Технологии программирования

Лекция №3

ИС, весна 2022

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);
```

Уровни абстракции

Уровни абстракции

Какие уровни абстракции можно выделить?

- Машинный код
- ASM и подобное
- Обычные компилируемые языки общего назначения: C, C++, C#
- Domain specific languages etc.

Non-zero-cost abstraction

- Основная причина, по которой стоит опускаться на уровни абстракции ниже – это цена.
- Абстракции не zero-cost, за них нужно платить перфомансом.

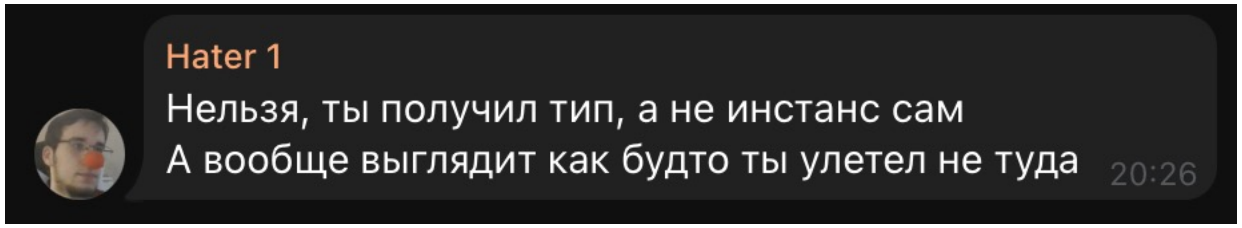
Reflection and codegen

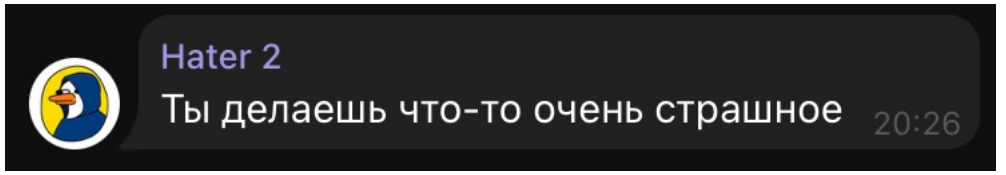
Поставим перед собой задачу

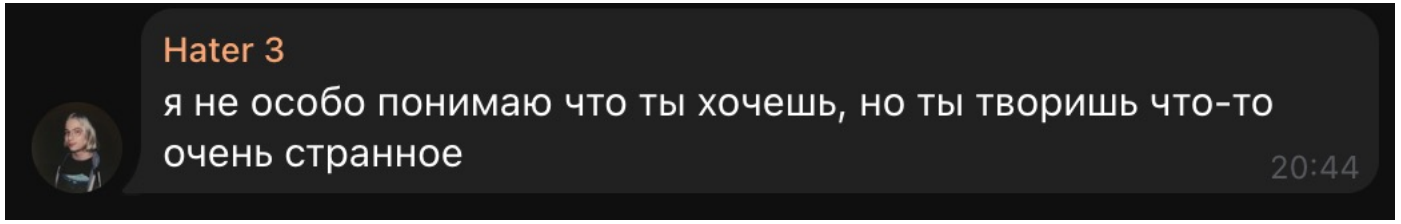
- Представим, что у нас есть задача:

Я хотел создать множество товаров, которое магазин определяет при создании. Через Assembly можно найти список классов, которые будут представлять каждый товар, они все помечены интерфейсом. У меня также есть Generic Класс, который будет существовать для каждого вида товаров, чтобы можно было удобно менять цену. Как раз таки на моменте с Generic я не могу дать ему знать о существовании товаров, потому что он принимает интерфейс, а не type

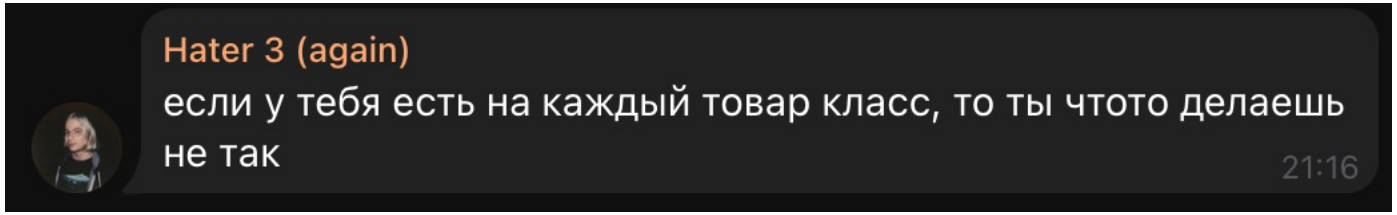
Хейтеры


Hater 1
Нельзя, ты получил тип, а не инстанс сам
