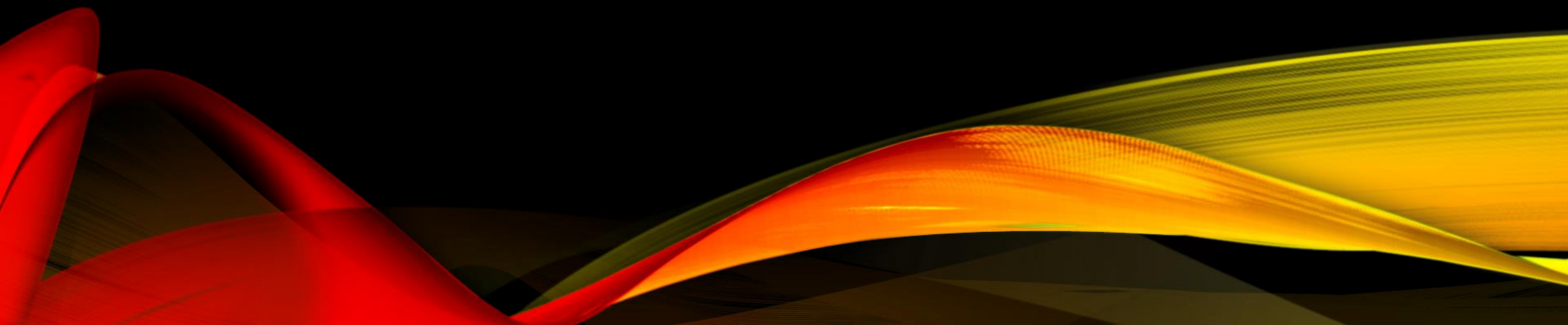
ЛЕКЦИЯ 11-12

- Модуль 3
- 14.02.2024
- Обобщённые типы: структура, интерфейс
- Виртуальность и перегрузка при наследовании обощенных классов
- Вариантности обобщённых интерфейсов

ЦЕЛИ ЛЕКЦИИ

- Познакомиться с обобщёнными интерфейсами, структурами и делегатами
- Расширить знания об ограничениях на обобщённые параметры
- Познакомиться со сравнимостью объектов обобщённых типов

ОБОБЩЁННЫЕ СТРУКТУРЫ



СИНТАКСИС ОБОБЩЁННОЙ СТРУКТУРЫ

Объявление структуры

```
struct StructIdentifier <T1, T2, …., TN> {
{
  // элементы структуры, в т.ч. работающих членами T1,…,TN
}
```

Имя обобщённой структуры

Имена обобщённых типов, используемых в структуре

Инстанцирование структуры

```
StructIdentifier <t1, t2, …, tN> structName =
                    new StructIdentifier <t1, t2, …, tN>(args);
```

Имя экземпляра структуры

Вызов конструктора

Названия конкретных типов, выступающих параметрами

ПРИМЕР. ОБОБЩЁННАЯ СТРУКТУРА

```
struct Segment<T>
{
    public (T, T) Begin { get; set; }
    public (T, T) End { get; set; }

    public Segment(T x1, T y1, T x2, T y2) =>
        (Begin, End) = ((x1, y1), (x2, y2));

    public override string ToString() => $"({Begin.Item1}; {Begin.Item2}) " +
        $"({End.Item1}; {End.Item2})";
}
```

```
Segment<double> line1 = new Segment<double>(2.5, 3.8, -1.4, 8.2);
Console.WriteLine(line1);

Segment<long> line2 = new Segment<long>(2323L, -2332L, 5000L, 3000L);
Console.WriteLine(line2);
```

ОГРАНИЧЕНИЕ НА ССЫЛОЧНЫЙ ТИП WHERE T : STRUCT

Смысл ограничения: запретить использовать в качестве типизирующих параметров ссылочные типы (классы, интерфейсы)

Ограничение позволяет использовать в качестве обобщенного типа только типы значений (структура). Попытка создать закрытый тип на основе ссылочного типа приведёт к ошибке компиляции

```
class DemoGenericClass<T> where T : struct
{
    public T Field { get; set; }
}
```

T может быть только типом значений

string – является ссылочным типом

```
// DemoGenericClass<string> demo = new DemoGenericClass<string>(); // error!
DemoGenericClass<int> demo = new DemoGenericClass<int>(); // ok!
```

ПРИМЕР: ОБОБЩЁННАЯ СТРУКТУРА

Это системный тип, описание по ссылке: https://learn.microsoft.com/ru-ru/dotnet/api/system.nullable-1?view=net-8.0

```
public struct Nullable<T> where T : struct
{
    public T Value { get; private set; }
    public bool HasValue { get; private set; }

    public Nullable(T value) => (Value, HasValue) = (value, true);

    public static implicit operator Nullable<T>(T value) => new(value);
    public static explicit operator T(Nullable<T> value) => value.HasValue
        ? value.Value
        : throw new InvalidOperationException("The value is not present.");

    public T ValueOrDefault() => HasValue ? Value : default;

    public override string ToString() => HasValue ? Value.ToString() : string.Empty;
}
```

Nullable-структура имеет смысл только для типов значений, не допускающих null.

