

# Технологии программирования

Лекция №2  
ИС, весна 2022

Вспомним базу

# Что такое тип и зачем?

- Тип – это спецификация работы с набором битов.
- Фундаментальные типы задаются архитектурами процессоров.
- Тип ссылочный: тип, значение которого содержит ссылку на его данные. Значение описываемое ссылочным типом указывает месторасположение другого значений.

# ValueType | ReferenceType

Из стандарта ECMA335:

- Тип значений : такой тип, экземпляр которого непосредственно содержит все свои данные.
- Тип ссылочный: тип, значение которого содержит ссылку на его данные. Значение описываемое ссылочным типом указывает месторасположение другого значений.

# Stack

- Содержит информацию о выполняемом потоке, цепочку вызванных методов, объявленные локальные переменные.
- При вызове конструктора структуры происходит выделение памяти на стеке под неё.
- При вызове конструктора класса, в куче выделяется память, создаётся экземпляр, на стек пишется ссылка на этот экземпляр.

# Heap

- Место, куда скидываются экземпляры.
- Намного больше места чем в стеке (но и не такая быстрая работа).
- Встроенный в рантайм механизм выделения памяти по запросу.
- Встроенный механизм по очистке данных.

# Boxing

- Процесс упаковки типа со стека на кучу.
- Во время боксинга для value type выделяется память на куче, он переносится туда.

System.Object



# System.Object

- Базовый тип, от которого наследуются все другие.

# Посмотрим на Object через WinDbg

- Запускаем код **var value = new object();**
- `!DumpHeap -type System.Object`
- Находим:
  - `00007ffc15f89480`      `x`      `xx System.Object`
- Идём глубже и дамвим его таблицу методов
  - `!dumpmt -md 00007ffc15f89480`

# Смотрим на методы

MethodDesc Table

Entry	MethodDesc	JIT	Name
00007FFC09DE0030	<a href="#">00007ffc09dd5608</a>	NONE	System.Object.Finalize()
00007FFC09DE0038	<a href="#">00007ffc09dd5618</a>	NONE	System.Object.ToString()
00007FFC09DE0040	<a href="#">00007ffc09dd5628</a>	NONE	System.Object.Equals(System.Object)
00007FFC09E14E28	<a href="#">00007ffc09dd5668</a>	PreJIT	System.Object.GetHashCode()
00007FFC09DE0028	<a href="#">00007ffc09dd55f8</a>	PreJIT	System.Object..ctor()
00007FFC09DE0018	<a href="#">00007ffc09dd55c8</a>	NONE	System.Object.GetType()
00007FFC09DE0020	<a href="#">00007ffc09dd55e0</a>	NONE	System.Object.MemberwiseClone()
00007FFC09DE0048	<a href="#">00007ffc09dd5638</a>	NONE	System.Object.Equals(System.Object, System.Object)
00007FFC09DE0050	<a href="#">00007ffc09dd5650</a>	NONE	System.Object.ReferenceEquals(System.Object, System.Object)

# Дефолтные реализации методов `System.Object`

- `ToString` - возвращает имя типа.
- `Equals` - для ссылочных типов сравнивает ссылки, для типов значений сравнивается значение полей.

# Object для универсальных коллекций

```
class Stack
{
    private readonly object[] _data = new object[10];
    private int _index = 0;

    3 references | 0 changes | 0 authors, 0 changes
    public void Add(object value)
    {
        _data[_index] = value;
        ++_index;
    }

    1 reference | 0 changes | 0 authors, 0 changes
    public object Pop()
    {
        object result = _data[_index];
        _index--;
        return result;
    }
}
```

```
var stack = new Stack();
stack.Add("M1");
stack.Add("M2");
stack.Add("M3");
Console.WriteLine(stack.Pop());
```

А что насчёт Value types?

# Value types

- Тоже наследуется от object.
- А значит может использоваться в нашей коллекции.

