ПИШНИК: BECOME HUMAN

Глава 19: Директивы Препроцессора.
Краткий Обзор Рефлексии и Атрибутов в C#

Автор презентации – Сагалов Даниил БПИ-196

Препроцессорные Директивы в C#

Стоит помнить, что у C# компилятора нет отдельного препроцессора.

Тем не менее, препроцессорные директивы позволяют задать некоторые особенности на первом этапе компиляции программы (в первую очередь, реализовать условную компиляцию). Важно понимать, что в отличие от C и C++ Вы не можете использовать препроцессорные директивы для создания макросов.

Помните, что директива препроцессора должна быть единственной инструкцией в строке.

#define и #undef

Препроцессорная директива **#define <Имя Символа>** позволяет определить некоторый символ, который при использовании в #if выражениях **возвращает true**; задать символу значение нельзя. Вы не можете использовать #define для объявления констант, как в C++.

Препроцессорная директива **#undef <Имя Символа>** наоборот определяет символ, **возвращающий false** при дальнейшем использовании в #if.

Символы, определённые через #define применимы для условной компиляции, когда Вы, например, хотите собрать тестовую версию, содержащую какой-то специфический код. При этом в релизной версии Вы можете исключить соответствующий символ.

Вы можете использовать #define и #undef **строго в самом начале файла, перед using-директивами, область видимости символа – файл, в котором он определён**. Определённые с помощью #define символы не конфликтуют с одноимёнными переменными. В дополнение к этому помните, что в многострочные комментарии в одной строке с препроцессорными директивами недопустимы (кроме #region, но в данном контексте как комментарии они восприниматься не будут.

Препроцессорные директивы **#if**, **#elif** и **#else** позволяют использовать определённые с помощью #define и #undef символы для получения различных сценариев компиляции. Препроцессорная директива **#endif** должна обязательно идти в конце #if … #elif … #else, иначе возникнет ошибка компиляции.

Для #if, #elif применимы операторы == и !=, также Вы можете использовать **!**, **||** и **&&** для проверок с несколькими символами, разрешается группировка символов с помощью скобок.

Помните, что Вы можете дополнительно определять символы с помощью параметра компилятора **-define**.

Также существует несколько предопределённых символов: DEBUG и TRACE (DEBUG автоматически устанавливается в зависимости от свойств конфигураций сборки). Ниже представлен набор предопределённых символов для разных версий .NET:

Версия .NET	Символ
.NET Framework	NETFRAMEWORK, NET20, NET35, NET40, NET45, NET451, NET452, NET46, NET461, NET462, NET47, NET471, NET472, NET48
.NET Standard	NETSTANDARD, NETSTANDARD1_0, NETSTANDARD1_1, NETSTANDARD1_2, NETSTANDARD1_3, NETSTANDARD1_4, NETSTANDARD1_5, NETSTANDARD1_6, NETSTANDARD2_0, NETSTANDARD2_1
.NET Core	NETCOREAPP, NETCOREAPP1_0, NETCOREAPP1_1, NETCOREAPP2_0, NETCOREAPP2_1, NETCOREAPP2_2, NETCOREAPP3_0, NETCOREAPP3_1

#if, #elif, #else
#endif

#warning, #error, #region

#warning <Текст> позволяет создавать собственное предупреждение компилятора (CS1030) в указанном месте кода. Зачастую используется при условной компиляции с #if.

#error <Текст> позволяет создавать собственную ошибку компиляции (CS1029) в указанном месте кода. Зачастую используется при несовместимом наборе условий компиляции.

#region <Текст> используется для структуризации кода в редакторе, так как позволяет создавать сворачиваемые именованные области кода. Обязательно требует парной закрывающей директивы **#endregion**. Попытка добавить многострочный комментарий на одной строке с #region приведёт к включению комментария в название области Обратите внимание, что #region не может накладывать на #if, однако при этом они могут быть вложены друг в друга:

```
#if NETCOREAPP3_1
#region ExampleRegion
        Console.WriteLine(“This is a .NET Core 3.1 app.");
#endif  // Ошибка компиляции - #endregion должна идти перед #endif
#endregion
```

Директива #line

Директива #line позволяет менять номер строки компилятора, имя файла или же исключать строку при подсчёте компилятором.

