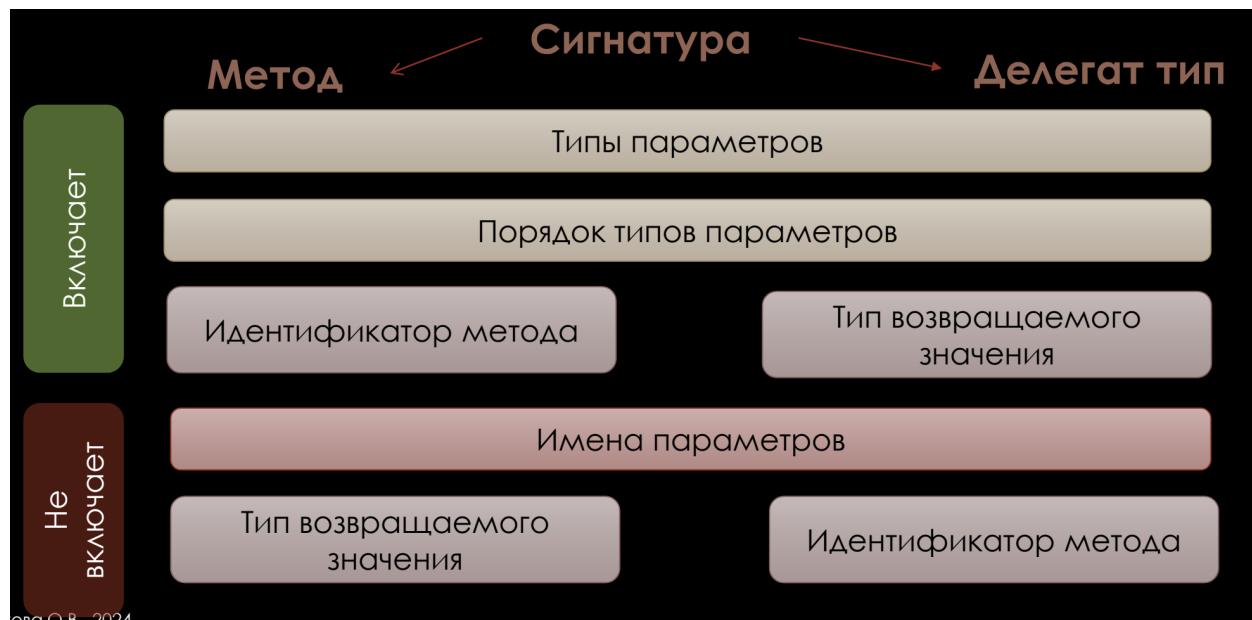


Lecture 1-2. Delegate

Делегат-тип является ссылочным типом данных.

- Делегат-тип является наследником класса Delegate
- Пользовательский тип не может быть напрямую унаследован из класса Delegate или MulticastDelegate
- Класс Delegate не является делегатом-типов
- Делегат-тип опечатан (sealed)

```
<модификатор доступа>delegate<тип_возвр_значения> Идентиф (<список параметров>)
```



Все делегаты-типы имеют следующие функциональные члены:

- `public Delegate[] GetInvocationList()` – метод, возвращающий массив объектов Delegate, содержащих информацию о каждом из вызываемых методов
- `public MethodInfo Method { get; }` – информация о последнем в списке вызовов методе

- `public object? Target { get; }` – объект, связанный с последним методом в списке вызовов или null, если метод статический

ВЫВОД:

```
value;  
using System;  
// Основная программа – класс Program.cs.  
class Program
```

```
{  
    static void Main()  
    {  
        DigitSplitter splitter = new(12345);  
        Row delRow = splitter.GetDigitsArray();  
        Console.WriteLine($"The method called is {delRow.Method}");  
        Console.WriteLine($"The target object is {delRow.Target}");  
    }  
}
```

```
// Объединение списков вызовов.  
IntFunc combination = (IntFunc)Delegate.Combine(func, sFunc);
```

```
Console.WriteLine("Separately via InvocationList:");  
foreach(Delegate d in func.GetInvocationList())  
{  
    Console.WriteLine(((IntFunc)d).Invoke(5));  
}
```

РЕКОМЕНДАЦИИ ПО ИСПОЛЬЗОВАНИЮ

Делегат

- Используется событийный шаблон проектирования
- Необходимо инкапсулировать статический метод
- Вызывающему коду не нужно обращаться к другим свойствам, методам или интерфейсам объекта, реализующего метод
- Требуется обеспечить легко реализуемую композицию
- Классу (в перспективе) может потребоваться более одной реализации метода

интерфейс

- Есть набор связанных методов, которые должна быть возможность вызывать
- Классу требуется только одна реализация метода
- Класс, использующий интерфейс, захочет привести этот интерфейс к другим типам интерфейсов или классов
- Реализуемый метод связан с типом или особенностями структуры класса: например, методы сравнения

Компараторы в `Sort<T>` предоставляют обе возможности обратного вызова:

Интерфейс

<code>Sort<T>(T[], IComparer<T>)</code>	Sorts the elements in an <code>Array</code> using the specified <code>IComparer<T></code> generic interface.
<code>Sort<T>(T[], Comparison<T>)</code>	Sorts the elements in an <code>Array</code> using the specified <code>Comparison<T></code> .

Делегат

```
public static void Sort<T> (T[] array, System.Collections.Generic.IComparer<T>? comparer);  
public delegate int Comparison<in T>(T x, T y);
```

Lecture 3: Anonymous function. Lambda

АНОНИМНАЯ ФУНКЦИЯ

```
System.Func<int, int, int> sumFunc = delegate (int x, int y)
{
    return x + y;
};
```

Для нестатических анонимных методов внешние переменные захватываются автоматически и доступны без дополнительных ограничений.

Различия анонимных методов и лямбда-выражений:

- Анонимные методы позволяют полностью опустить список параметров.
- Лямбда-выражения позволяют опускать и выводить типы параметров.
- Тело лямбда-выражения может быть выражением или блоком, тело анонимного метода – только блоком.
- Только лямбда-выражения имеют преобразования к совместимым типам дерева выражений.

ЛЯМБДА

```
Func<int, int> lambda4 = x => x + 1; // Тип входа и выхода.
Action lambda5 = () => Console.WriteLine("lambda"); // Без параметров
// С неявным указанием параметров:
Action<int, int> lambda6 = (x, y) => Console.WriteLine(x * y);
```

Параметры:

```
Func<int, int> lambda1 = (int x) => { return x + 1; };
Func<int, int> lambda2 = (x) => { return x + 1; }; // Без явного
Func<int, int> lambda3 = x => { return x + 1; }; // Без скобок
```

Типы всех параметров указываются либо полностью явно, либо полностью неявно.

Если у лямбда-выражения только один входной параметр, круглые скобки можно опустить.

Для лямбда-выражений без параметров круглые скобки обязательны:

```
Action lambda = () => Console.WriteLine("lambda");
```

Символ `_` (подчёркивание) можно использовать для пропуска двух и более параметров
(для одиночного `_` считается именем локальной переменной в целях обратной совместимости):

```
Func<int, int, int> lambda = (_, _) => 42;
```

Начиная с C# 10.0, для лямбда-выражений допускается явное указание типа возвращаемого значения перед списком параметров:

```
ByRefDel lambda7 = ref int (ref int val) => ref val;
delegate ref int ByRefDel(ref int x);
```

Пример работы с кортежами в лямбдах:

```
Func<(int, int), (int, int)> mod10 = ns => (ns.Item1 % 10, ns.I
```

РЕАЛИЗАЦИЯ ЧЛЕНОВ КЛАССОВ ЧЕРЕЗ ЛЯМБДА-ВЫРАЖЕНИЯ

Синтаксис выражений допустим для:

- методов
- свойства только для чтения
- свойств
- конструкторов
- деструкторов
- индексаторов

▼ Examples of use Lambda

ПРИМЕР ЛЯМБДА СИНТАКСИСА КОНСТРУКТОРА И МЕТОДА

```
public class Person
{
    public Person(string firstName) => fname = firstName;
    public Person(string firstName, string lastName)
    {
        fname = firstName;
        lname = lastName;
    }

    private string fname;
    private string lname;

    public override string ToString() => $"{fname} {lname}".Trim();
    public void DisplayName() => Console.WriteLine(ToString());
}
```

Однострочная запись конструктора

Методы, допускающие однострочную запись

ПРИМЕР ЛЯМБДА СИНТАКСИС СВОЙСТВА

```
public class Location
{
    private string locationName;

    public Location(string name)
    {
        locationName = name;
    }

    public string Name => locationName;
}
```

Свойство, дающее только возможность чтения

```
public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

Геттер и сеттер, записанные в лямбда-синтаксисе

ПРИМЕР ЛЯМБДА СИНТАКСИСА ИНДЕКСАТОРА

```
using System;
using System.Collections.Generic;

public class Sports
{
    private string[] types = { "Baseball", "Basketball", "Football",
                             "Hockey", "Soccer", "Tennis",
                             "Volleyball" };

    public string this[int i]
    {
        get => types[i];
        set => types[i] = value;
    }
}
```

Геттер и сеттер индексатора, записанные в лямбда-синтаксисе

Пример взят из официального руководства Microsoft

СТАТИЧЕСКИЕ АНОНИМНЫЕ ФУНКЦИИ

- Нестатические и не const переменные и поля не захватываются
- Статические члены и константы по прежнему доступны для захвата
- Нет доступа к ссылкам this/base, даже если такие имелись в охватывающей области видимости

АНОНИМНЫЕ МЕТОДЫ КАК ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ

Анонимные методы могут выступать в качестве возвращаемых значений в функциях, возвращающих экземпляры делегат-типов или **Expression**:

```
static Func<double, double, double, double> GetFunction(int a, int b, int c)
{
    return (a, b, c) => a * a * b + c;
```

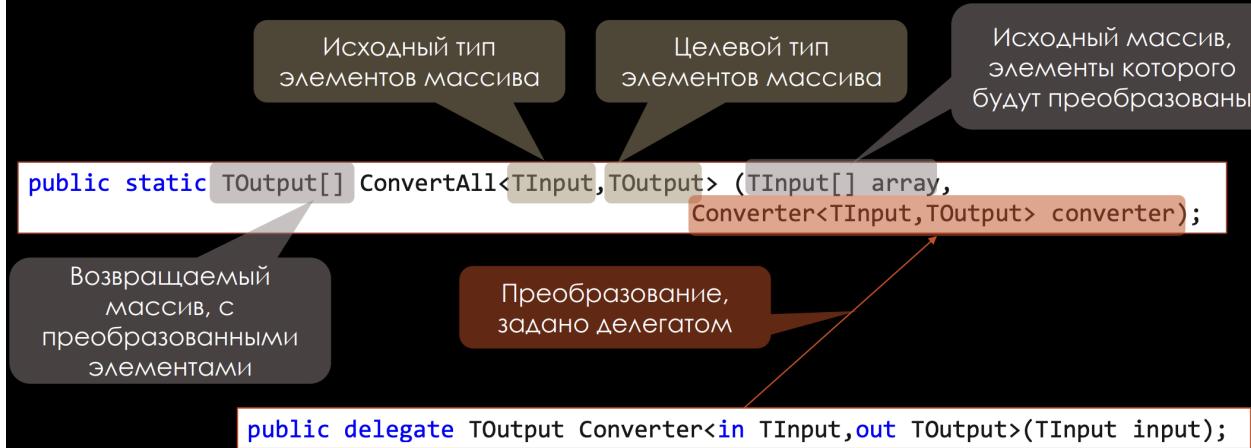
Уже упоминавшийся тип для дерева выражений

```
static Expression<Func<string, int>> GetExpression(string line)
{
    return (line) => int.Parse(line);
```

При использовании делегат-типов с модификатором `params` при объявлении анонимных методов ключевое слово `params` опускается

```
int[] ints = { 2, 6, 4, 5, 17, 8, 10, 3, 1, 14 };
// Конвертация всех чисел в double:
double[] doubles = Array.ConvertAll(ints,
    value => value >= 10 ? value + 0.1 : value * 0.1);
// Вывод всех чисел в полученном массиве.
Array.ForEach(doubles, value => Console.WriteLine($"{value:F1} "));
```

ПОДРОБНО О CONVERTALL



Сами по себе анонимные методы не имеют типа вне контекста. Они преобразуются компилятором либо к делегат-типов, либо к Expression.

Начиная с C# 10.0 появляется такое понятие, как естественные типы анонимных

методов. Компилятор может выводить тип по следующим правилам:

- Action/Func выводится при наличии подходящей сигнатуры;
- Генерируется автоматически (например, при наличии ref);
- Выведение типа для групп методов возможно только при отсутствии перегрузок/методов расширений;
- Вывод типа работает по ссылкам базового типа (Object/Delegate или Expression/LambdaExpression) по описанным выше правилам.

Lecture 4. Stream

Методы:

Имя	Назначение
Close()	Закрывает поток и освобождает ассоциированные с ним ресурсы.
Flush()	Освобождает буфер обмена, предварительно передав из него данные в информационный источник.
Read() ReadByte()	Читает байт или последовательность байтов из потока и перемещает текущую позицию на прочитанное число байтов
Seek()	Устанавливает текущую позицию в потоке
SetLength()	Устанавливает длину потока
Write() WriteByte()	Записывает в поток байт или последовательность байтов, перемещает указатель на количество записанных байтов

Свойства:

Имя	Назначение
CanRead CanSeek CanWrite	Проверяют для потока возможность чтения, записи и возможность изменения текущей позиции
Length	Длина потока в байтах
Position	Текущая позиция в байтах

Абстрактный класс, обеспечивающий чтение и запись байтов.

- Все классы, являющиеся потоками, являются наследниками Stream
- Служат для обеспечения общего способа просмотра источников данных
- Поток «экранирует» программиста от специфических особенностей операционной системы и физических устройств

```
public class FileStream : System.IO.Stream
```

Для создания экземпляра класса FileStream необходимо явно или неявно указать:

- Файл, с которым будет ассоциирован поток
- Способ доступа к файлу (FileAccess)
- Режим открытия файла (FileMode)
- Допуск к совместному использованию (FileShare)

```
FileStream (string имя_файла, FileMode режим, FileAccess доступ)
```

Методы File и FileInfo

Имя	Назначение
Create()	Создает новый файл и поток, ассоциированный с созданным файлом. Либо создает поток, ассоциированный с существующим файлом.
Open()	Открывает файл и возвращает ссылку на поток, ассоциированный с ним. С помощью параметра может быть указано назначение потока (чтение, запись и.т.д)
OpenRead()	Открывает файл и создает поток, предназначенный только для чтения.
OpenWrite()	Открывает файл и создает поток, предназначенный только для записи.

Исключения:

Имя	Причина генерации
IOException	Файл невозможно открыть из-за ошибки ввода-вывода.
FileNotFoundException	Файл невозможно открыть по причине его отсутствия.
ArgumentNullException	Имя файла представляет собой null- значение.
ArgumentException	Параметр Mode некорректен.
SecurityException	Пользователь не обладает правами доступа к файлу.
DirectoryNotFoundException	Имя каталога задано некорректно.

Lecture 5. Events

- Издатель (publisher) – тип, экземпляры которого генерируют события (raising an event). Позволяет другим объектам подписываться на рассылку событий и отписываться от неё
- Подписчик (subscriber) – тип, экземпляры которого способны подписываться на рассылку событий и обрабатывать связанные с ней данные
- Обработчик события (event handler) – метод обработки события. Как правило, используется объектом-подписчиком
- Данные события (event arguments) – информация, рассылаемая издателем всем подписчикам при возникновении события

Издатель:

- Позволяет подписываться и отписываться на рассылку событий
- Принимает циклически консольные команды до ввода команды «exit» (завершение работы). Если принятая команда «notify», все подписчики получают текущее время в качестве информации

```
public class Publisher
{
    public Action<DateTime> notificationAppeared;

    public void HandleCommands()
    {
        string command;
        do
        {
            command = Console.ReadLine();
            if (command == "notify" && notificationAppeared != null)
            {
                notificationAppeared(DateTime.Now);
            }
        } while (command != "exit");
    }
}
```

```
public class Publisher
{
    public event Action<DateTime> notificationAppeared;

    public void HandleCommands()
    {
        string command;
        do
        {
            command = Console.ReadLine();
            if (command == "notify" && notificationAppeared != null)
            {
                notificationAppeared(DateTime.Now);
            }
        } while (command != "exit");
    }
}
```

Поле делегат типа, что добавляет нее

Подписчик:

- Имеет собственное имя (строку)
- Подписывается на рассылку событий – имеет метод, соответствующий формату передаваемого сообщения

- В момент получения сообщения от издателя обрабатывает его – выводит текст со своим именем и содержащиеся в сообщении данные

```
public class Subscriber
{
    public string Name { get; init; }

    public Subscriber(string name) => Name = name;

    public void NotificationEventHandler(DateTime eventArgs)
        => Console.WriteLine($"{Name}: received notification " +
            $"on {eventArgs.ToShortDateString()}");
}
```

2) Метод сигнатура для обработки

```
public class Subscriber
{
    public string Name { get; init; }

    public Subscriber(string name) => Name = name;

    public void NotificationEventHandler(DateTime eventArgs)
        => Console.WriteLine($"{Name}: received notification " +
            $"on {eventArgs.ToShortDateString()}");
}
```

Метод-обработчик сигнатурой, т.к. основывается на типе

Lecture 6.

StandartEventsPattern

```
[модификатор доступа] void HandlerId(object source, EventArgs e)
{
    // тело обработчика
}
```

1. Добавить подписчика с помощью += к событию из п. 3
2. Объявить в типе-подписчике метод обработчик события, соответствующий сигнатуре делегата из п. 2
3. Добавить в тип-издатель код, ответственный за генерацию события и вызов метода из п. 4
4. Добавить в тип-издатель метод генерации события (как правило, с именем On<ИмяСобытия>) и организовать передачу аргументов событию в нём.
5. Объявить событие делегат-типа из п. 2 внутри типа-издателя
6. Выбрать делегат-тип для события:
 - а. Объявить делегат-тип с наследником EventArgs из п. 1 в качестве второго параметра
 - б. Использовать делегат-тип EventHandler<T>
7. Объявить класс для передачи данных о событии – наследник EventArgs

Lecture 7. Reflection

ЗАМЕНА ДАННЫХ

```
//User user = new User("Ivan");
//Type userType = user.GetType();

#region createObject
// Getting type's metadata from the type
Type userType = typeof(User);
// Getting constructor from metadata of type
ConstructorInfo? ci = userType?.GetConstructor(new Type[] { typeof(string) });
// Invoke constructor
object? user = ci?.Invoke(new object[] { "Petr" });
#endregion

var userFields = userType?.GetFields(BindingFlags.Instance | BindingFlags.NonPublic);
foreach (FieldInfo fi in userFields)
{
    if (fi.Name == "name")
    {
        var nameVal = fi.GetValue(user);
        Console.WriteLine($"Getted value:: {nameVal}");

        fi.SetValue(user, "Semen");
        nameVal = fi.GetValue(user);
        Console.WriteLine($"Getted value:: {nameVal}");
    }
}
```

При использовании этого фрагмента региона ниже нужно закомментировать или удалить

В этом регионе код создаёт объект типа, конструктор которого получен через рефлексию

- Атрибут Serializable не наследуется
- Атрибут NonSerialized наследуется

ЯВНОЕ УКАЗАНИЕ ЦЕЛИ АТРИБУТА:

Глобальные:

event field
method param
property return
type typevar
assembly module

```
public sealed class MyAttributeAttribute : System.Attribute
```

Типичные члены класса-атрибута:

- поля
- свойства
- конструкторы

Допускается:

- перегрузка конструкторов

- пустой конструктор добавляется автоматически, если не указано ни одного конструктора

Аргументы конструкторов должны быть известны в момент компиляции (константы)!

ТОЛЬКО ДЛЯ МЕТОДОВ



```
[ AttributeUsage( AttributeTargets.Method ) ]
public sealed class MyAttributeAttribute : System.Attribute
```

Имя	Значение	Значение по умолчанию
ValidOn	Хранит список типов целей к которым может применяться атрибут. Первый параметр конструктора должен быть значением перечислимого типа AttributeTargets.	
Inherited	Булево значение, указывающее может ли атрибут наследоваться производными классами декорированного типа.	true
AllowMultiple	Булево значение, указывающее можно ли к цели применять одновременно несколько экземпляров атрибута текущего типа.	false

