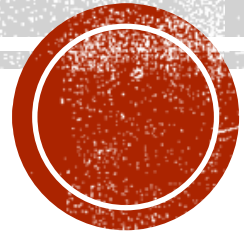
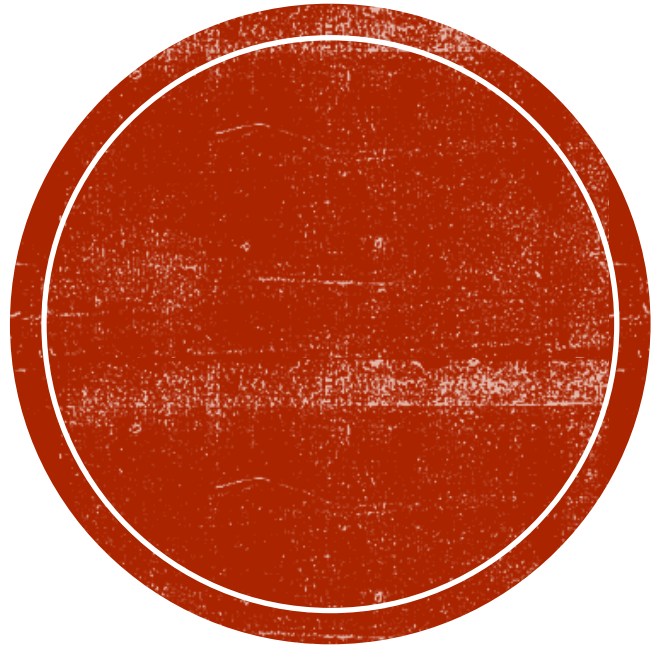
ЯЗЫКИ ПРИКЛАДНОГО ПРОГРАММИРОВАНИЯ

Лекция 2



УСТРОЙСТВО ПАМЯТИ В .NET

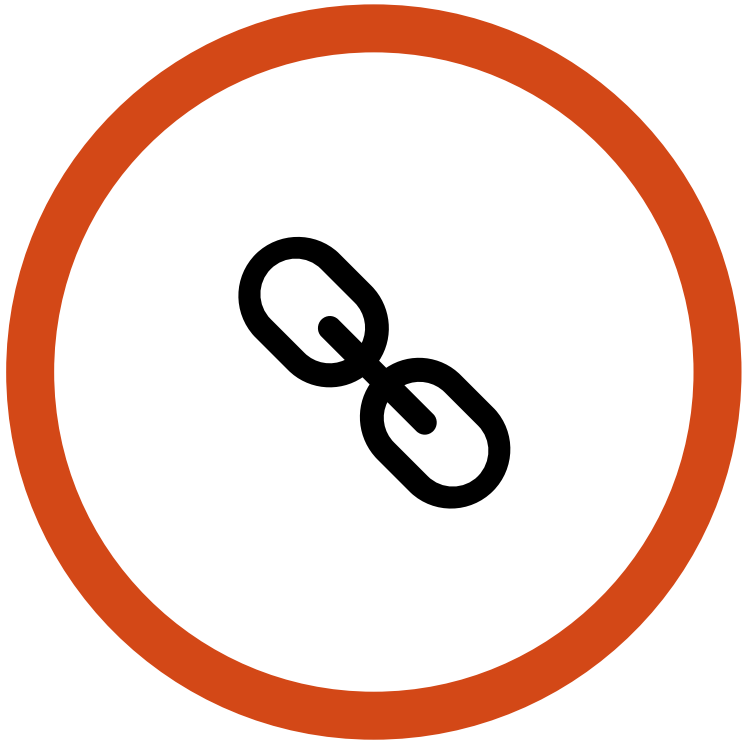


УСТРОЙСТВО ПАМЯТИ

- Стек
  - Имеет фиксированный размер
  - Не требует очистки (очищается автоматически при выходе за область видимости).
- Куча
  - Имеет динамический размер, ограниченный только объемом RAM*
  - Операции выполняются дольше чем на стеке

REFERENCE TYPE

- Ссылки храниться на стеке, объект хранится в куче
- Передача – по ссылке
- Требуется очистка памяти => нагрузка на GC
- Все классы – reference type