ИСПОЛЬЗОВАНИЕ ОБОБЩЁННОЙ СТРУКТУРЫ

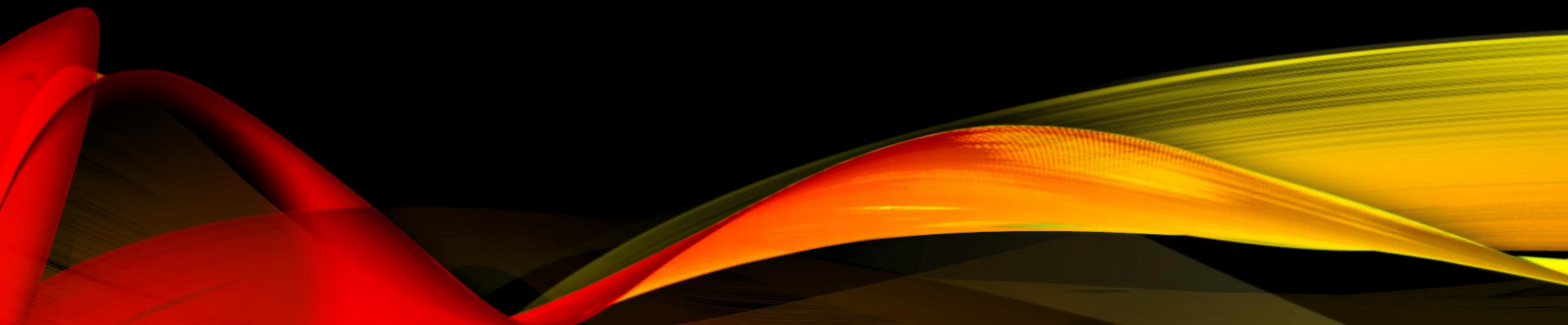
Значение типа по умолчанию.
Эквивалентно:
new(), default(Nullable<int>), new Nullable<int>().

```
using System;

Nullable<int> intVal = default;
Console.WriteLine($"{(intVal.HasValue ? $"{intVal.Value}" : "No value present")}.");
intVal = int.MaxValue;
Console.WriteLine($"inValue now contains: {intVal}");

Nullable<DateTime> dateVal = default;
Console.WriteLine($"The value for empty Nullable<DateTime>: {dateVal.ValueOrDefault()}");

// Строка ниже приведёт к ошибке компиляции - T не может быть ссылочным типом:
// Nullable<string> stringVal = "this won't work";
```

Вывод:

```
No value present.
inValue now contains: 2147483647
The value for empty Nullable<DateTime>: 01-Jan-01
00:00:00
```

ОБОБЩЁННЫЙ ИНТЕРФЕЙС



СИНТАКСИС ОБОБЩЁННОГО ИНТЕРФЕЙСА

Объявление интерфейса

```
interface InterfaceIdentifier <T1, T2, …., TN> {
{
  // члены интерфейса, в т.ч. работающие с типами T1,…,TN
}
```

Имя обобщённого интерфейса

Имена обобщённых типов – параметров интерфейса

Класс, реализующий обобщённый интерфейс

```
class ClassIdentifier<T1, T2,…, TN> :
                    InterfaceIdentifier <T1, T2, …, TN>;
```

Имя класса, реализующего интерфейс

Названия конкретных типов, выступающих параметрами

ПРИМЕР РЕАЛИЗАЦИИ ОБОБЩЁННОГО ИНТЕРФЕЙСА

```
interface ArifmeticCalculations<T>
{
    double Sum(T op1, T op2);
    double Sub(T op1, T op2);
}
```

Объявление обобщённого интерфейса

Реализация обобщённым типом

```
class Calculations<T> : ArifmeticCalculations<T>
{
    public double Sum(T op1, T op2) => Convert.ToDouble(op1) + Convert.ToDouble(op2);
    public double Sub(T op1, T op2) => Convert.ToDouble(op1) - Convert.ToDouble(op2);
}
```

Вывод:

```
5+6=11 5-6=-1
5.22+6=11,22 5.22-6=-0,78
```

```
//  Пробуем на целых типах.
Calculations<int> intCalc = new Calculations<int>();
Console.WriteLine($"5+6={intCalc.Sum(5, 6)} 5-6={intCalc.Sub(5, 6)}");

//  Пробуем на вещественных типах, отличных от double.
Calculations<float> floatCalc = new Calculations<float>();
Console.WriteLine($"5.22+6={floatCalc.Sum(5.22f, 6f):f2} 5.22-6={floatCalc.Sub(5.22f, 6f):f2}");
```


