OTUS

C# Developer. Professional

otus.ru

Проверить, идет ли запись

Меня хорошо видно && слышно?

Дженерики, их реализация и ограничения

Преподаватель

Виктор Дзицкий

TeamLead, Full Stack .Net Developer

Больше 10 лет опыта программирования на C# ;
Microsoft Certified Solutions Developer (MCSD)
Спикер на 4-х курсах Otus

Telegram: @Dzitskiy_dev, @Dzitskiy_net

Правила вебинара

Активно
участвуем

Off-topic обсуждаем
в группе OTUS
C#-2025-05

Задаем вопрос
в чат или голосом

Вопросы вижу в чате,
могу ответить не сразу

Условные обозначения

Индивидуально

Время, необходимое
на активность

Пишем в чат

Говорим голосом

Документ

Ответьте себе или
задайте вопрос

Цели вебинара

После занятия вы сможете:

1. Изучить, как в C# писать обобщенный код (с дженериками)
2. Познакомиться, как гибко настраивать ограничения
3. Применять ковариантность
4. Применять контрвариантность
5. Применить на практике знания по дженерикам

План вебинара

- Что такое дженерики
- Обобщенные классы и методы
- Что можно с ними делать
- Вариантность и контрвариантность
- Практика
- Рефлексия

Generics

Generics

Обобщение (Generic) — средство языка C#, позволяющее создавать программный код, содержащий единственное (типизированное) решение задачи для различных типов, с его последующим применением для любого конкретного типа (int, float, char и т.д.).

Преимущества Generics

- Переиспользование кода
- Типобезопасность
- Производительность

System.Collections.Generic

Содержит интерфейсы и классы, определяющие универсальные коллекции, которые позволяют пользователям создавать строго типизированные коллекции, обеспечивающие повышенную производительность и безопасность типов по сравнению с неуниверсальными строго типизированными коллекциями.

List<T>

Dictionary<TKey, TValue>

Queue<T>

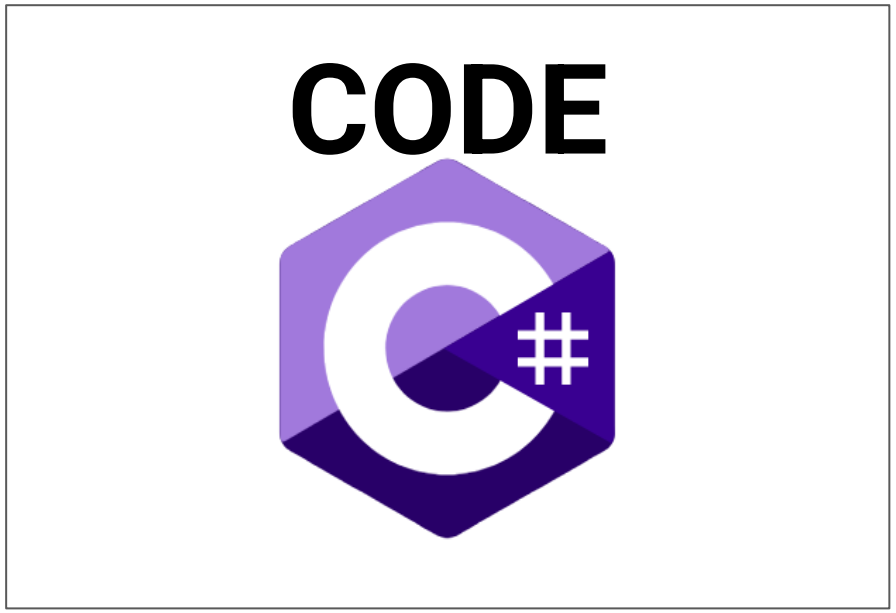
Stack<T>

Общий вид Generic

```
public class GenericType<T>
{
   private T val;

   public Intro(T v)
       => val = v;

   public void PrintMe()
   => Console.WriteLine($"I'm '{val}' and my type is
'{typeof(T)}’”);

   public void PrintMeType<K>(K v)
   => Console.WriteLine($"2. I'm '{v}' and my type is
 '{v.GetType()}'");
   }
```

```
var intIntro = new GenericType<int>(2);
intIntro.PrintMe();
var stringIntro = new GenericType<string>("hello");
stringIntro.PrintMe();

stringIntro.PrintMeType(2.0f);
stringIntro.PrintMeType(false);
stringIntro.PrintMeType<bool>(false);
```