```
[ AttributeUsage( AttributeTargets.Method | AttributeTargets.Constructor ) ]
public sealed class MyAttributeAttribute : System.Attribute
{ ...
```

Пример использования
AttributeUsage

Члены перечисления AttributeTargets:

All	Assembly	Class	Constructor
Delegate	Enum	Event	Field
GenericParameter	Interface	Method	Module
Parameter	Property	ReturnValue	Struct

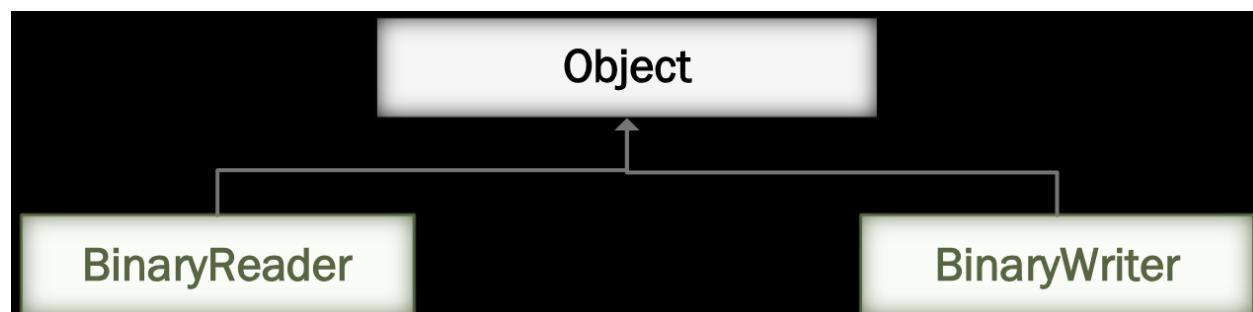
```
[ AttributeUsage( AttributeTargets.Class, // Required, positional
    Inherited = true, // Optional, named
    AllowMultiple = false ) ] // Optional, named
public sealed class MyAttributeAttribute : System.Attribute
```

Lecture 8. Serialization

- **Сериализация** – процесс преобразования структуры данных в последовательность байтов (или XML-узлов / JSON / etc.).
- **Десериализация** – восстановление структуры данных из последовательности байтов (или XML-узлов / JSON / etc.).

МЕХАНИЗМЫ СЕРИАЛИЗАЦИИ .NET

- контракты данных
- двоичная
- SOAP-сериализация
- XML-сериализация
- IXmlSerializable
- JSON-сериализация



```
BinaryWriter (Stream поток)
BinaryWriter (Stream поток, Encoding кодировка)
BinaryWriter (Stream поток, Encoding кодировка, Bool открыт)

BinaryReader (Stream поток)
BinaryReader (Stream поток, Encoding кодировка)
BinaryReader (Stream поток, Encoding кодировка, Bool открыт)
```

Средства записи:

- `public virtual void Write(byte[] буфер)`
- `public virtual void Write(byte[] буфер, int индекс-начало, int счетчик)`
- `public virtual void Write(char[] буфер, int индекс-начало, int счетчик)`
- `protected void Write7BitEncodedInt(int value)`
- `public virtual void Write(. . .)`

- `public virtual void Close()` – Закрывает бинарный и базовый потоки
- `public void Dispose()` – Освобождает ресурсы
- `public virtual void Flush()` – Очищает буфера
- `public virtual long Seek(int смещение, SeekOrigin точка_отсчета)` – устанавливает позицию записи

- `Close()`
- `PeekChar()` – “подсмотреть” символ (возвращает следующий доступный для чтения символ, не перемещая позицию байта или символа вперед).
- `Dispose()`

```
int Read() – чтение отдельного символа
void Read(byte[] буфер, int индекс-начало, int счётчик)
void Read(char[] буфер, int индекс-начало, int счётчик)

int Read7BitEncodedInt() – читает упакованное целое число
Если целое число будет помещаться в семь бит, целое число займет
только один байт. (ожидается, что целое число записали через
BinaryWriter.Write7BitEncodedInt())
```

Пример:

```

struct Discovery {
    public string Name { get; set; }
    public int Date { get; set; }
}

static class BinFileOp {
    public static void WriteData(string path)
    { ... }
    public static void ReadData(string path)
    { ... }
}

BinFileOp.WriteData(@"..\\..\\..\\data.bin");
Console.OutputEncoding = Encoding.UTF8;
Console.WriteLine();
BinFileOp.ReadData(@"..\\..\\..\\data.bin");

```

Данные из структуры «Изобретение» пишем в бинарный файл

Для чтения и записи создадим статический класс с двумя статическими методами

Тестовый код

<https://replit.com/@olgamaksimenkova/BinFi>

Запись и чтение бинарного файла:

```

public static void WriteData(string path)
{
    Discovery[] discoveries = {
        new Discovery { Name = "Радиоприемник", Date = 1895 },
        new Discovery { Name = "Мазер", Date = 1954 },
        new Discovery { Name = "Парашют", Date = 1911 },
        new Discovery { Name = "Гальванопластика", Date = 1840 },
        new Discovery { Name = "Коллайдер", Date = 1960 },
        new Discovery { Name = "Иконоскоп", Date = 1929 }
    };
    using (FileStream fs = File.Create(path))
    using (BinaryWriter bw = new BinaryWriter(fs))
        foreach (Discovery dis in discoveries)
        {
            bw.Write(dis.Name);
            bw.Write(dis.Date);
        }
}

```

```

public static void ReadData(string path)
{
    using (FileStream fs = new FileStream(path, FileMode.Open))
    using (BinaryReader br = new BinaryReader(fs))
    {
        while (true)
        try
        {
            string name = br.ReadString();
            int date = br.ReadInt32();
            Console.WriteLine("Name={0}, Date={1}", name, date);
        }
        catch (EndOfStreamException) { break; }
    }
}

```

```

using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

Point point = new Point(1,1);
BinaryFormatter bf = new BinaryFormatter();
using (FileStream fs = new FileStream("Point.bin", FileMode.Create))
{
    bf.Serialize(fs, point);
}

```

Сериализует / десериализует объект и полный граф связанных объектов в двоичном формате

Результат сериализации в файле Point.bin:

????яяя?????????JBinarySerialization, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null??BinarySerialization.Point???<X>k__BackingField<Y>k__BackingField?????????p?????p?

ДВОИЧНАЯ СЕРИАЛИЗАЦИЯ УНАСЛЕДОВАННЫХ ОБЪЕКТОВ

```

[Serializable]
public class Human {
    public string Name { get; set; }
    public int Age { get; set; }
    public Human() { }
    public Human(string name, int age) { 
        Name = name;
        Age = age;
    }
    public override string ToString() => $"Name={Name}, Age={Age}";
}

```

```

[Serializable]
public struct Department {
    public string Name { get; set; }
    public override string ToString() => $"Department={Name}";
}

```

```

[Serializable]
public class Professor : Human {
    public Department department { get; set; }
    public Professor() { }
    public Professor(string name, int age, Department department) : base(name, age) {
        this.department = department;
    }
    public override string ToString() => base.ToString() + ", " + department;
}

```

Максименкова О.В., 2024

```

Department dep = new Department();
dep.Name = "Software Engineering";
Professor prof = new Professor("Black", 46, dep);
BinaryFormatter binformatter = new BinaryFormatter();
using (FileStream fs = new FileStream("Deps.bin", FileMode.Create))
{
    // Выполнение сериализации:
    binformatter.Serialize(fs, prof);
}

Professor inProf = null;
using (FileStream fs1 = new FileStream("Deps.bin", FileMode.Open))
{
    inProf = (Professor)binformatter.Deserialize(fs1);
    Console.WriteLine("Restored object::");
    Console.WriteLine(inProf);
}

```

Максименкова О.В., 2024

Выход:

Restored object::
Name=Black, Age=46, Department=Software Engineering

Двоичная сериализация:

- Атрибуты
- Реализация интерфейса ISerializable
- [Serializable] – в объявлении типа
- [NonSerialized] – в объявлении игнорируемых полей

Атрибуты для методов:

- [OnSerializing] – перед сериализацией
- [OnSerialized] – после сериализации
- [OnDeserializing] – перед десериализацией
- [OnDeserialized] – после десериализации

Атрибут Serializable не наследуется

Атрибут NonSerialized наследуется

JSON. КОГДА ИСПОЛЬЗОВАТЬ?

- Передача данных на сервер (от сервера), особенно в web-приложениях
- Выполнение асинхронных AJAX-вызовов без перезагрузки страниц в webприложениях
- Работа с базами данных (особенно документно-ориентированными)
- Сохранение данных в локальном хранилище (сериализация)

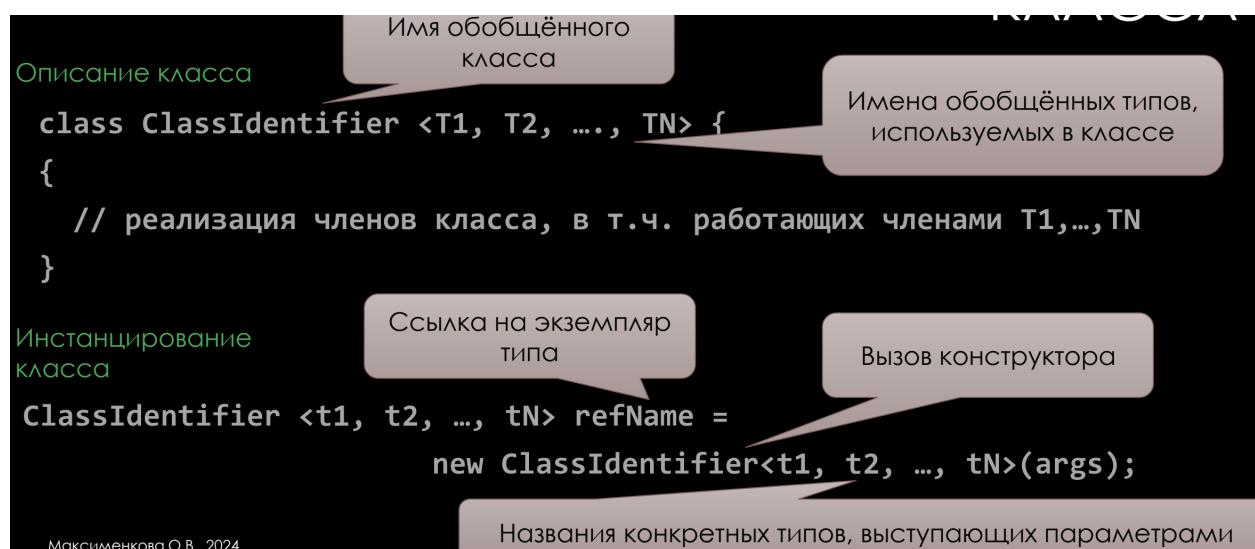
АТРИБУТЫ КОНТРАКТОВ ДАННЫХ

- [DataContract] – для сериализуемых типов;
- [DataMember] – для членов сериализуемых типов;
- [EnumMember] – для членов перечислений;

Lecture 9. Generics

ПРИМЕНИМОСТЬ ОБОБЩЕНИЙ

- классы
- структуры
- интерфейсы
- методы
- делегаты



Открыто-сконструированный тип	Закрыто-сконструированный тип
<pre>class ClassName<T1, T2> { // члены класса }</pre>	<code>ClassName<short, int></code>

• На этапе выполнения существуют только обобщённые типы/методы, в которых выполнена подстановка параметров; такие типы называют **закрытыми** или **закрытыми сконструированными**

• Объявленные обобщённые типы без подстановки аргументов типов называют **открытыми**

Ограничение	Описание Допустимых Типов-Аргументов
class?	Любой ссылочный тип (класс, интерфейс, делегат, массив или запись).
struct	Любой не-nullable тип значения. Автоматически включает в себя ограничение new() и не может сочетаться с ним явно. Несовместимо с ограничением unmanaged.
<ClassName>?	Тип-аргумент является типом ClassName или его наследниками.
<InterfaceName>?	Тип-аргумент обязан реализовывать интерфейс InterfaceName. На один параметр типа могут накладываться несколько ограничений на интерфейсы.
new()	Тип-аргумент должен иметь открытый конструктор без параметров. Несовместимо с ограничениями struct и unmanaged.
notnull (C# 8.0)	Тип-аргумент должен не допускать значения null.

T : U	Тип-аргумент совпадает с U или является его наследником. При использовании nullable-контекста T должен быть не-nullable типом, если U – не nullable-тип.
unmanaged	Тип-аргумент должен быть « неуправляемым ». Автоматически подразумевает наличие ограничений struct и new(), поэтому несовместимо с ними.
default (C# 9.0)	[Только для обобщённых методов] Используется для устранения неоднозначности, связанной с переопределением методов без ограничений (см. документацию).

Несовместимые друг с другом ограничения параметров типов:

- class, struct, unmanaged, notnull, default

СИНТАКСИС ОГРАНИЧЕНИЙ ПАРАМЕТРОВ ТИПОВ:

Для указания ограничений используется контекстно-ключевое слово **where**:

[Определение типа] where <Параметр типа> : <ограничения>

Для нескольких параметров определяются различные наборы предложений:

```
public class ValueList<T>
    where T : struct, IComparable<T>
{ }
```

```
public class LinkedList<T, U>
    where T : IComparable<T>
    where U : ICloneable
{ }
```

```
public class Dictionary<TKey, TValue>
    where TKey : IEnumerable<TKey>, new()
{ }
```

Смысл ограничения: безошибочное создание объекта типа T внутри обобщённого типа

Ограничение не позволяет использовать в качестве обобщенного типа класс, в котором отсутствует конструктор без параметров или умалчивающий конструктор

```
class DemoGenericClass<T> where T : new()
{
    T internalObject;
    public DemoGenericClass() => internalObject = new T();
}

class Number
{
    public int X { get; set; }
}

class BadNumber
{
    public int X { get; set; }
    public BadNumber(int x) { X = x; }
}

DemoGenericClass<Number> demo = new DemoGenericClass<Number>();
DemoGenericClass<BadNumber> d1 = new DemoGenericClass<BadNumber>();
```

Исходный код (<https://replit.com/@olgamaksimenkova/GenericsWhereNew>)

Умалчивающий конструктор

Нет конструктора без параметров

ОГРАНИЧЕНИЯ НА РОДИТЕЛЬСКИЙ КЛАСС WHERE T : BASENAME

Смысль ограничения: лимитировать множество классов, которые могут использоваться в качестве аргумента обобщённого типа

Ограничение не позволяет использовать в качестве обобщенного типа класс, не входящий в иерархию от BaseName класса. Попытка подсунуть класс из другой иерархии приводит к ошибке компиляции

ОГРАНИЧЕНИЕ НА ИНТЕРФЕЙС WHERE T : INTERFACENAME

Смысль ограничения: гарантировать наличие у типизирующего параметра определённых методов или его принадлежность к конкретному типу интерфейса

Ограничение не позволяет использовать в качестве типизирующих параметров типы, не реализующие все члены соответствующего интерфейса. При отсутствии у типа реализации хотя бы одного интерфейса происходит ошибка компиляции

Обобщённые типы могут наследоваться как от необобщённых типов, так и от других открытых и закрытых сконструированных:

```
// Открытый → закрытый сконструированный:
public class GenericDerived2<T> : GenericBase<int> { }

// Открытый сконструированный → необобщённый:
public class GenericDerived3<T> : NonGenericBase { }
```

```
public class GenericBase<T> { }
public class NonGenericBase { }
```

```
// Открытый → открытый:
public class GenericDerived1<U> : GenericBase<U> { }
```

Необобщённые типы могут наследоваться исключительно от закрытых сконструированных, т. к. в случае с открытыми параметром типа оказывается неопределённым, что недопустимо:

```
public class GenericBase<T> { }
public class NonGenericBase { }

// Необобщённый → закрытый сконструированный:
public class NonGenericDerived : GenericBase<byte> { }
```

Наследование от обобщённых типов с несколькими параметрами требует предоставить все необходимые аргументы обобщённому родителю:

```
public class GenericBase2<T, U> { }

// В обоих случаях все параметры родителя фиксируются:
public class GenericExample1<S, V> : GenericBase2<S, V> { }
public class GenericExample2<S> : GenericBase2<string, S> { }
```

ТРЕБОВАНИЯ К НАСЛЕДНИКУ ОБОБЩЁННОГО КЛАССА

- Обобщённые параметры $\langle T_1, T_2, \dots, T_N \rangle$ базового класса передаются в класс наследник
 - Наследник обобщённого класса – тоже обобщённый класс
 - Класс наследник может дополнять перечень обобщённых параметров базового типа
- Все конструкторы базового класса, получающие параметры обобщённого типа необходимо реализовать в наследнике с прямым вызовом конструкторов базового класса

ПРИМЕРЫ

```
class BaseClass<T> {
    public T BaseClassState { get; set; }
    public BaseClass(T obj) => BaseClassState = obj;
}

class InheritedClass<T> : BaseClass<T> {
    T InheritedClassState { get; set; }

    public InheritedClass(T objBC, T objIC) : base(objBC) => InheritedClassState = objIC;
    public override string ToString() {
        return InheritedClassState.ToString();
    }
}

InheritedClass<int> obj = new InheritedClass<int>(5, 6);
Console.WriteLine($"inherited class state:: {obj}");
```

В базовом классе присутствует конструктор с параметром T

Обязательно переопределяем в наследнике и передаём в родительский конструктор параметр!

```

class BaseClass<T1, T2, T3>{
    public T1 FieldT1 { get; private set; }
    public T2 FieldT2 { get; private set; }
    public T3 FieldT3 { get; private set; }

    public BaseClass(T1 op1, T2 op2, T3 op3) => (FieldT1, FieldT2, FieldT3) = (op1, op2, op3);
}

class InheritedClass<T1, T2, T3, T4> : BaseClass<T1, T2, T3> {
    public T4 FieldT4 { get; private set; }

    public InheritedClass(T1 op1, T2 op2, T3 op3, T4 op4) : base(op1, op2, op3) => FieldT4 = op4;
}

```

Три обобщённых параметра

Наследник добавляет еще один обобщённый параметр

```

InheritedClass<int, double, char, bool> obj =
    new InheritedClass<int, double, char, bool>(15, 3.14, 'H', true);

Console.WriteLine(obj.FieldT1);
Console.WriteLine(obj.FieldT2);
Console.WriteLine(obj.FieldT3);
Console.WriteLine(obj.FieldT4);

```

Исходный код
(<https://replit.com/@olgamaksimkova/GenericInheritAddType>)
Максименкова О.В., 2024

Lecture 10. Generics Inheritance Methods

Вариантность – свойство преобразования типов операторами.

Оператор – любая сущность языка C#, преобразующая тип данных в производные от него.

Преобразование затрагивает один тип: $T[]$, $Action <T>$

Преобразование затрагивает два типа: $T1$ $MethodId(T2)$, $Func<T1, T2>$

- $T > U$ «тип Т больше типа U»

Экземпляр типа Т можно заменить экземпляром типа U

- $T = U$ «тип Т равен типу U»

Замены U на Т и Т на U равновозможны

- $T \neq U$ «тип Т не сравним с типом U»

Замены не возможны, типы не сравнимы

Преобразование, осуществляемое оператором, проявляет свойство:

-

Ковариантности, если оно сохраняет отношение между парой типов после их преобразования в производные, позволяет использовать более конкретный тип.

-

Контравариантности, если оно заменяет отношение «больше» на «меньше», но сохраняет «равны» и «не сравнимы», позволяет использовать более общий тип.

Object > String

Ковариантность

```
Object[] arrObject = new Object[10];
arrObject = new String[2] { "abc", "kbk" };
```

Оператор T[] **ковариантно**
преобразует T, допустима замена
объекта Object[] на String[]

Ковариантность позволяет использовать производный тип с большей глубиной наследования, чем задано изначально, «сохраняет иерархию наследования»

Контравариантность

Action<T> **контравариантно** преобразует
T, допустима замена **объекта**
Action<string> на Action<object>

Object > String



```
public class A {
    public void ConvertString(string str)
    {
        str.ToUpper();
    }
    public void ConvertObject(object str) { }
}
```

```
A a = new A();
Action<string> ex = a.ConvertString;
Action<string> ex1 = a.ConvertObject;
```

Контравариантность позволяет использовать более общий тип (с меньшей глубиной наследования), чем заданный изначально, «обращает иерархию наследования».

Предполагается приведение объекта базового типа к производному типу:

```
class Base { }
class Derived : Base { }
Derived d = Base b;
```

Func<T, TRes> **ковариантно** преобразует
TRes и **контравариантно** преобразует T,
допустима замена **объекта** Func<string,
Object> на Func<Object, string>

```
public class A {
    public string ConvertString(object str) =>
        "aaa";
    public object ConvertObject(string str) =>
        str;
}
```

```
A a = new A();
Func<string, Object> ex = a.ConvertString;
Func<string, Object> ex1 = a.ConvertObject;
//Func<Object, string> ex2 = a.ConvertObject;
так нельзя
```

Свойства вариантиности представляют собой способ переноса наследования типов на производные от них типы — контейнеры, обобщённые типы, делегаты и т. п.

С помощью ковариантности и контравариантности можно неявно преобразовывать ссылки на типы коллекций, типы делегатов и аргументы обобщений.

Ковариантность сохраняет совместимость операции присваивания, а контравариантность заменяет ее на обратную.