Исходный код (https://replit.com/@olgamaksimenkova/SimpleGenericInterface)

ПРИМЕР РЕАЛИЗАЦИИ ОБОБЩЁННОГО ИНТЕРФЕЙСА ДЛЯ ПОДДЕРЖКИ ПЕРЕГРУЗКИ ОПЕРАЦИИ


```
/// <summary>
/// Похожий  тип можно найти в семинаре 7, модуль 2.
/// Описывает комплексные числа, с возможностью сложения.
/// </summary>
internal class Complex : ISummable<Complex>
{
    public double Re { get; set; }
    public double Im { get; set; }

    public Complex(double re, double im) => (Re, Im) = (re, im);
    /// <summary>
    /// Метод сложения текущего объекта с объектом other.
    /// Реализация даёт нам гарантии, что можно использовать этот тип, как параметр
    /// при ограничении с ISummable<T>
    /// </summary>
    /// <param name="other"></param>
    /// <returns></returns>
    public Complex Add(Complex other) => new Complex(Re+other.Re,Im+other.Im);
    public static Complex operator +(Complex lh, Complex rh) => lh.Add(rh);
}
```

```
internal interface ISummable<T>
{
    T Add(T other);
}
```

```
Calculator<Complex> cmxCalc = new Calculator<Complex>();
Complex cmx1 = new Complex(1, -2);
Complex cmx2 = new Complex(2, 3.2);
Console.WriteLine($"{(cmx1+cmx2).ToString()}");
```

ПРИМЕР ПЕРЕГРУЗКИ ОПЕРАЦИЙ ДЛЯ ОБОБЩЕННОГО ТИПА С ИСПОЛЬЗОВАНИЕМ DYNAMIC

```
internal class ArifOper<T>
{
    public T Value { get; set; }
    public static ArifOper<T> operator + (ArifOper<T> lh, ArifOper<T> rh)
    {
        dynamic res = (dynamic)lh.Value + rh.Value;
        return new ArifOper<T> { Value = res };
    }
    public static ArifOper<T> operator *(ArifOper<T> lh, ArifOper<T> rh)
    {
        dynamic res = (dynamic)lh.Value * rh.Value;
        return new ArifOper<T> { Value = res };
    }
    public override string ToString()=>Value.ToString();
}
```

```
ArifOper<int> intArif = new ArifOper<int>();
intArif.Value = 100;
//Console.WriteLine(intArif + 100);
ArifOper<int> intArif2 = new ArifOper<int>();
intArif2.Value = 100;
Console.WriteLine((intArif + intArif2).ToString());
```

ОШИБКА ПРИ РЕАЛИЗАЦИИ ОБОБЩЁННОГО ИНТЕРФЕЙСА

Представленный ниже код НЕ компилируется, т. к. IPrintable<TPrice> и IPrintable<decimal> могут совпадать, что недопустимо:

```
public interface IPrintable<T> {  public void Print(T value); }

public record Product<TPrice>(uint ID, TPrice Price)
    : IPrintable<TPrice>, IPrintable<decimal>
{
    public void Print(TPrice value)
        => System.Console.WriteLine($"ID: {ID}, Price: {Price}");

    public void Print(decimal value)
        => System.Console.WriteLine($"Price request for {ID}: {value}");
}
```

Данная реализация является недопустимой.

СРАВНЕНИЕ ОБЪЕКТОВ ОБОБЩЁННЫХ ТИПОВ

Для обеспечения сравнения двух объектов обобщённого типа T, обобщенный класс должен реализовывать:

- `System.IComparable<T>` или `System.Icomparable`
- `System.IEquatable<T>`

Если требуется сравнение «равно / не равно»

Если требуется сравнение «больше / меньше»

ПРИМЕР. ОБОБЩЁННЫЙ КЛАСС

```
internal class ComparableObject<T> where T :
    IComparable<T>,
    IEquatable<T>
{
    public T obj { get; }
    public ComparableObject(T initObj) => this.obj = initObj;
    public void Compare(T other)
    {
        string output = "";
        output += obj.CompareTo(other) > 0 ? ${obj} is greater" :
            obj.CompareTo(other) < 0 ? ${obj} is smaller" : ${obj} is the same";
    }
    public void EqualityCompare(T other)
    {
        if (obj.Equals(other)) { Console.WriteLine(${obj} is the same"); }
        else { Console.WriteLine(${obj} is not the same"); }
    }
}
```

Исходный код (https://replit.com/@olgamaksimenkova/GenericObjectComparison)