Общий вид Generic



https://github.com/Dzitsky/Otus.Generics.git

Value/reference types

Типы значений и ссылочные типы

- **Типы значений** (Value type) – переменные содержат экземпляр типа, хранятся в стеке (stack). Примеры: *int, bool, DateTime, struct*

- **Ссылочные типы** (Reference type) – переменные содержат ссылки на значения объектов в куче (heap). Примеры: *объекты классов (class*)

Применительно к дженерикам

```
public class MyList<T>
{
    public MyList()
    => _list = new List<T>();

    private List<T> _list;

    public void Add(T v)
    => _list.Add(v);
}
```

```
class Foo { }


var lString = new MyList<string>();
var lFoo = new MyList<Foo>();



var lInt = new MyList<int>();
var lBool = new MyList<bool>();
```

Применительно к дженерикам (ref)

```
var lString = new MyList<string>();
var lFoo = new MyList<Foo>();
```

- *String* и *Foo* – ссылочные типы
- Значит T – тоже ссылка, всегда одного размера
- Внутри MyList – работаем со T-ссылками
- Для MyList<T> - где T – ссылочный – компилятор создает один экземпляр типа MyList*ref*

Применительно к дженерикам (val)

```
var lInt = new MyList<int>();
var lBool = new MyList<bool>();
```

- *int* и *bool* – типы значений разного размера (4 и 1 байт соответственно)
- Значит T имеет разный размер под разный тип
- Внутри MyList – работаем с T разных типов
- Для MyList<T> - где T – тип значений – создается отдельный экземпляр типа (MyList^{bool} MyList^{int})

Что можно делать с дженериками

Наследование

```
// Базовый класс с коллекцией
public class MyCollection<T>
{
    public T[] Values { get; set; }
}

// Наследуем с трансляцией типа
public class MyExtraCollection<T>
            : MyCollection<T>
{
    public T[] MoreValues { get; set; }
}
```

```
// Класс с реализацией под double
public class ColOfNumbers
: MyCollection<double>
{
    public double GeomAvg()
    {
        var res = 1.0;
        var pow =  1.0 / Values.Length;
        foreach (var v in Values)
            res *= Math.Pow(v, pow);
        return res;
    }
}
```

Множественный дженерик

```
// Два обобщенных типа TKey и TValue
public class MyKeyValue<TKey, TValue>
{
    public MyKeyValue(
           TKey key,
           TValue value)
    {
        Key = key;
        Value = value;
    }

    public TKey Key { get; set; }
    public TValue Value { get; set; }
    public override string ToString()
    => $"Key={Key}, Value={Value}";
}
```

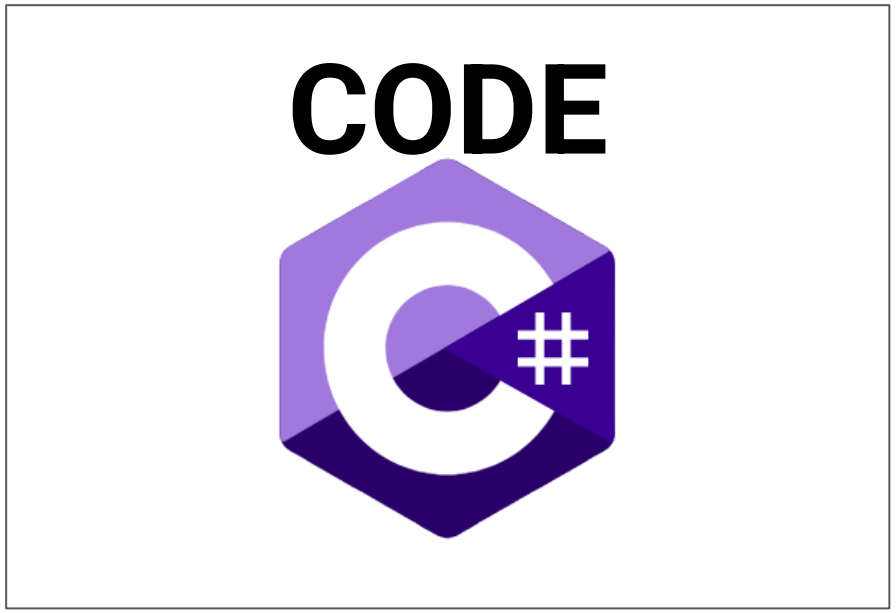
```
var kv1 = new MyKeyValue<int, string>(
1, "Hello");
var kv2 = new MyKeyValue<float, bool>(
1f, false);
```

default

Ключевое слово для инициализации значения по умолчания

Тип T	значение default(T)
Ссылочный (класс)	Null
int, double	0, 0.0
bool	false
Перечисление (enum) T	(T)0
Struct	Структура с полями заполненными по умолчанию

