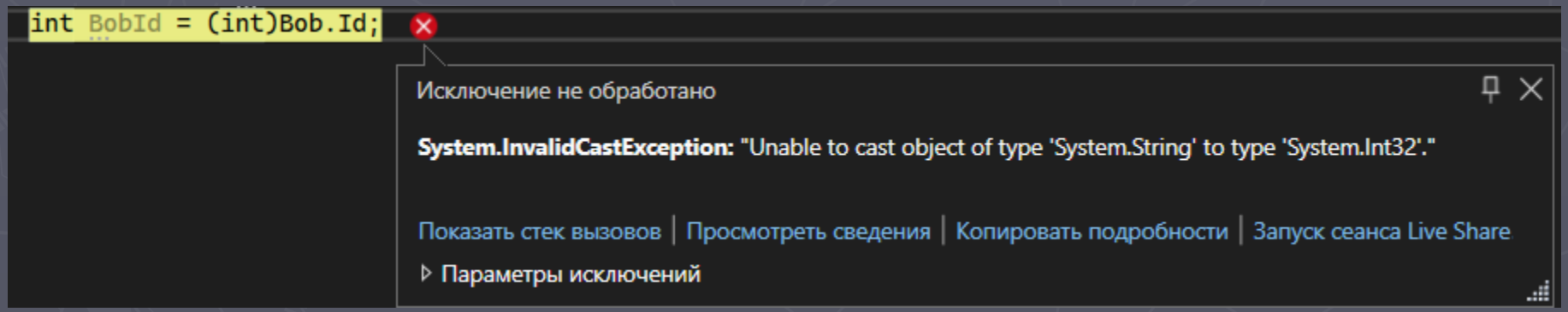
Обобщения

Generic

```
public class Exx
{
    Ссылок: 4
    public object Id { get; }
    Ссылок: 2
    public string Name { get; set; }
    Ссылок: 1
    public Exx(int id, string name)
    {
        Id = id;
        Name = name;
    }
    Ссылок: 1
    public Exx(string id, string name)
    {
        Id = id;
        Name = name;
    }
}
Ссылок: 0
```

```
private static void Main(string[] args)
{
    Exx Tom = new Exx(123, "Tom"); // упаковка в значения int в тип Object
    Exx Bob = new Exx("qq", "Bob");
    int TomId = (int)Tom.Id;   // Распаковка в тип int
    string BobId = (string)Bob.Id;
}
```

Упаковка и распаковка ведут к снижению производительности, так как системе надо осуществить необходимые преобразования.



другая проблема - проблема безопасности типов.

```
class Person<T>
{
    public T Id { get; set; }
    public string Name { get; set; }
    public Person(T id, string name)
    {
        Id = id;
        Name = name;
    }
}
```

Обобщенные типы позволяют не указывать конкретный тип, который будет использоваться

```
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* - параметризированный тип
- Определены для CLR – поддержка разных языков
- Открытый тип → закрытый тип

  Tlist<T> Tlist<int>

  экземпляры a,b,c

System.Collections.Generic

В CLR запрещено конструирование экземпляров открытых типов

```
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

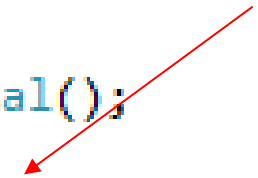
```
public class B<T>
{
    private A<T> one;
    private A<int> two;
}
```

- 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

```
class имя_класса<T> where T: тип_ограничения
```

В качестве ограничений мы можем использовать следующие типы:

- Классы
- Интерфейсы
- class
- struct
- new()

Ограничение на интерфейс

Аргумент типа должен являться заданным интерфейсом или реализовывать его. Можно указать несколько ограничений интерфейса. Заданный в ограничении интерфейс также может быть универсальным. В

```
where T : имя_интерфейса
```

где T — это имя параметра типа, а имя_интерфейса — конкретное имя ограничиваемого интерфейса

```
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 аргумента

