ЛЕКЦИЯ 7

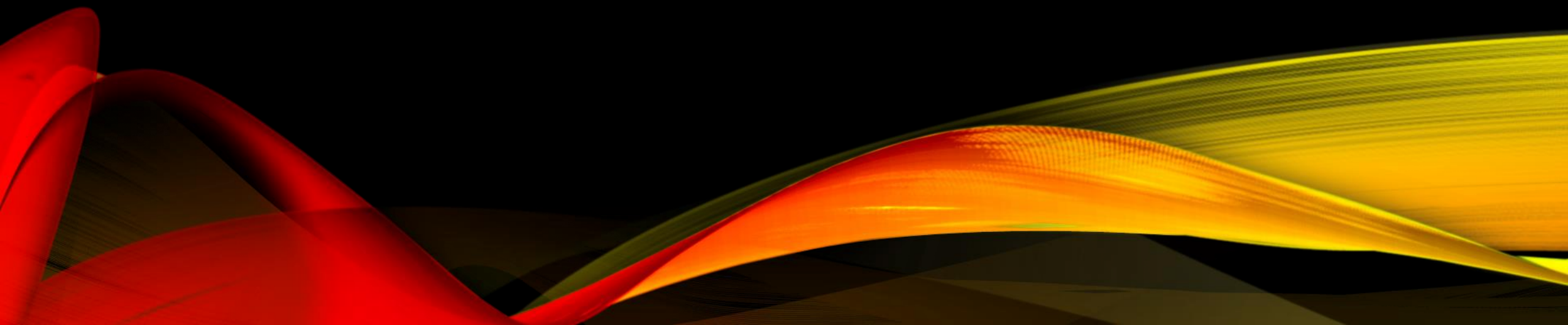
- Модуль 3
- 31.01.2024
- Основы рефлексии типов
- Атрибуты данных

ЦЕЛИ ЛЕКЦИИ

- Получить представление о рефлексии (отражении)

ОТРАЖЕНИЕ

Reflection



ОТРАЖЕНИЕ

- **Метаданные** – это данные о программах и используемых в них типах, хранимые в скомпилированных сборках

- **Отражение (рефлексия)** – механизм, позволяющий программе во время своего исполнения считывать метаданные сборок (чужих и своих)
- Классы по работе с отражением содержатся в пространстве имен `System.Reflection`

ПРЕИМУЩЕСТВА ИСПОЛЬЗОВАНИЯ ОТРАЖЕНИЯ

- можно создавать типы и вызвать их методы без предыдущих знаний об именах, которые помещаются в этих типах
  - динамическая идентификация типов
- не нужно на этапе компиляции знать информацию о типе, из которого будут получены метаданные, достаточно название типа
- представление типов, которые извлекаются из сборки в виде удобной объектной модели

КЛАСС TYPE

```
public abstract class Type : System.Reflection.MemberInfo, System.Reflection.IReflect
```

System.Type – основа всего отражения

ПРОГРАММНЫЕ СПОСОБЫ НАЗНАЧЕНИЯ ПЕРЕМЕННОЙ TYPE

1. Использование метода `System.Object.GetType()`
   - метод возвращает метаданные текущего объекта типа
2. Использование оператора `typeof()`
   - Объект не требуется, достаточно иметь объявленный тип
3. Использование статического метода `System.Type.GetType()`
   - Не нужно объявлять тип, информацию о котором нужно получить, достаточно знать только имя этого типа

ПРОГРАММНЫЕ СПОСОБЫ ПОЛУЧЕНИЯ ИНФОРМАЦИИ О ТИПЕ

Использование метода `System.Object.GetType()`

- метод возвращает метаданные текущего объекта типа

```
Point p = new Point(5,5);
Type testedType = p.GetType();

MethodInfo[] pointMI = testedType.GetMethods();
foreach (var method in pointMI)
{
    Console.WriteLine(method.Name);
}
```

Объект создан

```
get_IsEmpty
get_X
set_X
get_Y
set_Y
op_Implicit
op_Explicit
op_Addition
op_Subtraction
op_Equality
op_Inequality
Add
Subtract
Ceiling
Truncate
Round
Equals
Equals
GetHashCode
Offset
Offset
ToString
GetType
```

Информация может быть получена только об открытых полях и методах

ПРОГРАММНЫЕ СПОСОБЫ ПОЛУЧЕНИЯ ИНФОРМАЦИИ О ТИПЕ

Использование средства `typeof()`

- Объект не требуется, достаточно иметь объявленный тип

```
Type testedType = typeof(Point);

MethodInfo[] pointMI = testedType.GetMethods();
foreach (var method in pointMI)
{
    Console.WriteLine(method.Name);
}
```

