

# Обобщения

Generic



```
public class SuperArray
{
    int[] s;
}
```

класс является обобщённым

```
public class SuperArray<T>
{
    T[] s;
}
```

Тип – любой идентификатор  
**универсальный параметр**,  
так как вместо него можно  
подставить любой тип

```
SuperArray<int> iArr = new SuperArray<int>();
SuperArray <Stack<int>> stArr=new SuperArray<Stack<int>>>();
SuperArray<Person> perArr = new SuperArray<Person>();
```

# Обобщения (generics)

Механизм многократного использования алгоритмов

- ▶ *Обобщение* - параметризованный тип
- ▶ Определены для CLR – поддержка разных языков
- ▶ Открытый тип → закрытый тип

Tlist<T>

Tlist<int>

В CLR запрещено конструирование  
экземпляров открытых типов

экземпляры a,b,c

System.Collections.Generic

```
public class Student<T>
{ }
interface IAction<T>
{ }

static void main()
{
    Student<int> Nikita = new Student<int>();
    Student<string> Anna = new Student<string>();
}
```

Недостатки использования object:

- 1) InvalidCastException - два типа не совместимы друг с другом.
- 2) вероятность дополнительного потребления памяти и процессорного времени, если в ходе выполнения потребуется преобразовывать

# СВОЙСТВА

- 1) Универсальный тип может содержать другой универсальный тип

```
public class B<T>
{
    private A<T> one;
    private A<int> two;
}
```

- 2) Универсальные типы перегружаются на основе количества параметров

```
public class A { }
public class A<T> { }
public class A<T, U> { }
```

- 3) Универсальными могут быть классы, структуры, интерфейсы, делегаты, методы
- public void Method <R> (A<R> iA, B<R,T> iB)

```
public class Animal
{
    public void Move<T>(T distance)
    { }
}

static void Main(){
    Animal Носорог = new Animal();
    Носорог.Move(1);
    Носорог.Move("аршин");
    Носорог.Move<double>(45.6);
}
```

логическое  
выведение типов  
(type inference) →  
используется тип  
данных  
переменной, а не  
фактический тип  
объекта, на который  
ссылается

- ▶ 4) Могут содержать статические типы
- ▶ 5) Доступность конструируемых типов определяется на основе пересечения доступности универсального типа и типа в списке аргументов

```
public class lab2
{
    private class People { }
    public class Generic<T> { }
    private Generic<People> one;
    public Generic<People> two; // ошибка
}
```

- 5) могут использовать несколько универсальных параметров одновременно

```
class Transaction<U, V>
{
    public U FromAccount { get; set; }
    // с какого счета перевод
    public U ToAccount { get; set; }
    // на какой счет перевод
    public V Code { get; set; }
    // код операции
    public int Sum { get; set; }
    // сумма перевода
}
```


- 6) поддерживает механизм ограничений



В CLR существует механизм ограничений (constraints) - инструмент определения универсального типа с указанием допустимых для него аргументов типа

## *Ограничение на интерфейс*

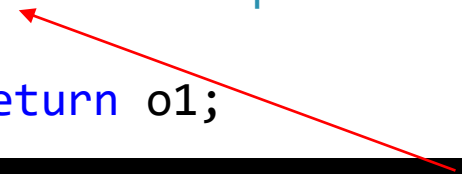
```
private static T Min<T>(T o1, T o2)
{
    if (o1.CompareTo(o2) < 0) return o1;
    return o2;
}
```



"T" не содержит определения для "CompareTo" и не удалось найти метод расширения "CompareTo", принимающий тип "T" в качестве первого аргумент

Ограничение сужает перечень типов, которые можно передать в обобщенном аргументе, и расширяет возможности по работе с этими типами.

```
public static T Min<T>(T o1, T o2) where T : IComparable<T>
{
    if (o1.CompareTo(o2) < 0) return o1;
    return o2;
}
```



указанный в T тип должен реализовывать обобщенный интерфейс IComparable того же типа (T).