А вообще выглядит как будто ты улетел не туда
20:26


Hater 2
Ты делаешь что-то очень страшное
20:26


Hater 3
я не особо понимаю что ты хочешь, но ты творишь что-то очень странное
20:44


Hater 4
Это очень плохо и совсем не объектно ориентированно. Я на 1000% уверен что прибегать к рефлексии не надо, то что ты хочешь можно сделать либо нормальными средствами, либо этого делать не стоит. Можешь ещё раз пояснить зачем тебе это?
21:13


Hater 3 (again)
если у тебя есть на каждый товар класс, то ты чтото делаешь не так
21:16

Проектируем решение

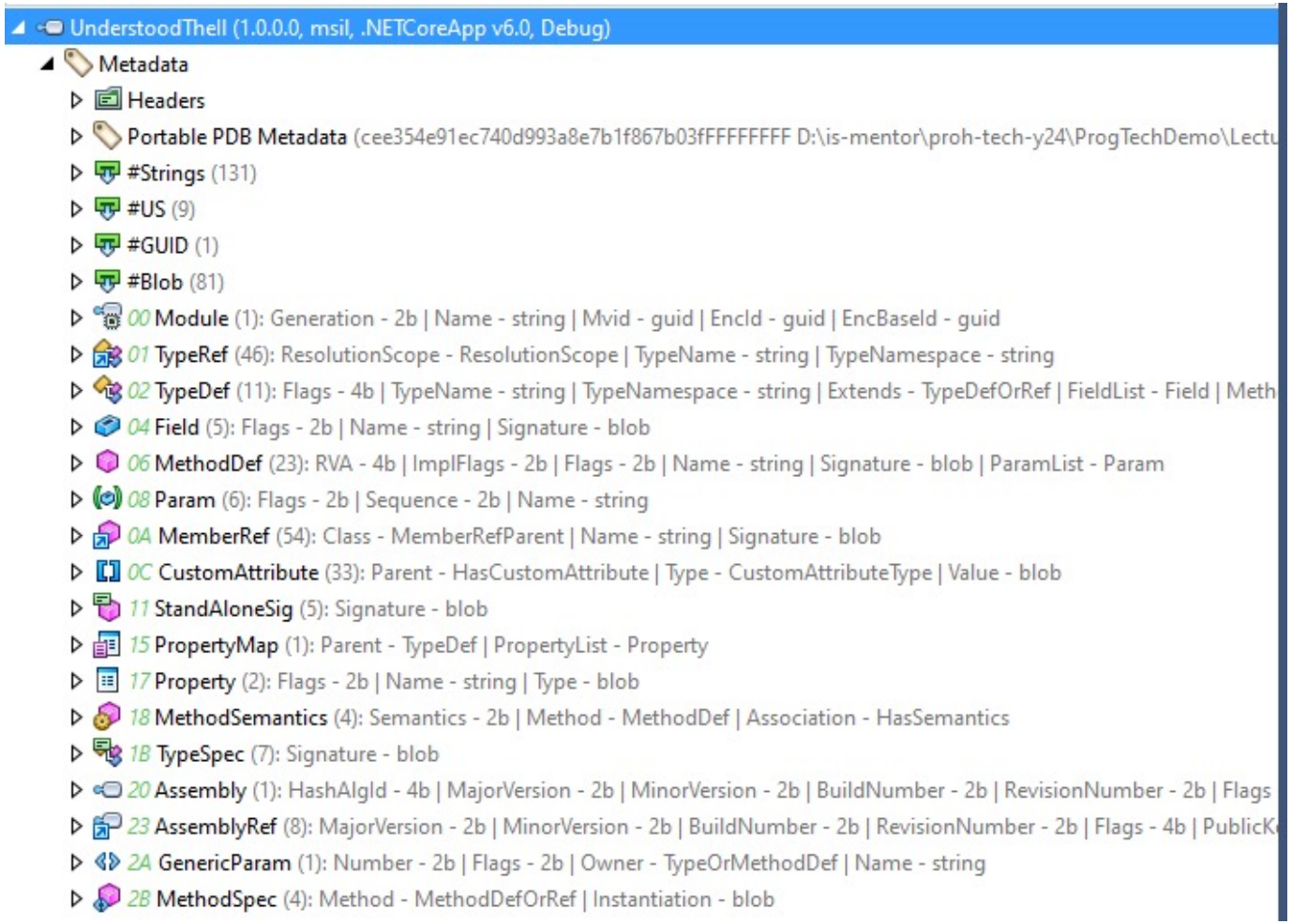
- Есть класс Item, который описывает то, что лежит в базе и возможность с базы получить эти данные.
- Есть интерфейс IShopItem, который должен быть реализован нашими продуктами.
- Ну и наш магазин, который это всё должен сгенерировать.

Assembly, метадата, манифест

- Assembly – это минимальная единица развёртывания .NET кода. Во время компиляции каждый проект собирается в свою Assembly. Любой nuget-пакет – это тоже Assembly, которую завернули в дополнительную обёртку.

- DLL можно вскрыть через ildasm или dotPeek и увидеть, что в нём описываются зависимости, версия, много другой информации про dll.

DLL в dotPeek

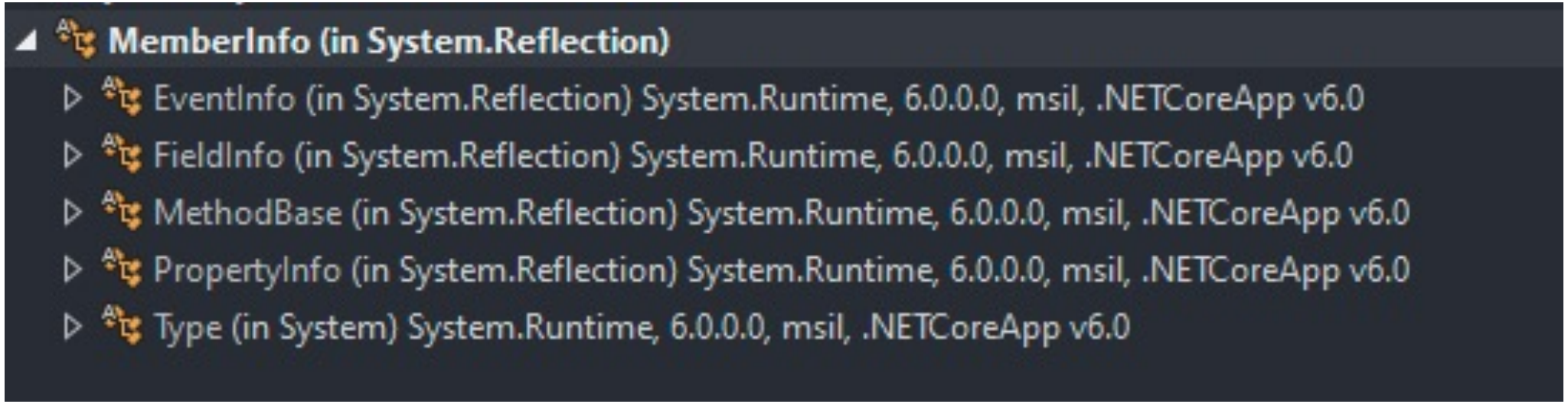


AssemblyBuilder, module builder

