Промышленное программирование

Лекция 3

❶ Обобщения
❷ Ковариантность и контравариантность
❸ Обобщённые контейнеры

❶ **Обобщения**
❷ Ковариантность и контравариантность
❸ Обобщённые контейнеры

Обобщённые методы

Шаблоны в C++
- Очень гибки
- Позволяют делать множество трюков
- Очень сложны

```cpp
template<typename T>
T GenericFunction(T item) {
    // С "T" и "item" можно делать всё что угодно.
    // Проблемы начнутся в момент инстанцирования.
    T a(1, 2, 3);
    cout << (a + item) * (1 + sin(item)) << endl;
    cout << item.GetUnicorn() << endl;
    return item;
}
```

Шаблоны в C#
- Не так гибки
- Проще в использовании
- Больше об отличиях с C++

```cs
public static void GenericMethod<T>(T item)
{
    // С "item" возможны только операции,
    // доступные на уровне System.Object.
    Console.WriteLine(item.Equals(null));
    Console.WriteLine(item.GetHashCode());
    Console.WriteLine(item.GetType());
    Console.WriteLine(item.ToString());
    Console.WriteLine();
}
```

Ограничения на параметры типа

Constraints on type parameters

`where T: struct`	Аргумент типа должен быть структурой
`where T: class`	Аргумент типа должен быть классом
`where T: new()`	Аргумент типа должен иметь открытый конструктор по умолчанию
`where T: U`	Аргумент типа должен удовлетворять одному из условий: • совпадать с типом U • быть наследником U, если U – класс • быть реализацией U, если U – интерфейс

Пример

```
static class Program {
    interface IFunction {
        double Compute(double x);
    }

    class LinearFunction : IFunction {
        public double LinearPart { get; } = 1;
        public double ConstPart { get; }
        public double Compute(double x) => LinearPart * x + ConstPart;
    }

    static double CreateDefaultAndCompute<T>(double x)
        where T : IFunction, new()
        => new T().Compute(x);

    static void Main() {
        Console.WriteLine(CreateDefaultAndCompute<LinearFunction>(5));
    }
}
```

❶ Обобщения
❷ **Ковариантность и контравариантность**
❸ Обобщённые контейнеры

Ковариантность

```
class Animal { }

class Mouse : Animal { }

interface IProducer<T> {
    T Produce();
}

class MouseProducer : IProducer<Mouse> {
    public Mouse Produce() => new Mouse();
}
```

```
class Animal { }

class Mouse : Animal { }

interface IProducer<out T> {
    T Produce();
}

class MouseProducer : IProducer<Mouse> {
    public Mouse Produce() => new Mouse();
}
```

```
IProducer<Animal> producer = new MouseProducer();
```

Ковариантность – возможность использовать типы,
находящиеся ниже в иерархии наследования, чем изначально указанный тип.
Ограничение: T становится невозможно использовать как тип для какого-либо входного аргумента.

Контравариантность

```
class Animal { }

class Mouse : Animal { }

interface IConsumer<T> {
    void Consume(T item);
}

class AnimalConsumer : IConsumer<Animal> {
    public void Consume(Animal animal) {}
}
```

```
class Animal { }

class Mouse : Animal { }

interface IConsumer<in T> {
    void Consume(T item);
}

class AnimalConsumer : IConsumer<Animal> {
    public void Consume(Animal animal) {}
}
```

```
IConsumer<Mouse> consumer = new AnimalConsumer();
```

Контравариантность – возможность использовать типы,
находящиеся выше в иерархии наследования, чем изначально указанный тип.
Ограничение: T становится невозможно использовать как тип для какого-либо выходного аргумента.

Инвариантность

Инвариантность – отсутствие ковариантности и контравариантности.

