ЯЗЫКИ ПРИКЛАДНОГО ПРОГРАММИРОВАНИЯ

Лекция 6

СТРУКТУРЫ

- **Value type**

- В конструкторе необходимо инициализировать все поля структуры.
- По умолчанию есть конструктор без параметров инициализирующий **default** значениями.

```
public struct Student
{
    public string Name;
    public int Age = 18;

    public override string ToString() => $"Name: {Name}  Age: {Age}";

    public Student(string name)
    {
        Name = name;
    }

    public Student(string name, int age) : this(name)
    {
        Age = age;
    }
}

public static class StructUseExample
{
    public static void CopyExample()
    {
        var anna = new Student("Anna", 19);
        var ivan = anna with { Name = "Ivan" };
    }
}
```

СТРУКТУРЫ

- Value type
- В конструкторе необходимо инициализировать все поля структуры.
- По умолчанию есть конструктор без параметров инициализирующий **default** значениями.

```
public record Record
{
    public string Name { get; init; }
    public int Age { get; init; }

    public Record(string name, int age)
    {
        Name = name;
        Age = age;
    }
}

public record ShortRecord(string Name, int Age);
```

RECORD

- Immutable reference type
- Компилятор генерирует:
  - Equals по значению
  - ToString
- Поддерживают with

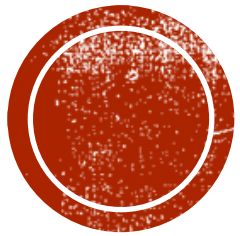
RECORD

```
1  var tom = new Person("Tom", 37);
2  var sam = tom with { Name = "Sam" };
3  Console.WriteLine($"{sam.Name} - {sam.Age}"); // Sam - 37
4
5  public record Person
6  {
7      public string Name { get; init; }
8      public int Age { get; init; }
9      public Person(string name, int age)
10     {
11         Name = name; Age = age;
12     }
13 }
```

- Поддерживают **with**

ALIASES

```
namespace Lecture2;
using printer = Console;

internal static class Program
{
    private static void Main(string[] args)
    {
        printer.WriteLine("Hello from printer");
        Console.WriteLine("Hello from console");
    }
}
```



GENERIC

```
internal class ListNode<ElemT>
{
    internal ElemT Value { get; }

    internal ListNode<ElemT>? Next { get; set; } = null;

    internal ListNode(ElemT val) => Value = val;
}

public class List<T>
{
    private ListNode<T>? _firstNode;

    public void Append(T value)
    {
        var node = new ListNode<T>(value);
        if (_firstNode == null)
            _firstNode = node;
        else
        {
            var prevNode = _firstNode;
            var nextNode = _firstNode.Next;
            while (nextNode != null)
            {
                prevNode = nextNode;
                nextNode = nextNode.Next;
            }
            prevNode.Next = node;
        }
    }

    // ... other methods of list
}
```

Generic class

- Аналог шаблонов из C++
- Позволяют обобщить логику и не дублировать реализации для разных типов
- Окончательное “определение” типа (класса) формируется только в момент его использования

- *На слайде пример реализации односвязного списка*

Generic interface (1)

```
public interface ICrudRepository<T> where T : notnull
{
    Guid Create(T item);

    T? Read(Guid id);

    void Update(Guid id, T itemToUpdate);

    T? Remove(Guid id);
}
```

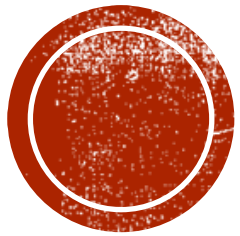
- Интерфейсы как и классы могут быть *«универсальными»*
- На тип может быть наложено ограничение в т.ч. Может требоваться реализация определенного интерфейса

- *На слайде пример интерфейса простого репозитория*

Generic Interface (2)

Реализация «универсального» интерфейса

```
public class InMemoryStringRepository : ICrudRepository<string>
{
    private readonly Dictionary<Guid, string> _items = new();

    public Guid Create(string item)
    {
        var id = Guid.NewGuid();
        _items[id] = item;
        return id;
    }

    public string? Read(Guid id) => _items.TryGetValue(id, out var item) ? item : null;

    public void Update(Guid id, string itemToUpdate) => _items[id] = itemToUpdate;

    public string? Remove(Guid id)
    {
        var itemContains = _items.TryGetValue(id, out var item);
        if (itemContains)
            _items.Remove(id);
        return item;
    }
}
```