Наследование, составной дженерик, значения по умолчанию



https://github.com/Dzitsky/Otus.Generics.git

Ограничения

Ограничения - new

Тип должен иметь конструктор без параметров

```
public class DemoConstraint<T> where T : new()
    {
        public DemoConstraint()
        {
            var v = new T();
            Console.WriteLine($"Hello {v}");
        }

        public void Do<K>() where K : new()
        {
            var v = new K();
            Console.WriteLine($"Hello {v}");
        }
    }
```

Ограничения – тип значений (struct)

```
// T - только value type (int, bool,
 DateTime, структура)
public class StructConstraint<T>
   where T: struct
{
    public bool AreEqual(T a, T b)
    {
        return a.Equals(b);
    }
}
```

```
// Можно
var asInt = new StructConstraint<int>();

// Ошибка компиляции
var asString = new StructConstraint<string>();
```

Ограничения – ссылочные типы (class)

```
// T - только reference type (классы)
public class ClassConstraint<T>
   where T: class
{
    public bool AreEqual(T a, T b)
    {
        return a == b;
    }
}
```

```
// Можно
var asString = new  ClassConstraint<string>();

// Ошибка компиляции
var asInt = new  ClassConstraint<int>();
```

Ограничения – несколько

```
public class StructConstraint<T, K>
    where T: struct
    where K: class, new()
{
     public bool AreEqual(T a, T b)
     {
         return a.Equals(b);
     }
}
```

```
// Можно
var asInt = new StructConstraint<int, string>();

// Ошибка компиляции
var asString = new StructConstraint<int, int>();
```

Ограничения – интерфейсы, классы

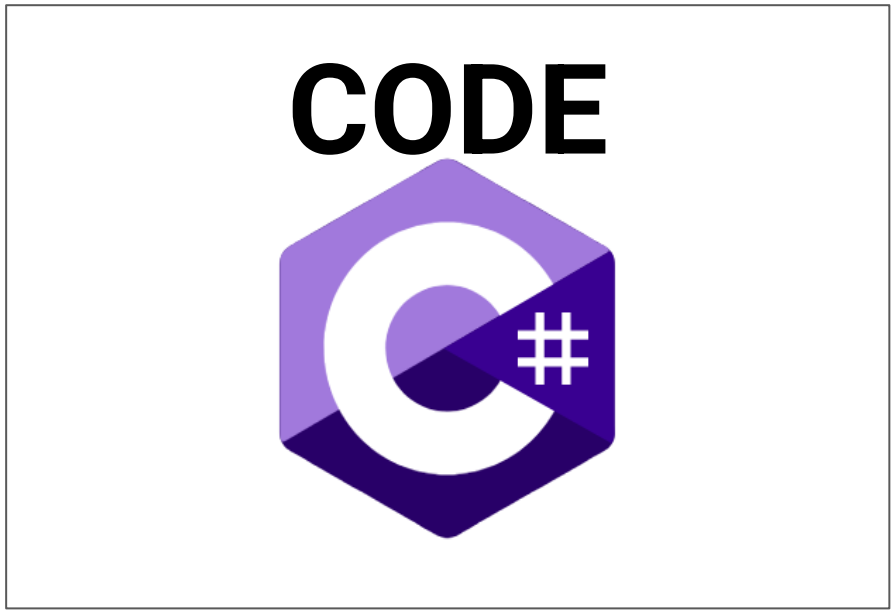
```
interface IVehicle{
    void Start();
    void Go();
}
class Auto : IVehicle
{
    public void Go()
    =>  Console.WriteLine("Vrum!");

    public void Start()
    =>  Console.WriteLine("Vruuuuuuuum
!");
}
```

```
class Rider<TV>
where TV : IVehicle
{
    public void RideAVehicle(TV vehicle)
    {
        vehicle.Start();
        vehicle.Go();
    }
}
```

Ограничения - прочие

where T : struct	Аргумент типа должен быть типом значения, не допускающим значения NULL.
where T : class	Аргумент типа должен быть ссылочным типом. Это ограничение также применяется к любому типу класса, интерфейса, делегата или массива.
where T : notnull	Аргумент типа должен быть типом, не допускающим значения NULL.
where T : unmanaged	Неуправляемый тип(int, double, bool, struct из неуправляемых типов, enum)
where T : new()	У типа есть конструктор без параметров
where T : U	Аргумент типа, указанный для T, должен быть аргументом, указанным для U, или производным от него.