ПРИМЕР. ОБЪЕКТ, УДОВЛЕТВОРЯЮЩИЙ УСЛОВИЯМ

```
internal class PositiveNumber : IComparable<PositiveNumber>, IEquatable<PositiveNumber>
{
    private double _value;
    public PositiveNumber(double numb) { _value = numb; }
    public double Value { get { return _value; } set { _value = value; } }

    public int CompareTo(PositiveNumber otherNumber) =>
        _value > otherNumber.Value ? -1 : _value < otherNumber.Value? 1 : 0;

    public bool Equals(PositiveNumber otherNumber) => _value == otherNumber.Value;
    public override bool Equals(object? obj) =>
        obj is PositiveNumber ? Equals((PositiveNumber)obj) : false;
    public override int GetHashCode() => _value.GetHashCode();
    public override string ToString() => $"{_value}";
}
```

```
PositiveNumber n1 = new(2.9);
PositiveNumber n2 = new(3.3);

ComparableObject<PositiveNumber> a = new ComparableObject<PositiveNumber>(n1);
a.Compare(n2);
a.EqualityCompare(n2);
a.EqualityCompare(n1);
```

ВАРИАНТНОСТЬ В C#

Продолжение

Вариантность обобщённых типов

ОПРЕДЕЛЕНИЯ

- Вариантность – свойство преобразования типов операторами
- Оператор – любая сущность языка C#, преобразующая тип данных в производные от него
  - Определение вариантности в C# (https://habr.com/ru/sandbox/91751/), в статье есть неточности в примерах, будьте внимательны

Преобразование затрагивает один тип

```
T[]

Action <T>
```

Преобразование затрагивает два типа

```
T1 MethodId(T2)

Func<T1, T2>
```

СООТНЕСЕНИЕ ТИПОВ

- T > U «тип T больше типа U»
  - Экземпляр типа T можно заменить экземпляром типа U
- T < U «тип T меньше типа U»
  - Экземпляр типа U можно заменить экземпляром типа T
- T = U «тип T равен типу U»
  - Замены U на T и T на U равновозможны
- T != U «тип T не сравним с типом U»
  - Замены не возможны, типы не сравнимы

Здесь оператор «больше», «меньше» и «равно» не понимается в арифметическом смысле

СВОЙСТВА ПРЕОБРАЗОВАНИЯ

Преобразование, осуществляемое оператором, проявляет свойство:

- Ковариантности, если оно сохраняет отношение между парой типов после их преобразования в производные
- Контравариантности, если оно заменяет отношение «больше» на «меньше», но сохраняет «равны» и «не сравнимы»
- Бивариантности, если отношения «больше» и «меньше» заменяются на «равны», а «равны» и «не сравнимы» сохраняются
- Инвариантности, если любое отношение, отличное от «равно» обращается в «не сравнимы»

КОВАРИАНТНОСТЬ

Ковариантность позволяет использовать производный тип с большей глубиной наследования, чем задано изначально

В случае необходимости использования параметров ковариантных типов для обобщённых интерфейсов и делегат-типов соответствующие параметры типа объявляются с ключевым словом out

```
// Ссылки в C# ковариантны: по ссылке типа родителя
// всегда можно разместить объект типа наследника:
Base derived = new Derived();

public class Base { }
public class Derived : Base { }
```

КОНТРАВАРИАНТНОСТЬ

- Контравариантность позволяет использовать более общий тип (с меньшей глубиной наследования), чем заданный изначально

```
class Base { }
class Derived : Base { }
```

- Предполагается приведение объекта базового типа к производному типу:
  - `< Derived d = Base b; >`

В обобщениях параметр контравариантного типа объявляется с ключевым словом in

ИНВАРИАНТНОСТЬ

Инвариантность допускает использование только изначально заданного типа

```
class Base { }
class Derived : Base { }
```

- Т.е. параметр инвариантного обобщённого типа не является ни ковариантным, ни контравариантным:
- `< Derived d = Base b; >  // Ошибка.`
- `< Base b = Derived d; >  // Ошибка.`

В обобщениях параметр типа без in/out является инвариантным (по умолчанию)!

КОВАРИАНТНОСТЬ И КОНТРАВАРИАНТНОСТЬ ИНТЕРФЕЙСОВ И КЛАССОВ

- Ковариантность и контравариантность в обобщениях C# появились с версии 4.0
  - Covariance and contravariance in C# (https://ericlippert.com/2007/10/16/covariance-and-contravariance-in-c-part-1/)

Для указания поддержки:

- **ковариантности** типом T используется ключевое слово out
  - `out T`
- **контравариантности** типом T используется ключевое слово in
  - `in T`
- **инвариантности** – отсутствие out/in

КОВАРИАНТНОСТЬ И КОНТРАВАРИАНТНОСТЬ ИНТЕРФЕЙСОВ

Ковариантный обобщённый интерфейс

```
interface InterfaceId<out T>
{
    // объявление методов
    // некоторые возвращают T
}
```

Контравариантный обобщённый интерфейс

```
interface InterfaceId<in T>
{
    // объявление методов
    // некоторые НЕ возвращают T
}
```