# Ограничение типа значения

гарантирует компилятору, что указанный аргумент типа будет иметь значимый тип

Но значимые типы с поддержкой `null` (`System.Nullable<T>`) не подходят под это ограничение

```
internal sealed class Mama<T> where T : struct
{
    public static T GiveSomething()
    {
        // Допускается, потому что у каждого значимого типа неявно
        // есть открытый конструктор без параметров
        return new T();
    }
}
```

## *Ограничение на базовый класс*

```
public class Figure { }  
public class Rectangle : Figure { }  
public class Computer { }  
  
public class LinkedSet<U> where U : Figure { }  
  
static void Main(){  
    LinkedSet<Rectangle> бусы = new LinkedSet<Rectangle>();  
    LinkedSet<Computer> компкласс = new LinkedSet<Computer>();  
}
```

# Ограничение ссылочного типа

```
public class Computer { }  
  
public class LinkedSet<U> where U : class { }
```

```
static void Main()  
{
```

Этому ограничению удовлетворяют все типы-классы, типы-интерфейсы, типы-делегаты и типы-массивы

```
    LinkedSet<Computer> компкласс = new LinkedSet<Computer>();  
    LinkedSet<int> рядфурье = new LinkedSet<int>();  
}
```

```
internal sealed class Mama<T> where T : class  
{  
    public void M()  
    {  
        T temp = null; // Допустимо  
    }  
}
```

При отсутствии у T ограничений код бы не скомпилировался

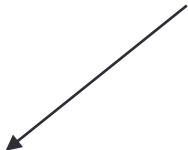
# Ограничение на конструктор

гарантирует компилятору, что указанный аргумент-тип будет иметь неабстрактный тип, имеющий открытый конструктор без параметров

```
public class Computer {  
    public Computer(int h){}  
}
```

```
public class LinkedSet<U> where U : new() { }
```

Требование  
предоставить  
конструктор без  
параметров



```
static void Main()  
{
```

```
    LinkedSet<Computer> компкласс = new LinkedSet<Computer>();  
    LinkedSet<int> рядфурье = new LinkedSet<int>();
```

```
}
```

# Ограничение на конструктор

```
class SomeClass { };  
    class TList<T> where T :new()  
    {  
        // Следующий код доступен благодаря ограничению на конструктор  
        T obj = new T();  
  
    }  
    static class Run{  
        public static void Main()  
        {  
            TList<SomeClass> Spisok = new TList<SomeClass>();  
        }  
    }  
}
```

## ► Особенности ограничения на конструктор

- последним по порядку
- ограничение `new ()` позволяет конструировать объект, используя только конструктор без параметров
- ограничение `new()` нельзя использовать одновременно с ограничением типа значения.

# Ограничение на связь параметров

```
public class Figure { }  
public class Rectangle : Figure { }
```

ОСНОВНОЕ      ДОПОЛНИТЕЛЬНЫЕ

```
public class LinkedSet<U, T> where U : class, T, new() { }
```

```
static void Main()  
{
```

↖ Ссылочным, разновидностью T и  
использовать конструктор без парам

```
    LinkedSet<Rectangle, Figure> чел = new LinkedSet<Rectangle, Figure>();  
    LinkedSet<Figure, String> рис = new LinkedSet<Figure, String>();
```

```
}
```



## ► Наследование

При переопределении виртуального обобщенного метода в переопределяющем методе должно быть задано то же число параметров-типов, а они, в свою очередь, наследуют ограничения, заданные для них методом базового класса

```
internal class Бабушка
{
    public virtual void М<T1>()
        where T1 : struct
    { }
}

internal sealed class Мама : Бабушка
{
    public override void М<T2>()
        where T2 : class
    { }
}
```

// Ошибка

Несоответствие

# Иерархии обобщенных (универсальных) классов

```
class Один<T>    {    }  
// Унаследованный обобщенный класс  
class Два<T> : Один <T>    { }  
// унаследованный класс с собственными параметрами  
class Три<T, V> : Один<T>    { }  
class Четыре<T, V, E, G> : Три<E,G>    { }  
// Обычный необобщенный класс  
class Пять    { }  
// Унаследованный от обычного класса обобщенный класс  
class Шесть<T> : Пять    { }
```


## ► Упрощенный синтаксис для ссылки на универсальный закрытый тип

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

# Значения по умолчанию

Так как параметр типа R не ограничен, он может иметь значимый или ссылочный тип, а приравнять переменную значимого типа к null нельзя, равно как и обратное

```
class TV<R>
{
    //    R model = 0;
    //    R model = null;
    R model = default(R);
}
```




# Статические члены

в CLR размещает статические поля типа в самом объекте-типе , каждый закрытый тип имеет свои статические поля

```
public class StatString<T>{  
    public static String name ;}  
  
static void Main()  
{  
    StatString<int>.name = "Матвей";  
    StatString<bool>.name = "Настя";  
}
```

