O(1)^*$ | $O(n)$ | $O(n)$ | No | Yes |
| LinkedList<T> | Doubly-linked list | $O(1)$ | $O(1)$ | $O(n)$ | No | No |
| Dictionary<K,V> | Hash table | $O(1)$ | $O(1)$ | $O(1)$ | No | No |
| HashSet<T> | Hash table | $O(1)$ | $O(1)$ | $O(1)$ | No | No |
| Queue<T> | Automatically resizable cyclic array | Amortized $O(1)$ | $O(1)$ | -- | No | No |
| Stack<T> | Automatically resizable array | Amortized $O(1)$ | $O(1)$ | -- | No | No |
| SortedDictionary<K,V> | Red-black tree | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | Yes (keys) | No |
| SortedList<K,V> | Sorted resizable array | $O(n)^{**}$ | $O(n)$ | $O(\log n)$ | Yes (keys) | Yes |
| SortedSet<T> | Red-black tree | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | Yes | No |

Интерфейсы ICollection и IList. ICollection<T> and ICollection

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
    int Count { get; }
    bool Contains (T item);
    void CopyTo (T[] array, int arrayIndex);
    bool IsReadOnly { get; }
    void Add(T item);
    bool Remove (T item);
    void Clear();
}
```

```
public interface ICollection : IEnumerable
{
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
    void CopyTo (Array array, int index);
}
```

IList IDictionary

Интерфейсы `IList<T>` и `IList`

```
public interface IList<T> : ICollection<T>, IEnumerable<T>,
IEnumerable
{
    T this [int index] { get; set; }
    int IndexOf (T item);
    void Insert (int index, T item);
    void RemoveAt (int index);
}

public interface IList : ICollection, IEnumerable
{
    object this [int index] { get; set }
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    int Add (object value);
    void Clear();
    bool Contains (object value);
    int IndexOf (object value);
    void Insert (int index, object value);
    void Remove (object value);
    void RemoveAt (int index);
}
```

Интерфейсы IReadOnlyList<T> (Framework 4.5)

```
public interface IReadOnlyList<out T> : IEnumerable<T>, IEnumerable
{
    int Count { get; }
    T this[int index] { get; }
}
```

Списки, очереди, стеки, наборы

List<T> и ArrayList

- Неупорядоченная коллекция похожая на массив
- Можно добавлять и извлекать элементы с использованием индексирования
- Динамически увеличивается в размерах по мере добавления значений в коллекцию, но автоматически не сжимается при удалении элементов из нее

```
public class List<T> : IList<T>, IReadOnlyList<T>
{
    public List ();
    public List (IEnumerable<T> collection);
    public List (int capacity);
    // Add+Insert
    public void Add (T item);
    public void AddRange (IEnumerable<T> collection);
    public void Insert (int index, T item);
    public void InsertRange (int index, IEnumerable<T> collection);
```

List<T> и ArrayList

```
// Remove
public bool Remove (T item);
public void RemoveAt (int index);
public void RemoveRange (int index, int count);
public int RemoveAll (Predicate<T> match);