# Открываем WinDbg

- Запускаем код **var boxed = (object)1;**
- `!DumpHeap -type Int32`
- Находим:
  - 00007ffc15f89480      1      24 System.Int32
- Идём глубже
  - `!dumpmt -md 00007ffc15f89480`



# Открываем WinDbg

```
0:013> !dumpmt -md 00007ffc15f89480
EEClass:          00007ffc15f74fa0
Module:           00007ffc15dd4000
Name:             System.Int32
mdToken:          000000000200014E
File:             C:\Program Files\dotnet\shared\Microsoft.NETCore.App\6.0.2\System.Private.CoreLib.dll
BaseSize:         0x18
ComponentSize:    0x0
DynamicStatics:   false
ContainsPointers  false
Slots in VTable:  115
Number of IFaces  in IFaceMap: 28
-----
MethodDesc Table
      Entry      MethodDesc      JIT Name
00007FFC15EF0030 00007ffc15ee5608 NONE System.Object.Finalize()
00007FFC15EF42E0 00007ffc15f885c0 NONE System.Int32.ToString()
00007FFC15EF42B0 00007ffc15f88548 NONE System.Int32.Equals(System.Object)
00007FFC15EF42D0 00007ffc15f88598 NONE System.Int32.GetHashCode()
00007FFC15EF4290 00007ffc15f884f8 NONE System.Int32.CompareTo(System.Object)
00007FFC15EF42A0 00007ffc15f88520 NONE System.Int32.CompareTo(Int32)
00007FFC15EF42C0 00007ffc15f88570 NONE System.Int32.Equals(Int32)
00007FFC15EF42F8 00007ffc15f88600 NONE System.Int32.ToString(System.IFormatProvider)
```

# Всё ещё универсальная коллекция

```
stack.Add(1);  
stack.Add(2);  
stack.Add(3);  
object result = stack.Pop();  
Console.WriteLine(result);
```

# ...НО С БОКСИНГОМ

```
stack.Add(1);  
stack.Add(2);  
stack.Add(3);  
object result = stack.Pop();  
Console.WriteLine(result);
```

■ readonly struct System.Int32  
Represents a 32-bit signed integer.

Boxing allocation: conversion from 'int' to 'object'  
requires boxing of value type

# JVM moment: object and struct

- Нет структур, нельзя объявить пользовательский не референсный тип.
- Примитивы отдельно от объектов.
- Integer / int
- Project Valhalla

# System.Enum

- Именованные инты.
- Чуть более понятные битовые операции.
- По дефолту нет проверки на безопасность типа.
- Значения без методов :с (но это поправимо!)

# Extension methods

- Тип, который объявлен вне и нет возможности его изменить.
- Метод интерфейса с реализацией дефолтной.
- Для енамов это возможность добавить методы.

```
public static class StudentExtensions
{
    public static string Hello(this Student student)
    {
        return student switch
        {
            Student.Ru => "Hello",
            Student.Other => "Соболезную",
        };
    }

    public static void F()
    {
        Console.WriteLine(Student.Ru.Hello());
    }
}
```

# JVM moment: System.Enum

```
enum Color {  
    Red,  
    Green,  
    Blue;  
  
    public char getFirstChar() {  
        return this  
            .toString()  
            .charAt(1);  
    }  
}
```

# Discriminated Unions

- DU - это сумма типов, в отличие от Tuple, который является произведением типов.

```
type Result =  
  | Success           // no string needed for success state  
  | ErrorMessage of string // error message needed
```



# Добавляем свой тип

```
public class GroupName
{
    4 references | 0 changes | 0 authors, 0 changes
    public int Course { get; }
    4 references | 0 changes | 0 authors, 0 changes
    public int Group { get; }
    4 references | 0 changes | 0 authors, 0 changes
    public int Specialization { get; }

    0 references | 0 changes | 0 authors, 0 changes
    public GroupName(int course, int group, int specialization)
    {
        Course = course;
        Group = group;
        Specialization = specialization;
    }
}
```