VALUE TYPE

- Хранится на стеке
- Стек ограничен => слишком большие **value type** «объекты» будут его быстро забивать
- Нет работы с кучей => нет нагрузки на **GC** => выше производительность
- Передаются по значению

BOXING

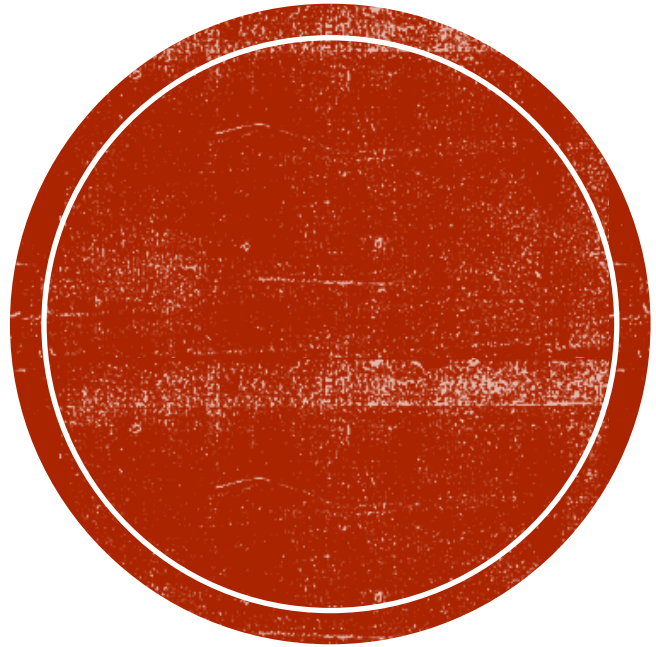
- Если работать с **value type** как с **Object** (классом) произойдет **boxing** - упаковка в «объект» и размещение в куче, что влечет накладные расходы
- Если привести «запакованный» **value type** обратно – произойдет распаковка – т.е. перемещение из кучи на стек

GARBAGE COLLECTOR (GC)

Задача **GC** – очистка кучи от уже не используемых объектов

- Куча (ссылки) делится на поколения (0, 1, 2)
- **GC**
  - проходит по объектам поколения, если на них нет корневых ссылок – удаляет
  - Если объект не был очищен несколько раз – он перемещается в следующее поколение.
  - Время от времени запускается дефрагментация.
- Кольцевые зависимости
- **Large object heap**

ПОДРОБНЕЕ ПРО ООП В C#



КОНСТРУКТОР

```
public class Message
{
    private string _title;
    private string _text;

    public int Id;

    public Message(int id)
    {
        Id = id;
    }

    public Message(string title, string text, int id) : this(id)
    {
        this._title = title;
        this._text = text;
    }

    public string Render() => $"\t {_title} \n\n  {_text}";
}
```

МЕТОДЫ

```
[модификаторы] тип_возвращаемого_значения название_метода ([параметры])
{
    // тело метода
}
```

МОДИФИКАТОРЫ ДОСТУПА

Модификатор	Текущий класс	Класс наследник из текущей сборки	Класс наследник из другой сборки	Класс из текущей сборки	Класс из другой сборки
private	+				
private protected	+	+			
protected	+	+	+		
internal	+	+		+	
protected internal	+	+	+	+	
public	+	+	+	+	+

МОДИФИКАТОРЫ КЛАССОВ

- *internal* – доступен только внутри сборки
- *public* – доступен в других сборках
- *abstract* – абстрактный класс
- *seald* - класс от которого нельзя наследоваться, упрощает работу JIT

STATIC

```
namespace HelloWorld
{
    internal static class Program
    {
        private static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

- **static class** – класс объект которого нельзя создать (содержит только **static** методы, поля, константы)
- **Static field** – статическое поле класса, доступно по имени класса (**ClassName.FieldName**).

СВОЙСТВА

```
public class Person
{
    private int _name;
    public int NameProperty
    {
        get { return _name; }
        set { _name = value; }
    }

    public int NamePropertyV2
    {
        get => _name;
        set => _name = value;
    }
}

public static class PropertyCallExample
{
    static void Call()
    {
        var c = new Person();
        c.NameProperty = 8;
        Console.WriteLine(c.NamePropertyV2);
    }
}
```

СВОЙСТВА