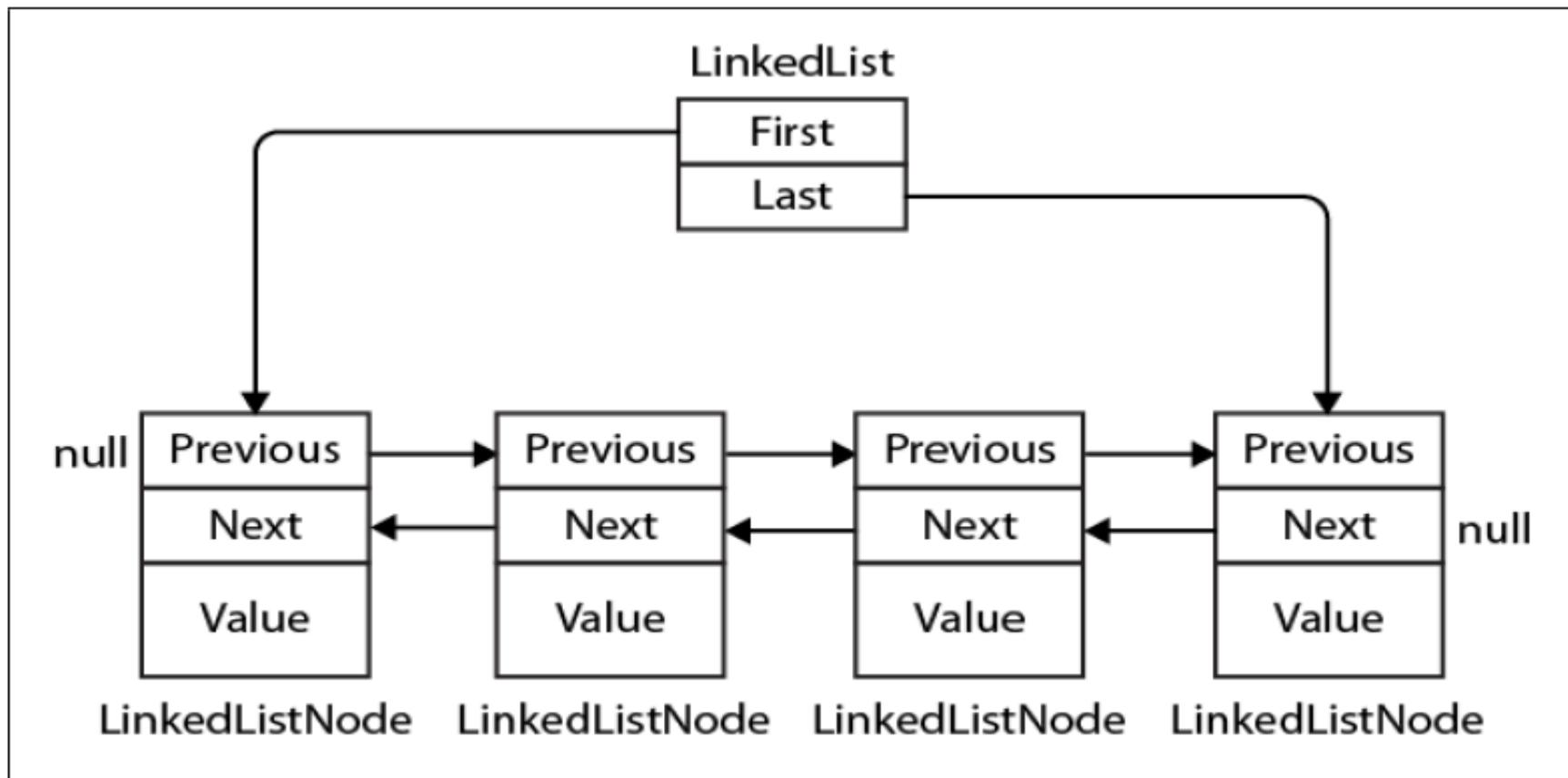
// Indexing
public T this [int index] { get; set; }
public List<T> GetRange (int index, int count);
public Enumerator<T> GetEnumerator();

// Exporting, copying and converting:
public T[] ToArray();
public void CopyTo (T[] array);
public void CopyTo (T[] array, int arrayIndex);
public void CopyTo (int index, T[] array, int arrayIndex, int count);
public ReadOnlyCollection<T> AsReadOnly();
public List<TOutput> ConvertAll<TOutput> (Converter <T,TOutput>
converter);
```

List<T> и ArrayList

