

Лекция 4

Рефлексия

Рефлексия

- Механизм рефлексии. Аннотации.
Методы, реализующие рефлексияю.

Обзор литературы

- <https://www.nookery.ru/understand-with-reflection/>
- <https://metanit.com/sharp/tutorial/14.1.php>
- <https://www.youtube.com/watch?v=vN6OXGQM1ac>
- <https://www.youtube.com/watch?v=i2W2wA-Udro>
- <https://blog.rc21net.ru/рефлексия-отражение-reflection-в-c-sharp/>

Атрибуты

- <https://www.youtube.com/watch?v=i2W2wA-Udro>
- <https://www.youtube.com/watch?v=4m3nAAekpdc>

Программа для декомпиляции

- <https://www.jetbrains.com/ru-ru/decompiler/>
- JetBrains.dotPeek.2020.3.3.web.exe (Бесплатная)

dotPeek

Что нового Возможности Скачать

ДЕКОМПИЛЯЦИЯ .NET

Декомпиляция сборок .NET в исходный код C#

dotPeek — это бесплатный самостоятельный инструмент, основанный на встроенном декомпиляторе **ReSharper**. Он надежно декомпилирует любую сборку .NET в эквивалентный код на C# или IL-код.

Декомпилятор поддерживает различные форматы файлов, включая библиотеки (.dll), исполняемые файлы (.exe) и файлы метаданных Windows (.winmd).

```
public override bool Execute()
{
    try
    {
        WorkMode workMode = (WorkMode) Enum.Parse(typeof(WorkMode), this.Mode);
        string metadata1 = this.InputFile.GetMetadata1("FullPath");
        if (string.IsNullOrEmpty(metadata1))
            throw new InvalidOperationException("The InputFile is required.");
        if (!File.Exists(metadata1))
            throw new InvalidOperationException(string.Format("The input file \"{0}\"", metadata1));
        string metadata2 = this.OutputFile.GetMetadata2("FullPath");
        if (string.IsNullOrEmpty(metadata2))
            throw new InvalidOperationException("The OutputFile is required.");
        return new ErrorDescriptionGenerator().Generate(workMode, metadata1, metadata2);
    }
    catch (Exception ex)
    {
        this.Log.LogError("Exception: " + (object) ex);
        this.Log.LogError(ex.StackTrace);
        return false;
    }
}
```

Рефлексия

Reflection (Отражение)



- **Отражение** (англ. *reflection*) — процесс, во время которого программа может отслеживать и модифицировать собственную структуру и поведение во время выполнения. Парадигма программирования, положенная в основу отражения, является одной из форм метапрограммирования.

[https://ru.wikipedia.org/wiki/Отражение_\(программирование\)](https://ru.wikipedia.org/wiki/Отражение_(программирование))

<https://www.youtube.com/watch?v=i2W2wA-Udro>

Рефлексия

- Манифест (Метаданные сборки) состоят из описания сборки: имя, версия, строгое имя, информация о культуре.
- Метаданные (Метаданные типов) включают пространство имен и имя типа, члены типа и параметры, если имеются.
- Ресурсы (Resources) – это объекты, которые используются кодом: строки, изображения, различные файлы.

Рефлексия

- Байт-код (псевдокод)—машинно – независимый код низкого уровня, генерируемый транслятором и исполняемый интерпретатором. Большинство инструкций байт-кода эквивалентны одной или нескольким командам ассемблера. Трансляция в байт-код занимает промежуточное положение между компиляцией в машинный код и интерпретацией.
- Байт-код называется так, потому что длина каждого кода операции — один байт, но длина кода команды различна. Каждая инструкция представляет собой однобайтовый код операции от 0 до 255, за которым следуют такие параметры, как регистры или адреса памяти.