ИСКЛЮЧЕНИЯ

ИСКЛЮЧЕНИЯ В C#

- Базовый класс – **Exception**

- Наиболее распространенные:
  - **NullReferenceException**
  - **OutOfMemoryException**
  - **ArgumentException** и **ArgumentNullException**
  - **IndexOutOfRangeException** и **ArgumentOutOfRangeException**
  - **NotImplementedException**
  - **AggregateException**
  - **ArithmeticException**

СОЗДАНИЕ СОБСТВЕННОГО ИСКЛЮЧЕНИЯ

```
public class CustomException : Exception
{
    public CustomException() { }
    public CustomException(string? message) : base(message) { }
    public CustomException(string? message, Exception? innerException) : base(message, innerException) { }
}
```

```
public partial class CallExample
{
    private static void Foo() => throw new OutOfMemoryException("MyCustomMessage");
    private static void Bar() => throw new ArgumentException();

    public static void ExceptionsCatchExample()
    {
        try
        {
            Foo();
        }
        catch (Exception e)
        {
            Console.WriteLine(e);
        }
    }
```

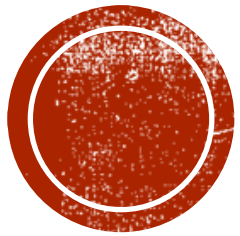
```
public static void FullCatchBlockExample()
{
    try
    {
        Foo();
        Console.WriteLine("No Errors in Foo");
    }
    catch (OutOfMemoryException e) when(e.Message.Contains("MyCustomMessage"))
    {
        Console.WriteLine("Filtered OutOfMemoryException was thrown");
    }
    catch (OutOfMemoryException e)
    {
        Console.WriteLine("Other OutOfMemoryException was thrown");
    }
    catch (Exception e)
    {
        Console.WriteLine(e);
        throw;
    }
    finally
    {
        Console.WriteLine("Finally be called anyway.");
    }
}
```

TRY-CATCH-FINALY

- Исключение последовательно пройдет через catch в порядке их написания
- Finaly выполняется в любом случае (даже если исключение не перехвачено)
- При помощи **throw;** можно выбросить пойманное исключение дальше после обработки в **catch** – такой вариант предпочтителен так как сохраняет **stack-trace**
- При помощи **when** можно накладывать дополнительные условия на **catch**

XML КОММЕНТАРИИ

```
/// <summary>
/// Interface for simple repository that allow save entities
/// </summary>
public interface ISaveRepository<T> where T : notnull
{
    /// <summary>
    /// Method for saving new entity to repository
    /// </summary>
    /// <param name="item">New entity for save. </param>
    /// <returns>Generated Id of new entity. </returns>
    /// <exception cref="ArgumentException"> If argument is already in repository. </exception>
    Guid Save(T item);
}

/// <inheritdoc cref="ISaveRepository{T}"/>
public class SaveRepositoryStub<T> : ISaveRepository<T> where T : notnull
{
    public Guid Save(T item) => Guid.NewGuid();
}
```



КОЛЛЕКЦИИ

IEnumerable<T>

```
public interface IEnumerable<T> : IEnumerable
{
    new IEnumerator<T> GetEnumerator();
}
```

```
public interface IEnumerator<T> : IEnumerator
{
    bool MoveNext();

    T Current { get; }

    void Reset();
}
```

- Простейший интерфейс коллекции, позволяющий перебирать данные в ней при помощи итератора.