ПРИМЕР ИНВАРИАНТНОГО ИНТЕРФЕЙСА (ILIST)

```
public class A
{
    public int X { get; set; }
    public int CompareTo(A other) => X.CompareTo(other.X);

}
public class B : A
{
    public int Y { get; set; }
}
```

```
// IList - инвариантен
IList<A> aList = new List<A>();
aList.Add(new A() { X = 5 });
aList.Add(new A() { X = 6 });
aList.Add(new B() { Y = 7 });

IList<B> bList = new List<B>();
//aList = bList;
//bList = aList;
//IList<A> list = new List<B>();
```

Объект с типом наследника добавить можно

Ссылку на список с типом наследника **нельзя** присвоить в ссылку с типом родителя (наоборот тоже)

ПРИМЕР КОНТРАВАРИАНТНОГО ИНТЕРФЕЙСА (ICOMPARABLE)

```
public class A : IComparable<A>
{
    public int X { get; set; }
    public int CompareTo(A other) => X.CompareTo(other.X);

}
public class B : A
{
    public int Y { get; set; }
}
```

```
A objA = new A();
objA.X = 5;
A objB = new B();
objB.X = 7;
// IComparable контравариантен
// Тип он использует, но не создаёт
IComparable<A>[] compObjs = {objA, objB};
Array.ForEach(compObjs, x =>
Console.Write($"{x}"));
IComparable<B>[] compObjs2 = {objB, objA};
// compOjbs = compObjs2;
```

Здесь мог быть пример ***ковариантного*** интерфейса **`IEnumerable`**,но его мы рассмотрим отдельно

ПРИМЕР. КОВАРИАНТНЫЙ ИНТЕРФЕЙС

```
public class User {
    public string Name { get; init; }
    public User(string name) => Name = name;

    public override string ToString() => Name;
}
```

```
public class Gamer : User {
    public Gamer(string name): base(name) { }
}
```

```
interface INamedObjectCreator<out T> {
    T CreateNamedObject(string name);
}
```

```
class GamerCreator : INamedObjectCreator<Gamer>
{
    public Gamer CreateNamedObject(string name) => new Gamer(name);
}
```

```
INamedObjectCreator<User> creator = new GamerCreator();
User someUser = creator.CreateNamedObject("Ivan");
Console.WriteLine(someUser);

INamedObjectCreator<Gamer> gamersCreator = new GamerCreator();
INamedObjectCreator<User> usersCreator = gamersCreator;
User gamer = usersCreator.CreateNamedObject("Semen");
Console.WriteLine(gamer);
```

Вывод:
Ivan
Semen

Исходный код (https://replit.com/@olgamaksimenkova/CovarContravarInterfaceDemo)

ПРИМЕР. КОНТРАВАРИАНТНЫЙ ИНТЕРФЕЙС

```
public class User {
    public string Name { get; init; }
    public User(string name) => Name = name;

    public override string ToString() => Name;
}
```

```
public class Gamer : User {
    public Gamer(string name): base(name) { }
}
```

```
interface IPrintName<in T>
{
    string PrintName(T obj);
}
```

```
class GameNamePrinter : IPrintName<User>
{
    public string PrintName(User user) => user.Name;
}
```

```
IPrintName<Gamer> printer = new GameNamePrinter();
Console.WriteLine(printer.PrintName(new Gamer("Ivan")));

IPrintName<User> anotherPrinter = new GameNamePrinter();
IPrintName<Gamer> gamerPrinter = anotherPrinter;

Console.WriteLine(gamerPrinter.PrintName(new Gamer("Semen")));
```

Вывод:
Ivan
Semen

Исходный код (https://replit.com/@olgamaksimenkova/ContraVarInterfaceDemo)

ПРИМЕР. КОВАРИАНТНЫЙ И КОНТРАВАРИАНТНЫЙ ИНТЕРФЕЙС

```
interface INamedObjectProcessing <in T, out TResult> {
    string PrintName(T obj);
    TResult CreateNamedObject(string name);
}
```

```
class NamedUsersProcessing : INamedObjectProcessing<User, Gamer> {
    public string PrintName(User user) => user.Name;
    public Gamer CreateNamedObject(string name) => new Gamer(name);
}
```

```
INamedObjectProcessing<Gamer, User> users =
                                new NamedUsersProcessing();
User user = users.CreateNamedObject("Ivan the User");
Console.WriteLine(user.Name);
users.PrintName(new Gamer("Semen the Gamer"));

INamedObjectProcessing<Gamer,Gamer> gamers =
                            new NamedUsersProcessing();
Gamer gamer = gamers.CreateNamedObject("Sasha the Gamer");
Console.WriteLine(gamers.PrintName(gamer));

INamedObjectProcessing<User, User> onlyUsers =
                            new NamedUsersProcessing();
User oneUser = onlyUsers.CreateNamedObject("Vitya the User");
Console.WriteLine(onlyUsers.PrintName(oneUser));
```

