Иллюстрации к курсу лекций по дисциплине «Программирование»

C#_25

Reflection and Attributes

Использованы материалы пособия Daniel Solis, Illustrated C# 7

Отражение и атрибуты

Метаданные и отражение

Медатаданные – это данные о программах и используемых в них типах, хранимые в скомпилированных сборках.

Отражение – механизм, позволяющий программе во время своего исполнения считывать метаданные сборок (чужих и своих).

Классы по работе с отражением содержатся в пространстве имен **System.Reflection**.

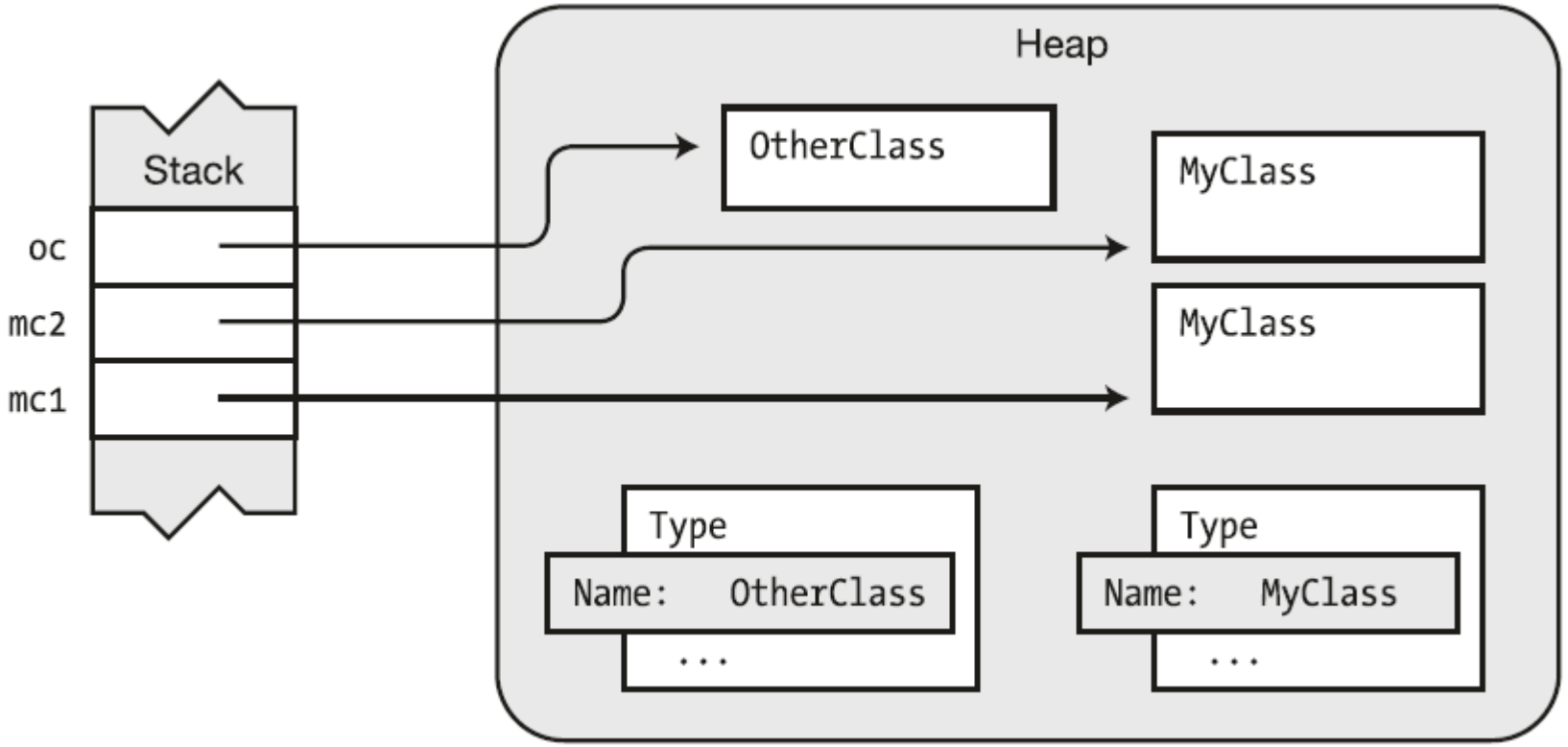
Класс Type

Для каждого типа, используемого в программе CLR создает его описание в виде экземпляра объекта типа **Type**.