```
// Other:  
public void Reverse(); // Reverses order of elements in list.  
public int Capacity { get;set; } // Forces expansion of internal array.  
public void TrimExcess(); // Trims internal array back to size.  
public void Clear(); // Removes all elements, so Count=0.
```

LinkedList<T>



Queue<T> и Queue

- FIFO структура данных
- Обеспечивает методы Enqueue и Dequeue вместо методов Add и Remove
- Растет автоматически при добавлении объектов в коллекцию, но автоматически не сжимается при удалении элементов из нее

```
public class Queue<T> : IEnumerable<T>, ICollection, IEnumerable
{
    public Queue();
    public Queue (IEnumerable<T> collection); // Copies existing elements
    public Queue (int capacity); // To lessen auto-resizing
    public void Clear();
    public bool Contains (T item);
    public void CopyTo (T[] array, int arrayIndex);
    public int Count { get; }
    public T Dequeue();
    public void Enqueue (T item);
    public Enumerator<T> GetEnumerator(); // To support foreach
    public T Peek();
    public T[] ToArray();
    public void TrimExcess();
}
```

Stack<T> и Stack

- LIFO структура данных
- Обеспечивает методы Push и Pop вместо методов Add и Remove
- Растет автоматически при добавлении объектов в коллекцию, но автоматически не сжимается при удалении элементов из нее

```
public class Stack<T> : IEnumerable<T>, ICollection, IEnumerable
{
    public Stack();
    public Stack (IEnumerable<T> collection); // Copies existing elements
    public Stack (int capacity); // Lessens auto-resizing
    public void Clear();
    public bool Contains (T item);
    public void CopyTo (T[] array, int arrayIndex);
    public int Count { get; }
    public Enumerator<T> GetEnumerator(); // To support foreach
    public T Peek();
    public T Pop();
    public void Push (T item);
    public T[] ToArray();
    public void TrimExcess();
}
```

BitArray

HashSet<T> и SortedSet<T> (ISet<T>)

- Методы Contains выполняются быстро с использованием основанного на хешировании поиска
- Не хранят дублированные элементы, молча игнорируют запросы на добавление дубликата
- Доступ к элементам по позициям не возможен

Словари. IDictionary, IDictionary <TKey, TValue>

Коллекция, в которой каждый элемент является парой ключ-значение

| Type | Internal structure | Retrieve by index? | Memory overhead (avg. bytes per item) | Speed: random insertion | Speed: sequential insertion | Speed: retrieval by key |
|------------------------|--------------------|--------------------|---------------------------------------|-------------------------|-----------------------------|-------------------------|
| Unsorted | | | | | | |
| Dictionary <K,V> | Hashtable | No | 22 | 30 | 30 | 20 |
| Hashtable | Hashtable | No | 38 | 50 | 50 | 30 |
| ListDictionary | Linked list | No | 36 | 50,000 | 50,000 | 50,000 |
| OrderedDictionary | Hashtable + array | Yes | 59 | 70 | 70 | 40 |
| Sorted | | | | | | |
| SortedDictionary <K,V> | Red/black tree | No | 20 | 130 | 100 | 120 |
| SortedList <K,V> | 2xArray | Yes | 2 | 3,300 | 30 | 40 |
| SortedList | 2xArray | Yes | 27 | 4,500 | 100 | 180 |

IDictionary <TKey, TValue>

```
public interface IDictionary <TKey, TValue> :  
ICollection <KeyValuePair <TKey, TValue>>, IEnumerable  
{  
    bool ContainsKey (TKey key);  
    bool TryGetValue (TKey key, out TValue value);  
    void Add (TKey key, TValue value);  
    bool Remove (TKey key);  
    TValue this [TKey key] { get; set; } // Main indexer - by key  
    ICollection <TKey> Keys { get; } // Returns just keys  
    ICollection <TValue> Values { get; } // Returns just values  
}
```

```
public struct KeyValuePair <TKey, TValue>  
{  
    public TKey Key { get; }  
    public TValue Value { get; }  
}
```

IDictionary

- Извлечение несуществующего ключа через индексатор дает в результате null (не приводит к генерации исключения)
- Членство проверяется с помощью Contains (ContainsKey)