# ...и немного бойлеркода

```
1 reference | 0 changes | 0 authors, 0 changes
public static bool operator==(GroupName left, GroupName right)
{
    return left.Equals(right);
}

0 references | 0 changes | 0 authors, 0 changes
public static bool operator!=(GroupName left, GroupName right)
{
    return !(left == right);
}

2 references | 0 changes | 0 authors, 0 changes
protected bool Equals(GroupName other)
{
    return Course == other.Course
        && Group == other.Group
        && Specialization == other.Specialization;
}

0 references | 0 changes | 0 authors, 0 changes
public override bool Equals(object? obj)
{
    if (ReferenceEquals(null, obj)) return false;
    if (ReferenceEquals(this, obj)) return true;
    if (obj.GetType() != this.GetType()) return false;
    return Equals((GroupName)obj);
}

0 references | 0 changes | 0 authors, 0 changes
public override int GetHashCode()
{
    return HashCode.Combine(Course, Group, Specialization);
}
```

# Records

0 references | - changes | -authors, -changes

```
public record GroupNameRecord(int Course, int Group, int Specialization);
```

- public get, init-only set
- Deconstruct
- Всё то, что мы руками сделали для своего типа.

# Immutable types

## Props:

- Проще следить за состоянием объекта
- Нет проблем с тем, что мы завязались на состояние, которое изменилось (например, GetHashCode)

## Cons:

- Сложно реализовать в языках, где изначально нет иммутабельности
- Аллокации, перфоманс

# Immutable strings

- Строки - концептуально не изменяемый тип
- Операции над строками = создание новой строки
- StringBuilder

# Generics

# Generics

- Логика без привязки к типу аргументов

```
public void Swap<T>(ref T a, ref T b)
{
    (a, b) = (b, a);
}
```

# Generics

...но не всегда.

```
public void UseAndDispose<T>(T value)
{
    WriteLine(value.ToString());
    value.Dispose();
}
```



# Generics constraints

...но не всегда.

```
public void UseAndDispose2<T>(T value)
    where T : IDisposable
{
    WriteLine(value.ToString());
    value.Dispose();
}
```

# Используем дженерики: Stack<T>

```
class Stack
{
    private readonly object[] _data = new object[10];
    private int _index = 0;

    3 references | 0 changes | 0 authors, 0 changes
    public void Add(object value)
    {
        _data[_index] = value;
        ++_index;
    }

    1 reference | 0 changes | 0 authors, 0 changes
    public object Pop()
    {
        object result = _data[_index];
        _index--;
        return result;
    }
}
```

```
class Stack<T>
{
    private readonly T[] _data = new T[10];
    private int _index = 0;

    public void Add(T value)
    {
        _data[_index] = value;
        ++_index;
    }

    public T Pop()
    {
        T result = _data[_index];
        _index--;
        return result;
    }
}
```

# Используем дженерики: Stack<T>

```
stack.Add(1);  
stack.Add(2);  
stack.Add(3);  
int result = (int)stack.Pop();  
Console.WriteLine(result);
```

```
var genericStack = new Stack<int>();  
genericStack.Add(1);  
genericStack.Add(2);  
genericStack.Add(3);  
Console.WriteLine(genericStack.Pop());
```

# Используем дженерики: Stack<T>

```
11  
12 var stack = new Stack();    stack = {Stack}  
13 var genericStack = new Stack<int>();    genericStack = {Stack}  
14  
15 stack.Add(1);  
16 stack.Add(2);  
17 stack.Add(3);  
18  
19 genericStack.Add(1);  
20 genericStack.Add(2);  
21 genericStack.Add(3);  
22  
23 return 0;    ≤ 1ms elapsed  
24
```