#line <Номер Строки> <Имя Файл> – задаёт номер следующей строки и/или имя файла, которым будет заменяться настоящее. Имя файла должно заключаться в двойные кавычки.

#line hidden – позволяет исключать одну или несколько строк кода, для прекращение исключения в конце укажите **#line**. #line hidden не влияет на имена файлов и номера строк, т. е. в случае возникновения ошибки в скрытом блоке компилятор укажет имя файла и номер ошибочной строки. При пошаговой отладке скрытые строки игнорируются даже при установке на них точек останова.

#line default – восстанавливает нумерацию строк с учётом всех ранее исключённых.

Директива #pragma

#pragma warning <disable/restore> <Список предупреждений> – позволяет отключать/включать предупреждения компилятора:

```
#pragma warning disable CS0168
    class Program {
        static void Main( ) {
    // Предупреждение о том, что i объявлена, но не используется не будет
            int i;
#pragma warning restore CS0168
            int j; // Предупреждение снова появится
        }
```

#pragma checksum <Имя Файла> <guid> <Байты суммы> – создаёт контрольные суммы для исходных файлов, помогает с отладкой страниц ASP.NET. GUID – глобальный уникальный идентификатор хэш-алгоритма. 3 параметр – байты суммы, чётное шестнадцатеричное число. При использовании нечётного возникнет предупреждение, а директива будет пропущена.

Рефлексия в C#

Рефлексия (Отражение) – механизм, позволяющий получать информацию о содержимом сборок, модулей и типов с помощью специальных объектов типа System.Type. Вы всегда можете получить объект типа Type с помощью **typeof** или метода **GetType( )** класса Object. Для работы с рефлексией используйте пространство имён **System.Reflection**. Помните, что так как класс System.Type абстрактный, полученные через GetType( )/typeof объекты будут иметь тип **System.RuntimeType**. Что примечательно, при работе с рефлексией все Runtime-типы создаются только один раз для каждого объекта, а также поддерживают операции сравнения.

Метаданные – специальные данные о программах и типах, используемых в них. Именно метаданные позволяют

Рефлексия может быть использована для:

- Осуществления доступа к атрибутам в метаданных программы.
- Для проверки и создания типов в сборке.
- Для создания экземпляров типов во время выполнения.
- Для создания типов во время выполнения (**System.Reflection.Emit**).
- Для выполнения позднего связывания, которое обеспечивает доступ к методам в типах, созданных во время выполнения.

System.Type

System.Type – абстрактный класс, представляющий определения других типов. Является основным способом получения метаданных. Объекты Type, представляющие тип, уникальны – две ссылки ссылаются на один и тот же объект Type только в том случае, если они представляют один и тот же тип. Это позволяет осуществлять сравнение объектов Type через проверку равенства ссылок.

Методы GetFields( ), GetProperties( ) и GetMethods( ) позволяют получать информацию об открытых членах типа.

```
class Program {
    Ссылок: 0
    static void Main() {
        int a = 12345;
        long b = 12345;
        int c = 12345;
        Type type_a = a.GetType();
        // False - System.Int32 и System.Int64 - разные типы
        Console.WriteLine(Object.ReferenceEquals(type_a, b.GetType()));
        // True - System.Int32 и System.Int32 - один тип
        Console.WriteLine(Object.ReferenceEquals(type_a, c.GetType()));
    }
}
```