```
public struct DictionaryEntry
{
    public object Key { get; set; }
    public object Value { get; set; }
}
```

Dictionary < TKey, TValue > и Hashtable

```
public interface IDictionary < TKey, TValue > :  
ICollection < KeyValuePair < TKey, TValue > >, IEnumerable  
{  
    bool ContainsKey ( TKey key );  
    bool TryGetValue ( TKey key, out TValue value );  
    void Add ( TKey key, TValue value );  
    bool Remove ( TKey key );  
    TValue this [ TKey key ] { get; set; } // Main indexer - by key  
ICollection < TKey > Keys { get; } // Returns just keys  
ICollection < TValue > Values { get; } // Returns just values  
}
```

```
public struct KeyValuePair < TKey, TValue >  
{  
    public TKey Key { get; }  
    public TValue Value { get; }  
}
```

Использование инициализаторов коллекции

Для добавления элементов в созданный экземпляр класса коллекции обычно используется метод Add

```
ArrayList al = new ArrayList();
al.Add("Value");
al.Add("Another Value");
```

Альтернативой написанию нескольких операторов с методом Add является использование инициализатора коллекции

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

```
ArrayList al2 = new ArrayList()
{
    new Person() {Name="James", Age =45},
    new Person() {Name="Tom", Age =31}
};
```

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

Инициализатор коллекции можно использовать только с коллекцией классов, предоставляющих метод Add

Настраиваемые коллекции и прокси

Для строго типизированных коллекций в приложении может возникать необходимость управления поведением ее элементов (например, добавлением, удалением)

- для запуска события при удалении или добавлении элемента
- для обновления свойств, связанных с удалением или добавлением элемента
- для обнаружения «несанкционированной» операции добавления/удаления и генерации исключения (например, при нарушении «бизнес-правила»)

Классы, представленные для этой цели в пространстве имен `System.Collections.ObjectModel`, являются оболочками или прокси, которые реализуют `IList<T>` или `IDictionary<,>`, за счет перенаправления на методы лежащей в основе коллекции

Каждая операция (`Add`, `Remove`, `Clear`) проходит через виртуальный метод, действующий в качестве «шлюза», с помощью которого можно «привязаться» с целью изменения или расширения нормального поведения коллекции

Настраиваемые коллекции и прокси

```
public class Collection<T> :  
    IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable  
{  
    // ...  
    protected virtual void ClearItems();  
    protected virtual void InsertItem (int index, T item);  
    protected virtual void RemoveItem (int index);  
    protected virtual void SetItem (int index, T item);  
    protected IList<T> Items { get; }  
}
```

Настраиваемые коллекции и прокси

```
public class Animal
{
    public string Name;
    public int Popularity;
    public Animal (string name, int popularity)
    {
        Name = name;
        Popularity = popularity;
    }
}
```

```
public class AnimalCollection : Collection <Animal>
{
    // AnimalCollection уже полностью готовый класс коллекция.
    // Никого дополнительного кода не требуется
}
```

```
public class Zoo // Класс который откроет класс AnimalCollection.
{
    // Он может иметь дополнительные члены.
    public readonly AnimalCollection Animals = new AnimalCollection();
}
```

Настраиваемые коллекции и прокси

```
class Programm
{
    static void Main()
    {
        Zoo zoo = new Zoo();
        zoo.Animals.Add (new Animal ("Kangaroo", 10));
        zoo.Animals.Add (new Animal ("Mr Sea Lion", 20));
        foreach (Animal a in zoo.Animals)
            Console.WriteLine (a.Name);
    }
}
```

Настраиваемые коллекции и прокси

```
public class Animal
{
    public string Name;
    public int Popularity;
    public Zoo Zoo { get; internal set; }

    public Animal (string name, int popularity)
    {
        Name = name;
        Popularity = popularity;
    }
}
```

Настраиваемые коллекции и прокси