```
1   AssemblyName assemblyName =
2           Assembly.GetExecutingAssembly().GetName();
3   AssemblyBuilder assemblyBuilder =
4           AssemblyBuilder.DefineDynamicAssembly(assemblyName, AssemblyBuilderAccess.Run);
5   ModuleBuilder moduleBuilder =
6           assemblyBuilder.DefineDynamicModule(nameof(UnderTheHoodOfIl));
7
```

Object > MemberInfo > Type

- MemberInfo – это базовый класс, от которого наследуются другие типы, которые описывают метаинформацию о различных типах.



Object > MemberInfo > Type

- В Type можно найти много информации про данные:
  - Название, namespace, сборка;
  - Является ли структурой, классом, enum’ом, generic’ом;
  - Получить список полей, конструкторов, методов и пр.

TypeInfo, GetType, typeof

- TypeInfo в целом позволяет получить около такие же данные с той лишь разницей, что создание TypeInfo - это уже процесс парсинга метаданных и получения всей информации, а Type предоставляет методы получения этой информации.
- Самый простые способы получения Type – это метод GetType() у экземляров и кейворд typeof() для типов.

TypeBuilder

- Рассмотрим, как можно создать тип с использованием билдера.
- TypeBuilder:
  - Имя
  - Атрибуты (класс, структура, интерфейс, вложенный, приватный etc)
  - Базовый тип, реализованные интерфейсы
  - Набор методов Define* (например, DefineConstructor)

MethodInfo, MethodBuilder

- MethodInfo (внезапно!) описывает метаинформацию о методах.
- Например:
  - Название;
  - Атрибуты: приватный, публичный, виртуальный;
  - Типы аргументов и возвращаемого значения.

Вызов метода через MethodInfo