Рефлексия

System.Reflection

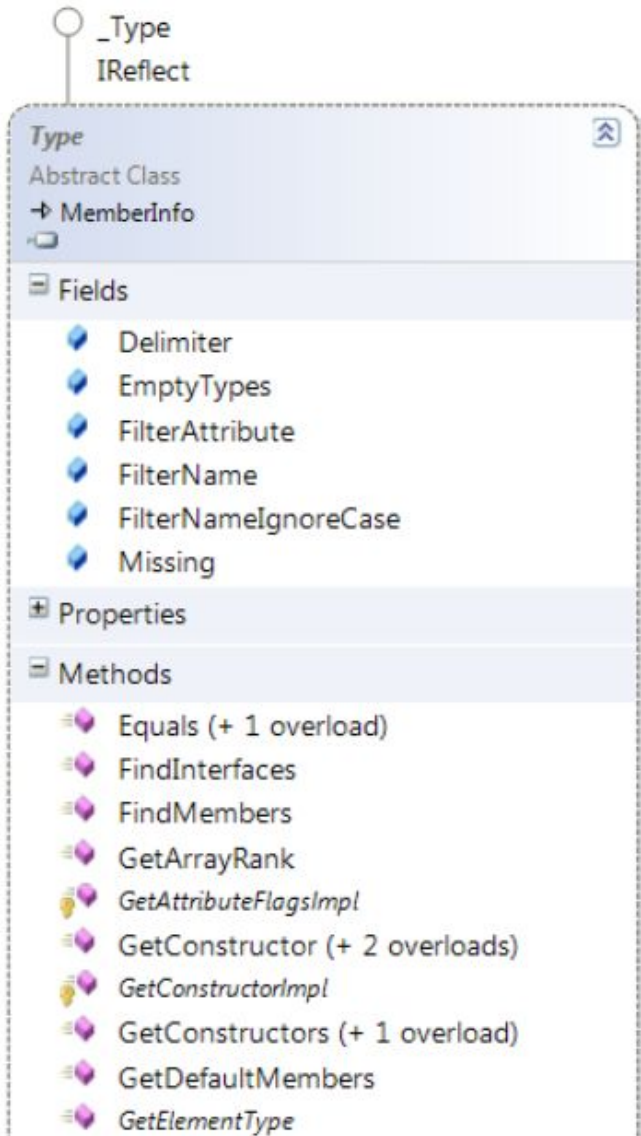
System.Reflection – пространство имен, которое содержит классы для применения рефлексии в языке C#.

<https://www.youtube.com/watch?v=vN6OXGQM1ac>

<https://docs.microsoft.com/ru-ru/dotnet/api/system.reflection?view=netframework-4.8>

Рефлексия

Класс Type



Type является корневым классом для функциональных возможностей рефлексии и основным способом доступа к метаданным.

С помощью членов класса **Type** можно получить сведения об объявленных в типе элементах: конструкторах, методах, полях, свойствах и событиях класса, а также о модуле и сборке, в которых развернут данный класс.

Рефлексия

Класс Type

Способы получения экземпляра

1. Вызов метода **GetType()** на экземпляре требуемого класса.
2. Вызов статического метода **GetType()** класса **Type**.
3. Использование оператора **typeof()**.

В приведенных выше примерах результатом будет ссылка на объект **Type**, содержащий информацию о целевом типе .

Рефлексия

Класс `Assembly`

Класс `Assembly` представляет собой сборку, которая является модулем с возможностью многократного использования, поддержкой версий и встроенным механизмом описания общезыковой исполняющей среды.

_Assembly
IEvidenceFactory
ICustomAttributeProvider
ISerializable

Assembly

Abstract Class

Properties

- CodeBase
- EntryPoint
- EscapedCodeBase
- Evidence
- FullName
- GlobalAssemblyCache
- HostContext
- ImageRuntimeVersion
- IsDynamic
- IsFullyTrusted
- Location
- ManifestModule
- PermissionSet
- ReflectionOnly
- SecurityRuleSet

Methods

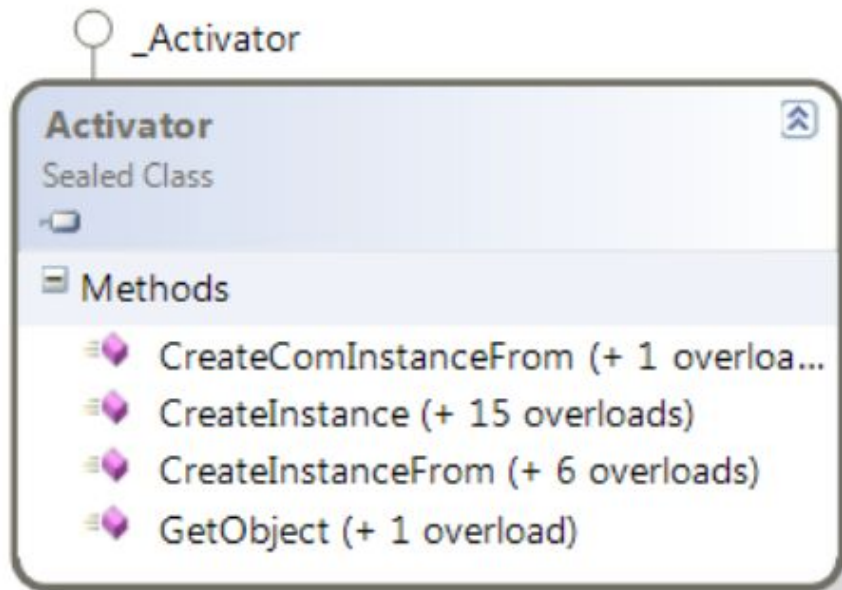
- Assembly
- CreateInstance (+ 2 overloads)
- CreateQualifiedName
- Equals
- GetAssembly

Рефлексия

Класс **Activator**

Класс **Activator** содержит методы для локального создания типов объектов.