Передаём тип

```
get_IsEmpty
get_X
set_X
get_Y
set_Y
op_Implicit
op_Explicit
op_Addition
op_Subtraction
op_Equality
op_Inequality
Add
Subtract
Ceiling
Truncate
Round
Equals
Equals
GetHashCode
Offset
Offset
ToString
GetType
```

Информация может быть получена только об открытых полях и методах

ПРОГРАММНЫЕ СПОСОБЫ ПОЛУЧЕНИЯ ИНФОРМАЦИИ О ТИПЕ

Использование статического метода `System.Type.GetType()`

- Не нужно объявлять тип, информацию о котором нужно получить, достаточно знать только имя этого типа

Передаём имя типа

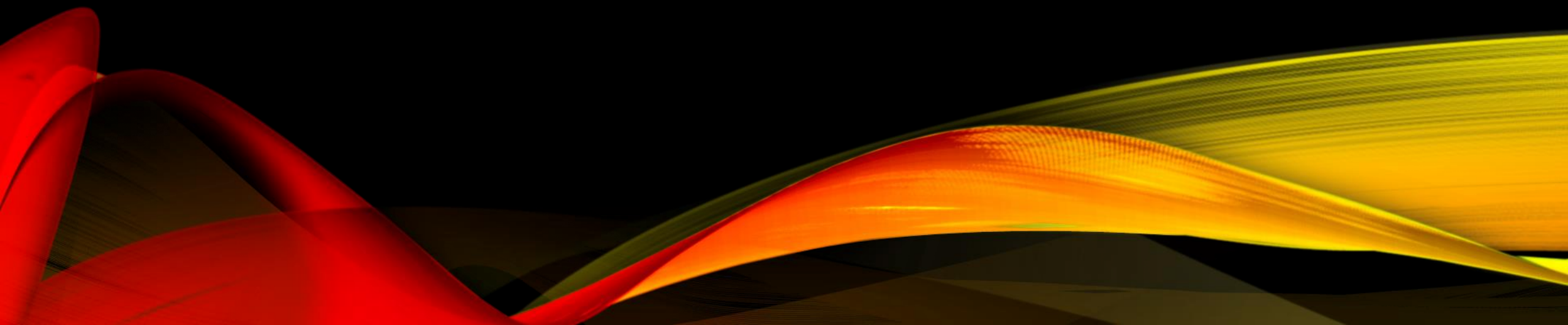
```
Type t = Type.GetType("System.Int32");

MethodInfo[] mi = t.GetMethods();
foreach (var method in mi)
{
    Console.WriteLine(method.Name);
}
```

```
CompareTo
CompareTo
Equals
Equals
GetHashCode
ToString
ToString
ToString
ToString
TryFormat
Parse
Parse
Parse
Parse
Parse
TryParse
TryParse
TryParse
TryParse
GetTypeCode
GetType
```

Информация может быть получена только об открытых полях и методах

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ РЕФЛЕКСИИ



ТИП ДАННЫХ ДЛЯ ЭКСПЕРИМЕНТОВ

```
public class User
{
    string name;
    public string Name { get => name; set => name = value; }
    // public User() { }
    public User(string? name) => this.Name = name ?? "Noname";
}
```

В примерах далее этот тип расположен в том же пространстве имён, что и тестовый код.

Самостоятельно разместите его в другом пространстве имён и поработайте с тестовым кодом примеров

УПРАВЛЕНИЕ ПОЛУЧАЕМЫМИ ДАННЫМИ

```
Type type = Type.GetType("User");
// Получение открытых полей и свойств
var usMembers = type?.GetMembers();

Console.WriteLine("Public members::");
foreach (MemberInfo mi in usMembers)
{
    Console.WriteLine(mi.Name);
}
```

```
// Получение закрытых членов типа
var hidUsMembers = type?.GetMembers(BindingFlags.NonPublic | BindingFlags.Instance);

Console.WriteLine("NonPublic members::");
foreach (MemberInfo mi in hidUsMembers)
{
    Console.WriteLine(mi.Name);
}
```

ЗАМЕНА ДАННЫХ

```
//User user = new User("Ivan");
//Type userType = user.GetType();
```