Пример Получения Информации о Типе

```
public static class Kowalski {
    public static event Action OnAnalysis;
    public static int weaponCount;
    // Скрываем ошибку о том, что поле не используется
#pragma warning disable CS0169
    private static string SecretMessage;
#pragma warning restore CS0169
    Ссылок: 0
    public static string[] comrades { get; set; }
        = { "Rico", "Skipper", "Private"};
    ссылка: 1
    public static Type Analysis() {
        OnAnalysis?.Invoke();
        return typeof(Kowalski);
    }
    Ссылок: 0
    static Kowalski() => weaponCount = int.MaxValue;
}
```

```
class Program {
    Ссылок: 0
    static void Main() {
        Type penguinInfo = Kowalski.Analysis();
        // Получаем информацию обо всех открытых полях
        Console.WriteLine("Fields:");
        foreach (var field in penguinInfo.GetFields())
            Console.WriteLine(field);
        Console.WriteLine(Environment.NewLine);
        // Выведет информацию обо всех открытых свойствах
        Console.WriteLine("Properties:");
        foreach (var property in penguinInfo.GetProperties())
            Console.WriteLine(property);
        Console.WriteLine(Environment.NewLine);
        // Выведет информацию обо всех открытых методах
        // (включая методы доступа к свойствам get/set)
        Console.WriteLine("Methods:");
        foreach (var method in penguinInfo.GetMethods())
            Console.WriteLine(method);
    }
}
```

Консоль отладки Microsoft Visual Studio

```
Fields:
Int32 weaponCount


Properties:
System.String[] comrades


Methods:
System.String[] get_comrades()
Void set_comrades(System.String[])
Void add_OnAnalysis(System.Action)
Void remove_OnAnalysis(System.Action)
System.Type Analysis()
System.Type GetType()
System.String ToString()
Boolean Equals(System.Object)
Int32 GetHashCode()
```

Атрибуты в C#

Атрибуты – специальная языковая конструкция, позволяющая добавлять метаданные к сборке. По соглашению об именовании для названий типов-атрибутов используется PascalCase, а в конце добавляется суффикс Attribute, который можно опустить при применении атрибута. Для применения атрибута необходимо добавить его перед членом, к которому он применяется в квадратных скобках.

Вы можете использовать несколько атрибутов к одной цели, перечислив их через запятую или же использовав квадратные скобки несколько раз. Вы можете указывать, к чему конкретно применяются атрибуты:

Значение	Применение
assembly	Вся сборка
module	Модуль текущей сборки
field	Поле в классе или структуре
event	Событие
method	Метод либо методы доступа к свойствам get и set
param	Параметры метода или параметры метода доступа set
property	Свойство
return	Возвращаемое значение метода, индексатора свойства или метода доступа к свойствам get
type	Структура, класс, интерфейс, перечисление или делегат

Задание 1

В результате выполнения фрагмента программы:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Xml.Serialization;

class A {
    public int y;

    public A() { }
}

class Program {
    static void Main() {
        XmlSerializer xml = new
            XmlSerializer(typeof(List<A>));
        FileStream fs = new FileStream("out.xml",
FileMode.Create);
        using (new StreamWriter(fs)) {
            List<A> list = new List<A>();
            for (int i = 0; i < 5; i++) {
                list.Add(new A());
                list[i].y = i + 1;
            }
            xml.Serialize(fs, list);
        }
        fs = new FileStream("out.xml", FileMode.Open);
        List<A> list2 = (List<A>)xml.Deserialize(fs);
        for (int i = 0; i < list2.Count; i++) {
            Console.Write(list2[i].y);
        }
    }
}
```

на экран будет выведено:

Примечание:

Если возникнет ошибка компиляции, введите: ***

Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---

Если возникнет ошибка исполнения или исключение, введите: +++

Задачи

Задание 2

В результате выполнения фрагмента программы:

```
using System;
using System.IO;
using System.Xml.Serialization;

public class A {
    public int x = 5;
    [NonSerialized]
    public int y = 7;

    public A() { }
}

class Program {
    static void Main() {
        XmlSerializer xml = new XmlSerializer(typeof(A));
        FileStream fs = new FileStream("out.xml",
FileMode.Create);
        using (new StreamWriter(fs)) {
            A a = new A();
            a.x = a.y + 10;
            xml.Serialize(fs, a);
        }
        fs = new FileStream("out.xml", FileMode.Open);
        A a2 = (A)xml.Deserialize(fs);
            Console.Write(a2.y + a2.x);
    }
}
```

на экран будет выведено:

ЗАДАЧИ

Задание 3

Выберите верные утверждения (*укажите все верные ответы*)**:**

1) XML-сериализация требует атрибут [Serializable].
2) XML-сериализация сериализует открытые и внутрисборочные поля класса.
3) XML-сериализация игнорирует атрибут [NonSerialized].
4) В JSON-сериализации нужно классы или структуры помечать атрибутом [DataMember], а их члены – [DataContract].
5) Бинарная сериализация требует наличия конструктора без параметров.

Задание 4

Про XML-сериализацию верно (*укажите все верные ответы*)**:**

1) Сериализует поля и методы.
2) При десериализации НЕ требует приведения типа к объекту, тип которого был сериализован.
3) Делает атрибут [Serializable] наследуемым.
4) Требует наличие конструктора без параметров.
5) Все сериализуемые поля должны быть public. В противном случае выбрасывается исключение.

Задание 5

Выберите все верные утверждения:

1) Атрибут [**NotSerialized**], как и атрибут [**Serializable**] может использоваться для классов.
2) Атрибуты [**NotSerialized**] и [**Serializable**] могут быть унаследованы.
3) В JSON-сериализации для сериализации используется метод WriteObject();
4) JSON-сериализация, в отличие от XML-сериализации, работает только с открытыми типами данных.
5) В бинарной сериализации, все классы, поля и свойства должны быть помечены атрибутом [**Serializable**].

Ответы

Задание	Ответ
1	+++ *(InvalidOperationException, тип закрытый)*
2	**24**
3	**3**
4	**4**
5	**3**

Задание 1

В результате выполнения фрагмента программы:

```
using System;
using System.Linq;

public class Integers1 {
    public int Hash;
    public int val;

    public Integers1(int x) {
        val = x;
        Hash = x ^ 2;
    }
}

public class Integers2 {
    public int Hash;
    public int val;

    public Integers2(int x) {
        val = x;
        Hash = x ^ 3;
    }
}

class Program {
    static void Main() {
        Integers1[] args1 = new Integers1[2];
        Integers2[] args2 = new Integers2[2];
        for (int i = 0; i < 2; i++) {
            args1[i] = new Integers1(i);
            args2[i] = new Integers2(i * 2);
        }
        var query = from a in args1
                    join b in args2 on a.Hash equals b.Hash
                    select a.val;
        foreach (var q in query)
            Console.Write(q);
    }
}
```

на экран будет выведено:

Задачи

Задание 2

В результате выполнения фрагмента программы:

```
using System;
using System.Linq;

class Program {
    static void Main() {
        var args = new int[] { 0, 1, 2, 3, 4, 5, 6 };
        var res = from t in args
                  where t >= 3 && t <= 5
                  select t % 4;
        foreach (var item in res) {
            Console.Write(item);
        }
    }
}
```

на экран будет выведено:

Задание 3

Ключевыми словами языка LINQ являются (*укажите все верные ответы*)**:**

1) **from.**
2) **select.**
3) **yield.**
4) **async.**
5) **await.**

ЗАДАЧИ

Задание 4

В результате выполнения фрагмента программы:

```
using System;
using System.Linq;

class Program {
    static void Main() {
        var A = new[] { 30, 26 };
        var B = new[] { 4, 7, 3 };
        var someInts = from a in A
                       from b in B
                       let mod = a % b
                       orderby mod
                       where mod > 1
                       select mod;
        foreach (var a in someInts)
            Console.Write(a);
    }
}
```

на экран будет выведено:

Задание 5

В результате выполнения фрагмента программы:

```
using System;

class Program {
    static void Main() {
        var integers = new[] {
            new { a = 3 },
            new { a = 4, b = 1 },
            new { a = 5, b = 8, c = 91 }
        };
        foreach (var rec in integers)
            Console.Write(rec.a);
    }
}
```

на экран будет выведено:

Примечание:

Если возникнет ошибка компиляции, введите: ***

Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---

Если возникнет ошибка исполнения или исключение, введите: +++

Ответы

Задание	Ответ
1	**13**
2	**301**
3	**12**
4	**22225**
5	***

Задачи

Задание 1

Выберите верные утверждения (*укажите все верные ответы*)**:**

1) На одной строке с препроцессорными директивами НЕ допускается использование многострочных комментариев.
2) Препроцессорные директивы допускают использование неограниченного количества пробелов между # и самой директивой.
3) Препроцессорные директивы должны быть использованы после директив `using`.
4) С помощью препроцессорных директив можно вызывать ошибки компиляции и предупреждения со своим текстом.
5) Препроцессорные директивы позволяют отключить ошибки компиляции.

Задание 2

Выберите верные утверждения (*укажите все верные ответы*)**:**

1) Использование препроцессорного `#if` допустимо с ключевыми словами `true` или `false`.
2) Возможно использовать `#define` с ключевыми словами языка C#.
3) Препроцессорные директивы позволяют отключить предупреждающие сообщения компилятора.
4) Вместо else-if в препроцессорных `#if` и `#else` должно использоваться `#elseif`.
5) В препроцессорных `#if` НЕ допускается использование | и & для условий.

Задание 3

В результате выполнения фрагмента программы:

```
#define A
#region B

using System;

class Program {
#endregion
    static void Main() {
#if A && !B
        int x = 5;
        int y = 0;
        Console.Write(x + y);
#else
        int z = x / y;
        Console.Write(z / z);
#endif
    }
}
```

на экран будет выведено:

Примечание:
*Если возникнет ошибка компиляции, введите: ****
Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---
Если возникнет ошибка исполнения или исключение, введите: +++

ЗАДАЧИ

Задание 4

Целью атрибута может быть (*укажите все верные ответы*)**:**

1) Входной параметр метода.
2) Константа класса.
3) Публичный индексатор.
4) Статическое свойство.
5) Событие.

Задание 5

При помощи рефлексии можно извлечь информацию о(-б) (*укажите все верные ответы*)**:**

1) Полях, объявленных в классе.
2) Реализованных классом интерфейсах.
3) Статических методах класса.
4) Свойствах структуры.
5) Индексаторов структуры.

Задачи

Задание 6

В результате выполнения фрагмента программы:

```
using System;
using System.Reflection;

struct MyStruct {
    public static int X;
    public int Y;
    static int Z;
    int W;
    internal static int V;
}

class Program {
    static void Main() {
        FieldInfo[] fs = typeof(MyStruct).GetFields();
        foreach (var item in fs) {
            Console.Write(item.Name);
        }
    }
}
```

на экран будет выведено:

Примечание:
*Если возникнет ошибка компиляции, введите: ****
Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---
Если возникнет ошибка исполнения или исключение, введите: +++

Задание 7

В результате выполнения фрагмента программы:

```
using System;
using System.Reflection;

struct MyStruct {
    public static int X;
    public int Y;
    static int Z;
    int W;
    internal static int V;
}

class Program {
    static void Main() {
        FieldInfo[] fs =
typeof(MyStruct).GetFields(BindingFlags.NonPublic |
BindingFlags.Instance | BindingFlags.Static);
        foreach (var item in fs) {
            Console.Write(item.Name);
        }
    }
}
```

на экран будет выведено:

Примечание:
*Если возникнет ошибка компиляции, введите: ****
Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---
Если возникнет ошибка исполнения или исключение, введите: +++

Ответы

Задание	Ответ
1	**124**
2	**1235** (*#elif, не #elseif*)
3	**5**
4	**1245**
5	**12345** (*GetDefaultMembers( ) и GetInterfaces( )*)
6	**YX**
7	**WZV**