Метод **CreateInstance()** создает экземпляр типа, определенного в сборке путем вызова конструктора, который наилучшим образом соответствует заданным аргументам.



Рефлексия

Reflection (Отражение)



- **Отражение** (англ. *reflection*) — процесс, во время которого программа может отслеживать и модифицировать собственную структуру и поведение во время выполнения. Парадигма программирования, положенная в основу отражения, является одной из форм метапрограммирования.

[https://ru.wikipedia.org/wiki/Отражение_\(программирование\)](https://ru.wikipedia.org/wiki/Отражение_(программирование))

<https://www.youtube.com/watch?v=i2W2wA-Udro>

Пример 1

```
using System;
using System.Reflection;

namespace Console_Reflection_Lesson_1
{
    class Program
    {
        static void Main(string[] args)
        {
            Type type = Type.GetType("Console_Reflection_Lesson_1.User");
            var members = type.GetMembers();

            foreach (MemberInfo memberInfo in members)
            {
                Console.WriteLine(memberInfo.Name);
            }
            Console.ReadLine();
        }
    }
}
```

```
Ссылки: 0
class User
{
    private int temp = 7;
    Ссылки: 0
    public string Name { get; set; }
    Ссылки: 0
    public string Age { get; set; }
}
```

```
get_Name
set_Name
get_Age
set_Age
Equals
GetHashCode
GetType
ToString
.ctor
Name
Age
```

Пример 1

```
get_Name  
set_Name  
get_Age  
set_Age  
Equals  
GetHashCode  
GetType  
ToString  
.ctor  
Name  
Age
```

Индексаторы get, set

**4 стандартных метода,
являющихся общими для
всех типов данных**

Конструктор класса

Свойства

Метод GetType

Метод **GetType** – возвращает текущий Type.

Перегрузки

GetType ()	Возвращает текущий Type.
GetType (String)	Возвращает объект Type с указанным именем, учитывая при поиске регистр.
...	...

Возвращаемое значение

- Type Тип с указанным именем, если он существует;
- в противном случае — значение null.

Метод GetMembers

Метод **GetMembers** – получает члены (свойства, методы, поля, события и т. д.) текущего объекта **Type**.

Перегрузки

GetMembers()	Возвращает все открытые члены текущего объекта Type .
GetMembers(BindingFlag)	При переопределении в производном классе ищет члены, определенные для текущего объекта Type , используя указанные ограничения привязки.
<ul style="list-style-type: none">• Массив объектов MemberInfo, представляющий все открытые члены текущего типа Type.• -или- Пустой массив типа MemberInfo, если у текущего типа Type нет открытых членов.	

Пример 2

- Добавим в метод `GetMembers`, два флага
 - `NonPublic` - все непубличные
 - `Instance` - выбираем не статичные элементы, а типы которые представляют собой экземплярные элементы

```
var members = type.GetMembers(BindingFlags.NonPublic|BindingFlags.Instance);
```

- [BindingFlags](#) Для определения элементов, включаемых в поиск, можно использовать следующие флаги фильтра:
- Укажите `BindingFlags.Instance`, чтобы включить методы экземпляра.
- Укажите `BindingFlags.Static`, чтобы включить статические методы.
- Укажите `BindingFlags.Public`, чтобы включить в поиск открытые методы.
- Укажите `BindingFlags.NonPublic` для включения в поиск неоткрытых методов (т. е. закрытых, внутренних и защищенных методов). Возвращаются только защищенные и внутренние методы базовых классов. закрытые методы в базовых классах не возвращаются.
- Укажите, `BindingFlags.FlattenHierarchy` следует `public` ли включать и `protected` статические элементы вверх по иерархии; `private` статические члены в наследуемых классах не включаются.
- Укажите `BindingFlags.Default` отдельный, чтобы вернуть пустой [MethodInfo](#) массив.
- [BindingFlags](#) Для изменения работы поиска можно использовать следующие флаги модификаторов:
- `BindingFlags.DeclaredOnly` значение, чтобы искать только члены, объявленные в [Type](#), а не члены, которые были просто унаследованы.