Исходный код (https://replit.com/@olgamaksimenkova/ConraVarAndConVaInterface)

СРАВНЕНИЕ КОВАРИАНТНОСТИ И КОНТРАВАРИАНТНОСТИ

```
class Base { }
class Derived : Base { }
```

```
// Совместимость операции присваивания:
string str = "test";
object obj = str; // Объект типа наследника присваивается по ссылке базового типа.

// Ковариантность:    public interface IEnumerable<out T> : System.Collections.IEnumerable
IEnumerable<Derived> strings = new List<Derived>();
IEnumerable<Base> objects = strings; // Сохраняется совместимость типов операции =.

// Контравариантность:  public delegate void Action<in T>(T obj);
static void SetObject(Base o) { }
Action<Base> actObject = SetObject;
Action<Derived> actString = actObject; // обратная операции = //совместимость типов
```

КОВАРИАНТНОСТЬ И КОНТРАВАРИАНТНОСТЬ ДЕЛЕГАТ-ТИПОВ

- При объявлении делегатов:

При отсутствии in/out – инвариантность (точное совпадение типа)!

```
public delegate void Action<in T>(T obj);

public delegate TResult Func<out TResult>();

public delegate TResult Func<in T1, out TResult>(T1 arg1);

public delegate TOutput Converter<in TInput, out TOutput>(TInput input);
```

контравариантность

ковариантность

контравариантность

ковариантность

контравариантность

ковариантность

ПРИМЕНЕНИЕ КОВАРИАНТНОСТИ И КОНТРВАРИАНТНОСТИ

- При объявлении делегат-типов:

```
public delegate void Action<in T>(T obj);
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg1);
```

- При использовании делегатов:

```
static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return ""; }
static void SetString(string str) { }
```

```
static void Test() {
    // Ковариантность (тип возвращаемого значения):
    Func<object> del = GetString; // string вместо object.

    // Контравариантность (тип входного параметра):
    Action<string> del2 = SetObject; // object вместо string.
}
```

КОВАРИАНТНОСТЬ И КОНТРВАРИАНТНОСТЬ

Ковариантность параметров типа доступна только для **обобщённых интерфейсов** и **делегатов** (generic delegate)

Обобщённые интерфейсы и делегаты могут иметь и ковариантные и контравариантные параметры типа одновременно:

```
delegate TResult Func<in T1, out TResult>(T1 arg1);
```

- **Вариантность применяется только к ссылочным типам**
  - если указать тип значения для аргумента вариативного типа, то этот параметр типа становится в результате инвариантным
- **Вариативность не применима к многоадресным делегатам** (multicast delegate).
  - для заданных двух делегатов типов `Action<Derived>` и `Action<Base>` нельзя объединять первый делегат со вторым, несмотря на то что результат будет безопасным типом. Вариантность позволяет присвоить второй делегат переменной типа `Action<Derived>`, но делегаты можно объединять, только если их типы точно совпадают

Ковариантность (**out**) – для параметров типа, которые используются только как выходные параметры, контравариантность (**in**) – для --"-- входных параметров

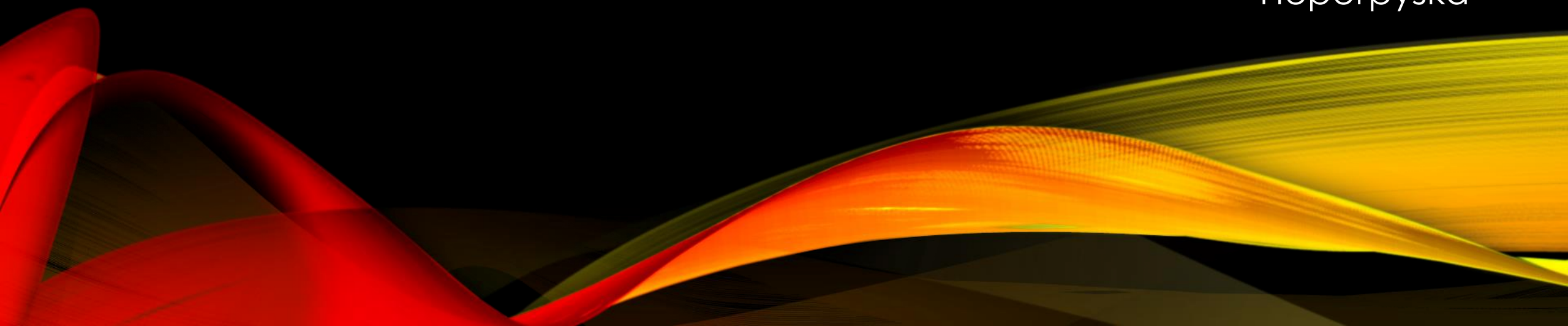
ПРИМЕР КОВАРИАНТНОСТИ И КОНТРАВАРИАНТНОСТИ

```
public class Type1 { }
public class Type2 : Type1 { }
public class Type3 : Type2 { }

public class Program  {
    public static Type3 MyMethod(Type1 t) {
        return t as Type3 ?? new Type3();
    }

    static void Main()  {
        Func<Type1, Type3> f0 = MyMethod;
        Func<Type2, Type2> f1 = f0;
        // Ковариантный тип возвр. значения и контравариантный тип параметра.
        Func<Type3, Type1> f2 = f1;
        Type1 t1 = f2(new Type3());
    }
}
```