```
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 на базовый класс

Аргумент типа должен иметь базовый класс или производный от него класс.

```
public class Figure { }
public class Rectengle : Figure { }
public class Computer { }

public class LinkedSet<U> where U : Figure { }

static void Main(){
    LinkedSet<Rectengle> бусы = new LinkedSet<Rectengle>();
    LinkedSet<Computer> компкласс = new LinkedSet<Computer>();
```

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

```
where T : struct
```

```
struct UI
{
    public UI(string Name, int Age)
    {
        this.Name = Name;
        this.Age = Age;
    }

    public string Name;
    public int Age;
}

// Обобщенный класс, использующий ограничение на тип значения
class UserInfo<T> where T : struct
{
    T obj;

    public UserInfo(T ob)
    {
        obj = ob;
    }
}

class Program
{
    static void Main()
    {
        UI user1 = new UI(Name: "Alexandr", Age: 26);
        UserInfo<UI> user = new UserInfo<UI>(user1);
    }
}
```

Ограничение ссылочного типа

Аргумент типа должен быть ссылочным типом. Это ограничение также применяется к любому типу класса, интерфейса, делегата или массива.

where T : class

Ограничение ссылочного типа

```
public class Computer { }

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

static void Main()
{

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

}
```

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

Ограничение на конструктор

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

where T : new()

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

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

static void Main()
{

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

}
```

Требование предоставить конструктор без параметров

```
using System;

namespace ConsoleApplication1
{
    interface IUserInfo
    {
        string Name { get; set; }
        int Age { get; set; }
    }

    // Создадим класс, реализующий интерфейс IUserInfo
    class AllInfoUser : IUserInfo
    {
        public AllInfoUser() { }
        public AllInfoUser(string Family, string uName, int uAge)
        {
            this.Family = Family;
            this.Name = uName;
            this.Age = uAge;
        }

        public string Name { get; set; }
        public int Age { get; set; }
        public string Family { get; set; }

        public override string ToString()
        {
            string s = String.Format("Информация о пользователе: \n{0} {1} {2}\n",this.Name,Fa
mily,this.Age);
            return s;
        }
    }
```

```
// Обобщенный класс использующий ограничение на интерфейс и конструктор
class Info<T> where T : IUserInfo, new()
{
    T[] UserList;
    int i;

    // Следующий код доступен благодаря ограничению на конструктор
    T obj = new T();

    public Info()
    {
        UserList = new T[3];
        i = 0;
    }

    public void Add(T obj)
    {
        if (i == 3) return;
        UserList[i] = obj;
        i++;
        return;
    }

    public void ReWrite()
    {
        foreach (T t in UserList)
            Console.WriteLine(t.ToString());
    }
}
```

```
class Program
{
    static void Main()
    {
        Info<AllInfoUser> database1 = new Info<AllInfoUser>();
        database1.Add(new AllInfoUser(uName: "Alex", Family: "Erohin", uAge: 26));
        database1.Add(new AllInfoUser(uName: "Alexey", Family: "Volkov", uAge: 28));
        database1.Add(new AllInfoUser(uName: "Dmitryi", Family: "Medvedev", uAge: 50));

        database1.ReWrite();

        Console.ReadLine();
    }
}
```

- 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.

Параметры типа в качестве ограничений

Аргумент типа, указанный для T, должен быть аргументом, указанным для U, или производным от него.

where T : U

последовательно можно задать ограничения к каждому из них

```
class Messenger<T, P>
    where T : Message
    where P: Person
{
    public void SendMessage(P sender, P receiver, T message)
    {
        Console.WriteLine($"Отправитель: {sender.Name}");
        Console.WriteLine($"Получатель: {receiver.Name}");
        Console.WriteLine($"Сообщение: {message.Text}");
    }
}
class Person
{
    public string Name { get;}
    public Person(string name) => Name = name;
}

class Message
{
    public string Text { get; } // текст сообщения
    public Message(string text) =>  Text = text;
}
```