Вне зависимости от количества экземпляров типа в программе всегда создается только один объект типа **Type** в программе.

Объекты типа Type в программе

MyClass mc1 = new MyClass(), mc2 = new MyClass();
OtherClass oc = new OtherClass();



Type t = mc1.GetType();

Основные члены класса Type

Имя члена	Тип	Описание
Name	свойство	Возвращает имя типа.
Namespace	свойство	Возвращает пространство имен, содержащее описание типа.
Assembly	свойство	Возвращает сборку, в которой объявлен тип. Для обобщенных типов возвращается сборка в которой определен тип.
GetFields	метод	Возвращает список полей типа.
GetProperties	метод	Возвращает список свойств типа.
GetMethods	метод	Возвращает список методов типа.

Получение информации о типе объекта

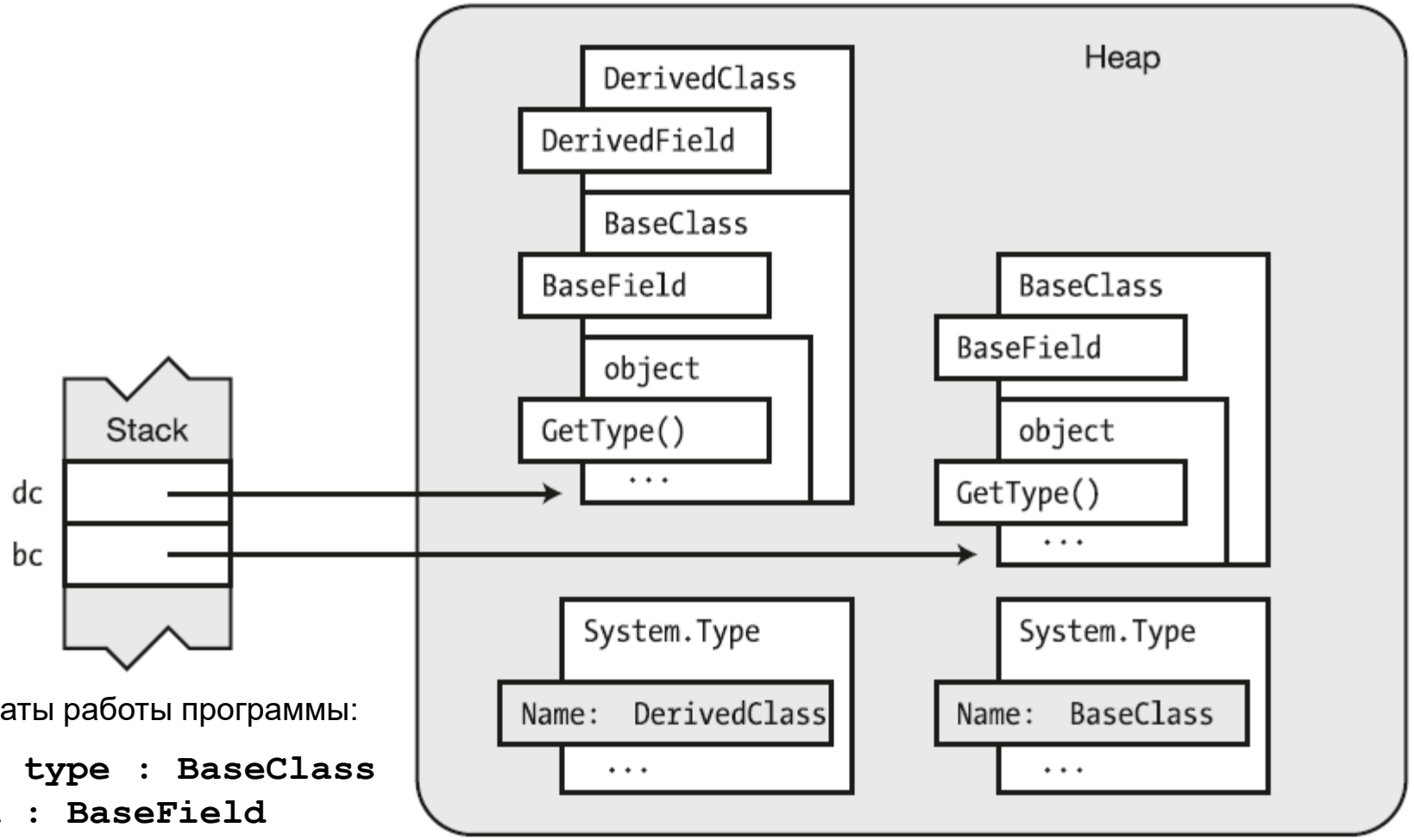
```
public class BaseClass {
    public int BaseField = 0;
}
public class DerivedClass : BaseClass {
    public int DerivedField = 0;
}
public class Program {
    public static void Main() {
        var bc = new BaseClass();
        var dc = new DerivedClass();
        BaseClass[] bca = new BaseClass[] { bc, dc };
        foreach (var v in bca) {
            Type t = v.GetType();
            Console.WriteLine($"Object type : { t.Name }");
            FieldInfo[] fi = t.GetFields();
            foreach (var f in fi)
                Console.WriteLine($" Field : { f.Name }");
            Console.WriteLine();
        }
    }
}
```