- *На слайде – несколько упрощенный код из стандартной библиотеки*

```
public interface ICollection<T> : IEnumerable<T>
{
    int Count { get; }

    bool IsReadOnly { get; }

    void Add(T item);

    void Clear();

    bool Contains(T item);

    // CopyTo copies a collection into an Array, starting at a particular
    // index into the array.
    //
    void CopyTo(T[] array, int arrayIndex);


    bool Remove(T item);
}
```

ICOLLECTION<T>

- *На слайде – несколько упрощенный код из стандартной библиотеки*

FOREACH

```
public static void ForeachCall()
{
    var emails = new string[] { "petya@kek.ru", "sasha@rar.ru", "masha@bst.ru" };

    foreach (var email in emails)
    {
        Console.WriteLine(email);
    }
}
```

МАССИВЫ

- Массив – объект класса **Array**, а не указатель/область в памяти.
- Реализует **Ienumerable** и **Icollection**
- Обладает дополнительным набором полей и методов, например **Length** – размер массива.
- Подробнее – см https://docs.microsoft.com/ru-ru/dotnet/api/system.array?view=net-6.0

List<T>

- Список, в который можно добавлять элементы
- «Под капотом» является динамическим массивом
- Реализует **Ienumerable<T>** и **Icollection<T>**

ДРУГИЕ ПОЛЕЗНЫЕ КОЛЛЕКЦИИ

- ArrayList
- HashSet
- Dictionary
- Queue
- Stack
- LinkedList

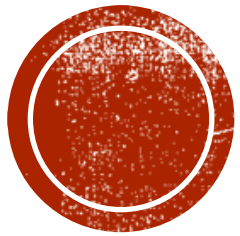
```
public class GameObject2D
{
    private int _x;
    private int _y;

    public GameObject2D(int x, int y)
    {
        _x = x;
        _y = y;
    }

    public void ChangePosition(int newX, int newY)
    {
        PositionsHistory.Add((x:_x, y: _y));
        _x = newX;
        _y = newY;
    }

    public readonly List<(int x, int y)> PositionsHistory = new();

    public (int x, int y) GetPosition => (_x, _y);
}
```

КОРТЕЖИ

- Предоставляют способ работы с набором значений

ЛОКАЛЬНЫЕ ФУНКЦИИ

```
public class LocalFunctionExample
{
    public void Method(int x)
    {
        int temp = 42;

        Console.WriteLine(InnerMethod());
        Console.WriteLine(InnerMethod());
        Console.WriteLine(InnerMethod());

        int InnerMethod()
        {
            temp += x;
            return temp;
        }
    }
}
```



REF, IN, OUT

REF

```
public static void RefExample(ref int param)
{
    param++;
    Console.WriteLine(param);
}

public static void RefExampleCall()
{
    int variable = 42;
    RefExample(ref variable);
    Console.WriteLine(variable);

    // Console:
    // 43
    // 43
}
```

- Используется для передачи параметров по ссылке

IN

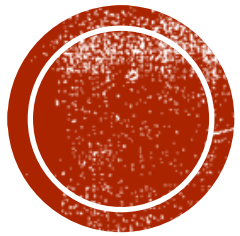
```
public static void InExample(in int param)
{
    // param - не может быть измененен

    Console.WriteLine(param);
}
```

- указывает, что параметр будет передаваться в метод по ссылке, однако внутри метода его значение нельзя будет изменить.

OUT

```
public static void OutExample(out int param)
{
    param = 42;
    Console.WriteLine(param);
}

public static void OutExampleCall()
{
    OutExample(out var variable);
    Console.WriteLine(variable);

    int otherVariable;
    OutExample(out otherVariable);
}
```

- Модификатор **out** - для возврата значения по ссылке из метода.
- В теле метода обязательно должно быть присвоено значение.
- Позволяют вернуть несколько значений из метода.



ДЕЛЕГАТЫ

ДЕЛЕГАТЫ

- Делегат – это специальный объект указывающий на метод и позволяющий вызвать его.
- Для объявления используется ключевое слово **delegate**, далее указывается возвращаемый тип, название и параметры.
- Метод соответствует делегату, если они имеют одинаковый возвращаемый тип и тот же набор параметров, **in**, **out**, **ref** также учитываются.
- Делегат может быть обобщенным (**generic**).

