Атрибуты

Атрибуты представляют собой расширяемый механизм для добавления специальной информации к элементам кода (сборкам, типам, членам, возвращаемым значениям, параметрам и параметрам обобщенных типов).

Классы атрибутов

Атрибут определяется классом, который унаследован (прямо или косвенно) от абстрактного класса `System.Attribute`. Чтобы присоединить атрибут к элементу кода, надо указать перед элементом кода имя типа атрибута в квадратных скобках:

```
[ObsoleteAttribute]
public class Foo { ... }
```

Этот атрибут приводит к тому, что компилятор выдаст предупреждение, если производится ссылка на тип или член, помеченный как устаревший (obsolete).

По соглашению имена всех типов атрибутов завершаются словом `Attribute`. Это соглашение распознается компилятором C# и позволяет опускать данный суффикс, когда присоединяется атрибут:

```
[Obsolete]
public class Foo { ... }
```

Упрощенный пример описания `ObsoleteAttribute`:

```
public sealed class ObsoleteAttribute : Attribute { ... }
```

Перечисления и атрибут Flags

Перечисление — это специальный тип значения, который позволяет указывать группу именованных числовых констант:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Это перечисление можно применять следующим образом:

```
BorderSide topSide = BorderSide.Top;
bool isTop = topSide == BorderSide.Top; // true
```

Каждый член перечисления имеет лежащее в его основе целочисленное значение.

По умолчанию:

• лежащие в основе значения относятся к типу `int`;

• членам перечисления присваиваются константы 0, 1, 2, ... (в порядке их объявления).

Можно указать другой целочисленный тип:

```
public enum BorderSide : byte { Left, Right, Top, Bottom }
```

Для каждого члена перечисления можно также указывать явные значения:

```
public enum BorderSide : byte { Left=1, Right=2, Top=10, Bottom=11 }
```

Члены, которым значения не были присвоены, получают значения на основе инкрементирования последнего явно указанного значения:

```
public enum BorderSide : byte { Left=1, Right, Top=10, Bottom }
```

Преобразования перечислений

Экземпляр перечисления может быть преобразован в и из лежащего в основе целочисленного значения с помощью явного приведения:

```
int i = (int)BorderSide.Left;
BorderSide side = (BorderSide)i;
bool leftOrRight = (int)side <= 2;
```

Числовой литерал 0 в выражении `enum` явного приведения не требует:

```
BorderSide b = 0;
if (b == 0) ... // Приведение не требуется
```

Причины:

(1) первый член перечисления часто используется как «стандартное» значение;

(2) для типов комбинированных перечислений значение 0 означает «отсутствие флагов».

Перечисления флагов

Члены перечислений можно комбинировать. Чтобы предотвратить неоднозначности, члены комбинируемого перечисления требуют явного присваивания значений (обычно степени двойки):

```
[Flags]
public enum BorderSides { None=0, Left=1, Right=2, Top=4, Bottom=8 }
```

Для работы со значениями комбинированного перечисления применяются побитовые операции:

```
BorderSides leftRight = BorderSides.Left | BorderSides.Right;
if ((leftRight & BorderSides.Left) != 0)
  Console.WriteLine("Includes Left"); // Includes Left
string formatted = leftRight.ToString(); // "Left, Right"
BorderSides s = BorderSides.Left;
s |= BorderSides.Right;
Console.WriteLine(s == leftRight); // True
s ^= BorderSides.Right; // Переключает значение BorderSides.Right
Console.WriteLine(s); // "Left"
```

К типу комбинируемого перечисления должен применяться атрибут `Flags`. Если объявить такое перечисление без атрибута `Flags`, то комбинировать члены по-прежнему можно, но вызов `ToString` на экземпляре перечисления будет выдавать число, а не последовательность имен.

Комбинируемые перечисления обычно имеют имена во множественном числе.

Для удобства члены комбинаций могут быть помещены в само объявление перечисления:

```
[Flags]
public enum BorderSides
{
  None=0,
  Left=1, Right=2, Top=4, Bottom=8,
  LeftRight = Left | Right,
  TopBottom = Top | Bottom,
  All = LeftRight | TopBottom
}
```

Операции над перечислениями

```
= == != < > <= >= + - ^ & | ~ += -= ++ -- sizeof
```

Побитовые, арифметические и операции сравнения возвращают результат обработки лежащих в основе целочисленных значений. Сложение разрешено для перечисления и целочисленного типа, но не для двух перечислений.

Безопасность перечислений

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Поскольку тип перечисления может быть приведен к лежащему в основе целому типу и наоборот, фактическое значение может выходить за пределы допустимых границ для членов перечисления:

```
BorderSide b = (BorderSide)12345;
Console.WriteLine(b); // 12345
```

Побитовые и арифметические операции могут давать в результате аналогичные недопустимые значения:

```
BorderSide b = BorderSide.Bottom;
b++; // Ошибка не возникает
```

Недопустимый экземпляр `BorderSide` может нарушить работу кода:

```
void Draw(BorderSide side)
{
  if (side == BorderSide.Left) {...}
  else if (side == BorderSide.Right) {...}
  else if (side == BorderSide.Top) {...}
  else {...} // Предполагается BorderSide.Bottom
}
```

Вариант решения проблемы — добавление дополнительной конструкции `else`:

```
else if (side == BorderSide.Bottom) {...}
else throw new ArgurnentException("Invalid BorderSide: " + side, “side”);
```

Другой вариант — явная проверка значения перечисления методом Enum.IsDefined:

```
BorderSide side = (BorderSide)12345;
Console.WriteLine(Enum.IsDefined(typeof(BorderSide), side)); // False
```