```
Messenger<Message, Person> telegram = new Messenger<Message, Person>();
Person tom = new Person("Tom");
Person bob = new Person("Bob");
Message hello = new Message("Hello, Bob!");
telegram.SendMessage(tom, bob, hello);
```

Ограничение на связь параметров

```
public class Figure { }
public class Rectengle : Figure { }

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

static void Main()
{

    LinkedSet<Rectengle, Figure> чел = new LinkedSet<Rectengle, Figure>();
    LinkedSet<Figure,String> рис = new LinkedSet<Figure,Single>();

}
```

основное дополнительные

Ссылочным, разновидностью T и использовать конструктор без парам

Если для универсального параметра задано несколько ограничений, то они должны идти в определенном порядке:

- Название класса, class, struct. Причем мы можем одновременно определить только одно из этих ограничений
- Название интерфейса
- new()

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 M<T1>()
              where T1 : struct
            {  }
        }
internal sealed class Мама : Бабушка
        {
            public override void M<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         { }
```

Варианты наследования обобщений:

Первый вариант заключается в создание класса-наследника, который типизирован тем же типом, что и базовый

```
class Person<T>
{
    public T Id { get;}
    public Person(T id)
    {
        Id = id;
    }
}
```

```
class UniversalPerson<T> : Person<T>
{
    public UniversalPerson(T id) : base(id) { }
}
```

```
Person<string> person1 = new Person<string>("34");
Person<int> person3 = new UniversalPerson<int>(45);
UniversalPerson<int> person2 = new UniversalPerson<int>(33);
Console.WriteLine(person1.Id);
Console.WriteLine(person2.Id);
Console.WriteLine(person3.Id);
```

Второй вариант: создание обычного необобщенного класса-наследника.

```
class StringPerson : Person<string>
{
    public StringPerson(string id) : base(id) { }
}
```

при наследовании у базового класса надо явным образом определить используемый тип:

```
StringPerson person4 = new StringPerson("438767");
Person<string> person5 = new StringPerson("43875");
// так нельзя написать
//Person<int> person6 = new StringPerson("45545");
Console.WriteLine(person4.Id);
Console.WriteLine(person5.Id);
```

Теперь в производном классе в качестве типа будет использоваться тип string.

Третий вариант представляет типизацию производного класса параметром совсем другого типа, отличного от универсального параметра в базовом классе

```
class IntPerson<T> : Person<int>
{
    public T Code { get; set; }
    public IntPerson(int id, T code) : base(id)
    {
        Code = code;
    }
}
```

для базового класса также надо указать используемый тип

```
IntPerson<string> person7 = new IntPerson<string>(5, "r4556");
Person<int> person8 = new IntPerson<long>(7, 4587);
Console.WriteLine(person7.Id);
Console.WriteLine(person8.Id);
```

тип IntPerson типизирован еще одним типом, который может не совпадать с типом, который используется базовым классом.

Значения по умолчанию

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

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

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

```
class Person<T>
{
    public static T? code;
    public T Id { get; set; }
    public string Name { get; set; }
    public Person(T id, string name)
    {
        Id = id;
        Name = name;
    }
}
```

```
Person<int> tom = new Person<int>(546, "Tom");
Person<int>.code = 1234;

Person<string> bob = new Person<string>("abc", "Bob");
Person<string>.code = "meta";

Console.WriteLine(Person<int>.code);        // 1234
Console.WriteLine(Person<string>.code);    // meta
```