Ограничения



https://github.com/Dzitsky/Otus.Generics.git

Ковариантность и контравариантность

Общие определения

- Применяется к интерфейсам-дженерикам и делегатам
- Управляют приведением типов, находящимися в одной иерархии наследования
- Решают проблему приведения одного дженерика к другому

Проблема

```
class Vehicle{ }

class Automobile : Vehicle { }
```

```
var a = new Automobile();
var b = new Vehicle();

// Так можно
b = a;
```

```
interface IDemo<T>{}

class ClassDemo<T>:IDemo<T>{}
```

```
IDemo<Automobile> a = new ClassDemo<Automobile>()
IDemo<Vehicle> b = new ClassDemo<Vehicle>();

// а так - нельзя
b = a;
```

Ковариантность

Ковариантность – на выходе операции выводить более конкретный тип, чем заданный изначально

Используется ключевое слово **out**

```
interface ICoVar<out T> {}

class CoVar<T> : ICoVar<T>{}


ICoVar<Automobile> auto = new CoVar<Automobile>();
ICoVar<Vehicle> vec = new CoVar<Vehicle>();

// Теперь можно приводить ICor<Automobile> к Icor<Vehicle>
vec = auto;
```

Контравариантность

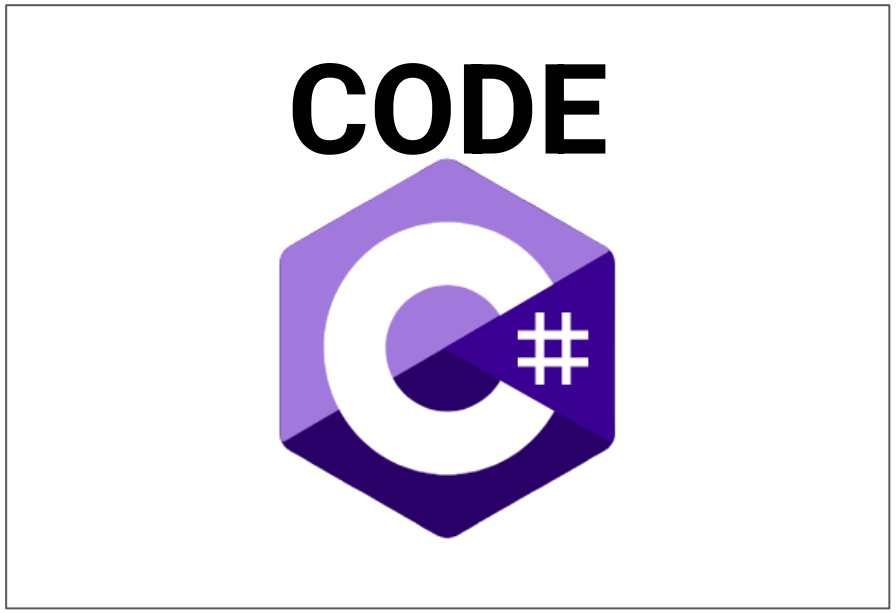
Контравариантность – позволяет во входном параметре использовать более универсальный тип

Используется ключево слово **in**

```
interface IContrVar<in T> {
    void Build(T v);
}

class ContrVar<T> : IContrVar<T>
{
    public void Build(T v) {}
}
```

```
IContrVar<Automobile> autocontr = new ContrVar<Vehicle>();

autocontr.Build(new Automobile());
```

Ковариантность и Контрвариатность



https://github.com/Dzitsky/Otus.Generics.git

Практика

Выводы и рефлексия

Тезисы

1. Изучили механизм дженериков в C#
2. Разобрались, как настраивать различные расширения
3. На практике попробовали, как использовать дженерики

Вопросы

Заполните, пожалуйста, опрос

Спасибо за внимание! Приходите на следующие вебинары