Метод Enum.IsDefined *не работает с перечислениями флагов*. Следующий вспомогательный метод (учитывающий поведение Enum.ToString()) возвращает true, если перечисление флагов является допустимым:

```
static bool IsFlagDefined(Enum e)
{
  decimal d;
  return !decimal.TryParse(e.ToString(), out d);
}
[Flags]
public enum BorderSides
{
  None=0,
  Left=1, Right=2, Top=4, Bottom=8,
  LeftRight = Left | Right,
  TopBottom = Top | Bottom,
  All = LeftRight | TopBottom
}
static void Main()
{
  for (int i = 0; i <= 16; i++)
  {
    BorderSides side = (BorderSides)i;
    Console.WriteLine (IsFlagDefined(side) + " ("+
      Enum.IsDefined(typeof(BorderSides), side) + ") " + side);
  }
}
```

Результат:

```
True (True) None
True (True) Left
True (True) Right
True (True) LeftRight
True (True) Top
True (False) Left, Top
True (False) Right, Top
True (False) LeftRight, Top
True (True) Bottom
True (False) Left, Bottom
True (False) Right, Bottom
True (False) LeftRight, Bottom
True (True) TopBottom
True (False) Left, TopBottom
True (False) Right, TopBottom
True (True) All
False (False) 16
```

Если в предыдущей программе убрать атрибут Flags, то результат будет следующим:

```
True (True) None
True (True) Left
True (True) Right
True (True) LeftRight
True (True) Top
False (False) 5
False (False) 6
False (False) 7
True (True) Bottom
False (False) 9
False (False) 10
False (False) 11
True (True) TopBottom
False (False) 13
False (False) 14
True (True) All
False (False) 16
```

Атрибуты (продолжение)

Именованные и позиционные параметры атрибутов

Атрибуты могут иметь параметры:

```
[XmlElement("Customer", Namespace="http://aa/bb.com")]
public class CustomerEntity { ... }
```

Этот атрибут обеспечивает XML-сериализацию класса `CustomerEntity` в виде XML-элемента по имени `Customer`, принадлежащего пространству имен `http://aa/bb.com`:

Параметры атрибутов бывают *позиционные* и *именованные*. В предыдущем примере первый аргумент является позиционным параметром, а второй — именованным параметром. Позиционные параметры соответствуют параметрам открытых конструкторов типа атрибута. Именованные параметры соответствуют открытым полям или открытым свойствам типа атрибута.

При указании атрибута должны включаться позиционные параметры, которые соответствуют одному из конструкторов класса атрибута. Именованные параметры являются необязательными.

Цели атрибутов

Неявно целью атрибута является элемент кода, который находится непосредственно за атрибутом, и обычно это тип или член типа. Тем не менее, атрибуты можно присоединять и к сборке. Это требует явного указания *цели* атрибута, например:

```
[assembly: CLSCompliant(true)]
```

Атрибут `CLSCompliant` контролирует соответствие общеязыковой спецификации (Comшon Language Specification — CLS). Если в этой сборке содержится описание следующего вида

```
public int SetValue(UInt32 value) { ... }
```

то компилятор выведет предупреждение, так как тип `UInt32` не входит в CLS. Подавить предупреждение можно с помощью следующего атрибута:

```
[CLSCompliant(false)]
public int SetValue(UInt32 value) { ... }
```

Указание нескольких атрибутов

Для одного элемента кода можно указывать несколько атрибутов. Атрибуты могут быть заданы либо внутри единственной пары квадратных скобок (и разделяться запятыми), либо в отдельных парах квадратных скобок (или с помощью комбинации двух способов):

```
[Serializable, Obsolete, CLSCompliant(false)]
public class Bar { ... }

[Serializable] [Obsolete] [CLSCompliant(false)]
publc class Bar { ... }

[Serializable, Obsolete]
[CLSCompliant(false)]
public class Bar { ... }
```

Атрибут AttributeUsage

`AttributeUsage` — это атрибут, применяемый к классам атрибутов. Он сообщает компилятору о том, как должен использоваться целевой атрибут:

```
public sealed class AttributeUsageAttribute : Attribute
{
  public AttributeUsageAttribute(AttributeTargets validOn);
  public bool AllowMultiple { get; set; }
  public bool Inherited { get; set; }
  public AttributeTargets ValidOn{ get; }
}
```

Свойство `AllowMultiple` управляет тем, может ли определяемый атрибут применяться к одной и той же цели более одного раза. Свойство `Inherited` указывает на то, должен ли атрибут, примененный к базовому классу, также применяться к производным классам (или в случае методов — должен ли атрибут, применен-

ный к виртуальному методу, также применяться к переопределенным методам). Свойство ValidOn определяет набор целей (классов, интерфейсов, свойств, методов, параметров и т.д.), к которым может быть присоединен этот атрибут. Оно принимает любую комбинацию значений перечисления AttributeTargets, которое содержит следующие члены:

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

Пример:

```
[AttributeUsage(AttributeTargets.Delegate | AttributeTargets.Enum
| AttributeTargets.Struct | AttributeTargets.Class, Inherited = false)]
public sealed class SerializableAttribute : Attribute {}
```

Свойства атрибута и параметры его конструктора должны относиться к следующим типам:

- запечатанный примитивный тип (bool, byte, char, double, float, int, long, short или string);
- тип Type;
- тип перечисления;
- одномерный массив любого из упомянутых выше типов.