Пример 2

- Добавим в метод GetMembers, два флага
 - NonPublic - все непубличные
 - Instance - выбираем не статичные элементы, а типы которые представляют собой экземплярные элементы

```
var members = type.GetMembers(BindingFlags.NonPublic|BindingFlags.Instance);
```

```
Finalize  
MemberwiseClone  
_temp  
<Name>k__BackingField  
<Age>k__BackingField
```

Защищенные элементы
System.object

```
private int _temp = 7;
```

Поля которые скрывают
значения свойства

Способы получения

экземпляра

```
//1 - способ получения экземпляра класса Type
///метаописание типа по строковому наименованию
Type type1 = Type.GetType("Console_Reflection_Lesson_1.User");
Console.WriteLine("1-й способ: " + type1);
var user = new User();
//2 - способ получения экземпляра класса Type
///метаописание типа из объекта реализующий тип, вызывая этот тип методом GetType
type1 = user.GetType();
Console.WriteLine("2-й способ: " + type1);
//3 - способ получения экземпляра класса Type
///метаописание типа по самому типу
type1 = typeof(User);
Console.WriteLine("3-й способ: " + type1);
```

```
1-й способ: Console_Reflection_Lesson_1.User
2-й способ: Console_Reflection_Lesson_1.User
3-й способ: Console_Reflection_Lesson_1.User
```

Пример 3

Теперь попробуем менять с помощью метаданных сами данные

```
private static void Main(string[] args)
{
    User user = new User();
    Type type = user.GetType();
    var fields = type.GetFields(BindingFlags.NonPublic | BindingFlags.Instance);
    foreach (FieldInfo fieldInfo in fields)
    {
        if (fieldInfo.Name=="_temp")
        {
            var value = fieldInfo.GetValue(user);
            Console.WriteLine("До: {0}", value );

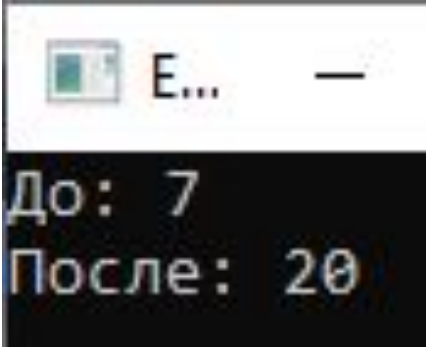
            fieldInfo.SetValue(user, 20);

            value = fieldInfo.GetValue(user);
            Console.WriteLine("После: {0}", value);
        }
    }
    Console.ReadLine();
}
```

Ссылка: 0
class User

```
{
    private int temp = 7;
    Ссылка: 0
    public string Name { get; set; }
    Ссылка: 0
    public string Age { get; set; }
}
```

С помощью механизма рефлексии мы изменили private поля.
Но этим не следует злоупотреблять!!!



```
E...
До: 7
После: 20
```

Метод GetField

Метод **GetField** – возвращает поля текущего объекта **Type**.

Перегрузки

GetFields ()

Возвращает все открытые поля текущего объекта **Type**.

GetFields
(BindingFlags)

При переопределении в производном классе ищет поля, определенные для текущего объекта **Type**, используя указанные ограничения привязки.

Пример 4

Рассмотрим пример создания
новых объектов с помощью
рефлексии

В итоге получаем экземпляр
класса User, но он у нас будет
не типизированным:
`object user`.

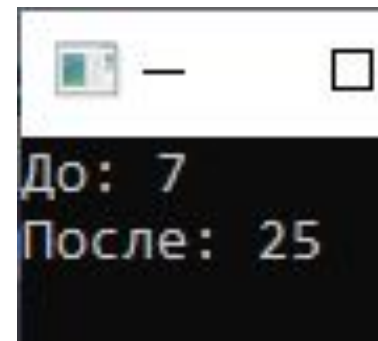
```
Ссылка: 0
class Program
{
    2
    Ссылка: 0
    static void Main(string[] args)
    {
        //метаописание типа по самому типу
        Type type = typeof(User);

        ConstructorInfo constructorInfo = type.GetConstructor(new Type[] { });
        object user = constructorInfo.Invoke(new object[] { });

        var fields = type.GetFields(BindingFlags.NonPublic|BindingFlags.Instance);

        foreach (FieldInfo fieldInfo in fields)
        {
            if (fieldInfo.Name == "_temp")
            {
                var value = fieldInfo.GetValue(user);
                Console.WriteLine("До: {0}", value);
                fieldInfo.SetValue(user, 25);
                value = fieldInfo.GetValue(user);
                Console.WriteLine("После: {0}", value);
            }
        }
        Console.ReadLine();
    }
}

class User
{
    private int temp = 7;
    Ссылка: 0
    public string Name { get; set; }
    Ссылка: 0
    public string Age { get; set; }
}
```



```
До: 7
После: 25
```


Пример 4

`class System.Reflection.ConstructorInfo`

Обнаруживает атрибуты конструктора класса и предоставляет доступ к метаданным конструктора.