```
public class AnimalCollection : Collection <Animal>
{
    private Zoo zoo;
    public AnimalCollection (Zoo zoo) { this.zoo = zoo; }
    protected override void InsertItem (int index, Animal item)
    {
        base.InsertItem (index, item);
        item.Zoo = zoo;
    }
    protected override void SetItem (int index, Animal item)
    {
        base.SetItem (index, item);
        item.Zoo = zoo;
    }
    protected override void RemoveItem (int index)
    {
        this [index].Zoo = null;
        base.RemoveItem (index);
    }
    protected override void ClearItems()
    {
        foreach (Animal a in this) a.Zoo = null;
        base.ClearItems();
    }
}
```

Определение эквивалентности и порядка

Стандартные протоколы для определения эквивалентности

- операции == и !=
- виртуальный метод Equals типа object
- интерфейс IEquatable<T>

Стандартные протоколы для определения порядка

- операции < и >
- интерфейс IComparable и IComparable<T>

Стандартная реализация сравнения эквивалентности и порядка отображает то, что является наиболее естественным для типа

Определение эквивалентности и порядка

Подключаемые протоколы

- позволяют подключаться на альтернативное поведение эквивалентности и порядка
- позволяют использовать словарь или отсортированную коллекцию с типом ключа, не обладающим внутренней возможностью эквивалентности или сравнения

IEqualityComparer и IEqualityComparer<T> (EqualityComparer)

- Выполняют подключаемое сравнение эквивалентности
- Распознаются Hashtable and Dictionary

IComparer и IComparer<T>(Comparer)

- выполняют подключаемое сравнение порядка
- распознаются отсортированными словарями и коллекциями и Array.Sort

IStructuralEquatable и IStructuralComparable

Выполняют структурное сравнение на классах и массивах

IEqualityComparer и IEqualityComparer<T>

```
public interface IEqualityComparer<T>
{
    bool Equals (T x, T y); //является ли этот ключ таким же как и другой
    int GetHashCode (T obj); // какой целочисленный код у этого ключа
}

public interface IEqualityComparer // Nongeneric version
{
    bool Equals (object x, object y);
    int GetHashCode (object obj);
}
```

IEqualityComparer и IEqualityComparer<T>

```
public abstract class EqualityComparer<T> : IEqualityComparer,
    IEqualityComparer<T>
{
    public abstract bool Equals (T x, T y);
    public abstract int GetHashCode (T obj);

    bool IEqualityComparer.Equals (object x, object y);
    int IEqualityComparer.GetHashCode (object obj);
    public static EqualityComparer<T> Default { get; }
}
```

IEqualityComparer и IEqualityComparer<T>

```
public class Customer
{
    public string LastName;
    public string FirstName;
    public Customer (string last, string first)
    {
        LastName = last;
        FirstName = first;
    }
}
```

```
public class LastFirstEqComparer : EqualityComparer <Customer>
{
    public override bool Equals (Customer x, Customer y)
    {
        return x.LastName == y.LastName && x.FirstName == y.FirstName;
    }
    public override int GetHashCode (Customer obj)
    {
        return (obj.LastName + ";" + obj.FirstName).GetHashCode();
    }
}
```

IEqualityComparer и IEqualityComparer<T>

```
Customer c1 = new Customer ("Bloggs", "Joe");
Customer c2 = new Customer ("Bloggs", "Joe");
...
Console.WriteLine (c1 == c2); // False
Console.WriteLine (c1.Equals (c2)); // False
...
var d = new Dictionary<Customer, string>();
d[c1] = "Joe";
Console.WriteLine (d.ContainsKey(c2)); // False
```

```
var eqComparer = new LastFirstEqComparer();
var d = new Dictionary<Customer, string> (eqComparer);
d [c1] = "Joe";
Console.WriteLine (d.ContainsKey (c2)); // True
```