ДЕЛЕГАТЫ

```
public static class OtherClass
{
    public static void Hello() => Console.WriteLine("Hey bro!");
    public static double Sum(double f, double s) => f + s;
}

internal delegate void ActionDelegate();

internal delegate R BinaryOperation<T, R>(T first, T second);

public class DelegatesExample
{
    private static void WriteHelloWorld() => Console.WriteLine("Hello World");

    public static void SimpleDelegateExample()
    {
        ActionDelegate ad = WriteHelloWorld;
        ActionDelegate delegateFromOtherClass = OtherClass.Hello;
        BinaryOperation<double, double> binaryOperation = OtherClass.Sum;

        ad(); // Hello World
        delegateFromOtherClass(); // Hey bro!
        Console.WriteLine(binaryOperation(10, 20)); // 30
    }
}
```

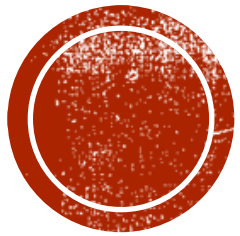
```
public class DelegatesExample2
{
    public static void SomeOtherExample()
    {
        ActionDelegate ad = WriteHelloWorld;
        ad();
        // Hello World

        ad += OtherClass.Hello;
        ad += OtherClass.Hello;
        ad();
        // Hello World
        // Hey bro!
        // Hey bro!

        ad -= OtherClass.Hello;
        ad -= OtherClass.Hello;
        ad?.Invoke();
        // Hello World

        ad -= WriteHelloWorld;
        ad?.Invoke();
        // No console
    }
}
```

ДЕЛЕГАТЫ

- Делегат может указывать на множество методов сразу. Все методы в делегате попадают в список вызова (invocation list).
- При вызове делегата все методы из этого списка последовательно вызываются.
- Для добавления методов в делегат применяется операция +=, для удаления -=

ДЕЛЕГАТЫ

- При удалении методов из делегата фактически будет создаваться новый делегат, который в списке вызова методов будет содержать на один метод меньше.
- При удалении метода не добавленного в делегат, результат - null.
- Если делегат содержит несколько ссылок на один и тот же метод, то операция -= начинает поиск с конца списка вызова делегата и удаляет только первое найденное вхождение.
- Если подобного метода в списке вызова делегата нет, то операция -= не имеет никакого эффекта.



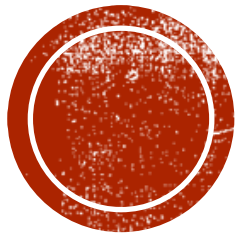
АНОНИМНЫЕ МЕТОДЫ

- Анонимные методы используются для создания экземпляров делегатов, они не могут существовать сами по себе, используется:
  - Для инициализации экземпляра делегата
  - в качестве аргумента для параметра метода (делегата)

```
public class AnonymousMethodsExamples
{
    private delegate int BinaryOperation(int first, int second);

    private static void CallDelegate(BinaryOperation op)
    {
        Console.WriteLine(op(5, 6));
    }

    public static void Exec()
    {
        BinaryOperation operation = delegate (int x, int y)
        {
            return x + y;
        };

        CallDelegate(delegate(int x, int y)
        {
            return x * y;
        });
    }
}
```

АНОНИМНЫЕ МЕТОДЫ

- Если анонимный метод использует параметры, то они должны соответствовать параметрам делегата.
- Если для анонимного метода не требуется параметров, то скобки с параметрами опускаются.
- Параметры можно опустить если их не планируется использовать, однако, параметры не могут быть опущены, если один или несколько параметров определены с модификатором out.

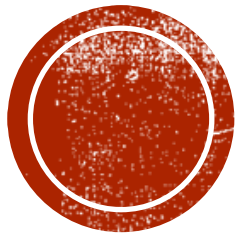


ЛЯМБДА-ВЫРАЖЕНИЯ

ЛЯМБДА ВЫРАЖЕНИЯ

```
public class LambdasExample
{
    private delegate int BinaryOperation(int first, int second);

    private static void CallDelegate(BinaryOperation op)
    {
        Console.WriteLine(op(5, 6));
    }

    public static void Exec()
    {
        BinaryOperation operation = (f, s) => f + s;
        CallDelegate((f, s) => f * s);

        var f = (int x) => x * x;
        Console.WriteLine(f(10));
    }

    private static double S(double radius) => Math.PI * 2 * radius;
}
```

- «Синтаксический сахар» для записи анонимных методов, более емкий и лаконичный.
- Тип данных – делегат.



