

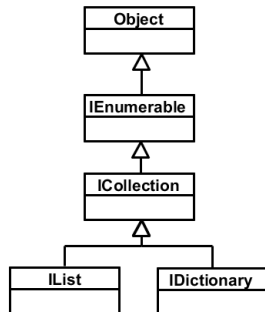
# Контейнеры и генерики

Юрий Литвинов  
yurii.litvinov@gmail.com

27.03.2018г

# Интерфейсы контейнеров

- ▶ `IEnumerable` — штука, из которой можно последовательно получать элементы
- ▶ `ICloneable` — штука, от которой можно делать глубокую копию
- ▶ `ICollection` — абстрактная коллекция
- ▶ `IDictionary` — расширение `ICollection`, абстрактный словарь
- ▶ `IList` — расширение `ICollection`, коллекция, к элементам которой можно обращаться по индексу



# Энумератор

- ▶ Абстрагирует обход коллекции, не может её модифицировать
  - ▶ Обобщённая позиция внутри коллекции
  - ▶ Как правило, требует доступа к внутренним полям коллекции
- ▶ Реализует интерфейс `IEnumerator`
  - ▶ Свойство `Current`
    - ▶ Изначально — перед первым элементом
  - ▶ `MoveNext()`
    - ▶ Возвращает `false` после последнего элемента
  - ▶ `Reset()`
    - ▶ Можно не реализовывать, тогда кидает `NotSupportedException`
- ▶ Компилятор знает про `IEnumerable`:

```
foreach (var i in list) {  
    Console.Write(i);  
}
```
- ▶ Инвалидируется при изменении коллекции (но `Current` продолжает работать)

## Негенериковые коллекции

- ▶ ArrayList, реализует IList, ICollection, IEnumerable, ICloneable
- ▶ BitArray, реализует ICollection, IEnumerable, ICloneable
- ▶ Hashtable, реализует IDictionary, ICollection, IEnumerable, ICloneable
- ▶ Queue, реализует ICollection, IEnumerable, ICloneable
- ▶ SortedList, реализует IDictionary, ICollection, IEnumerable, ICloneable
- ▶ Stack, реализует ICollection, IEnumerable, ICloneable

# Почему негенериковые коллекции — плохо

- ▶ boxing/unboxing
  - ▶ `list.Add(1);`
- ▶ Типобезопасность
  - ▶ `list.Add(1);`
  - ▶ `list.Add("hello");`
- ▶ Понижающие касты
  - ▶ `var str = list[1] as string;`
- ▶ Поэтому придумали генерики:  
`var list = new List<string>();`  
`list.Add("hello");`  
`var str = list[0];`
  - ▶ Так обычно пишут в книгах «C# для суперпрофессионалов», но это не совсем правда...

# Типизация

- ▶ Данные — это просто строки битов
- ▶ Пример — теория множеств
  - ▶  $\{\}$  — 1
  - ▶  $\{\{\}\}$  — 2
  - ▶ Парадокс Рассела
    - ▶ “Пусть  $A$  — множество всех множеств, которые не содержат себя в качестве своего элемента. Содержит ли  $A$  само себя в качестве элемента?”
- ▶ Типы ограничивают допустимые операции над значениями
- ▶ Виды типизации
  - ▶ Динамическая или статическая
  - ▶ Сильная или слабая

# Ad-hoc полиморфизм

- ▶ Перегрузка
- ▶ Приведение

Пример:

1 + 1.0

- ▶ "+" (**int** + **int**, **int** + **float**, **float** + **int**, **float** + **float**)
- ▶ "+" (**int** + **int**, **float** + **float**)
- ▶ "+" (**float** + **float**)

# Универсальный полиморфизм

- ▶ Полиморфизм подтипов (сабтайпинг, наследование)
  - ▶ 1..10 — подынтервал 1..100, следовательно, подтип
  - ▶ Принцип подстановки Лисков
- ▶ Параметрический полиморфизм
  - ▶ `id: x: 'T -> x: 'T`
  - ▶ `id<int>(2)`
  - ▶ `id<string>("Cthulhu fhtagn!")`
  - ▶ `List<'T>` — набор параметрически полиморфных функций



# Типы

- ▶ Элементарные типы
- ▶ Конструкторы типов
  - ▶ Подъязык для описания сложных типов
- ▶ Структурное равенство и равенство по имени
- ▶ Выражения над типами
  - ▶ Генерик — это функция, принимающая набор параметров-типов и возвращающая тип

Подробности: Cardelli, Luca, and Peter Wegner. "On understanding types, data abstraction, and polymorphism." *ACM Computing Surveys (CSUR)* 17.4 (1985): pp. 471-523.