```
static bool Foo<T> (T x, T y)
{
    bool same = EqualityComparer<T>.Default.Equals(x, y);
    //изначально проверяет реализует ли тип Т интерфейс IEquatable
    ...
}
```

IComparer и IComparer<T>

Переключение специальной логики в отсортированных словарях и коллекциях

```
public interface IComparer
{
    int Compare(object x, object y);
}
public interface IComparer<in T>
{
    int Compare(T x, T y);
}
```

```
public abstract class Comparer<T> : IComparer, IComparer<T>
{
    public static Comparer<T> Default { get; }
    public abstract int Compare (T x, T y); // Implemented by you
    int IComparer.Compare (object x, object y); // Implemented for you
}
```

IComparer и IComparer<T>

```
class Wish
{
    public string Name;
    public int Priority;
    public Wish (string name, int priority)
    {
        Name = name;
        Priority = priority;
    }
}
```

```
class PriorityComparer : Comparer<Wish>
{
    public override int Compare (Wish x, Wish y)
    {
        // Гарантирует, что никогда не будет противоречия с Equals
        //эквивалентность придирчивее сравнения
        if (object.Equals (x, y)) return 0;
        return x.Priority.CompareTo (y.Priority);
    }
}
```

StringComparer

```
public abstract class StringComparer : IComparer, IComparer<string>,
IEqualityComparer, IEqualityComparer<string>
{
    public abstract int Compare (string x, string y);
    public abstract bool Equals (string x, string y);
    public abstract int GetHashCode (string obj);
    public static StringComparer Create (CultureInfo culture,
        bool ignoreCase);
    public static StringComparer CurrentCulture { get; }
    public static StringComparer CurrentCultureIgnoreCase { get; }
    public static StringComparer InvariantCulture { get; }
    public static StringComparer InvariantCultureIgnoreCase { get; }
    public static StringComparer Ordinal { get; }
    public static StringComparer OrdinalIgnoreCase { get; }
}
```

IStructuralEquatable and IStructuralComparable

Структурная эквивалентность и сравнение порядка в виде подключаемых вариантов для других типов

```
public interface IStructuralEquatable
{
    bool Equals (object other, IEqualityComparer comparer);
    int GetHashCode (IEqualityComparer comparer);
}

public interface IStructuralComparable
{
    int CompareTo (object other, IComparer comparer);
}
```

IStructuralEquatable and IStructuralComparable

```
int[] a1 = { 1, 2, 3 };
int[] a2 = { 1, 2, 3 };