Результат:

```
Object type : BaseClass
 Field : BaseField

Object type : DerivedClass
 Field : DerivedField
 Field : BaseField
```

Получение информации о типе объекта



Результаты работы программы:

```
Object type : BaseClass
 Field : BaseField

Object type : DerivedClass
 Field : DerivedField
 Field : BaseField
```

Пример с использованием typeof()

```
public class BaseClass {
    public int BaseField;
}
public class DerivedClass : BaseClass {
    public int DerivedField;
}
public class Program {
    public static void Main() {
        Type tbc = typeof(DerivedClass); // typeof == GetType()
        Console.WriteLine($"Object type : { tbc.Name }");
        FieldInfo[] fi = tbc.GetFields();
        foreach (var f in fi)
            Console.WriteLine($"         Field : { f.Name }");
    }
}
```

Результаты работы программы:

Object type : DerivedClass

Field : DerivedField

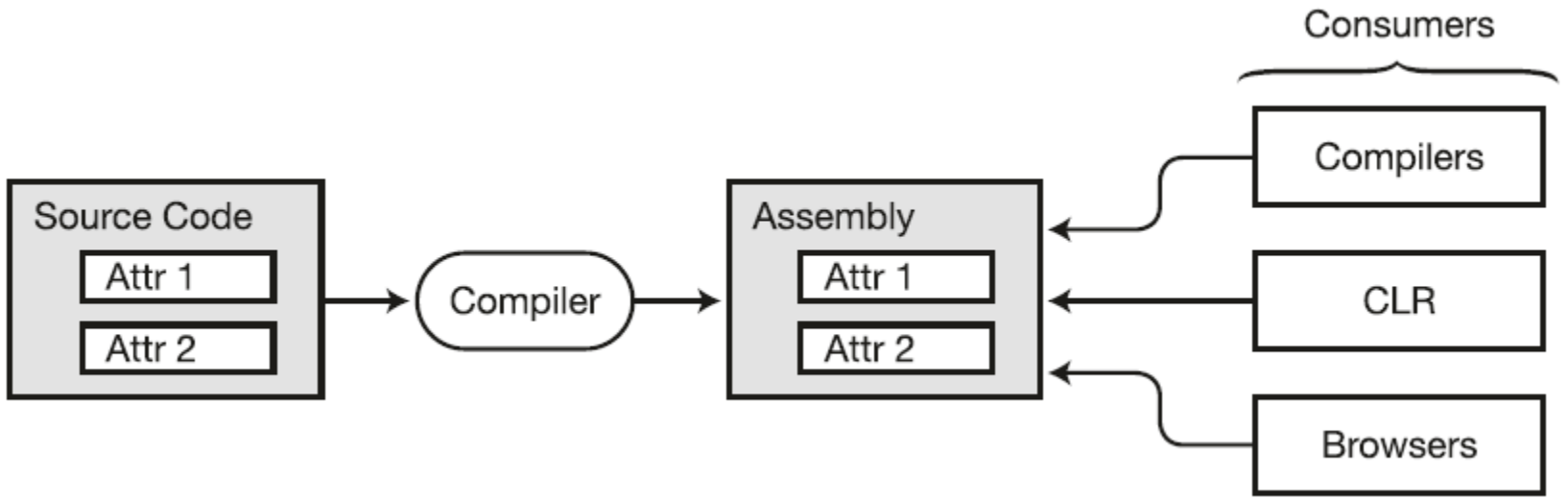
Field : BaseField

Что такое атрибут (attribute)?

Атрибут – это языковая конструкция, позволяющая добавлять метаданные к программной сборке.

Шаги:

1) Применение атрибута (библиотечного или определенного программистом) к коду программы.
2) Компилятор добавляет метаданные об атрибутах в сборку
3) Программа во время выполнения может анализировать метаданные.



Соглашение об именовании атрибутов

Для создания идентификаторов классов, обозначающих атрибуты используйте:

- Pascal-стиль;
- суффикс *Attribute*.

Важно:
При применении атрибута к цели вы можете опустить суффикс **Attribute**.

Например, для SerializableAttribute и MyAttributeAttribute, можно указать сокращенные имена Serializable и MyAttribute.

Применение (applying) атрибута

```
// применение атрибута
[ Serializable ]
public class MyClass
{ ...


// применение атрибута с параметром
[ MyAttribute("Simple class", "Version 3.57") ]
public class MyOtherClass
{ ...
```

Предопределенные и зарезервированные атрибуты

```
// применяем атрибут к методу
[Obsolete("Use method SuperPrintOut")]
static void PrintOut(string str) {
      Console.WriteLine(str);