```
BindingFlags bindingFlags = BindingFlags.Instance | BindingFlags.NonPublic;

var callOverReflection = new CallOverReflection();
Type currentType = callOverReflection.GetType();
MethodInfo methodInfo = currentType.GetMethod(
    name: "PrivateMethod",
    bindingFlags)!;
methodInfo.Invoke(callOverReflection, parameters: Array.Empty<object?>());
```

IL, ILGenerator

Мы уже обсудили, что там внутри где-то всё работает с IL кодом. И вот мы дошли до момента, когда мы тоже можем пописать IL код!

Во время создания нашего типа мы объявили MethodBuilder и дошли до того, что можем получить ILGenerator этого метода и “реализовать” метод.

```
MethodAttributes methodAttributes =
    MethodAttributes.Public
    | MethodAttributes.Final
    | MethodAttributes.Virtual;

MethodBuilder methodBuilder = typeBuilder.DefineMethod(
    nameof(IShopItem.GetPrice),
    methodAttributes,
    typeof(int), Type.EmptyTypes);

ILGenerator ilGen = methodBuilder.GetILGenerator();
```

OpCodes

```
static int F(int i)
{
    int value = i * 11;
    Console.Write(value);
    return value;
}
```

```
.method assembly hidebysig static
    int32 '<<Main>$>g__F|0_0' (
        int32 i
    ) cil managed
{
    // Method begins at RVA 0x2059
    // Code size 11 (0xb)
    .maxstack 8

    IL_0000: ldarg.0
    IL_0001: ldc.i4.s 11
    IL_0003: mul
    IL_0004: dup
    IL_0005: call void [System.Console]System.Console::Write
    IL_000a: ret
} // end of method '<Program>$'::'<<Main>$>g__F|0_0'
```

Как создать экземпляр

- Представим ситуацию, когда мы знаем какой-то тип и нам нужно создать его экземпляр. Есть три варианта:
  - **where T : new()**
  - **System.Activator.CreateInstance()**
  - **FormatterServices.GetUninitializedObject()**

GetUninitializedObject