Существуют два набора статических полей



# Сравнение экземпляров параметра типа

```
public static bool CheckM <T> (T element, T[] masElementov)
{
    foreach (T v in masElementov)
        if (v == element) // Ошибка!
            return true;

    return false;
}
```

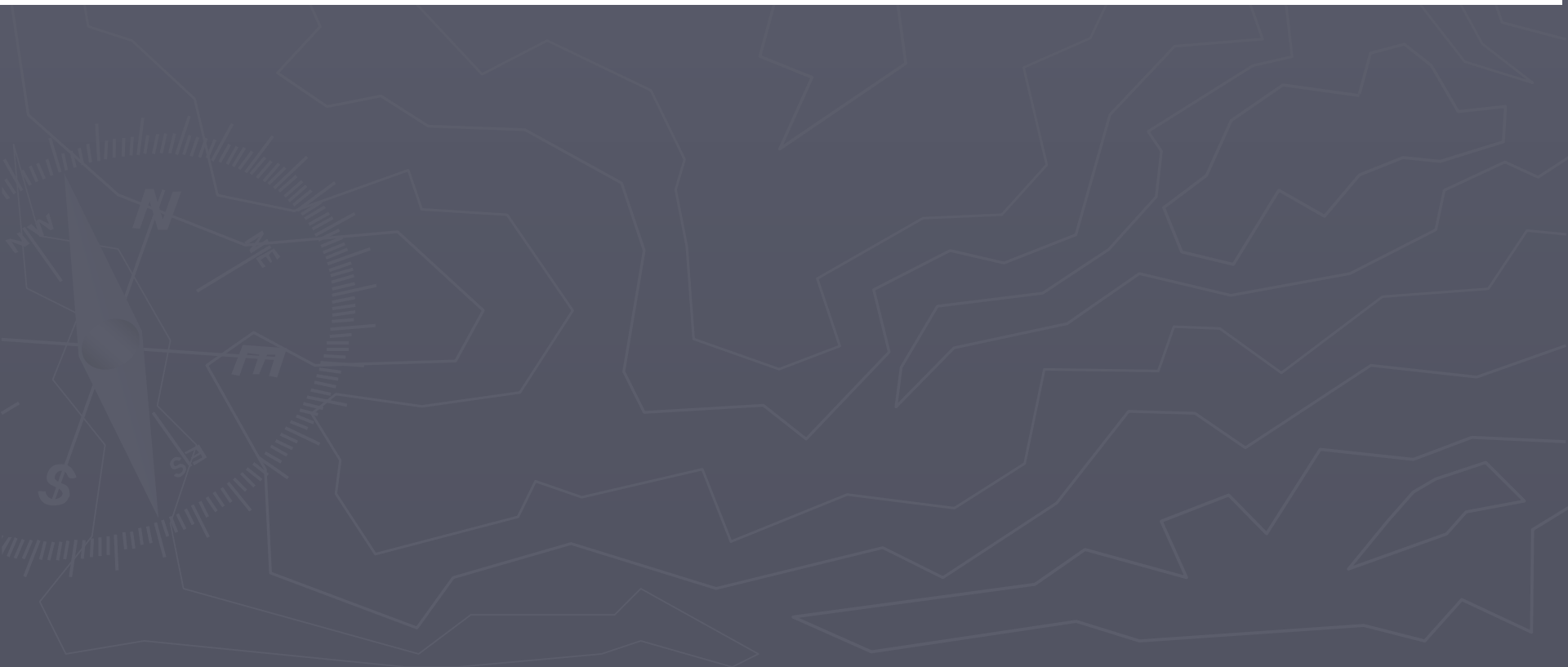
у T нет ограничений, и хотя можно сравнивать две переменные ссылочного типа, сравнивать две переменные значимого типа допустимо лишь в том случае, когда значимый тип перегружает оператор ==

```
public static bool CheckM<T>(T element, T[] masElementov) where T : IEquatable<T>
{
    foreach (T v in masElementov)
        if (v.Equals(element)) // OK
            return true;

    return false;
}
```

```
public static bool CheckM<T>(T element, T[] masElementov) where T : IComparable <T>
{
    foreach (T v in masElementov)
        if (v.CompareTo(element)>=0) // OK
            return true;

    return false;
}
```