```
public struct Department
{
    public string Name { get; set; }
    public override string ToString() => $"Department={Name}";
}

public class Human
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Human() { }
    public Human(string name, int age) => (Name, Age) = (name, age);
    public override string ToString() => $"Name={Name}, Age={Age}";
}

public class Professor : Human
{
    public Department department { get; set; }
    public Professor() { }
    public Professor(string name, int age, Department department) : base(name, age) =>
        this.department = department;
    public override string ToString() => $"{base.ToString()}, {department}";
}
```

ПРИМЕР

ПРИМЕР

```
// Для примера берём делегат-тип с возвратом базового типа
// и ссылкой на массив также базового типа.
public delegate Human DealWithProfs(Human[] profs);

public class Program
{
    // Тип возврата - наследник Human.
    public static Professor GetOne(Human[] humans) => new Professor();
    public static Human GetTwo(Professor[] profs) => new Human();

    public static void Main()
    {
        Human[] hums = { new Human(), new Human(), new Human()};
        Professor[] profs = {new Professor(), new Professor()};
        // Нет точного совпадения с сигнатурой делегата по типу возвращаемого значения.
        // Преобразование возможно за счёт ковариантности (тип возврата шире).
        DealWithProfs dp = GetOne;
        Professor p = GetOne(hums); // С вызовом всё ок!
        Professor p1 = GetOne(profs); // С вызовом всё ок – проявляется ковариантность массива.

        Human[] hums2 = profs; // Явное проявление ковариантности массива.
        // Так нельзя. Тип параметра шире типа параметра делегата.
        // DealWithProfs dp1 = GetTwo;
    }
}
```

Обобщённые методы могут быть объявлены в:

- Классах
- Структурах
- Интерфейсах

Обобщённый метод:

```
class ClassIdentifier {return_type MethodIdentifier<T1, T2, ..., TN> (param_list) {} }

obj.MethodIdentifier(parameters)

obj.MethodIdentifier<T1,...,TN>(parameters)
```

```
public static class ArrayReverseTool {
    static public void ReverseAndPrint<T>(T[] arr) {
        Array.Reverse(arr);
        Array.ForEach(arr, elem => Console.Write(elem + " "));
        Console.WriteLine();
    }
}
```

Вывод:

```
11 9 7 5 3
3 5 7 9 11
third second first
first second third
2.345 7.891 3.567
3.567 7.891 2.345
```

Исходный код
(<https://replit.com/@olgamaksimenkova/DemoGenericMethods4Array>)
Максименкова О. В. 2024

```
int[] intArray = { 3, 5, 7, 9, 11 };
string[] stringArray = { "first", "second", "third" };
double[] doubleArray = { 3.567, 7.891, 2.345 };
ArrayReverseTool.ReverseAndPrint<int>(intArray);
ArrayReverseTool.ReverseAndPrint(intArray);
ArrayReverseTool.ReverseAndPrint<string>(stringArray);
ArrayReverseTool.ReverseAndPrint(stringArray);
ArrayReverseTool.ReverseAndPrint<double>(doubleArray);
ArrayReverseTool.ReverseAndPrint(doubleArray);
```

Обобщённый делегат:

```
delegate ret_type DelegateIdentifier <T1, T2, ..., TN> (arg_list);
DelegateIdentifier<T1, T2,..., TN> refDel<t1, t2, ..., tN>;
var myDel = new Func<int, int, string>(PrintString);
Console.WriteLine($"Total: {myDel(15, 13)}");
static string PrintString(int p1, int p2) => (p1 + p2).ToString();
public delegate TR Func<T1, T2, TR>(T1 p1, T2 p2);
```

Lecture 11-12. Generic Types Interface Variance

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; }
```

Т может быть только типом значений

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

string – является ссылочным типом

Обобщённый интерфейс

```
interface InterfaceIdentifier <T1, T2, ..., TN> { T1,...,TN }

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

```

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

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}");

```

Нельзя:

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

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

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

- 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`.

ИНВАРИАНТНОСТЬ

Инвариантность допускает использование только изначально заданного типа, 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)!

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`

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; }
}
```

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

Объект с типом
наследника добавить
можно

```
// 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>();
```

Microsoft .NET Core 2.0

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.WriteLine($"{x}"));
IComparable<B>[] compObjs2 = {objB, objA};
// compObjs = compObjs2;
```

Microsoft .NET Core 2.0

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/@olgaminsenкова/ContraVarAndConValInterface>)

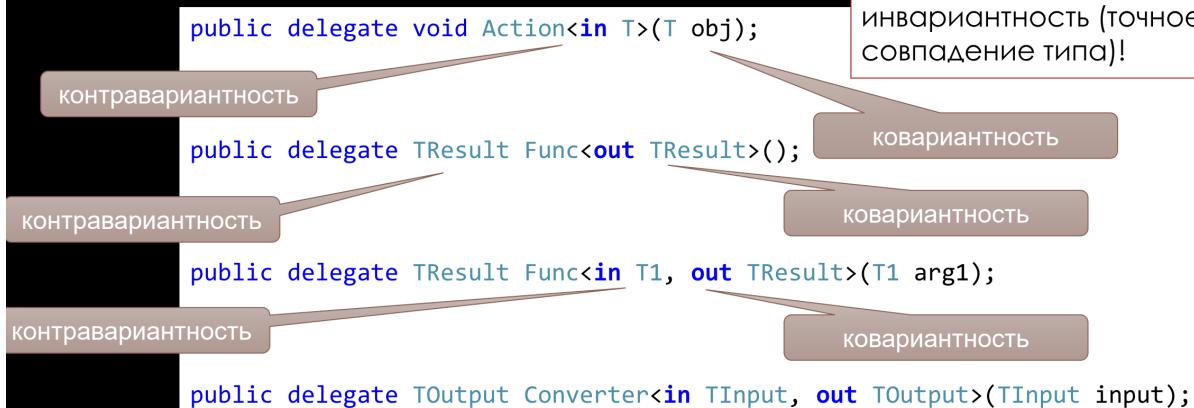
Максименкова О.В., 2024

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

// Ковариантность: public interface I Enumerable<out T> : System.Collections.IEnumerable
I Enumerable<Derived> strings = new List<Derived>();
I Enumerable<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
```

- При объявлении делегатов:



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);
```

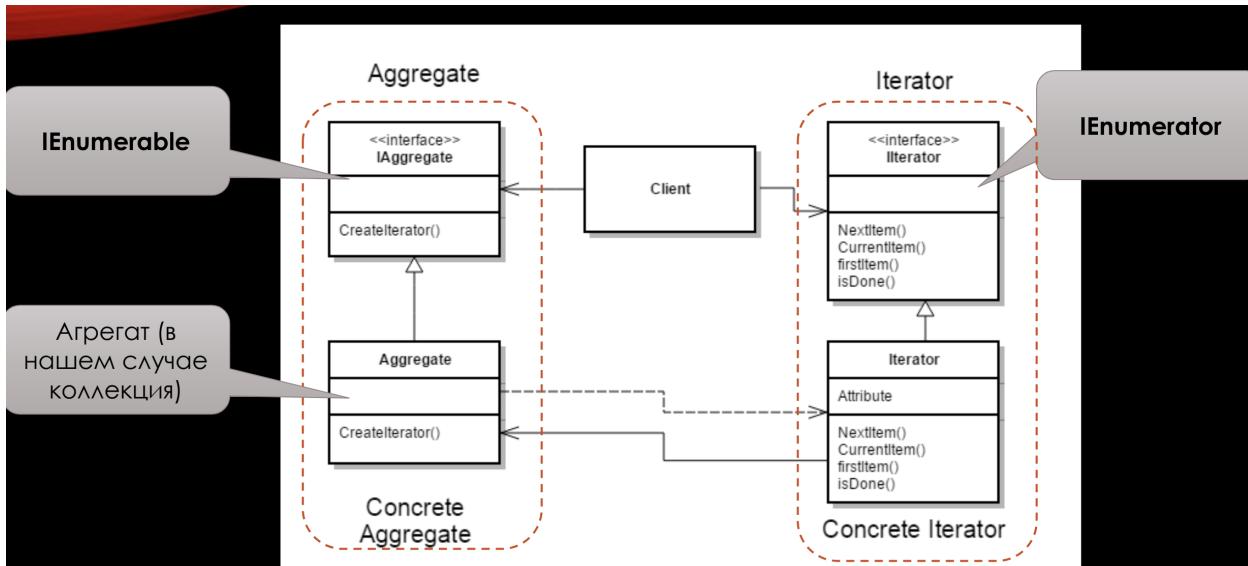
- Вариантность применяется только к ссылочным типам
если указать тип значения для аргумента вариативного типа, то этот параметр типа становится в результате инвариантным.
- Вариативность не применима к многоадресным делегатам (multicast delegate)
для заданных двух делегатов типов Action<Derived> и Action<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

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

Lecture 13-14. Iterators, Numerators



СПОСОБЫ СОЗДАНИЯ ИТЕРИРУЕМЫХ КОЛЛЕКЦИЙ

- Интерфейсы `IEnumerable` / `IEnumerator`
- Обобщённые интерфейсы `IEnumerable<T>` / `IEnumerator<T>`
- Конструкция, в которой интерфейсы не применяются (универсальная типизация - чёёё??)

Итераторы .NET односторонние и используются только для чтения

Чтобы получить экземпляр итератора, вызывается метод `GetEnumerator()` интерфейса `IEnumerable` (каждый раз возвращается новый экземпляр итератора)

Члены интерфейса `IEnumerator`:

- `object Current { get; }` — возвращает текущий элемент агрегата
- `bool MoveNext()` — осуществляет переход к следующему элементу агрегата, `false` возвращает, если достигнут конец последовательности
- `void Reset()` — устанавливает итератор в начало агрегата

`yield return` применяется для предоставления следующего значения итератора

Пример

```
class Numbers {
    public uint Numb { get; set; }
    public Numbers(uint n) => Numb = n;
    public IEnumrator<uint> GetEnumrator() => IterMethod();
}

public IEnumrator<uint> IterMethod() {
    uint newX = Numb;
    do {
        uint d = newX % 10;
        yield return d;
        newX = newX / 10;
    } while (newX != 0);
}
```