В следующем классе определяется атрибут для содействия системе автоматизированного модульного тестирования. Он указывает, что метод должен быть протестирован, количество повторений теста и сообщение, выдаваемое в случае неудачи:

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class TestAttribute : Attribute
{
  public int Repetitions;
  public string FailureMessage;
  public TestAttribute() : this(1) { }
  public TestAttribute(int repetitions) { Repetitions = repetitions; }
}
```

Код класса Foo с методами, которые декорированы атрибутом Test разнообразными способами:

```
class Foo
{
  [Test]
  public void Method1() { ... }
  [Test(20)]
  public void Method2() { ... }
  [Test(20, FailureMessage="Debugging Time!")]
  public void Method3() { ... }
}
```

Извлечение атрибутов во время выполнения

Два стандартных способа извлечения атрибутов во время выполнения:

- вызов метода GetCustomAttributes на любом объекте Type или Memberinfo (возвращает все атрибуты);
- вызов метода Attribute.GetCustomAttribute или Attribute.GetCustomAttributes.

Последние два метода перегружены для приема любого объекта рефлексии, который соответствует допустимой цели атрибута (Type, Assembly, Module, MemberInfo или ParameterInfo).

Пример:

```
foreach (Methodlnfo mi in typeof(Foo).GetMethods())
{
  TestAttribute att = (TestAttribute)Attribute.GetCustomAttribute(mi,
    typeof(TestAttribute));
  if (att ! = null)
    Console.WriteLine("Method {0} will be tested; reps= {1}; msg= {2}",
      mi.Name, att.Repetitions, att.FailureMessage);
```

Вывод выглядит следующим образом:

```
Method Method1 will be tested; reps=1; msg=
Method Method2 will be tested; reps=20; msg=
Method Method3 will be tested; reps=20; msg=Debugging Time!
```

Пример реального применения атрибута Test:

```
foreach (MethodInfo mi in typeof(Foo).GetMethods())
{
  TestAttribute att = (TestAttribute)Attribute.GetCustomAttribute(mi,
    typeof(TestAttribute));
  if (att != null)
    for (int i = 0; i < att.Repetitions; i++)
      try
      {
        mi.Invoke (new Foo(), null); // Вызвать метод без аргументов
      }
      catch (Exception ex) // Оформить исключение ex как внутреннее (InnerException)
      {
        throw new Exception("Error: " + att.FailureMessage, ex);
      }
```

Пример, в котором выводится список атрибутов, присутствующих в заданном типе:

```
[Serializable, Obsolete]
class Sample
{
  static void Main()
  {
    Attribute[] atts = Attribute.GetCustomAttributes(typeof(Test));
    foreach(var att in atts)
      Console.WriteLine(att);
  }
}
```

```
System.ObsoleteAttribute
System.SerializableAttribute
```

Некоторые атрибуты, доступные в сценариях Unity

ContextMenuAttribute

Связывает метод (без параметров) с пунктом контекстного меню компонента, соответствующего данному скрипту, в инспекторе. Метод не должен быть статическим.

```
using UnityEngine;
public class ContextTesting : MonoBehaviour
{
    [ContextMenu("Do Something")]
    void DoSomething()
    {
        Debug.Log("Perform operation");
    }
}
```

ContextMenuItemAttribute

Добавляет пункт к контекстному меню поля соответствующего компонента в инспекторе.

```
using UnityEngine;
using System.Collections;
public class ExampleClass : MonoBehaviour
{
```

```
    [ContextMenuItem("Reset", "ResetBiography")]
    [Multiline(8)]
    public string playerBiography = "";
    void ResetBiography()
    {
        playerBiography = "";
    }
}
```

DisallowMultipleComponent

Не позволяет добавлять более одного компонента данного типа (или производного типа) к GameObject.

HeaderAttribute

Добавляет заголовок перед указанным набором полей в инспекторе.

```
using UnityEngine;
using System.Collections;
public class ExampleClass : MonoBehaviour
{
    [Header("Health Settings")]
    public int health = 0;
    public int maxHealth = 100;
    [Header("Shield Settings")]
    public int shield = 0;
    public int maxShield = 0;
}
```

HelpURLAttribute

Добавляет интернет-ссылку на описание данного компонента (ссылка связывается с кнопкой помощи).

```
using UnityEngine;
[HelpURL("http://example.com/docs/MyComponent.html")]
public class MyComponent { ... }
```

HideInInspectorAttribute

Не отображает публичное поле в инспекторе.

```
using UnityEngine;
using System.Collections;
public class ExampleClass : MonoBehaviour
{
    [HideInInspector]
    public int p = 5;
    ...
}
```

MultilineAttribute

Связывает с данным строковым полем многострочное поле ввода в инспекторе; может содержать параметр, равный числу строк (по умолчанию 3). При большем числе строк полосы прокрутки не возникают.

RangeAttribute

Отображает вещественное или целочисленное поле в виде ползунка, ограничивая тем самым его допустимый диапазон.

```
public RangeAttribute(float min, float max);
```

RequireComponentAttribute

Позволяет автоматически добавить к GameObject компонент, который необходим для нормальной работы добавляемого скрипта. Проверка отсутствия требуемого компонента (и при необходимости его добавление) выполняется только в момент добавления скрипта к GameObject.

```
using UnityEngine;
```

```
// PlayerScript требует подключения компонента Rigidbody
[RequireComponent(typeof(Rigidbody))]
public class PlayerScript : MonoBehaviour
{
    Rigidbody rb;
    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }
    void FixedUpdate()
    {
        rb.AddForce(Vector3.up);
    }
}
```

SerializeField

Обеспечивает сериализацию (в частности, отображение в инспекторе) закрытого или защищенного поля компонента. Для открытых полей сериализация выполняется автоматически. Указанная сериализация является внутренним механизмом Unity и не связана со стандартными механизмами сериализации .NET Framework.

Сериализация свойств и статических полей не поддерживается.

Сериализуемые типы:

- все классы, наследуемые от `UnityEngine.Object`, в частности, `GameObject`, `Component`, `MonoBehaviour`, `Texture2D`, `AnimationClip`;
- все базовые типы (`int`, `string`, `float`, `bool`);
- некоторые встроенные типы (`Vector2`, `Vector3`, `Vector4`, `Quaternion`, `Matrix4x4`, `Color`, `Rect`, `LayerMask`);
- массивы с сериализуемыми элементами;
- списки `List` с сериализуемыми элементами;
- перечисления;
- структуры.

Словари `Dictionary` не являются сериализуемыми типами.