При использовании этого фрагмента регион ниже нужно закомментировать или удалить

```
#region createObject
// Getting type's metadata from the type
Type userType = typeof(User);
// Getting constructor from metadata of type
ConstructorInfo? ci = userType?.GetConstructor(new Type[] { typeof(string) });
// Invoke constructor
object? user = ci?.Invoke(new object[] { "Petr" });
#endregion
```

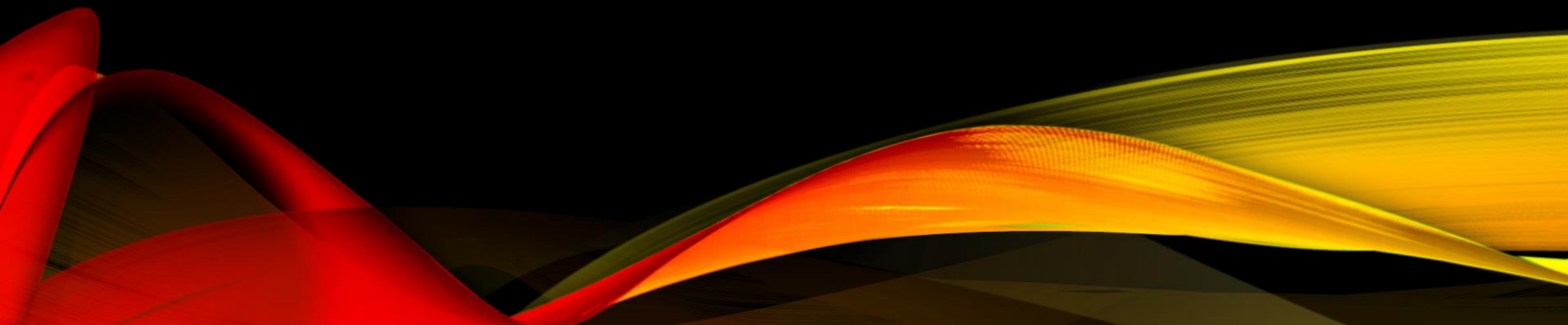
В этом регионе код создаёт объект типа, конструктор которого получен через рефлексию

```
var userFields = userType?.GetFields(BindingFlags.Instance | BindingFlags.NonPublic);
foreach (FieldInfo fi in userFields)
{
    if (fi.Name == "name")
    {
        var nameVal = fi.GetValue(user);
        Console.WriteLine($"Getted value:: {nameVal}");

        fi.SetValue(user, "Semen");
        nameVal = fi.GetValue(user);
        Console.WriteLine($"Getted value:: {nameVal}");
    }
}
```

АТРИБУТЫ

Attributes



АТРИБУТ

- **Атрибут** – это языковая конструкция, позволяющая добавлять метаданные к программной сборке
  - Метаданные – это «данные, описывающие данные»

```
// Пример применения атрибута к типу данных.
[ Serializable ]
public class MyClass
{ ...
```

- Атрибут **Serializable** – указывает на возможность сериализации объекта, к которому применён данный атрибут
- **NonSerialized** – указывает на невозможность сериализации

КОНФИГУРИРОВАНИЕ КЛАССОВ

- Атрибут `Serializable` **не наследуется**
- Атрибут `NonSerialized` **наследуется**

Атрибут – механизм (способ) сообщения дополнительной информации

[Serializable]

```
[Serializable]
class Student
```

[NonSerialized]

```
[NonSerialized]
private string Surname;
```

ЧТО ТАКОЕ АТРИБУТ (ATTRIBUTE)?

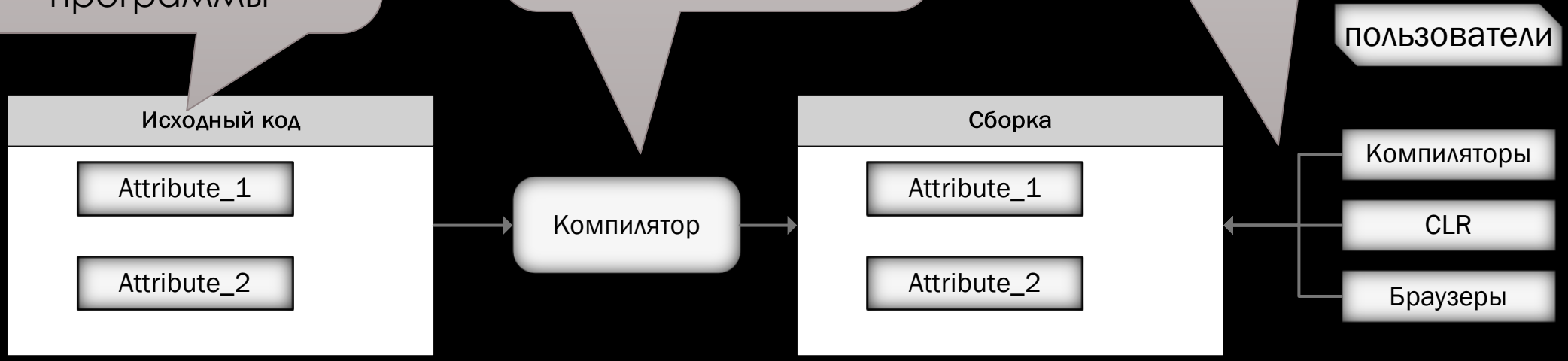
Способ получения знаний о существующем коде