```
Numbers n = new Numbers(7096);
foreach(var i in n)
{
    Console.WriteLine(i);
}
```

Вывод:
6
9
0
7

```
class ColorEnumrator : IEnumrator
{
    string[] _colNames;
    int _pos = -1;
    public ColorEnumrator(string[] otherNames)
    { // Конструктор.
        _colNames = new string[otherNames.Length];
        for (int i = 0; i < otherNames.Length; i++)
            _colNames[i] = otherNames[i];
    }
    public object Current { get => _colNames[_pos]; } // Current.
    public bool MoveNext()
    { // MoveNext().
        if (_pos < _colNames.Length - 1) { _pos++; return true; }
        else return false;
    }
    public void Reset() => _pos = -1; // Reset().
} // End of class ColorEnumrator.
```

```
class MyColors : IEnumrable
{
    string[] _colors = { "Red", "Yellow", "Blue" };
    // GetEnumerator()
    public IEnumrator GetEnumerator() => new ColorEnumrator(_colors);
} // End of class MyColors.
```

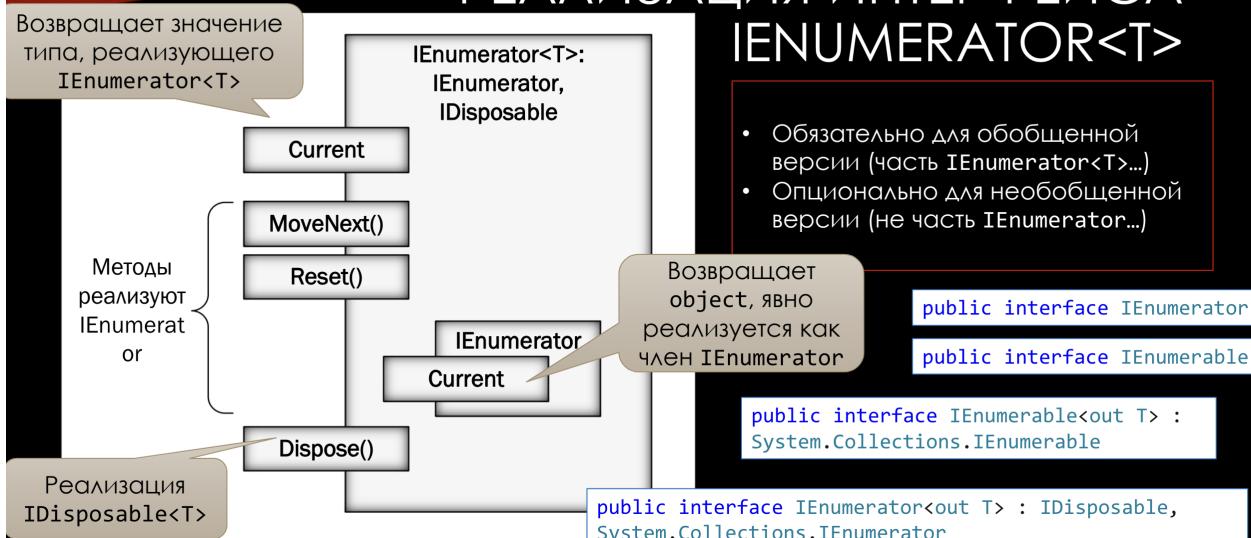
Коллекция с итератором

```
// шаблон проектирования итератора
class MyClass { // утиная типизация
    public IEnumerator<string> GetEnumerator()
    {
        return IteratorMethod();
    }
    public IEnumerator<string> IteratorMethod()
    {
        yield return ... ;
    }
}
void Main()
{
    MyClass mc = new MyClass();
    foreach (string s in mc)
    ....
    foreach (string s in mc.IteratorMethod())
    ....
}
```

Итерируемая коллекция с итератором

```
// шаблон проектирования итератора
class MyClass { // утиная типизация
    public IEnumerator<string> GetEnumerator()
    {
        return IteratorMethod().GetEnumerator();
    }
    public IEnumerable<string> IteratorMethod()
    {
        yield return ... ;
    }
}
void Main()
{
    MyClass mc = new MyClass();
    foreach (string s in mc)
    ....
    foreach (string s in mc.IteratorMethod())
    ....
}
```

РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА IENUMERATOR<T>

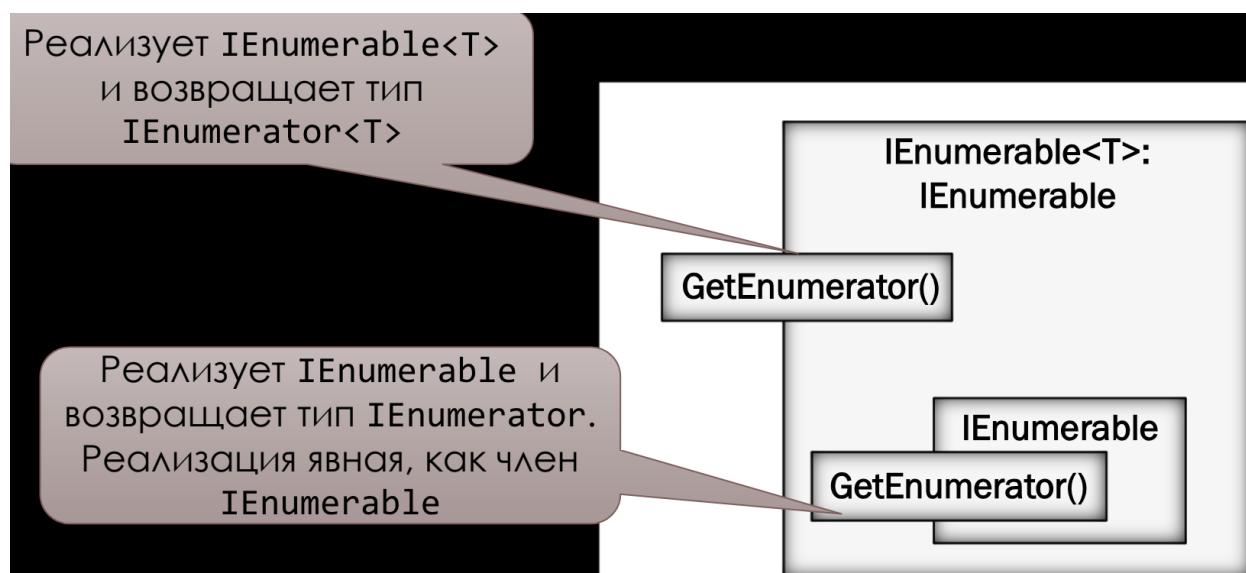


КЛАСС, РЕАЛИЗУЮЩИЙ ИНТЕРФЕЙС IENUMERATOR<T>

```
using System.Collections;
using System.Collections.Generic;
class MyGenEnumerator: IEnumerator< T > {
    public T Current { get; } // Ienumerator<T>--Current
    object IEnumerator.Current { get { ... } } // Ienumerator--Current
    public bool MoveNext() { ... } // Ienumerator--MoveNext
    public void Reset() { ... } // Ienumerator--Reset
```

```
    public void Dispose() { ... } // IDisposable--Dispose  
}
```

РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА IENUMERABLE<T>

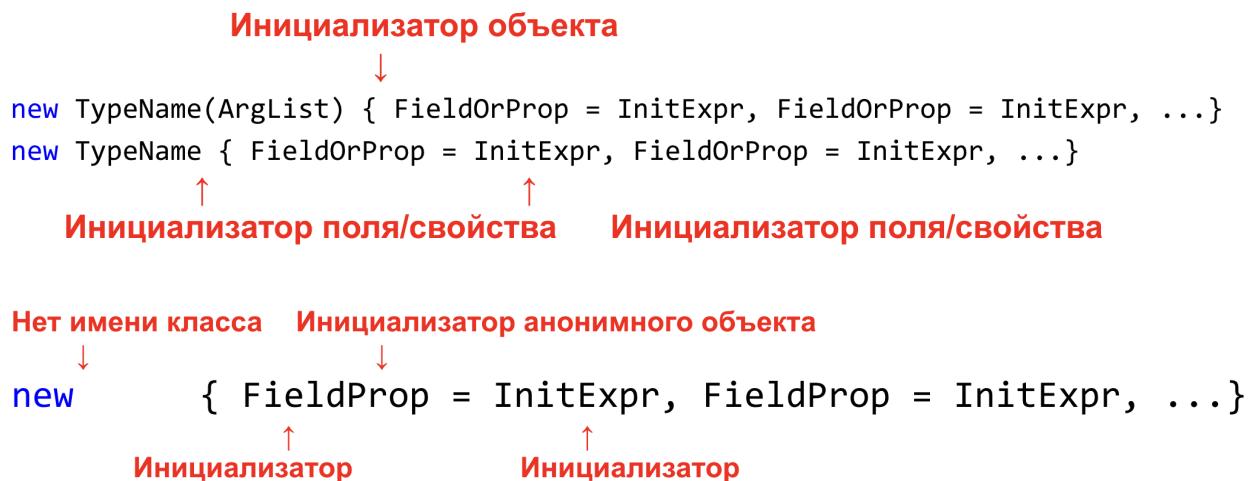


```
using System.Collections;  
using System.Collections.Generic;  
class MyGenEnumerable: IEnumerable<T>  
{  
    public IEnumerator<T> GetEnumerator() { ... } // IEnumerable<T>  
    IEnumerator IEnumerable.GetEnumerator() { ... } // IEnumerable  
}
```

Lecture 15-16. LINQ

Анонимный тип – создаваемый компилятором ссылочный тип (прямой наследник `object`), инкапсулирующий свойства только для чтения

- Инкапсулирует свойства только для чтения в один объект без необходимости предварительного создания типа; другие члены класса недопустимы
- Методов и событий в анонимном типе быть не может!
- Имя типа известно на уровне компилятора и не доступно на уровне исходного кода
- Тип каждого свойства (только для чтения) выводится компилятором



Создается объект **ссылочного** типа со свойствами только для чтения!

Пример:

```
public class Student // Студент.  
{  
    public int StID { get; set; }  
    public string LastName { get; set; }  
}
```

```
public class CourseStudent // Дисциплина.  
{  
    public string CourseName { get; set; }  
    public int StID { get; set; }  
}
```

```
Student st = new Student { StID = 1, LastName="Ivanov" };  
CourseStudent course = new CourseStudent { CourseName = "Programming", StID = 1 };
```

// Обязательно используем анонимный тип для идентификатора allInfo.

