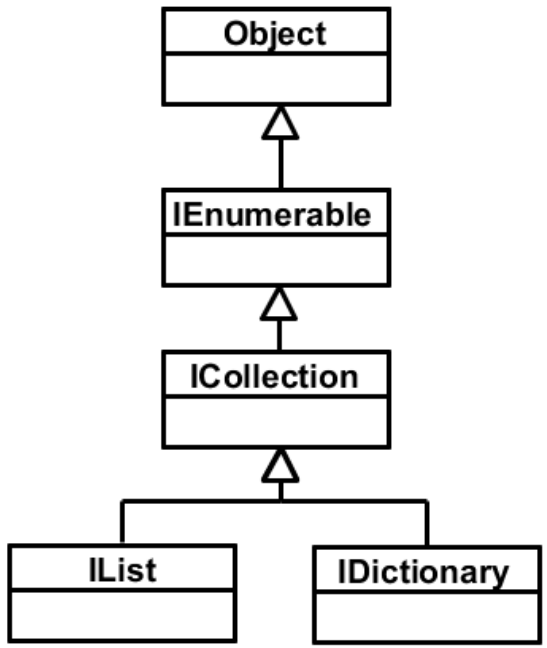
# Контейнеры и генерики

Юрий Литвинов

3

# 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
- 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...

# Полиморфизм

- Ad-hoc
  - Перегрузка
  - Приведение
- Универсальный
  - 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
  - На самом деле, функтор над категорией типов

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 инстанциировании порождается новый код (как в C++)
  - Для каждого параметра ссылочного типа байт-код переиспользуется (как в Java)
    - Но не происходит стирание

Генерик-методы:

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

# 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()
    => string.Format("[{0}, {1}]", 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);
```

# Генерики и вложенные классы

TODO

# 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

# 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 => 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 => 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 => 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();
}
```

# Подробности

Взгляд на генерики и вариантность с точки зрения теории категорий:
http://tomasp.net/blog/variance-explained.aspx/