Добавление специальной информации в код

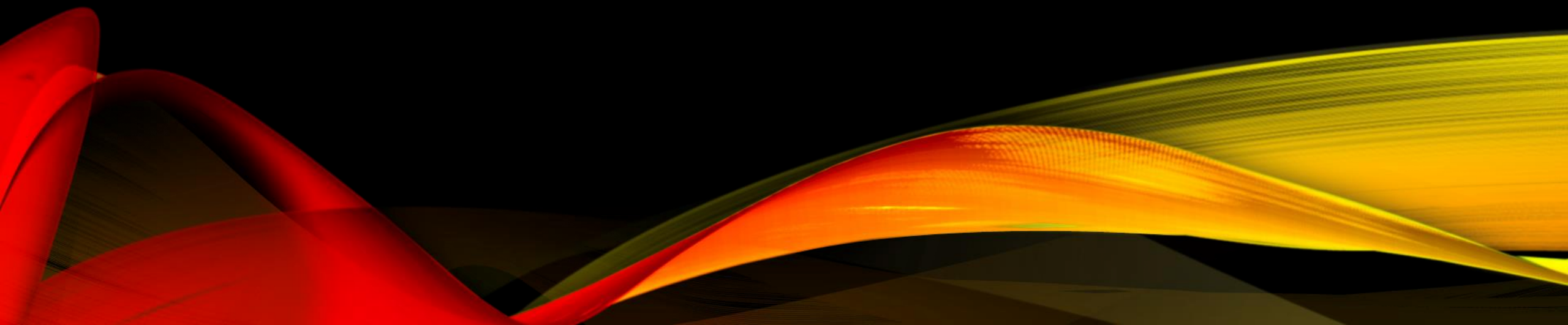
1. Применение атрибута (библиотечного или самописного) к коду программы

2. Компилятор добавляет метаданные об атрибутах в сборку

3. Программа во время выполнения может анализировать метаданные



ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ АТРИБУТОВ



ТИП ДАННЫХ ДЛЯ ЭКСПЕРИМЕНТОВ

```
class Segment
{
    public double A { get; set; }
    public double B { get; set; }
    [Demo]
    //[Demo(Color = ConsoleColor.Red)] // [DemoAttribute]

    public double Lenght { get => B - A; }
}
```

Для некоторых примеров этот код необходимо раскомментировать и использовать вместо стоящего выше

В примерах далее этот тип расположен в том же пространстве имён, что и тестовый код.

Самостоятельно разместите его в другом пространстве имён и поработайте с тестовым кодом примеров

ПРИМЕР КЛАССА, ОПИСЫВАЮЩЕГО АТРИБУТ

```
// User's attribute example
class DemoAttribute : Attribute
{
    #region attrMod
    //public ConsoleColor Color { get; set; }
    #endregion
}
```

Для некоторых примеров этот код необходимо раскомментировать

ПРИМЕР ИСПОЛЬЗОВАНИЯ АТРИБУТА БЕЗ ЗНАЧЕНИЯ

```
Segment segment = new Segment();
Type segmentType = segment.GetType();

// get all properties from Segment
var props = segmentType.GetProperties();

foreach(PropertyInfo pi in props)
{
    var attrs = pi.GetCustomAttributes(typeof(DemoAttribute),false);

    Console.WriteLine(attrs.Length !=0 ? pi.Name : "Nothing");
}
```