Console.WriteLine(a1.Equals(a2)); // False

IStructuralEquatable se1 = a1; // массивы реализуют оба интерфейса
Console.WriteLine(se1.Equals(a2, EqualityComparer<int>.Default)); // True
```

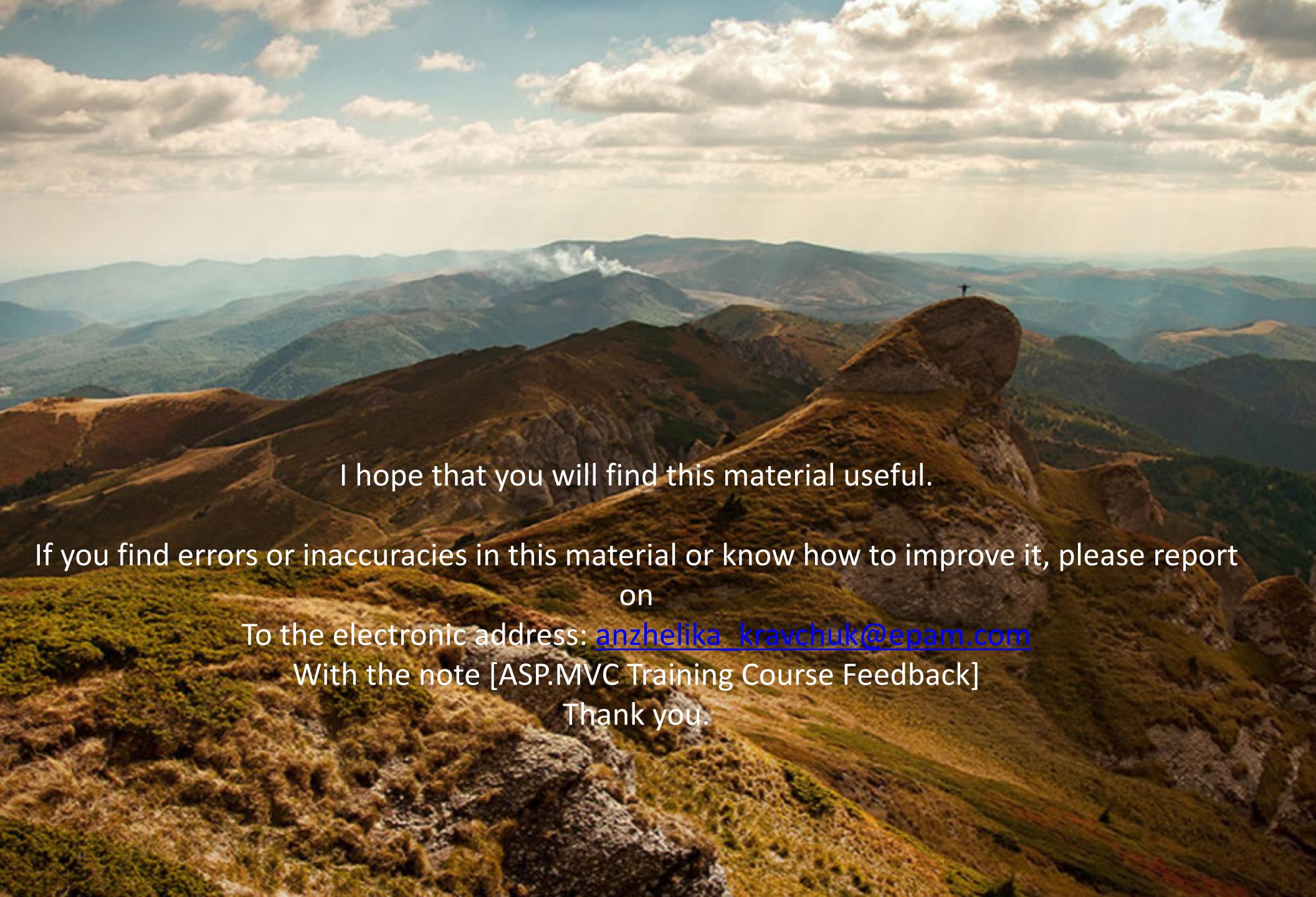
```
string[] a1 = "the quick brown fox".Split();
string[] a2 = "THE QUICK BROWN FOX".Split();

IStructuralEquatable se1 = a1;

bool isFalse = a1.Equals(a2);
bool isTrue = se1.Equals(a2, StringComparer.InvariantCultureIgnoreCase);
```

A wide-angle photograph of a mountainous landscape under a bright sky with scattered clouds. In the foreground, rocky terrain and green slopes are visible. A prominent, rounded mountain peak rises in the center-right. On top of this peak stands a small figure of a person, appearing as a tiny dark dot against the light-colored rock. The background features a range of mountains fading into the distance.

Thank you for attention!

A wide-angle landscape photograph of a mountainous region. In the foreground, there are rocky, grassy slopes. A prominent peak on the right side has a small figure standing on its rocky ridge. The background features a range of mountains under a sky filled with scattered, bright clouds.

I hope that you will find this material useful.

If you find errors or inaccuracies in this material or know how to improve it, please report
on

To the electronic address: anzhelika_kravchuk@epam.com

With the note [ASP.MVC Training Course Feedback]

Thank you.