```
using UnityEngine;
public class SomePerson : MonoBehaviour
{
    public string firstName = "John"; // Сериализуется
    private int age = 40; // Не сериализуется
    [SerializeField]
    private bool hasHealthPotion = true; // Сериализуется
    ...
}
```

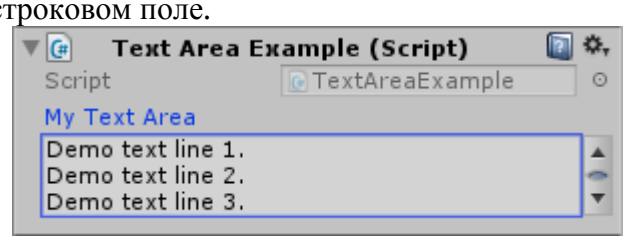
SpaceAttribute

Добавляет перед указанным полем дополнительный вертикальный промежуток в инспекторе.

```
using UnityEngine;
public class ExampleClass : MonoBehaviour
{
    public int health = 0;
    public int maxHealth = 100;
    [Space(10)]
    public int shield = 0;
    public int maxShield = 0;
}
```

TextAreaAttribute

Расширенный вариант MultilineAttibute, позволяющий задавать минимальное и максимальное количество строк в многострочном компоненте и добавляющий полосу прокрутки при увеличении строк в данном строковом поле.



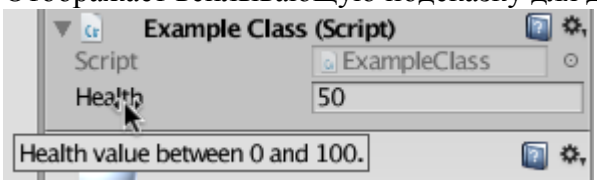

```
public TextAreaAttribute();
public TextAreaAttribute(int minLines, int maxLines);
```

`minLines`: минимальное число строк в многострочном поле;

`maxLines`: максимальное число строк в многострочном поле (при превышении этого числа появляется полоса прокрутки).

TooltipAttribute

Отображает всплывающую подсказку для данного поля в инспекторе.




```
using UnityEngine;
using System.Collections;
public class ExampleClass : MonoBehaviour
{
    [Tooltip("Health value between 0 and 100.")]
    public int health = 0;
}
```

Сериализация

Сериализация — это действие по превращению находящегося в памяти объекта или набора объектов, ссылающихся друг на друга, в плоское представление в виде потока байтов или XML·узлов, который может быть сохранен или передан по сети. Десериализация получает поток данных и восстанавливает его в объект или набор объектов в памяти.

Двоичный сериализатор

Двоичная сериализация хорошо автоматизирована и может обрабатывать сложные наборы объектов с минимальным вмешательством.

Сделать тип поддерживающим двоичную сериализацию можно либо с помощью атрибутов, либо путем реализацией интерфейса `ISerializable`.

Базовые возможности

Тип делается сериализируемым с помощью единственного атрибута:

```
[Serializable]
public sealed class Person
{
  public string Name;
  public int Age;
}
```

Атрибут `[Serializable]` инструктирует сериализатор о необходимости сериализации *всех* полей (как закрытых, так и открытых). Каждое поле само должно допускать сериализацию, иначе сгенерируется исключение. Примитивные типы .NET, такие как `string` и `int`, поддерживают сериализацию.

Атрибут `Serializable` не наследуется, так что подклассы не будут автоматически сериализируемыми, если их не пометить этим атрибутом.

В случае автоматических свойств (`{ get; [private|protected] set; }`) механизм двоичной сериализации сериализирует лежащие в основе поля, генерируемые компилятором. Однако имена таких полей могут изменяться при перекомпиляции типа, нарушая совместимость с существующими сериализированными данными. Обойти эту проблему можно либо за счет устранения автоматических свойств в типах `[Serializable]`, либо путем реализации интерфейса `ISerializable`.

Чтобы сериализировать экземпляр `Person`, необходимо создать объект-*форматер* и вызвать метод `Serialize`. Примером форматера является класс `BinaryFormatter`, определенный в пространстве имен `System.Runtirne.Serialization.Formatters.Binary` (имеется также форматер `SoapFormatter`).

Пример сериализации объекта `Person` с помощью `BinaryForrnatter`:

```
Person p = new Person() { Name = "George", Age = 25 };
IFormatter formatter = new BinaryFormatter();
using (FileStream s = File.Create("serialized.bin"))
  formatter.Serialize(s, p);
```

Все данные, необходимые для реконструирования объекта `Person`, записываются в файл serialized.bin. Метод `Deserialize` восстанавливает объект:

```
using (FileStream s = File.OpenRead("serialized.bin"))
{
  Person p2 = (Person)formatter.Deserialize(s);
  Console.WriteLine(p2.Name +" "+ p2.Age);  // George 25
}
```

Сериализированные данные включают полные сведения о типе и сборке, поэтому если попытаться привести результат десериализации к совпадающему типу `Person` из другой сборки, то возникнет ошибка. Десериализатор восстанавливает объектные ссылки полностью в их исходном состоянии. Это касается коллекций, которые трактуются просто как сериализируемые объекты, подобные любым другим (все коллекции, определенные в пространствах имен `System.Collections.*`, помечены как сериализируемые).

Механизм двоичной сериализации может обрабатывать сложные наборы объектов, не требуя специальной поддержки (кроме обеспечения возможности сериализации всех участвующих членов).

Атрибуты двоичной сериализации

[NonSerialized]

Механизм двоичной сериализации реализует *политику отключения*. Поля, которые не должны сериализироваться, требуется явно пометить с помощью атрибута [NonSerialized]:

```
[Serializable]
public sealed class Person
{
  public string Name;
  public DateTime DateOfBirth;
  // Поле Age может быть вычислено, поэтому в сериализации не нуждается:
  [NonSerialized] public int Age;
}
```

Несериализированные члены при десериализации всегда получают пустое значение или null, даже если инициализаторы полей или конструкторы устанавливают их по-другому.

[OnDeserializing], [OnDeserialized]

Десериализация пропускает все обычные конструкторы, а также инициализаторы полей. Это не особенно важно, когда в сериализации принимают участие все поля, но может привести к проблемам, если некоторые поля исключены через [NonSerialized]. Пример:

```
public sealed class Person
{
  public string Name;
  public DateTime DateOfBirth;
  [NonSerialized] public int Age;
  [NonSerialized] public bool Valid = true;
  public Person() { Valid = true; }
}
```

В десериализированном объекте Person поле Valid будет иметь значение false — несмотря на его установку в true внутри конструктора и инициализатора поля.

Для решения проблемы нужно определить специальный «конструктор» десериализации с помощью атрибута [OnDeserializing]. Метод, помеченный этим атрибутом, будет вызываться *перед* десериализацией:

```
[OnDeserializing]
void OnDeserializing(StreamingContext context)
{
  Valid = true;
}
```

Для обновления вычисляемого поля Age можно также использовать метод, помеченный атрибутом [OnDeserialized] (этот метод запускается *после* десериализации):

```
[OnDeserialized]
void OnDeserialized(StreamingContext context)
{
  TimeSpan ts = DateTime.Now – DateOfBirth;
  Age = ts.Days / 365; // Примерный возраст в годах
}
```

[OnSerializing], [OnSerialized]

Механизм двоичной сериализации также поддерживает атрибуты [OnSerializing] и [OnSerialized]. С их помощью помечается метод, подлежащий выполнению до или после сериализации.

[OptionalField]

Добавление к классу нового поля нарушает совместимость с данными, которые уже сериализированы, если не пометить это новое поле атрибутом [OptionalField].

Атрибут OptionalField сообщает десериализатору о том, что если в потоке данных не встречаются данные поля, то он должен считать эти поля несериализированными. Эти недостающие поля можно обработать в методе [OnDeserializing].

Возможна также ситуация, когда десериализатор обнаруживает постороннее поле. Двоичный форматер в этой ситуации отбрасывает посторонние данные.

Двоичная сериализация с помощью ISerializable

Реализация интерфейса `ISerializable` предоставляет типу полный контроль над прохождением его двоичной сериализации и десериализации. Класс, реализующий данный интерфейс, должен быть дополнительно помечен атрибутом `[Serializable]`.

Определение интерфейса `ISerializable`:

```
public interface ISerializable
{
  void GetObjectData(SerializationInfo info, StreamingContext context);
}
```

Метод `GetObjectData` запускается при сериализации; его работа заключается в наполнении объекта `SerializationInfo` (словарь пар «имя/значение») данными из всех полей, которые необходимо сериализировать:

```
public virtual void GetObjectData(SerializationInfo info, StreamingContext context)
{
  info.AddValue("Name", Name);
  info.AddValue("DateOfBirth", DateOfBirth);
}
```

Сами значения могут иметь любой сериализируемый тип; при необходимости платформа .NET Framework будет выполнять рекурсивную сериализацию. В словаре разрешено хранить значения `null`.

Целесообразно объявить метод `GetObjectData` как `virtual` (если класс не является `sealed`). Это позволит подклассам расширять сериализацию без необходимости в повторной реализации интерфейса `ISerializable`.

В дополнение к реализации интерфейса `ISerializable` тип, управляющий собственной сериализацией, должен предоставить *конструктор десериализации*, который принимает такие же два параметра, как и метод `GetObjectData`. Этот конструктор может быть объявлен с любым уровнем доступа — исполняющая среда все равно его найдет (обычно он объявляется как `protected`).

Пример конструктора десериализации:

```
protected Person(SerializationInfo si, StreamingContext sc)
{
  Name = si.GetString("Name");
  DateOfBirth = (DateTime)si.GetValue("DateOfBirth", typeof(DateTime));
}
```

Для широко используемых типов в классе `SerializationInfo` есть типизированные методы `Get*`, такие как `GetString`, предназначенные для упрощения реализации конструкторов десериализации. В случае указания имени, для которого данные не существуют, генерируется исключение.

Создание подклассов из сериализируемых классов

Рассмотрим следующую иерархию классов:

```
[Serializable]
public class Person
{
  public string Name;
  public int Age;
}
[Serializable]
public class Student : Person
{
  public string Course;
}
```

В этом примере классы Person и Student являются сериализируемыми, и оба они задействуют стандартное поведение сериализации исполняющей среды, так как ни один из них не реализует интерфейс ISerializable.

Предположим, что разработчик класса Person решил по какой-то причине реализовать интерфейс ISerializable и предоставить конструктор десериализации, чтобы управлять сериализацией Person. Новая версия Person может иметь такой вид:

```
[Serializable]
public class Person : ISerializable
{
  public string Name;
  public int Age;
  public virtual void GetObjectData(SerializationInfo si, StrearningContext sc)
  {
    si.AddValue("Name", Name);
    si.AddValue("Age", Age);
  }
  protected Person(SerializationInfo si, StrearningContext sc)
  {
    Name = si.GetString("Name");
    Age = si.GetInt32("Age");
  }
  public Person() { }
}
```

Несмотря на возможность работы с экземплярами Person, это изменение нарушает сериализацию экземпляров Student. Сериализация экземпляра Student выглядит успешной, однако поле Course в типе Student не сохраняется в потоке, поскольку реализации метода ISerializable.GetObjectData в классе Person ничего не известно о членах типа, производного от Student. Вдобавок десериализация экземпляров Student генерирует исключение, потому что исполняющая среда не найдет конструктор десериализации в Student.

Решение упомянутой проблемы предусматривает реализацию интерфейса ISerializable с самого начала для сериализируемых классов, которые являются открытыми и незапечатанными.

Класс Student должен быть реализован следующим образом:

```
[Serializable]
public class Student: Person
{
  public string Course;
  public override void GetObjectData(SerializationInfo si, StrearningContext sc)
  {
    base.GetObjectData(si, sc);
    si.AddValue("Course", Course);
  }
  protected Student(SerializationInfo si, StreamingContext sc) : base(si, sc)
  {
    Course = si.GetString("Course");
  }
  public Student() { }
}
```

Сериализация XML

Платформа .NET Framework предлагает в пространстве имен System.Xml.Serialization отдельный механизм сериализации XML под названием XmlSerializer. Он используется для сериализации типов .NET в XML-файлы.

Как и в случае механизма двоичной сериализации, возможны два подхода:

• добавить к типам соответствующие атрибуты (определенные в System.Xml.Serialization);

• реализовать интерфейс IXmlSerializable.

Однако, в отличие от механизма двоичной сериализации, реализация интерфейса (т. е. IXmlSerializable) полностью избегает использования механизма, оставляя на разработчика самостоятельное написание кода сериализации с участием классов XmlReader и XmlWriter. Мы не будем рассматривать этот подход.

Сериализация на основе атрибутов

Для применения класса XmlSerializer необходимо создать его экземпляр и вызвать метод Serialize или Deserialize с потоком и интересующим объектом. Предположим, что определен следующий класс:

```
public class Person
{
  public string Name;
  public int Age;
}
```

Приведенный ниже код сохраняет объект Person в XML-файл и затем его восстанавливает:

```
Person p = new Person ();
p.Name = "Peter";
p.Age = 30;
XmlSerializer xs = new XmlSerializer(typeof(Person));
using (Stream s = File.Create("person.xml"))
  xs.Serialize (s, p);
Person p2;
using (Stream s = File.OpenRead("person.xml"))
  p2 = (Person)xs.Deserialize(s);
Console.WriteLine (p2.Name + " " + p2.Age); // Peter 30
```

Методы Serialize и Deserialize могут работать с объектами Stream, XmlWriter/XmlReader или TextWriter/TextReader. Результирующий XML:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>Peter</Name>
  <Age>30</Age>
</Person>
```

Класс XmlSerializer способен сериализировать типы, не имеющие ни одного атрибута — такие как тип Person. По умолчанию он сериализирует все *открытые* поля и свойства типа. Исключить члены из числа сериализируемых можно посредством атрибута XmlIgnore:

```
public class Person
{
  [XmlIgnore]
  public DateTime DateOfBirth;
}
```

В отличие от механизма двоичной сериализации класс XmlSerializer не распознает атрибут [OnDeserializing], а вместо него при десериализации полагается на конструктор *без параметров*, генерируя исключение, если он отсутствует. (Заметим, что в приведенном выше примере класс Person имеет неявный конструктор без параметров.) Это, в частности, обеспечивает инициализацию всех полей перед десериализацией:

```
public class Person
{
  public bool Valid = true;
  // Выполняется перед десериализацией
}
```

XmlSerializer может сериализировать почти любой тип, в том числе

• примитивные типы, типы DateTime, TimeSpan, Guid, а также их версии, допускающие null;
• тип byte[] (который преобразуется с использованием кодировки Base64);
• любой тип, реализующий интерфейс IXmlSerializable;
• любой тип коллекции.

Десериализатор является переносимым в плане версий: он не возбуждает исключения, если элементы или атрибуты отсутствуют либо встречаются лишние данные.

Атрибуты, имена и пространства имен

По умолчанию поля и свойства сериализируются в XML-элементы. Потребовать, чтобы взамен применялся XML-атрибут, можно следующим образом:

```
[XmlAttribute]
public int Age;
```

Именем элемента или атрибута можно управлять:

```
public class Person
{
  [XmlElement("FirstName")]
  public string Name;
  [XmlAttribute("RoughAge")]
  public int Age;
}
```

```
<Person RoughAge="30" ... >
  <FirstName>Peter</FirstName>
</Person>
```

Стандартное пространство имен XML является пустым. Для указания пространства имен XML атрибуты [XmlElement] и [XmlAttribute] поддерживают именованный аргумент Namespace. Можно также назначить имя и пространство имен самому типу с помощью атрибута [XmlRoot]:

```
[XmlRoot("Candidate", Namespace = "http://mynamespace/test/")]
public class Person { ... }
```

Порядок следования XML-элементов

Класс XmlSerializer записывает элементы в том порядке, в каком они определены в классе. Такое поведение можно изменить, указывая значение для именованного аргумента Order в атрибуте XmlElement:

```
public class Person
{
  [XmlElement(Order=2)] public string Name;
  [XmlElement(Order=1)] public int Age;
}
```

Если аргумент Order применяется, то должен присутствовать везде.

Десериализатор не учитывает порядок следования элементов.

Подклассы и дочерние объекты

Предположим, что корневой тип имеет два подкласса:

```
public class Person { public string Name; }
public class Student : Person { }
public class Teacher : Person { }
```

и реализован метод для сериализации корневого типа:

```
public void SerializePerson (Person p, string path)
{
  XmlSerializer xs = new XmlSerializer (typeof(Person));
  using (Stream s = File.Create (path))
    xs.Serialize (s, p);
}
```

Чтобы данный метод работал с объектом Student или Teacher, экземпляр XmlSerializer должен быть информирован о существовании упомянутых подклассов. Сделать это можно двумя способами. Первый из них — регистрация каждого подкласса с помощью атрибута XmlInclude:

```
[XmlInclude(typeof(Student))]
[XmlInclude(typeof(Teacher))]
public class Person { public string Name; }
```

Второй способ — указание каждого подтипа при конструировании экземпляра XmlSerializer:

```
Xmlserializer xs = new XInlSerializer(typeof(Person),
  new Type[] { typeof(Student), typeof(Teacher) });
```

В любом случае сериализатор реагирует помещением подтипа в атрибут xsi:type:

```
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="Student">
  <Name>Peter</Name>
</Person>
```

Зная этот атрибут, десериализатор затем создает объект типа Student, а не Person.