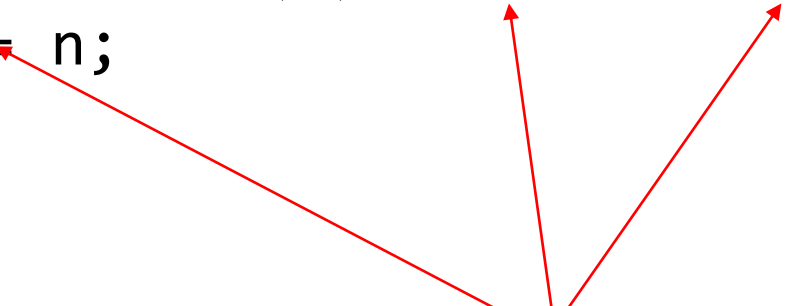


# Использование переменных универсального типа в качестве операндов

```
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;
}
```



нельзя применять к операндам типа T

ограничивает поддержку обобщений в среде CLR



# Ковариантность интерфейсов (делегатов)

- 1) средство, разрешающее методу возвращать тип, производный от класса, указанного в параметре типа
- 2) для интерфейсов и делегатов
- 3) распространяться только на тип, возвращаемый методом
- 4) только для ссылочных типов
- 5) ковариантный тип нельзя использовать в качестве ограничения в интерфейсном методе.

тип, производный от класса, указанного в параметре

```
public interface IInformation<out UY>
{
    UY GetInfo();
}
```

Аргумент-тип может быть преобразован от класса к одному из его базовых классов

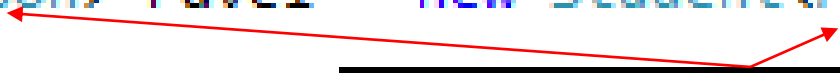
```
interface IStudy<out T> { }

class Student<T>:IStudy< T> { }

class Person { }
class Men : Person { }

static void Main()
{
    Person Sasha = new Men();
    IStudy<Person> Vika = new Student<Person>();

    IStudy<Person> Pavel = new Student<Men>();
}
```



Разрешено возвращать тип,  
производный от класса,  
указанного в параметре типа

# Контравариантность интерфейсов (делегатов)

- ▶ средство, позволяющим методу использовать аргумент, тип которого относится к базовому классу, указанному в соответствующем параметре типа
- ▶ для ссылочных типов
- ▶ параметр контравариантного типа можно применять только к аргументам методов

разрешает методу использовать аргумент, тип которого относится к базовому классу

```
public interface IDemo <in RR>  
{  
    void Show(RR obj);  
}
```

Параметр-тип может быть преобразован от класса к классу, производному от него.


```
interface IStudy<out T> { }

class Student<T>:IStudy< T> { }

class Person { }
class Men : Person { }

static void Main()
{
    Person Sasha = new Men();
    IStudy<Person> Vika = new Student<Person>();

    IStudy<Person> Pavel = new Student<Men>();
    IStudy<Men> Nikita = new Student<Person>();
}
```



Нельзя использовать аргумент, тип которого относится к базовому классу, указанному в параметре типа

```
interface IStudy<in T> { }
```

**Контравариантность**

```
class Student<T>:IStudy< T> { }
```

```
class Person { }
```

```
class Men : Person { }
```

```
static void Main()
```

```
{
```

```
    Person Sasha = new Men();
```

```
    IStudy<Person> Vika = new Student<Person>();
```

```
    IStudy<Person> Pavel = new Student<Men>();
```

```
    IStudy<Men> Nikita = new Student<Person>();
```

```
}
```

Теперь можно использовать аргумент,  
тип которого относится к базовому  
классу, указанному в параметре типа

делает его контравариантным

делает его ковариантным

```
public delegate R Func<in T, out R>(T arg);
```

```
static void Main() {
```

```
Func<Object, String> del1 = null;
```

```
// можно привести к типу Func с другими  
// параметрами-типами:
```

```
Func<String, Object> del2 = del1;
```

```
// Явного приведения типа не требуется
```

```
Object e = del2("");
```

# Особенности

- ▶ Вариантность неприменима для значимых типов из-за необходимости упаковки (boxing)
- ▶ Недопустима для параметра-типа, если при передаче аргумента используются out и ref

```
delegate void S <in T>(ref T t);
```

- ▶ Компилятор может самостоятельно проверить являются ли параметры обобщенного типа вариантными

- У свойств, индексаторов, событий, операторных методов, конструкторов и деструкторов не может быть параметров-ТИПОВ.