# Генерики в .NET

- ▶ `System.Collections.Generic`  
`List<string> listOfStrings = new List<string>();`
- ▶ Не требуют исходного кода генерика
  - ▶ Информация о параметрах-типах есть в байт-коде
- ▶ Не выполняют boxing, если параметр-тип — тип-значение
  - ▶ Для каждого параметра типа-значения при инстанцировании порождается новый код (как в C++)
  - ▶ Для каждого параметра ссылочного типа байт-код переиспользуется (как в Java)
    - ▶ Но не происходит стирание

Генерик-методы:

```
int[] myInts = {1, 5, 2, 8, 4};  
Array.Sort<int>(myInts);
```

# Выражения инициализации

```
int[] myArrayOfInts = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

```
var myGenericList = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

```
var myList = new ArrayList { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

## Свои генерик-методы

```
static void Swap(ref int a, ref int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```



```
static void Swap<T>(ref T a, ref T b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

# Использование

```
int a = 10, b = 90;  
Swap<int>(ref a, ref b);
```

```
string s1 = "Hello", s2 = "There";  
Swap<string>(ref s1, ref s2);
```

```
bool b1 = true, b2 = false;  
Swap(ref b1, ref b2);
```

# Генерик-классы

```
public class Point<T>
{
    private T xPos;
    private T yPos;

    public Point(T xVal, T yVal) {
        xPos = xVal;
        yPos = yVal;
    }

    public T X {
        get { return xPos; }
        set { xPos = value; }
    }

    public override string ToString() => $"{xPos}, {yPos}";

    public void ResetPoint() {
        xPos = default(T);
        yPos = default(T);
    }
}
```

# Использование

```
Point<int> p = new Point<int>(10, 10);  
Point<double> p2 = new Point<double>(5.4, 3.3);
```

```
var p = new Point<int>(10, 10);  
var p2 = new Point<double>(5.4, 3.3);
```

## Рекомендации по именованию

- ▶ Параметр тип должен называться **T** или хотя бы начинаться с **T**
  - ▶ Например, `Data<TKey, TValue>`
- ▶ Если генерики отличаются только параметрами-типами и находятся в разных файлах, то:
  - ▶ Можно добавлять число параметров-типов в имя файла через “”
    - ▶ Например, `Data‘1.cs`, `Data‘2.cs`
  - ▶ Можно параметры-типы включать в имена файлов в фигурных скобках
    - ▶ Например, `Data{T}.cs`, `Data{TKey,TValue}.cs`



# Открытые и закрытые типы

Как оно устроено внутри

- ▶ **Открытый тип** — тип с неспецифицированными параметрами-типами, создание его экземпляров невозможно
- ▶ **Закрытый тип** — тип, у которого для всех параметров-типов указаны фактические типы

```
internal sealed class DictionaryStringKey<TValue>
    : Dictionary<String, TValue> {
}
```

```
public static class Program {
    public static void Main() {
        // Открытый тип
        Type t = typeof(Dictionary<,>);
        // Открытый тип
        t = typeof(DictionaryStringKey<>);
        // Закрытый тип
        t = typeof(DictionaryStringKey<Guid>);
    }
}
```

# Статические конструкторы

Каждый закрытый тип имеет **свой** объект-тип, статический конструктор вызовется для каждого параметра-типа

```
internal sealed class GenericTypeThatRequiresAnEnum<T> {  
    static GenericTypeThatRequiresAnEnum() {  
        if (!typeof(T).IsEnum) {  
            throw new ArgumentException("T must be an enumerated type");  
        }  
    }  
}
```

Обычно так делать не надо, есть ограничения

# Способ прострелить себе ногу

© Андрей Акиншин