МЕТОДЫ ПРИ НАСЛЕДОВАНИИ ОБОБЩЁННЫХ ТИПОВ

Виртуальность

Перегрузка



Далее на слайдах приведены коды, нуждающиеся в переосмыслении и доработке

Некоторые решения являются архитектурно ошибочными и приведены для примеров и исправления

ПРИМЕР. ПЕРЕОПРЕДЕЛЕНИЕ МЕТОДОВ В ОБОБЩЁННЫХ ТИПАХ (1)

```
public class Number<T> where T: struct {
    protected T _value;
    // Свойство виртуально, на случай переопределения в наследнике.
    public virtual T Value { get => _value;
        set => _value = value;
    }
    // В этом типе нет ограничений на значение.
    // Но метод виртуален, чтобы могли работать наследники.
    public virtual bool IsAcceptable(T newValue) => true;
    public Number() { }
    public Number(T value) => Value = value;

    public override string ToString() => _value.ToString();
}
```

Исходный код https://replit.com/@olgamaksimenkova/GenericLimintedNumbers

ПРИМЕР. ПЕРЕОПРЕДЕЛЕНИЕ МЕТОДОВ В ОБОБЩЁННЫХ ТИПАХ (2)

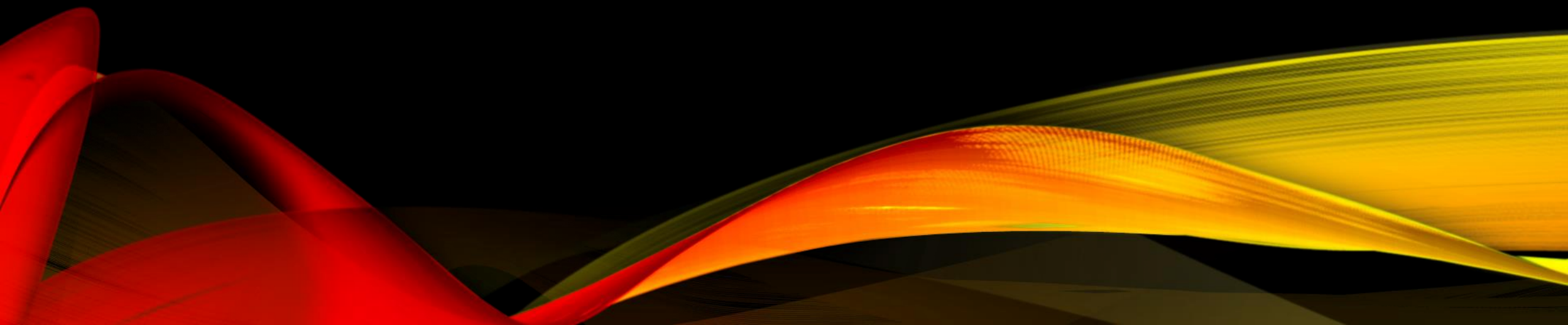
```
public class LimitedNumber<T> : Number<T> where T: struct {
   // Критерий для ограничения значения числа.
    public Predicate<T> Criterion { set; private get; }
    protected LimitedNumber() { }
    // Конструктор с критерием и значением.
    public LimitedNumber(Predicate<T> criterion, T initValue)  {
        // Если критерий не передан -- сразу исключение.
        if (criterion == null) throw new ArgumentNullException();
        Criterion = criterion;
        // Значения, не отвечающие критерию не должны попасть в объект.
        if (criterion(initValue))   _value = initValue;
        else throw new ArgumentOutOfRangeException();
    }
    // В интерфейс отправляем метод проверки значения на соответствие критерию.
    public override bool IsAcceptable(T newValue) {
        if(Criterion(newValue)) return true;
        return false;
    }
}
```

ПРИМЕР. ПЕРЕОПРЕДЕЛЕНИЕ МЕТОДОВ В ОБОБЩЁННЫХ ТИПАХ (3)

```
Number<int> number = new(5);
LimitedNumber<int> number2 = new(x => x > 0, 6);
LimitedNumber<int> number3 = new(x => x < 0, -6);

int[] testData = { -9, 1, 2, 3, 16, -22 };
foreach (int cur in testData)    {
    if (number2.IsAcceptable(cur))
       Console.WriteLine($"{cur} is ok to {number2}");
    if (number3.IsAcceptable(cur))
       Console.WriteLine($"{cur} is ok to {number3}");
}
Console.WriteLine();
Number<int>[] arrayOfNumbers = {number, number2, number3 };
foreach(var i in arrayOfNumbers)  {
    foreach (int cur in testData)   {
        if (i.IsAcceptable(cur))
             Console.WriteLine($"{cur} is ok to {i}");
    }
}
```