ПРИМЕР ИСПОЛЬЗОВАНИЯ АТРИБУТА СО ЗНАЧЕНИЕМ

```
Segment segment = new Segment();
Type segmentType = segment.GetType();

// get all properties from Segment
var props = segmentType.GetProperties();

foreach(PropertyInfo pi in props)
{
    var attrs = pi.GetCustomAttributes(typeof(DemoAttribute),false);
    Console.WriteLine(attrs.Length !=0 ? pi.Name : "Nothing");

    if (attrs.Length > 0)
    {
        var segAttr = (DemoAttribute)attrs[0];
        Console.ForegroundColor = segAttr.Color;
        Console.WriteLine($"{pi.Name}");
        Console.ResetColor();
    }

}
```

Значения атрибутов – константы и назначаются во время компиляции, изменить во время исполнения их нельзя

Чтобы не было ошибок компиляции, надо заменить атрибут в описании типа Segment

РЕКОМЕНДАЦИИ ДЛЯ ПОЛЬЗОВАТЕЛЬСКИХ АТРИБУТОВ

- Класс атрибута должен представлять некоторое состояние цели, к которой он применяется
- Если для атрибута требуются поля, добавьте параметрический конструктор для сбора значений (в него же добавьте опциональные параметры с умалчиваемыми значениями, если требуется)
- Не реализовывайте публичные методы или другие функциональные члены (за исключением свойств)
- В целях безопасности объявите класс атрибута опечатанным (`sealed`)
- Используйте атрибут `AttributeUsage` при объявлении собственного атрибута, чтобы явно указать множество целей вашего атрибута

ИСПОЛЬЗОВАНИЕ НЕСКОЛЬКИХ АТРИБУТОВ

- Стековый вариант использования
  - `[Serializable] // Stacked`
  - `[MyAttribute("Simple class", "Version 3.57")]`

- Вариант перечисления через запятую
  - `[MyAttribute("Simple class", "Version 3.57"), Serializable] // Comma separated`

ПРИМЕНЕНИЕ К ПОЛЯМ И МЕТОДАМ

- Применение к полям
  - `[MyAttribute("Holds a value", "Version 3.2")]`
  - `public int MyField;`
- Применение к методам
  - `[Obsolete]`
  - `[MyAttribute("Prints out a message.", "Version 3.6")]`
  - `public void PrintOut()`
  - `{`
  - `...`
  - `}`

ЯВНОЕ УКАЗАНИЕ ЦЕЛИ АТРИБУТА

Глобальные:

event	field
method	param
property	return
type	typevar
assembly	module

```
[method: MyAttribute("Prints out a message.", "Version 3.6")]
[return: MyAttribute("This value represents ...", "Version 2.3")]
public long ReturnSetting()
{
    ...
}
```

ГЛОБАЛЬНЫЕ АТРИБУТЫ

```
// Содержимое файла AssemblyInfo.cs

[assembly: AssemblyTitle("SuperWidget")]
[assembly: AssemblyDescription("Implements the SuperWidget product.")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("McArthur Widgets, Inc.")]
[assembly: AssemblyProduct("Super Widget Deluxe")]
[assembly: AssemblyCopyright("Copyright © McArthur Widgets 2012")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
```

ПОЛЬЗОВАТЕЛЬСКИЕ АТРИБУТЫ

```
                имя атрибута
                     ↓
public sealed class MyAttributeAttribute : System.Attribute
{                                ↑                    ↑
...                          суффикс          базовый класс
```

Типичные члены класса-атрибута:

- поля
- свойства
- конструкторы

СОЗДАНИЕ КОНСТРУКТОРОВ КЛАССА-АТРИБУТА

```
public MyAttributeAttribute(string desc, string ver)
{
   Description = desc;
   VersionNumber = ver;
}
```

Допускается:

- перегрузка конструкторов
- пустой конструктор добавляется автоматически, если не указано ни одного конструктора

ПРИМЕНЕНИЕ КОНСТРУКТОРОВ

```
[MyAttribute("Holds a value")]   // конструктор с одним параметром
public int MyField;

[MyAttribute("Version 1.3", "Galen Daniel")]   // конструктор с двумя параметрами
public void MyMethod()
{ ...
```

Использование конструктора без параметров:

```
[MyAttr]
class SomeClass ...

[MyAttr()]
class OtherClass ...
```

Аргументы конструкторов должны быть известны в момент компиляции (константы)!