}
public static void Main() {
      PrintOut("Start of Main");
}
```

При компиляции:

Warning CS0618 'Program.PrintOut(string)' is obsolete: 'Use method SuperPrintOut'

Результат работы:

```
Start of Main
```

```
Предупреждения можно отключать:
// отключает предупреждение 618 (Obsolete)
#pragma warning disable 618
// отключает ВСЕ предупреждения
#pragma warning disable
```

Использование атрибута для генерации ошибки

Флаг ошибки компиляции

↓

```
[ Obsolete("Use method SuperPrintOut", true) ]
static void PrintOut(string str)
{ ...
```

При компиляции:

Error CS0619 'Program.PrintOut(string)' is obsolete: 'Use method SuperPrintOut'

Препроцессор. **#define** <символ>

Директива **#define** может находиться только в начале файла и позволяет **определить** символ (но не задать его значение).

Символы можно использовать для указания условий компиляции.
Для проверки символов можно использовать директивы **#if** или **#elif.**
Для условной компиляции также можн использовать **ConditionalAttribute** (след. слайд).

Символ можно определить с помощью ключа компилятора **-define**. Для отмены определения символа служит директива **#undef**.

Символы не конфликтуют с одноименными переменными (это разные сущности).

Атрибут Conditional

```
// попробуйте закомментировать строку
#define DoTrace                          // Conditional
// попробуйте закомментировать строку
public class Program
{
        [Conditional("DoTrace")]         // Conditional
        static void TraceMessage(string str)
        { Console.WriteLine(str); }

        public static void Main() {
            TraceMessage("Start of Main");
            Console.WriteLine("Doing work in Main.");
            TraceMessage("End of Main");
        }
}
```

Результаты работы:

Start of Main
Doing work in Main.
End of Main

Атрибуты Caller*

Информационные атрибуты:

- **CallerFilePath,**
- **CallerLineNumber,**
- **CallerMemberName.**

Могут использоваться только в заголовках методов перед опциональными параметрами (со значением по умолчанию).