- Ковариантность и контравариантность работает только с обобщёнными интерфейсами и обобщёнными делегатами
- Обобщённые классы могут иметь только инвариантные параметры типа

Covariance and contravariance
Covariance and contravariance in generics

❶ Обобщения
❷ Ковариантность и контравариантность
❸ Обобщённые контейнеры

Некоторые обобщённые интерфейсы

```
namespace System;

public interface IEquatable<T>
{
    bool Equals(T other);
}

public interface IComparable<in T>
{
    int CompareTo(T other);
}
```

```
namespace System.Collections.Generic;

public interface IEqualityComparer<in T>
{
    bool Equals(T x, T y);
    int GetHashCode(T obj);
}

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

Итераторы

```
namespace System.Collections.Generic;

public interface IEnumerator<out T> :
    System.Collections.IEnumerator,
    System.IDisposable
{
    T Current { get; }

    // Унаследовано от IEnumerator
    object? Current { get; }

    bool MoveNext();
    void Reset();

    // Унаследовано от IDisposable
    void Dispose();
}
```

```
namespace System.Collections.Generic;

public interface IEnumerable<out T>
    : System.Collections.IEnumerable
{
    IEnumerator<T> GetEnumerator();

    // Унаследовано от IEnumerable
    IEnumerator GetEnumerator();
}
```

```
foreach (var item in items)
    // ...
```

Коллекции

```
public interface IReadOnlyCollection<out T>
    : IEnumerable<T> {
    int Count { get; }
}

public interface ICollection<T>
    : IEnumerable<T> {
    int Count { get; }
    bool IsReadOnly { get; }

    void Add(T item);
    void Clear();
    bool Contains(T item);
    void CopyTo (T[] array, int arrayIndex);
    bool Remove(T item);
}

public interface ISet<T>
    : ICollection<T> {
    ...
}
```

```
public class Stack<T> : IReadOnlyCollection<T>, ... {
    // Push, Pop, ...
}

public class Queue<T> : IReadOnlyCollection<T>, ... {
    // Enqueue, Dequeue, ...
}

public class LinkedList<T>
    : ICollection<T>, IReadOnlyCollection<T>, ... { }

public class HashSet<T> : ISet<T>, ... {
    public IEqualityComparer<T> Comparer { get; }
    ...
}

public class SortedSet<T> : ISet<T>, ... {
    public IComparer<T> Comparer { get; }
    ...
}
```

Списки

```
public interface IList<T> : ICollection<T>
{
    T this[int index] { get; set; }

    int IndexOf(T item);
    void Insert(int index, T item);
    void RemoveAt(int index);
}
```

```
public class List<T> : IList<T>, ...
{
    public T this[int index] { get; set; }
    public int Count { get; }
    public int Capacity { get; set; }

    public void Add(T item);
    public void Insert(int index, T item);
    public bool Remove(T item);
    public void RemoveAt(int index);
    public void Clear();
    public void Sort();
    ...
}
```

Словари

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

public interface IDictionary<TKey, TValue>
    : ICollection<KeyValuePair<TKey, TValue>>, ...
{
    TValue this[TKey key] { get; set; }
    ICollection<TKey> Keys { get; }
    ICollection<TValue> Values { get; }

    void Add(TKey key, TValue value);
    bool ContainsKey(TKey key);
    bool Remove(TKey key);
    bool TryGetValue(TKey key, out TValue value);
}
```

```
public class Dictionary<TKey, TValue>
    : IDictionary<TKey, TValue>
{
    public IEqualityComparer<TKey> Comparer { get; }
    ...
}

public class SortedDictionary<TKey, TValue>
    : IDictionary<TKey, TValue>
{
    public IComparer<TKey> Comparer { get; }
    ...
}
```

```
var dict = new Dictionary<string, int>();
foreach (var (key, value) in dict)
    // ...
```

Массивы

Arrays (C# Programming Guide)

Заключение

1. Обобщения в C# намного ограниченнее, чем шаблоны в C++
2. Ковариантность полезна для модели производителя
3. Контравариантность полезна для модели потребителя
4. Обобщённые контейнеры:
    1. Stack
    2. Queue
    3. LinkedList
    4. HashSet
    5. List
    6. Dictionary