```
class ObjectWithCtor
{
    private readonly int _value;

    public static void Show()
    {
        new ObjectWithCtor().Write();
        var typeless :object  = FormatterServices
            .GetUninitializedObject(typeof(ObjectWithCtor));
        var value = (ObjectWithCtor)typeless;
        value.Write();
    }

    public ObjectWithCtor() => _value = 10;
    public bool Ok() => _value == 10;
    public void Write() => Console.WriteLine(_value);
}
```

Демо нашего магазина

Объявляем сигнатуры

```
public interface IShopItem
{
    public int GetPrice();
}

public class Item
{
    public string Name { get; set; }
    public int Price { get; set; }
}
```

Создаём билдеры

```
1   AssemblyName assemblyName =
2           Assembly.GetExecutingAssembly().GetName();
3   AssemblyBuilder assemblyBuilder =
4           AssemblyBuilder.DefineDynamicAssembly(assemblyName, AssemblyBuilderAccess.Run);
5   ModuleBuilder moduleBuilder =
6           assemblyBuilder.DefineDynamicModule(nameof(UnderTheHoodOfIl));
7
```

Создаём типы

```
foreach (Item item in items)
{
    TypeBuilder typeBuilder = moduleBuilder.DefineType(
        item.Name,
        attr: TypeAttributes.Public,
        parent: null,
        interfaces: new[] {typeof(IShopItem)});

    typeBuilder.DefineDefaultConstructor(
        MethodAttributes.Private);
```

Добавляем метод

```
MethodAttributes methodAttributes =
    MethodAttributes.Public
    | MethodAttributes.Final
    | MethodAttributes.Virtual;

MethodBuilder methodBuilder = typeBuilder.DefineMethod(
    nameof(IShopItem.GetPrice),
    methodAttributes,
    typeof(int), Type.EmptyTypes);

ILGenerator ilGen = methodBuilder.GetILGenerator();
```

Добавляем имплементацию

```
ILGenerator ilGen = methodBuilder.GetILGenerator();

ilGen.Emit(opcode: OpCodes.Ldc_I4, item.Price);
ilGen.Emit(opcode: OpCodes.Ret);
```

Создаём тип и экземпляр

```
Type? newType = typeBuilder.CreateType();
object? createTypeValue = FormatterServices
    .GetUninitializedObject(newType);
shopItems.Add((IShopItem)createTypeValue);
```

Получаем результат

```
List<Item> items = Database.GetItems();

var stillShopBut = new StillShopBut();
List<IShopItem> shopItems = stillShopBut.CreateShopItems(items);
foreach (IShopItem shopItem in shopItems)
{
    WriteLine($"{shopItem.GetType()} : {shopItem.GetPrice()}");
}
```

Codegen

Виды генерации

- Статик
  - Конвертация
  - CodeDom
  - T4
  - Анализаторы и кодфиксеры
- Компайл
  - Препоцессинг
  - Во время компиляции
- Рантайм
  - Работа с метамданными
  - Работа с байтами

Конвертация между языками

- Плюсы: уже всё написано, просто нужно вставить и запустить
- Минусы: очень сложно, скорее всего нормальной тулы нет или она не поддерживает почти всё, что есть в языке. Кейсы перевода языка странные, а выхлоп очень плохой.

T4

- Плюсы: очень наглядно, ещё до запуска неплохо так видно что будет генерироваться. Минимальный порог вхождения т.к. почти стринг билдер.
- Минусы: нужно самому отвечать за валидность сгенерированных данных, это просто стрингбилдер на максималках. Чтобы узнать о том, что забыл поставить скобочку в вызове метода, нужно запустить.