```
var allInfo = new { ID = st.StID, LastName = st.LastName, Course = course.CourseName };  
Console.WriteLine(allInfo);
```

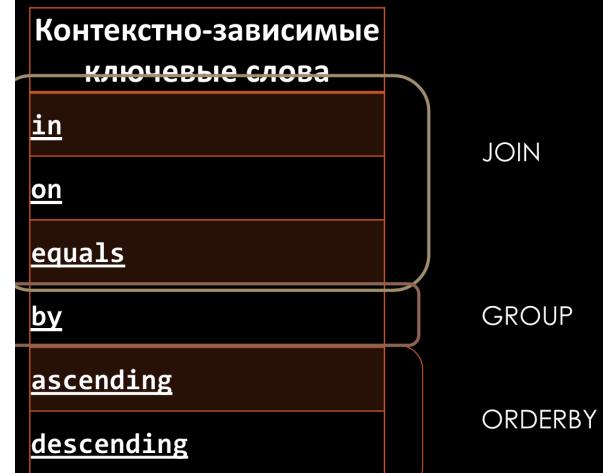


Технологии LINQ – это языковые конструкции, обеспечивающие согласованное функционирование запросов для объектов, реляционных БД и XML

Кроме того, LINQ-запрос может написать к любой коллекции, поддерживающей интерфейс IEnumerable / IEnumerable<T>

Методы:

from
where
select
group
into
orderby
join
let



Name Category	Метод	Описание категории
Restriction Ограничение	Where	Возвращает подмножество объектов из последовательности, выбирая их на основе критерия (предиката).
Projection Отображение	Select SelectMany	Выбирает элементы последовательности и создает из них другую последовательность, с элементами, возможно, другого типа
Partitioning Разбиение	Take TakeWhile Skip SkipWhile	Выбирает (возвращает) или отбрасывает (пропускает) объекты последовательности
Join Соединение	Join GroupJoin	Возвращает перечислимый (<code>IEnumerable<T></code>) объект, который соединяет элементы двух последовательностей, согласно заданному критерию.
Concatenation Конкатенация	Concat	"Склейивает" две последовательности в одну

Name Category	Метод	Описание категории
Ordering Упорядочивание	OrderBy OrderByDescending ThenBy ThenByDescending Reverse	Упорядочивает элементы последовательности на основе заданного критерия.
Grouping Группировка	GroupBy	Группирует элементы последовательности на основе заданного критерия.
Set Множества	Distinct Union Intersect Expect	Выполняет на элементах последовательности (последовательностей) теоретико-множественные операции.
Conversion Преобразования	Cast OfType AsEnumerable ToArray ToList ToDictionary ToLookup	Преобразует последовательности к различным формам, таким как массивы, списки, словари.
Equality Эквивалентность	SequenceEqual	Сравнивает (на равенство) две последовательности.

Name Category	Метод	Описание категории
Element Элемент	<code>DefaultIfEmpty</code> <code>First</code> <code>FirstOrDefault</code> <code>Last</code> <code>LastOrDefault</code> <code>Single</code> <code>SingleOrDefault</code> <code>ElementAt</code> <code>ElementAtOrDefault</code>	Возвращает конкретный элемент последовательности.
Generation Генерация	<code>Range</code> <code>Repeat</code> <code>Empty</code>	Генерирует последовательности.
Quantifiers Квантификаторы	<code>Any</code> <code>All</code> <code>Contains</code>	Возвращает логические значения, определяющие истинность заданного предиката на текущей последовательности
Aggregate Агрегация	<code>Count</code> <code>LongCount</code> <code>Sum</code> <code>Min</code> <code>Max</code> <code>Average</code> <code>Aggregate</code>	Возвращает отдельное значение, представляющее запрашиваемую характеристику последовательности.

Аксименкова О.В., 2024

всегда имя и обобщенный первый
 public + static параметр параметр
 ↓ ↓ ↓
`public static int Count<T>(this IEnumerable<T> source);`

`public static T First<T>(this IEnumerable<T> source);`
`public static IEnumerable<T> Where<T>(this IEnumerable<T> source, ...);`
 ↑ ↑
 тип возврата признак метода расширения

```
int[] numbers = { 2, 5, 28 };
```

// Возвращает ссылку с типом перечисления:

```
IEnumerable<int> lowNums = from n in numbers  
    where n < 20  
    select n;
```

```
int numsCount = (from n in numbers // Возвращает int.  
    where n < 20  
    select n).Count();
```

```
Console.WriteLine($"numsCount = {numsCount}"); // Вывод: 2.
```

```
var students = new[] { // массив объектов анонимного типа  
    new { LName="Jones", FName="Mary", Age=19, Major="History" },  
    new { LName="Smith", FName="Bob", Age=20, Major="CompSci" },  
    new { LName="Fleming", FName="Carol", Age=21, Major="History" }  
};  
// Используем анонимный тип select new { s.LastName, s.FirstName, s.Major };  
  
var query = from s in students  
    select new { s.LName, s.FName, s.Major };  
foreach (var q in query)  
    Console.WriteLine($"{q.FName} {q.LName} -- {q.Major}");
```

```
group student by student.Major;  
      ↑       ↑  
    КЛ. СЛОВО     КЛ. СЛОВО
```

Вывод:
Ключ группы History
Jones, Mary
Fleming, Carol
Ключ группы CompSci
Smith, Bob

```
var students = new[] { // массив объектов анонимного типа  
    new { LName="Jones", FName="Mary", Age=19, Major="History" },  
    new { LName="Smith", FName="Bob", Age=20, Major="CompSci" },  
    new { LName="Fleming", FName="Carol", Age=21, Major="History" }  
};  
  
var queryVar = from student in students  
    group student by student.Major;  
foreach (var s in queryVar) { // перечисляем группы  
    Console.WriteLine($"Ключ группы {s.Key}"); // s.Key – ключ группы  
    foreach (var t in s) // перечислим элементы группы  
        Console.WriteLine($"{t.LName}, {t.FName}");  
}
```

```
var groupA = new[] { 3, 4, 5, 6 };  
var groupB = new[] { 6, 7, 8, 9 };  
var someInts = from a in groupA           // Первая коллекция.  
    from b in groupB           // Вторая коллекция.  
    let sum = a + b           // Создана переменная для суммы.  
    where sum == 12           // Условие-фильтр.  
    select new { a, b, sum = a + b }; // Объект анонимного типа  
  
someInts.ToList().ForEach(s => Console.WriteLine($"{s} "));
```

Lecture 17. (Multi+) Threading

Пример нового потока:

```
class Program
{
    public static void Main()
    {
        Thread funcThread = new Thread(Print);
        funcThread.Start();

        for (int i = 0; i < 50; i++)
            Console.Write(0);
    }

    static void Print()
    {
        for (int i = 0; i < 50; i++)
            Console.Write(1);
    }
}
```

Методы взаимодействия потоков (захваты управления):

- Взаимоисключении (мьютексы [mutually exclusive access])
- Семафоры
- Критические секции
- События

lock – использование объекта-заглушки для обеспечения эксклюзивного доступа к критической секции кода в одном процессе

```

class ThreadSafe
{
    private readonly object _locker = new object();

    public void Method()
    {
        // Этот код может выполняться несколькими потоками.
        lock (_locker)
        {
            // Этот код может выполняться только одним потоком.
        }
        // Этот код может выполнятся несколькими потоками.
    }
}

```

Monitor – использование объекта-заглушки для обеспечения эксклюзивного доступа к критической секции кода в одном процессе

```

private static object _locker = new object();
private static int _counter = 0;

public static void CounterInc()
{
    Monitor.Enter(_locker); // Блокировка захвачена.
    try
    {
        _counter++;
    }
    finally
    {
        Monitor.Exit(_locker); // Блокировка освобождена.
    }
}

```

Мьютекс - это особый вид двоичного семафора, который используется для обеспечения механизма блокировки

Преимущества:

- Не возникает условий гонки, поскольку в критической секции в один момент времени находится только один процесс
- Обеспечивается целостность данных
- Это простой механизм блокировки, который активируется процессом перед входом в критическую секцию и освобождается при выходе из нее

Недостатки:

- Если после входа в критическую секцию поток заснет или будет вытеснен высокоприоритетным процессом, ни один другой поток не сможет войти в критическую секцию
- Когда предыдущий поток покидает критическую секцию, в нее могут войти только другие процессы
- Реализация мьютекса может привести к занятому ожиданию, что приводит к тряте процессорного времени

Mutex – частный случай семафора, применяющийся для ограничения доступа к ресурсу, при этом освободить ресурс может только занявший его поток

```
private static Mutex mtx = new Mutex();

public static void Method()
{
    mtx.WaitOne();
    try
    {
        // Критическая секция, выполняющаяся одним потоком.
    }
    finally
    {
        mtx.ReleaseMutex(); // Блокировка освобождена.
    }
}
```

Семафор – это обычная целочисленная неотрицательная переменная, с которой можно работать только с помощью двух специальных неделимых (атомарных) операций Ожидание (Wait, P) и Сингал (Signal, V)

Преимущества:

- Несколько потоков могут получить доступ к критической секции одновременно
- Одновременно к критической секции будет обращаться только один процесс, однако допускается работа нескольких потоков
- Семафоры являются машиннонезависимыми, поэтому их следует запускать через микроядро
- Гибкое управление ресурсами

Недостатки:

- Содержит инверсию приоритета
- Операции семафора (ожидание, сигнал) должны быть реализованы корректно, чтобы избежать дедлоков, что приводит к потере модульности, поэтому семафоры нельзя использовать в крупномасштабных системах
- Семафор чувствителен к ошибкам при программировании, что может провоцировать дедлеки и нарушению свойства взаимного исключения
- ОС должна отслеживать все вызовы операций ожидания и сигналов

Semaphore – служит для ограничения количества потоков, получающих доступ к ресурсу. Подходит для контроля пулов ресурсов, позволяет получить именованный системный семафор

```
class Program
{
    private static Semaphore smph = new Semaphore(2,2);
    public static void Method()
    {
        smph.WaitOne();
        try
        {
            // Критическая секция, выполняющаяся одним потоком.
        }
        finally
        {
            smph.Release(); // Блокировка освобождена.
        }
    }
}
```

SemaphoreSlim – служит для ограничения количества потоков, получающих доступ к ресурсу. Рекомендован к использованию для синхронизации на

уровне приложения