Находятся в пространстве имен
`System.Runtime.CompilerServices`

```
using System.Runtime.CompilerServices;
```

Использование атрибутов Caller*

```
public static void MyTrace(string message,
[CallerFilePath] string fileName = "",
[CallerLineNumber] int lineNumber = 0,
[CallerMemberName] string callingMember = "")
{
  Console.WriteLine($"File: { fileName }");
  Console.WriteLine($"Line: { lineNumber }");
  Console.WriteLine($"Called From: { callingMember }");
  Console.WriteLine($"Message: { message }");
}

public static void Main() => MyTrace("Simple message");
```

Результаты работы:
File: D:\...\05_CallerAttributes.cs
Line: 25
Called From: Main
Message: Simple message

Если у параметра метода нет умалчиваемого значения - ошибка:
Error CS4021 The CallerFilePathAttribute may only be applied to parameters with default values.

Атрибут DebuggerStepThrough

Используется для предотвращения отладки кода.

Может использоваться в:

- **классах,**
- **структурах,**
- **конструкторах,**
- **методах,**
- **аксессорах.**

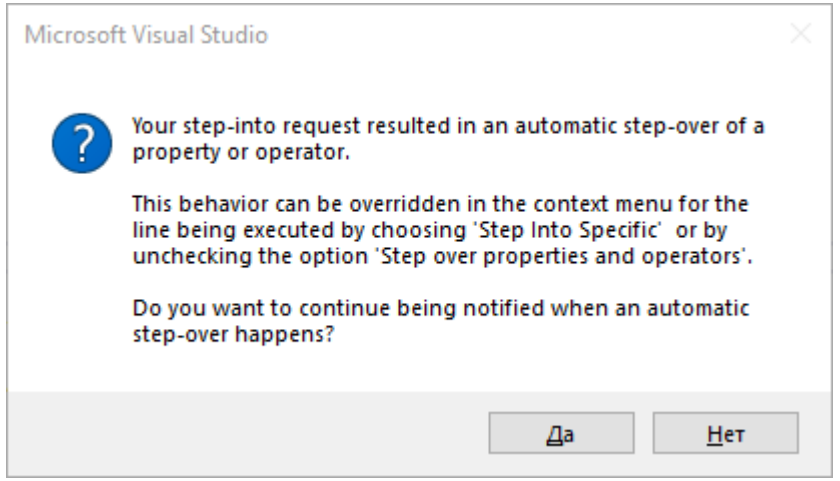
Определен в пространстве имен **`System.Diagnostics`.**

```
using System.Diagnostics;
```

Использование DebuggerStepThrough

```
public class Program {
        int x = 1;
        int X {
            get { return x; }
            [DebuggerStepThrough] // нельзя зайти отладчиком в аксессор
            set {
                x = x * 2;
                x += value;
            }
        }
        public int Y { get; set; }
        public static void Main() {
            Program p = new Program();
            p.IncrementFields();
            p.X = 5;
            Console.WriteLine($"X = { p.X }, Y = { p.Y }");
        }
        [DebuggerStepThrough] // нельзя зайти отладчиком в метод
        void IncrementFields()
        {   X++; Y++; }
}
```

Результаты работы:
X = 13, Y = 1



Другие предопределенные атрибуты

- **CLSCompliant** – указывает, что общедоступные члены должны быть проверены компилятором на соответствие CLS. CLS-сборки могут использоваться из любого .NET-совместимого языка (не поддерживаются uint / ulong).
- **Serializable** – указывает на возможность сериализации.
- **NonSerialized** – указывает на невозможность сериализации.
- **DLLImport** – указывает на реализацию в неуправляемом коде.
- **WebMethod** – указывает, что доступ к методу должен предоставляться через Веб-службу (протокол SOAP).
- **AttributeUsage** – указывает к каким типам программных конструкций можно применять атрибут. Используется только при объявлении атрибутов.