`ConstructorInfo` `Type.GetConstructor(Type[] types)` (+ 2 перегрузки)

Выполняет поиск открытого конструктора экземпляра, параметры которого соответствуют типам, содержащимся в указанном массиве.

★ Предложение IntelliCode на основе этого контекста

```
ConstructorInfo constructorInfo = type.GetConstructor(new Type[] { });
```

Получаем метаописание конструктора, нашего класса. Поскольку конструктор класса `User` пустой, передаем пустой массив.

`object` `ConstructorInfo.Invoke(object[] parameters)` (+ 3 перегрузки)

Вызывает конструктор, который определяется экземпляром с указанными параметрами, чтобы этим параметрам присваивались стандартные значения, которые используются нечасто.

★ Предложение IntelliCode на основе этого контекста

Примечание. Два раза нажмите клавишу TAB, чтобы вставить фрагмент кода "Invoke".

```
object user = constructorInfo.Invoke(new object[] { });
```

Вызываем реальный конструктор. Поскольку конструктор без параметров, передаем пустой массив.

`class System.Type`

Представляет объявления типов для классов, интерфейсов, массивов, значений, перечислений параметров, определений указанном источнике.

универсальных типов и открытых или закрытых сконструированных универсальных типов. Исходный код .NET Framework для этого типа см. в

Рефлексия

Reflection (Отражение)

Отражение

Атрибуты



Механизм при помощи которого разработчик сообщает дополнительную информацию о том или ином объекте называется **атрибутом**.

Пример 5

Рассмотрим пример с атрибутами

```
class MySimpleAttribute: Attribute
{
}
```

Ссылка: 0

```
static void Main(string[] args)
```

```
{
    User user = new User();
    Type type = user.GetType();
```

```
    var properties = type.GetProperties();
```

```
    foreach (PropertyInfo propertyInfo in properties)
```

```
    {
        var attributes = propertyInfo.GetCustomAttributes(typeof(MySimpleAttribute), false);
        if (attributes.Length > 0)
            Console.WriteLine(propertyInfo.Name);
    }
```

```
    Console.ReadKey();
}
```

Ссылка: 0

```
class User
```

```
{
    private int _temp = 7;
    Ссылка: 0
    public string Name { get; set; }
```

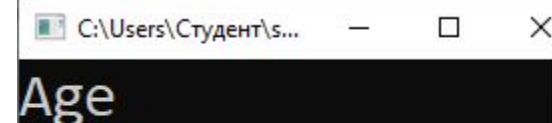
```
    [MySimple]
```

Ссылка: 0

```
    public string Age { get; set; }
```

```
}
```

Атрибут сам по себе ни какой нагрузки не несет. Он расширяет таблицу метаданных.



C:\Users\Студент\s... — □ ×
Age

Пример 6

Рассмотрим усложненный пример с атрибутам, с включением дополнительного свойства

```
class MySimpleAttribute: Attribute
{
    Ссылка: 2
    public int Number { get; set; }
}
```

```
static void Main(string[] args)
{
    User user = new User();
    Type type = user.GetType();
    // получаем описание всех свойств
    var properties = type.GetProperties();
    foreach (PropertyInfo propertyInfo in properties)
    {
        var attributes = propertyInfo.GetCustomAttributes(typeof(MySimpleAttribute), false);
        if (attributes.Length > 0)
        {
            var attributte = (MySimpleAttribute)attributes[0];
            Console.WriteLine("Property name: {0}, attribute value: {1}", propertyInfo.Name, attributte.Number);
        }
    }
    Console.ReadKey();
}
```

Ссылка: 2

```
class User
{
    private int temp = 7;
    Ссылка: 0
    public string Name { get; set; }

    [MySimple(Number = 5)]
    Ссылка: 0
    public string Age { get; set; }
}
```

C:\Users\Студент\source\repos\Console_Reflection_Lesson_5\Console

Property name: Age, attribute value: 5

Пример 7

Ограничим область действия атрибута.

По умолчанию область действия атрибута распространяется к чему угодно (класс, методы, свойства, ...)

```
//задаем область действия атрибута только к свойству
```

```
[AttributeUsage(AttributeTargets.Property)]
```

```
Ссылка: 3
```

```
class MySimpleAttribute: Attribute
```

```
{
```

```
    Ссылка: 2
```

```
    public int Number { get; set; }
```

```
}
```

Рассмотреть примеры