Именем, находящимся в XML-атрибуте type, можно управлять, применяя к подклассу атрибут [XmlType]:

```
[XmlType("Candidate")]
public class Student : Person { }
```

Результат:

```
<Person xmlns:xsi=" ... " xsi:type="Candidate">
```

Сериализация дочерних объектов

Класс XmlSerializer автоматически рекурсивно обрабатывает объектные ссылки, такие как поле HomeAddress в Person:

```
public class Person
{
  public string Name;
  public Address HomeAddress = new Address();
}
public class Address
{
  public string Street, PostCode;
}
```

Например:

```
Person p = new Person();
p.Name = "Peter";
p.HomeAddress.Street = "Odo St";
p.HomeAddress.PostCode = "6020";
```

Результирующий XML:

```
<Person ... >
  <Name>Peter</Name>
  <HomeAddress>
    <Street>Odo St</Street>
    <PostCode>6020</PostCode>
  </HomeAddress>
</Person>
```

Создание подклассов из дочерних объектов

Предположим, что нужно сериализировать класс Person, который может ссылаться на подклассы класса Address следующим образом:

```
public class Address { public string Street, PostCode; }
public class USAddress : Address { }
```

```
public class AUAddress : Address { }
public class Person
{
  public string Name;
  public Address HomeAddress = new USAddress();
}
```

В зависимости от того, как должен структурироваться XML, решить задачу можно двумя отличающимися способами. Если требуется, чтобы имя элемента всегда соответствовало имени поля или свойства с подтипом, записанным в атрибуте `type`:

```
<Person ... >
  <HomeAddress xsi:type="USAddress">
    ...
  </HomeAddress>
</Person>
```

то необходимо применять атрибут `[XmlInclude]` для регистрации каждого подкласса с классом `Address`:

```
[XmlInclude(typeof(AUAddress))]
[XmlInclude(typeof(USAddress))]
public class Address
{
  public string Street, PostCode;
}
```

С другой стороны, если нужно, чтобы имя элемента отражало имя подтипа, давая примерно такой результат:

```
<Person ... >
  <USAddress>
    ...
  </USAddress>
</Person>
```

то вместо этого понадобится указывать множество атрибутов `[XmlElement]` для поля или свойства родительского типа:

```
public class Person
{
  public string Name;
  [XmlElement("Address", typeof(Address))]
  [XmlElement("AUAddress", typeof(AUAddress))]
  [XmlElement("USAddress", typeof(USAddress))]
  public Address HomeAddress = new USAddress();
}
```

Каждый атрибут `[XmlElement]` сопоставляет имя элемента с типом. В случае принятия такого подхода атрибуты `[XmlInclude]` для типа `Address` не потребуются (хотя их наличие не нарушит сериализацию).

Если опустить имя элемента в `[XmlElement]` (и указать только тип), то будет использоваться стандартное имя типа (на которое оказывает влияние атрибут `[XmlType]`, но не `[XmlRoot]`).

Сериализация коллекций

Класс `XmlSerializer` распознает и сериализирует конкретные типы коллекций, не требуя какого-то вмешательства:

```
public class Person
{
  public string Name;
  public List<Address> Addresses = new List<Address>();
}
public class Address
```

```
{
  public string Street, PostCode;
}
```

Результирующий XML выглядит следующим образом:

```
<Person ... >
  <Name> ... </Name>
  <Addresses>
    <Address>
      <Street> ... </Street>
      <Postcode> ... </Postcode>
    </Address>
    <Address>
      <Street> ... </Street>
      <Postcode> ... </Postcode>
    </Address>
  </Addresses>
</Person>
```

Атрибут `[XmlArray]` позволяет переименовывать внешний элемент (т.е. `Addresses`).

Атрибут `[XmlArrayItem]` позволяет переименовывать внутренние элементы (т. е. элементы `Address`).

Пример:

```
public class Person
{
  public string Name;
  [XmlArray("PreviousAddresses")]
  [XmlArrayItem("Location")]
  public List<Address> Addresses = new List<Address>();
}
```

```
<Person ... >
  <Name> ... </Name>
  <PreviousAddresses>
    <Location>
      <Street> ... </Street>
      <Postcode> ... </Postcode>
    </Location>
    <Location>
      <Street> ... </Street>
      <Postcode> ... </Postcode>
    </Location>
  </PreviousAddresses>
</Person>
```

Атрибуты `XmlArray` и `XmlArrayItem` также позволяют указывать пространства имен XML.

Возможна сериализация коллекций без внешнего элемента:

```
<Person ... >
  <Name> ... </Name>
  <Address>
    <Street> ... </Street>
    <Postcode> ... </Postcode>
  </Address>
  <Address>
    <Street> ... </Street>
    <Postcode> ... </Postcode>
```

```
  </Address>
</Person>
```

Для этого к полю или свойству коллекции необходимо добавить атрибут [XmlElement]:

```
public class Person
{
  ...
  [XmlElement("Address")]
  public List<Address> Addresses new List<Address>();
}
```

Работа с элементами коллекции, являющимися подклассами

Правила для элементов коллекции, являющихся подклассами, естественным образом следуют из других правил, применяемых к подклассам. Чтобы закодировать элементы, являющиеся подклассами, с помощью атрибута type, например:

```
<Person ... >
  <Name> ... </Name>
  <Addresses>
    <Address xsi:type="AUAddress">
...
```

понадобится добавить атрибуты [XmlInclude] к базовому типу (Address), как делалось ранее. Это работает независимо от того, подавляется сериализация внешнего элемента или нет.

Если элементы, являющиеся подклассами, должны именоваться в соответствии с их типом, например:

```
<Person ... >
  <Name> ... </Name>
  <!--начало необязательного внешнего элемента-->
    <AUAddress>
      <Street> ... </Street>
      <Postcode> ... </Postcode>
    </AUAddress>
    <USAddress>
      <Street> ... </Street>
      <Postcode> ... </Postcode>
    </USAddress>
  <!--конец необязательного внешнего элемента-->
</Person>
```

то для поля или свойства коллекции потребуется задать множество атрибутов [XmlArrayItem] или [XmlElement].