0

Summary Events Mem

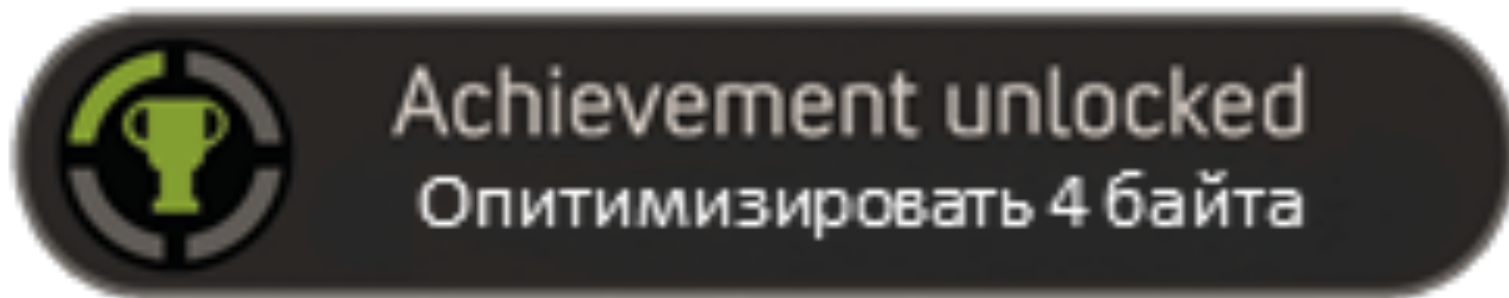
Take Snapshot View

ID	Time	Objects (Diff)
1	0.23s	3,974 ( n/a )
2	0.23s	3,978 (+4 ↑)
3	0.23s	3,978 (+0)

# Используем дженерики: Stack<T>

Managed Memory			Compare With Baseline: Snapshot #1
Filter types		<input checked="" type="checkbox"/> Show Just My Code	<input type="checkbox"/> Collapse small object types
Object Type	Count Diff. ▼	Size Diff. (Bytes)	
 Int32	+3	+72	
 Microsoft.Extensions.HotReload.HotReloadAgent	+1	+56	

Используем дженерики: `Stack<T>`



# JVM moment: Generics

- Нельзя использовать примитивы в дженериках
- Типы известны только до этапа компиляции, после они затираются

# Generics: итог

- Меньше дублирования кода
- Перфоманс
- Проверки типов во время компиляции



LINQ

# LINQ



# IEnumerable

- Базовый интерфейс для всех перечислений.
- IEnumerable / IEnumerator
- Большие иерархии: IEnumerable -> ICollection -> IList -> List

# First-class type

- Может быть параметром функции.
- Может быть возвращаемым значением функции.
- Может быть объектом присвоения.
- Может сравниваться.

# Delegates, Action, Func

- Делегат – это объект, который знает, как вызывать метод.
- Func – это заранее созданные делегаты.
- Action – это костыль над отсутствием Unit.

# Lambda expressions

- Лямбды – это делегаты.
- Вывод типов в шарпе работает не так хорошо, как хочется.
- Лямбды тесно связаны с механизмом замыкания.

# LINQ

- Generic логика работы с коллекциями.
- Экстеншен метод поверх базового интерфейса.
- Fluent interface.
- Отложенное выполнение.

# Pipelines в F#

- В функциональных языках намного проще реализовать цепочки вызовов.

```
let finalSeq =  
    seq { 0..10 }  
    ▷ Seq.filter (fun c -> (c % 2) = 0)  
    ▷ Seq.map ((* ) 2)  
    ▷ Seq.map (sprintf "The value is %i.")
```



# JVM moment: Stream API

- Менее гибкий инструмент ввиду отсутствия экстеншен методов.
- Ряд проблем, связанных со стримами для примитивов.

```
Integer odd = collection  
    .stream()  
    .filter(p -> p % 2 != 0)  
    .reduce((c1, c2) -> c1 + c2)  
    .orElse(0);
```