```
public class ClassWithProperties
{
    public int Property { get; set; }

    public int PropertyWithPrivateSet { get; private set; }

    public int ImmutableProperty { get; }

    public int ComputedProperty => 42 + _field;
}
```

Инициализатор и init

```
public class Character
{
    public string Name { get; init; }
    public int Age { get; init; }

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

public class InitExample
{
    public static void InitCallExample()
    {
        var character = new Character()
        {
            Name = "Aragorn",
            Age = 87
        };
        Console.WriteLine($"{character}");
    }
}
```

READONLY И CONST

Readony

- Нельзя изменить
- Присваивается в конструкторе или при объявлении
- Может применяться к полям и свойствам, как статическим – так и нет

Const

- Нельзя изменить
- Определяется на этапе компиляции
- Доступ – аналогично **static** полю класса

НАСЛЕДОВАНИЕ

```
public abstract class Unit
{
    public string Name { get; }
    public int Hp { get; protected set; }

    protected Unit(string name, int hp)
    {
        Name = name;
        Hp = hp;
    }

    public virtual void TakeDamage(int damage) => Hp -= damage;
    public abstract string GetPhrase();
}
```

- В C# нет множественного наследования

- **abstract** – метод должен быть реализован в наследнике
- **Virtual** – метод может быть переопределен в наследниках при помощи **override**

НАСЛЕДОВАНИЕ

```
public class Undead : Unit
{
    private const int DefaultHp = 100;

    public Undead() : base("Undead", DefaultHp) { }

    public override void TakeDamage(int damage) => Hp -= (int) Math.Round(damage / 2.0);

    public override string GetPhrase() => "...";
}

public class Worker : Unit
{
    private const int DefaultHp = 10;
    public int BuildPower { get; }

    public Worker(int buildPower) : base("Worker", DefaultHp)
    {
        BuildPower = buildPower;
    }

    public override string GetPhrase() => "Опять работа?!";
}
```

ИНТЕРФЕЙСЫ

- *interfacre*

- Содержат только* объявления методов, свойств и т.п.
- Класс может реализовывать один или несколько интерфейсов.

** - в последних версиях C# добавили реализацию по умолчанию для методов интерфейса*

РЕКОМЕНДАЦИИ

Названия интерфейсов начинать с **I**

Для любого **public** класса делать интерфейс – так будет проще тестировать + абстрагировать **API** от реализации.

Где возможно – принимать интерфейсы а не конкретные реализации (в основном для классов с логикой, для моделей – это лишнее).

Делить интерфейсы по ответственности

ИНТЕРФЕЙСЫ - ПРИМЕР

```
public interface IApiClient
{
    int GetFollowersCount(string userId);

    List<string> GetFollowers(string userId);

    void FollowUser(string userId, string followerId);
}

public class MockApiClient : IApiClient
{
    public int GetFollowersCount(string userId) => 42;

    public List<string> GetFollowers(string userId) => new() { "Adam", "Eva" };

    public List<string> Followed => new();
    public void FollowUser(string userId, string followerId) => Followed.Add(followerId);
}
```

ИНТЕРФЕЙСЫ – ПРИМЕР

```
public class FollowingService
{
    private readonly IApiClient _apiClient;

    public FollowingService(IApiClient apiClient)
    {
        _apiClient = apiClient;
    }

    public int FollowAndGetFollowersCount(string userId, string followerId)
    {
        _apiClient.FollowUser(userId, followerId);
        // some other app logic ...
        return _apiClient.GetFollowersCount(followerId);
    }
}

public static class InterfaceCaller
{
    public static void App()
    {
        var service = new FollowingService(new RealApiClient());
        service.FollowAndGetFollowersCount("Adam", "Eva");
    }

    public static void Test()
    {
        var mock = new MockApiClient();
        var service = new FollowingService(mock);

        service.FollowAndGetFollowersCount("Adam", "Eva");

        if (mock.Followed[0] != "Eva")
            Console.WriteLine("TEST FAILED");
    }
}
```

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

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");
    }
}
```