ИМПЕРАТИВНЫЙ VS. ДЕКЛАРАТИВНЫЙ СТИЛИ

Императивный стиль выражения

- `MyClass mc = new MyClass(“Hello”, 15);`

Декларативный стиль выражения

- `[MyAttribute(“Holds a value”)]`

ПАРАМЕТРЫ КОНСТРУКТОРОВ: ПОЗИЦИОННЫЕ И ИМЕНОВАННЫЕ

позиционный ↓ | именованный ↓ | именованный ↓

```
[MyAttribute("An excellent class", Reviewer="Amy McArthur", Ver="0.7.15.33")]
```

↑ равно | ↑ равно

```
public sealed class MyAttributeAttribute : System.Attribute {
    public string Description;
    public string Ver;
    public string Reviewer;
    public MyAttributeAttribute(string desc) { // единственный формальный параметр
        Description = desc;
    }
}
```

три аргумента ↓

```
[MyAttribute("An excellent class", Reviewer="Amy McArthur", Ver="7.15.33")]
class MyClass { ... }
```

ОГРАНИЧЕНИЕ ИСПОЛЬЗОВАНИЯ АТРИБУТОВ (ATTRIBUTEUSAGE)

```
                    только для методов
                          ↓
[ AttributeUsage( AttributeTargets.Method ) ]
public sealed class MyAttributeAttribute : System.Attribute
{ ...
```

Имя	Значение	Значение по умолчанию
ValidOn	Хранит список типов целей к которым может применяться атрибут. Первый параметр конструктора должен быть значением перечислимого типа AttributeTargets.	
Inherited	Булево значение, указывающее может ли атрибут наследоваться производными классами декорированного типа.	true
AllowMultiple	Булево значение, указывающее можно ли к цели применять одновременно несколько экземпляров атрибута текущего типа.	false

КОНСТРУКТОР ATTRIBUTEUSAGE

```
[ AttributeUsage( AttributeTargets.Method | AttributeTargets.Constructor ) ]
public sealed class MyAttributeAttribute : System.Attribute
{ ...
```

Члены перечисления *AttributeTargets:*

All	Assembly	Class	Constructor
Delegate	Enum	Event	Field
GenericParameter	Interface	Method	Module
Parameter	Property	ReturnValue	Struct

Пример использования AttributeUsage

```
[ AttributeUsage( AttributeTargets.Class,   // Required, positional
        Inherited = true,      // Optional, named
        AllowMultiple = false ) ]     // Optional, named
public sealed class MyAttributeAttribute : System.Attribute
{ ...
```

ДОСТУП К АТРИБУТУ. МЕТОД ISDEFINED

```
[ReviewComment("Check it out", "2.4")]
class MyClass { }

public static void Main()
{
    MyClass mc = new MyClass(); // создаем объект
    Type t = mc.GetType(); // получаем тип объекта
    bool isDefined = // проверяем тип на наличие атрибута
    t.IsDefined(typeof(ReviewCommentAttribute), false);
    if (isDefined)
        Console.WriteLine($"ReviewComment применен к типу {t.Name}");
}
```

Результат работы программы:

```
ReviewComment применен к типу AttrDemo.MyClass
```

ИСПОЛЬЗОВАНИЕ МЕТОДА GETCUSTOMATTRIBUTES

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class MyAttributeAttribute : Attribute
{
    public string Description { get; set; }
    public string VersionNumber { get; set; }
    public string ReviewerID { get; set; }
    public MyAttributeAttribute(string desc, string ver)
    {
        Description = desc;
        VersionNumber = ver;
    }
}

[MyAttribute("Check it out", "2.4")]
class MyClass { }
```

Результат работы программы:

```
Description : Check it out
Version Number : 2.4
Reviewer ID :
```

```
public static void Main() {
    Type t = typeof(MyClass);
    object[] AttArr = t.GetCustomAttributes(false);// без наследования
    foreach (Attribute a in AttArr)
    {
        MyAttributeAttribute attr = a as MyAttributeAttribute;
        if (null != attr)
     {
          Console.WriteLine($"Description : { attr.Description }");
          Console.WriteLine($"Version Number : { attr.VersionNumber }");
          Console.WriteLine($"Reviewer ID : { attr.ReviewerID }");
        }
    }
}
```

ССЫЛКИ

- C# Reflection - Type class [http://dotnetpattern.com/csharp-reflection-type-class]
- Type Class [http://learn.microsoft.com/en-us/dotnet/api/system.type?view=net-6.0]
- Exercises in .NET with Andras Nemes, Dynamically invoking a constructor with Reflection in .NET C#[http://dotnetcodr.com/2014/10/08/dynamically-invoking-a-constructor-with-reflection-in-net-c/]
- Атрибуты C#: обо всех аспектах [http://habr.com/ru/articles/468287/]