```
class Foo<T>
{
    public static int Bar;
}

void Main()
{
    Foo<int>.Bar++;
    Console.WriteLine(Foo<double>.Bar);
}
```

Вообще, мутабельные static-поля в generic-классах не нужны. Ещё более вообще, мутабельные static-поля не нужны.

# Генерики и наследование

Подстановка параметра-типа на наследование никак не влияет

```
internal sealed class Node<T> {  
    public T data;  
    public Node<T> next;  
    public Node(T data) : this(data, null) {}  
    public Node(T data, Node<T> next) {  
        this.data = data; this.next = next;  
    }  
}  
  
private static void SameDataLinkedList() {  
    Node<Char> head = new Node<Char>('C');  
    head = new Node<Char>('B', head);  
    head = new Node<Char>('A', head);  
}
```

# Генерики и наследование, пример

```
internal class Node {  
    protected Node next;  
    public Node(Node next) => this.next = next;  
}
```

```
internal sealed class TypedNode<T> : Node {  
    public T data;  
    public TypedNode(T data) : this(data, null) {}  
    public TypedNode(T data, Node next) : base(next) {  
        this.data = data;  
    }  
}
```

```
private static void DifferentDataLinkedList() {  
    Node head = new TypedNode<Char>('.');  
    head = new TypedNode<DateTime>(DateTime.Now, head);  
    head = new TypedNode<String>("Today is ", head);  
}
```

# Идентичность типов-генериков

Иногда вместо

```
List<DateTime> dtl = new List<DateTime>();
```

пишут

```
internal sealed class DateTimeList : List<DateTime> { }
```

...

```
DateTimeList dtl = new DateTimeList();
```

Теряется эквивалентность типов:

```
Boolean sameType = (typeof(List<DateTime>) == typeof(DateTimeList));
```

Можно

```
using DateTimeList = System.Collections.Generic.List<System.DateTime>;
```

## Генерики и вложенные классы

Вложенные в генерик классы могут использовать его параметры-типы (и сами считаются генериками)

```
public class Outermost<T>
{
    public class Inner<U>
    {
        public class Innermost1<V> {}
        public class Innermost2 {}
    }
}
```

Объект вложенного класса нельзя создать, не указав параметры-типы объемлющего класса

# Ограничения

```
public class MyGenericClass<T> where T : new()  
public class MyGenericClass<T> where T : new(), class  
public class MyGenericClass<T, U>  
    where T : new()  
    where U: class
```

Доступные ограничения:

- ▶ **where T : struct**
- ▶ **where T : class**
- ▶ **where T : new()**
- ▶ **where T : NameOfBaseClass**
- ▶ **where T : NameOfInterface**



# Пример 1

## Primary constraint

```
internal sealed class PrimaryConstraintOfStream<T> where T : Stream
{
    public void M(T stream)
    {
        stream.Close();
    }
}
```

## Пример 2

### Primary constraint

```
internal sealed class PrimaryConstraintOfClass<T> where T : class {  
    public void M()  
    {  
        // Поскольку T - ссылочный тип, присвоить ему null допустимо  
        T temp = null;  
    }  
}
```

## Пример 3

### Secondary constraint

```
private static List<TBase> ConvertIList<T, TBase>(IList<T> list)
    where T : TBase
{
    List<TBase> baseList = new List<TBase>(list.Count);
    for (int index = 0; index < list.Count; index++)
    {
        baseList.Add(list[index]);
    }
    return baseList;
}
```

## Пример 4

### Constructor constraint

```
internal sealed class ConstructorConstraint<T> where T : new() {  
    public static T Factory() {  
        // Тут подойдут любые типы-значения и  
        // ссылочные типы с public-конструктором без параметров  
        return new T();  
    }  
}
```

# Ограничения и касты

Так работать не будет:

```
private static void CastingAGenericTypeVariable1<T>(T obj) {  
    int x = (int) obj;  
    string s = (string) obj;  
}
```

А так будет:

```
private static void CastingAGenericTypeVariable2<T>(T obj) {  
    int x = (int) (object) obj;  
    string s = (string) (object) obj;  
}
```

И так:

```
private static void CastingAGenericTypeVariable3<T>(T obj) {  
    string s = obj as string;  
}
```

# Генерики и сравнения

Так работает:

```
private static void ComparingAGenericTypeVariableWithNull<T>(T obj) {  
    if (obj == null) { /* Никогда не вызовется для типов-значений */ }  
}
```

А так нет:

```
private static void ComparingTwoGenericTypeVariables<T>(T o1, T o2) {  
    if (o1 == o2) { }  
}
```

# Генерики и операторы

И так не работает:

```
private static T Sum<T>(T num) where T : struct {  
    T sum = default(T) ;  
    for (T n = default(T) ; n < num ; n++)  
    {  
        sum += n;  
    }  
  
    return sum;  
}
```

# Вариантность

```
public void f(Tuple<object, object> x)
{
    ...
}
```

```
f(new Tuple<object, object>(apple1, apple2));
f(new Tuple<Apple, Apple>(apple1, apple2)); // Ошибка компиляции
```

Чтобы нельзя было делать так:

```
public void f(Tuple<object, object> x)
{
    x.Item1 = new Battleship();
}
```



# Виды вариантности

- ▶ **Ковариантность** —  $A \leq B \Rightarrow G<A> \leq G<B>$

```
void PrintAnimals(IEnumerable<Animal> animals) {
    for (var animal in animals)
        Console.WriteLine(animal.Name);
}
```

— IEnumerable<любой наследник Animal> тоже ок, IEnumerable ковариантен

- ▶ **Контравариантность** —  $A \leq B \Rightarrow G<B> \leq G<A>$

```
void CompareCats(IComparer<Cat> comparer) {
    var cat1 = new Cat("Otto");
    var cat2 = new Cat("Troublemaker");
    if (comparer.Compare(cat2, cat1) > 0)
        Console.WriteLine("Troublemaker wins!");
}
```

— IComparer<любой предок Cat> тоже ок, IComparer контравариантен

- ▶ **Инвариантность** —  $A \leq B \Rightarrow G<A>$  и  $G<B>$  никак не связаны

- ▶ Пример с Tuple выше

# Ковариантность массивов

```
string[] a = new string[1];
```

```
object[] b = a;
```

```
b[0] = 1;
```

— System.ArrayTypeMismatchException, ошибка времени выполнения!

# Вариантность функциональных типов

## Контравариантность по типам аргументов

```
public class A
```

```
{
```

```
    public static void f(Func<string, object> a)
```

```
    {
```

```
        a("1");
```

```
    }
```

```
}
```

```
...
```

```
Func<object, object> b = x => x.ToString();
```

```
A.f(b);
```

# Вариантность функциональных типов

Ковариантность по возвращаемому значению

```
Func<Object, ArgumentException> fn1 = null;  
Func<Object, Exception> fn2 = fn1;
```

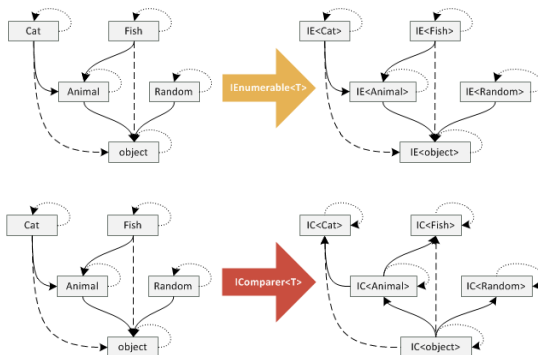
Обратите внимание, ref-параметры сразу делают функцию инвариантной

# Явное указание вариантности для интерфейсов

```
public interface IContainer<out T>
{
    T GetItem();
}
```

```
public interface IContainer<out T>
{
    void SetItem(T item); // ошибка компиляции
    T GetItem();
}
```

# Как это можно себе представлять



<http://tomasp.net/blog/variance-explained.aspx/>