ПРИМЕР ПРОБЛЕМНОГО ОБОБЩЕНИЯ

«Писатель в консоль»



ПРИМЕР. ОБОБЩЁННЫЙ ПИСАТЕЛЬ В КОНСОЛЬ (1)

```
public class CustomConsoleWriter<T> where T : IDailyInfo, IImportantInfo {
    ConsoleColor dailyInfo = ConsoleColor.Green;
    ConsoleColor importantInfo = ConsoleColor.Red;

    public void ConsoleWriteDailyInfo(T objectToWrite) {
        Console.ForegroundColor = dailyInfo;
        Console.WriteLine(objectToWrite.GetDailyInfo());
        Console.ResetColor();
    }
    public void ConsoleWriteImportantInfo(T objectToWrite) {
        Console.ForegroundColor = importantInfo;
        Console.WriteLine(objectToWrite.GetImportantInfo());
        Console.ResetColor();
    }
}
```

ПРИМЕР. ОБОБЩЁННЫЙ ПИСАТЕЛЬ В КОНСОЛЬ (2)

```
public interface IDailyInfo
{
    string GetDailyInfo();
}
```

```
public interface IImportantInfo
{
    string GetImportantInfo();
}
```

```
public class User : IImportantInfo, IDailyInfo {
    public string Name { get; set; }
    public string Email { get; set; }
    public string Password { get; set; }
    public User() { }
    public User(string name, string email, string password) =>
                    (Name, Email, Password) = (name, email, password);
    public string GetDailyInfo() => $"Name={Name};Email={Email}";
    public string GetImportantInfo() => $"PWD={Password}";

}
```

ПРИМЕР. ОБОБЩЁННЫЙ ПИСАТЕЛЬ В КОНСОЛЬ (3)

```
public class Gamer : User
{
    public string FavoriteGame { get; set; }
    public Gamer() { }
    public Gamer(string name, string email, string password, string favoriteGame) :
        base(name, email, password) =>
       FavoriteGame = favoriteGame;
    public new string GetImportantInfo() =>
           base.GetImportantInfo() + $";FG={FavoriteGame}";
}
```

ПРИМЕР. ОБОБЩЁННЫЙ ПИСАТЕЛЬ В КОНСОЛЬ (4)

```
CustomConsoleWriter<User> usersWriter = new CustomConsoleWriter<User>();

User testUser = new User("Joe Doe", "joe@hotmail.com", "12345");
Gamer testGamer = new Gamer("Mary Smith", "mary@gmail.com", "2345", "Genshin Impact");

usersWriter.ConsoleWriteDailyInfo(testUser);
usersWriter.ConsoleWriteImportantInfo(testUser);

usersWriter.ConsoleWriteDailyInfo(testGamer);
usersWriter.ConsoleWriteImportantInfo(testGamer);

CustomConsoleWriter<Gamer> gamersWriter = new CustomConsoleWriter<Gamer>();

Console.WriteLine(testGamer.GetImportantInfo());

gamersWriter.ConsoleWriteDailyInfo(testGamer);
gamersWriter.ConsoleWriteImportantInfo(testGamer);
```

ПРОБЛЕМЫ В РЕШЕНИИ ПРИМЕРА «ПИСАТЕЛЬ В КОНСОЛЬ»

- Реализация обобщенного типа CustomConsoleWriter не зависит от обобщённого параметра
- Код методов ConsoleWriteDailyInfo и ConsoleWriteImportantInfo идентичен с точностью до вызова
- При использовании объекта-наследника в качестве типизирующего параметра наблюдается некорректное поведение: на экран выводится не вся информация об объекте
- Это следствие нарушения принципа инверсии зависимостей SOLID

ИСПОЛЬЗОВАННАЯ И РЕКОМЕНДОВАННАЯ ЛИТЕРАТУРА

- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-interfaces
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-delegates

  https://docs.microsoft.com/en-us/dotnet/api/system.action-1
- https://docs.microsoft.com/en-us/dotnet/api/system.func-1
- https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/concepts/covariance-contravariance/
- https://www.bestprog.net/ru/2021/06/30/c-generics-basic-concepts-generic-classes-and-structures-ru/
- https://professorweb.ru/my/csharp/charp_theory/level11/11_14.phphttps://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-interfaces
- https://www.bestprog.net/ru/2021/06/30/c-generics-basic-concepts-generic-classes-and-structures-ru/