```
class Program
{
    private static SemaphoreSlim smph = new SemaphoreSlim(2,2);
    public static void Method()
    {
        smph.Wait();
        try
        {
            // Секция, ограниченная для 2 потоков.
        }
        finally
        {
            smph.Release(); // Блокировка освобождена.
        }
    }
}
```

AutoResetEvent, ManualResetEvent – служит для управления потоками с помощью сигналов и событий, т.е. позволяет составить систему оповещений между потоками

```
class Program
{
    private static AutoResetEvent arv = new(false);
    public static void Method()
    {
        // В этом потоке оповещаем о событии
        arv.Set();
    }
    public static void Method_1()
    {
        // В этом потоке ожидаем события
        arv.WaitOne();
    }
}
```

Пул потоков – специальный набор рабочих потоков приложения, управляемых системой

Для чего нужны пулы потоков?

- Абстракция над созданием потока (поток создаётся без явного указания программистом)
- В уже созданных потоках исполняются разные задачи (это способствует снижению затрат на создание потока и балансировки загрузки процессора)
- Управление пропускной способностью (ограничения или снятие ограничений на использование ядер ЦП)

Особенности:

- Статический класс System.Threading.ThreadPool
- Реализация на основе очереди делегатов
- Используется коллекция ConcurrentQueue<>
- Количество потоков в пуле лимитировано только объёмом доступной памяти
- Необработанные исключения в потоках пула приводят к завершению процессов, но есть исключения, которые используются для управления потоком выполнения программы:
 - ThreadAbortException (только .NET Framework приложения)
 - AppDomainUnloadedException из-за выгрузки домена приложения, в котором выполняется поток
 - CLR или ведущий процесс прерывает выполнение потока путём создания внутреннего исключения

```
/// <summary>
/// Синхронизация потоков
/// </summary>
static void Main()
{
    Thread tr = new Thread(Print3);
    tr.Start();
    tr.Join(); // Ожидание окончания потока tr
    for (int k = 0; k < 5; k++)
        Console.WriteLine("Поток_1");
    Console.WriteLine("Нажмите любую клавишу!");
    Console.ReadKey(true);
}
```

`Thread.CurrentThread` – статическое свойство (тип Thread)

`IsAlive` – свойство (тип bool, поток запущен и не завешен)

`Name` – свойство (тип string, имя потока, write-once)

`Start()` – запуск потока (перевод в состояние running)

`Start(аргумент)` – запуск потока с передачей аргумента

`Join()` – синхронизация (блокирует выполнение вызывающего потока до завершения потока, представленного экземпляром)

`Join(аргумент)` – синхронизация с таймаутом

`Sleep(time)` – приостановить исполнение потока на time мс

`Thread.Sleep(0)` – возможность переключиться на исполнение другого потока

Lecture 18. TPL Async Await

Основан на типах Task и Task<TResult>, необходимые для представления асинхронных операций.

Библиотека параллельных задач (Task Parallel Library, TPL) – набор открытых типов и API-интерфейсов из пространств имён System.Threading и System.Threading.Task.

Ограничения производительности ввода-вывода, рекомендовано использовать await для операции, которая возвращает Task или Task<T>, внутри метода async:

- Код ожидает чего-либо, например, данных из источника
- Ограничение ресурсов процессора, рекомендовано использоваться await для операции, которая запускается в фоновом потоке методом Task.Run
- Код выполняет сложные вычисления, например, перерасчёт комплексного состояния

Задача (Task) — это конструкция, реализующая модель асинхронной обработки на основе обещаний (Promise)

- Модель "обещает", что задача будет выполнена позже, позволяя взаимодействовать с помощью "обещаний" с чистым API
- Класс Task представляет одну задачу, которая не возвращает значение
- Класс Task<T> представляет одну задачу, которая возвращает значение типа T

ПАТТЕРН WAIT-UNTIL-DONE ЧЕРЕЗ TASK<T>

```
// объявление метода для асинхронного вызова
static long Sum(int x, int y) {
    Console.WriteLine("\t\tInside Sum");
    Thread.Sleep(100);
    return x + y;
}

public static void Main01Wait_Until_Done() {
    Task<long> task = new Task<long>(() => Sum(3, 5));
    Console.WriteLine("Before Start");
    task.Start();
    Console.WriteLine("After Start");
    Console.WriteLine("Doing stuff");
    long result = task.Result; // Wait for end and get result
    Console.WriteLine("After task.Result: {0}", result);
}
```

Результат:
 Before Start
 After Start
 Doing stuff
 Inside Sum
 After task.Result: 8

ПАТТЕРН POLLING ЧЕРЕЗ TASK<T>

```
static long Sum(int x, int y) {
    Console.WriteLine("\t\tInside Sum");
    Thread.Sleep(100);
    return x + y;
}

public static void Main02Polling() {
    Task<long> task = new Task<long>(() => Sum(3, 5));
    task.Start();
    Console.WriteLine("After Start");
    // Check whether the async method is done.
    while (!task.IsCompleted) {
        Console.WriteLine("Not Done");
        // Continue processing, even though in this case it's just busywork.
        for (long i = 0; i < 10000000; i++) ; // Empty statement
    }
    Console.WriteLine("Done");
    long result = task.Result; // Call EndInvoke to get result and clean up.
    Console.WriteLine("Result: {0}", result);
}
```

Результаты выполнения программы:
 After Start
 Not Done
 Inside Sum
 Not Done
 Not Done
 Not Done
 Done
 Result: 8

ПАТТЕРН CALLBACK ЧЕРЕЗ TASK<T>

```

static long Sum(int x, int y) {
    Console.WriteLine("\t\tInside Sum");
    Thread.Sleep(100);
    return x + y;
}

static void CallWhenDone(long result) {
    Console.WriteLine("\t\tInside CallWhenDone.");
    Console.WriteLine("\t\tThe result is: {0}.", result);
}

public static void Main03Callback() {
    Task<long> task = new Task<long>(() => Sum(3, 5));
    task.ContinueWith(t => CallWhenDone(t.Result)); ←
    task.Start();
    Console.WriteLine("After Start");
    Console.WriteLine("task.Result: {0}", task.Result);
    // Console.ReadKey(true); // Необязательно
}

```

ASYNC + AWAIT

```

static long Factorial(int factor) {
    long res = 1;
    for (int k = 1; k <= factor; k++)
        res *= k;
    System.Threading.Thread.Sleep(1500);
    return res;
}

// запуск вычисления факториала в отдельном потоке
static async Task<long> FactorialAsync(int factor) {
    Console.WriteLine("2 - Запуск асинхронного потока!");
    var eee = await Task.Run(() => Factorial(factor));
    Console.WriteLine("4 - Завершен асинхронный поток!");
    return eee; // long
}

```

BackgroundWorker - пример

```

using System;
using System.ComponentModel; // BackgroundWorker
using System.Threading;

namespace DemoLect_MultiThreading {
    public class BackgroundWorkerDemo {
        public static void Main() { // Основной поток
            BackgroundWorker bgw = new BackgroundWorker();
            bgw.WorkerReportsProgress = true;
            bgw.DoWork += (sender, args) => {
                for (int n = 0; n++ < 50;) {
                    Thread.Sleep(50); // Задержка потока
                    Console.Write("W");
                    if (n % 5 == 0)
                        bgw.ReportProgress(2 * n);
                }
            };
            bgw.ProgressChanged += (sender, args) => {
                Console.WriteLine(args.ProgressPercentage + "%");
            };

            bgw.RunWorkerCompleted += (sender, args) => {
                Console.WriteLine("\r\nCancelled!");
            };

            Console.WriteLine("Основной поток выводит точки!");
            bgw.RunWorkerAsync(); // Фоновый поток запущен

            while (true) { // Цикл основного потока, Ctrl + C для прерывания
                Console.Write(".");
                // Искусственная задержка основного потока:
                for (int k = 0; k < int.MaxValue / 300; k++) ;
            }
        }
    }
}

```

Доп для любимого ПИво

В объявлении анонимного метода обязательно используется ключевое слово delegate,

В коде программы анонимный метод доступен только через ссылку с типом делегата подходящей сигнатуры,

Тело анонимного метода заключается в фигурные скобки,

Список формальных параметров анонимного метода может быть опущен (не указан)

Делегат-тип является специального вида классом,

Делегат-тип является наследником Delegate,

Делегат-тип опечатан (то есть отмечен модификатором sealed)

Экземпляр делегата:

может представлять несколько методов одновременно,

может представлять метод с типом возвращаемого значения,

прописанным в определении типа делегата,

может представлять метод без параметров, может быть связан с анонимным методом,

может представлять перегруженный метод

Обобщёнными не могут быть перечисления и события.

Реализация обобщённого метода может быть размещена в необобщённом классе, обобщенном классе, обобщённой структуре, необобщённой структуре.

Реализация обобщённого метода может быть размещена в необобщённом абстрактном классе, обобщённом абстрактном классе, необобщённой структуре.

Итератор должен возвращать следующий элемент из совокупности элементов,
`yield break` служит для прерывания итератора

Допустимо объявление переменной `int yield = 4,`

yield имеет специальное назначение в блоке итератора,
В теле итератора допустимо использовать оператор
yield break

Итератор должен возвращать следующий элемент из совокупности элементов,
`yield break` служит для прерывания итератора

Классы, представляющие адаптеры потоков данных: `StreamReader`,
`StreamWriter`, `BinaryReader`, `BinaryWriter`

Все файлы, представляющие потоки данных в C#,
являются наследниками класса `Stream`,
Потоки данных поддерживают операции чтения, записи и поиска,
С потоком данных связаны ресурсы, которые требуется освобождать после завершения работы с потоком

Способы доступны программисту для предоставления программе возможностей получения информации о типах объектов во время исполнения: метод `System.Object.GetType()`, оператор `typeof()`, метод `System.Type.GetType()`

При двоичной сериализации сериализуются открытые и закрытые поля.,
Атрибут `[Serializable]` необходим для двоичной сериализации.,
При десериализации объекта с использованием `BinaryFormatter` конструкторы объекта не вызываются.

XML-сериализация не выполняет преобразования методов, индексаторов, закрытых полей или свойств, объявленных только для чтения.,
При XML-сериализации возможно указать имя элемента или атрибута.,
При XML-сериализации у объекта обязательно должен быть конструктор без параметров.

BinaryFormatter сериализует все нестатические поля класса без атрибута [NonSerialized]

Событие – это член класса или структуры,
Событие может быть объявлено с использованием модификатора static,
Событие описывается на основе уже описанного делегата-типа,
Событие описывается при помощи служебного слова event
Событие может быть описано / декларировано в классе, в интерфейсе, в структуре