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;
}
```

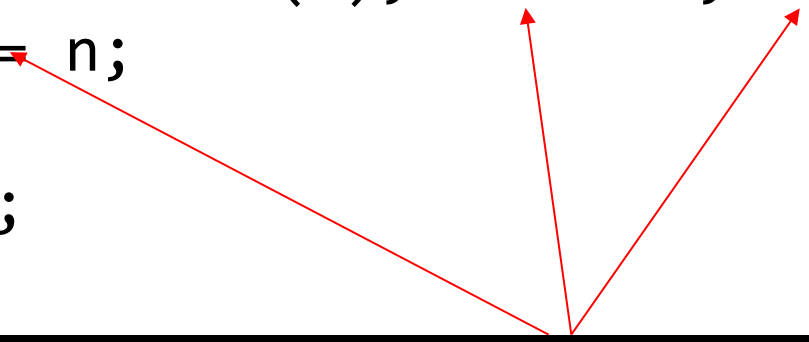
у T нет ограничений, и хотя можно сравнивать две переменные ссылочного типа, сравнивать две переменные значимого типа допустимо лишь в том случае, когда значимый тип перегружает оператор ==

```
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: позволяет использовать более конкретный тип, чем заданный изначально

Контравариантность: позволяет использовать более универсальный тип, чем заданный изначально

Инвариантность: позволяет использовать только заданный тип

Ковариантность интерфейсов

Обобщенные интерфейсы могут быть ковариантными, если к универсальному параметру применяется ключевое слово out. Такой параметр должен представлять тип объекта, который возвращается из метода.

out в определении интерфейса указывает, что данный интерфейс будет ковариантным.

```
interface IMessenger<out T>
{
    T WriteMessage(string text);
}
class EmailMessenger : IMessenger<EmailMessage>
{
    public EmailMessage WriteMessage(string text)
    {
        return new EmailMessage($"Email: {text}");
    }
}
```

```
IMessenger<Message> outlook = new EmailMessenger();
Message message = outlook.WriteMessage("Hello World");
Console.WriteLine(message.Text);    // Email: Hello World


IMessenger<EmailMessage> emailClient = new EmailMessenger();
IMessenger<Message> messenger = emailClient;
Message emailMessage = messenger.WriteMessage("Hi!");
Console.WriteLine(emailMessage.Text);    // Email: Hi!
```

```
1  interface IMessenger<T>
```

то мы столкнулись бы с ошибкой в строке

```
1  IMessenger<Message> outlook = new EmailMessenger();  // ! Ошибка
2
3  IMessenger<EmailMessage> emailClient = new EmailMessenger();
4  IMessenger<Message> messenger = emailClient;  // ! Ошибка
```

Поскольку в этом случае невозможно было бы привести объект IMessenger<EmailMessage> к типу IMessenger<Message>

При создании ковариантного интерфейса надо учитывать, что универсальный параметр может использоваться только в качестве типа значения, возвращаемого методами интерфейса. Но не может использоваться в качестве типа аргументов метода или ограничения методов интерфейса.

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

Для создания контравариантного интерфейса надо использовать ключевое слово in.

```
interface IMessenger<in T>
{
    void SendMessage(T message);
}
class SimpleMessenger : IMessenger<Message>
{
    public void SendMessage(Message message)
    {
        Console.WriteLine($"Отправляется сообщение: {message.Text}");
    }
}
```

мы можем переменной типа IMessenger<EmailMessage> передать объект IMessenger<Message> или SimpleMessenger

```
IMessenger<EmailMessage> outlook = new SimpleMessenger();
outlook.SendMessage(new EmailMessage("Hi!"));

IMessenger<Message> telegram = new SimpleMessenger();
IMessenger<EmailMessage> emailClient = telegram;
emailClient.SendMessage(new EmailMessage("Hello"));
```

Если бы ключевое слово in не использовалось бы, то мы не смогли бы это сделать. То есть объект интерфейса с более универсальным типом приводится к объекту интерфейса с более конкретным типом.

Контравариантность интерфейсов

При создании контрвариантного интерфейса надо учитывать, что универсальный параметр контрвариантного типа может применяться только к аргументам метода, но не может применяться к возвращаемому результату метода.

Контравариантность интерфейсов (делегатов)

- средство, разрешающим 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> { }

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>();
}
```

Контравариантность

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

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.