ACTION, PREDICATE, FUNC

Action

```
public class DefaultDelegatesExample
{
    private Action _simpleAction = () => Console.WriteLine("Fail");
    private Action<int> _simpleActionWithParameter = (param) => Console.WriteLine(param);

    public delegate void Action();
    public delegate void Action<in T>(T obj);
    public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
    // ...
}
```

- Встроенный делегат
- Действие, которое ничего не возвращает
- Может принимать до 16 параметров

PREDICATE

- Встроенный делегат
- Принимает один параметр и возвращает **bool**

```
public Predicate<int> _predicate = (x) => x > 0;
```

FUNC

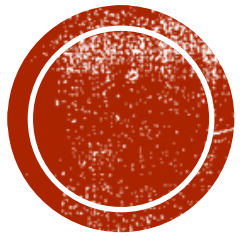
```
private Func<int> _func = () => 42;
private Func<int, int, int> _funcWithParameters = (p1, p2) => p1 * p2;

public delegate TResult Func<out TResult>();
public delegate TResult Func<out TResult, in T>(T obj);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// ...
```

- Встроенный делегат
- Действие, которое имеет результат (функция/метод)
- Может принимать до 16 параметров

Action

```
private Func<int> _func = () => 42;
private Func<int, int, int> _funcWithParameters = (p1, p2) => p1 * p2;

public delegate TResult Func<out TResult>();
public delegate TResult Func<out TResult, in T>(T obj);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// ...
```

- Встроенный делегат
- Действие, которое имеет результат (функция/метод)
- Может принимать до 16 параметров



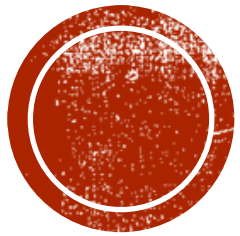
EVENT

```
public class Account
{
    public Account(int startBalance)
    {
        _balance = startBalance;
    }

    public event Action<int>? BalanceUpdated;

    private int _balance;

    public int Balance
    {
        get => _balance;
        set
        {
            _balance = value;
            BalanceUpdated?.Invoke(_balance);
        }
    }
}

public static class Caller
{
    public static void Call()
    {
        var account = new Account(100);
        account.BalanceUpdated += newBalance => Console.WriteLine($"Текущий баланс: {newBalance}");

        account.Balance -= 10;
        account.Balance -= 20;
        account.Balance -= 30;
    }
}
```

СОБЫТИЯ

- Позволяют перейти от линейного выполнения программы к обработке событий.

- События объявляются с помощью ключевого слова event, после которого указывается тип делегата, который представляет событие.
- С событием связывается один или несколько обработчиков.
- Для добавления обработчика события применяется операция +=:
- Обработчик может быть не определен, поэтому при вызове события лучше его всегда проверять на null.
- **add/remove accessors**



ЗАМЫКАНИЯ

ЗАМЫКАНИЕ

- Замыкания – функция имеющая доступ и использующая переменные из внешнего контекста выполнения/окружения.
- Замыкание состоит из:
  - Внешнего метода, который определяет область видимости и в которой определены переменные и параметры
  - переменные и параметры (лексическое окружение), которые определены во внешней функции
  - вложенная функция

```
public class ClosureExample
{
    public static void LambdaExample()
    {
        var outerFn = () =>
        {
            var x = 10;
            return () => Console.WriteLine(++x);
        };

        var fn = outerFn();

        fn();   // 11
        fn();   // 12
        fn();   // 13
    }

    delegate int Operation(int n);

    public static void InnerFunctionExample()
    {
        var fn = Multiply(5);

        Console.WriteLine(fn(5));   // 25
        Console.WriteLine(fn(6));   // 30
        Console.WriteLine(fn(7));   // 35

        Operation Multiply(int n)
        {
            int Inner(int m)
            {
                return n * m;
            }
            return Inner;
        }
    }
}
```

ЗАМЫКАНИЕ НА ПРАКТИКЕ

- На практике замыкание можно получить:
  - При помощи лямбда-выражения
  - При помощи локальной функции