Указывайте множество атрибутов [XmlArrayItem], если нужно *включить* внешний элемент коллекции:

Указывайте множество атрибутов [XmlElement], если нужно *исключить* внешний элемент коллекции:

Специальный механизм сериализации данных в Unity

Данные для скриптов Unity можно сохранять с помощью методов класса PlayerPrefs.

При этом для Windows эти данные сохраняются в реестре по следующему пути:

HKEY_CURRENT_USER | Software | <Имя компании> | <Имя проекта>

Имя компании и имя проекта можно получить и изменить с помощью команды меню среды Unity Edit | Project settings | Player.

По умолчанию имя компании имеет вид DefaultCompany.

Если методы класса PlayerPrefs вызываются не при выполнении программы, а в редакторе Unity (как в приводимом далее примере), то данные сохраняются по другому пути:

HKEY_CURRENT_USER | Software | Unity | Unity Editor | <Имя компании> | <Имя проекта>

Пример использования атрибутов и сериализации данных в сценарии Unity

```
using UnityEngine;
using System;
using System.Collections;
using System.Collections.Generic;
using System.Xml.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;

[HelpURL("https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-
guide/concepts/serialization/")]
public class SerAttrDemo : MonoBehaviour
{
    [Flags]
    public enum BorderSides
    {
        None=0,
        Left=1, Right=2, Top=4, Bottom=8,
        LeftRight = Left | Right,
        TopBottom = Top | Bottom,
        All = LeftRight | TopBottom
    }
    [Serializable]
    public class GameData
    {
        [SerializeField] [Range(1,100)]
        [Tooltip("Может принимать значения от 1 до 100")]
        int privateField = 1;
        [HideInInspector]
        public int hidedField = 1;
        [TextArea(1,3)] [ContextMenuItem("Clear","ClearComment")]
        public string Comment;
        [XmlAttribute]
        public float X;
        [XmlAttribute]
        public float Y;
        [XmlAttribute]
        public float Z;
        [Space(10)]
        public float RotX;
        public float RotY;
        public float RotZ;
        [Space(10)]
        public float ScaleX;
        public float ScaleY;
        public float ScaleZ;
    }

   public BorderSides borderSides;
```

```
    [Header("Serialized data")]
    [ContextMenuItem("Get Transform","GetTransform")]
    [ContextMenuItem("Set Transform","SetTransform")]
    public GameData data = new GameData();

    public void ClearComment()
    {
        data.Comment = "";
    }
    private void GetTransform()
    {
        data.X = transform.position.x;
        data.Y = transform.position.y;
        data.Z = transform.position.z;
        data.RotX = transform.localRotation.eulerAngles.x;
        data.RotY = transform.localRotation.eulerAngles.y;
        data.RotZ = transform.localRotation.eulerAngles.z;
        data.ScaleX = transform.localScale.x;
        data.ScaleY = transform.localScale.y;
        data.ScaleZ = transform.localScale.z;
    }
    private void SetTransform()
    {
        Transform t = transform;
        transform.position = new Vector3 (data.X, data.Y, data.Z);
        transform.rotation = Quaternion.Euler(data.RotX,
            data.RotY, data.RotZ);
        transform.localScale = new Vector3(data.ScaleX,
            data.ScaleY, data.ScaleZ);
    }
    [ContextMenu("Save XML")]
    public void SaveXML()
    {
        SaveXML("GameData.xml");
    }
    public void SaveXML(string fileName)
    {
        XmlSerializer serializer = new XmlSerializer(typeof(GameData));
        FileStream file = File.Create(fileName);
        serializer.Serialize(file, data);
        file.Close();
        print(data.X);
    }
    [ContextMenu("Load XML")]
    public void LoadXML()
    {
        LoadXML("GameData.xml");
    }
    public void LoadXML(string fileName)
    {
        if(!File.Exists(fileName)) return;
```

```
        XmlSerializer serializer = new XmlSerializer(typeof(GameData));
        FileStream file = File.OpenRead(fileName);
        data = serializer.Deserialize(file) as GameData;
        file.Close();
        print(data.X);
    }
    [ContextMenu("Save Binary")]
    public void SaveBinary()
    {
        SaveBinary("GameData.sav");
    }
    public void SaveBinary(string fileName)
    {
        BinaryFormatter bf = new BinaryFormatter();
        FileStream file = File.Create(fileName);
        bf.Serialize(file, data);
        file.Close();
    }
    [ContextMenu("Load Binary")]
    public void LoadBinary()
    {
        LoadBinary("GameData.sav");
    }
    public void LoadBinary(string fileName)
    {
        if(!File.Exists(fileName)) return;
        BinaryFormatter bf = new BinaryFormatter();
        FileStream file = File.OpenRead(fileName);
        data = bf.Deserialize(file) as GameData;
        file.Close();
    }
    [ContextMenu("Save PlayerPrefs")]
    public void SavePlayerPrefs()
    {
        PlayerPrefs.SetString("Comment", data.Comment);
        PlayerPrefs.SetFloat("X", data.X);
        PlayerPrefs.SetFloat("Y", data.Y);
        PlayerPrefs.SetFloat("Z", data.Z);
        PlayerPrefs.SetFloat("RotX", data.RotX);
        PlayerPrefs.SetFloat("RotY", data.RotY);
        PlayerPrefs.SetFloat("RotZ", data.RotZ);
        PlayerPrefs.SetFloat("ScaleX", data.ScaleX);
        PlayerPrefs.SetFloat("ScaleY", data.ScaleY);
        PlayerPrefs.SetFloat("ScaleZ", data.ScaleZ);
    }
    [ContextMenu("Load PlayerPrefs")]
    public void LoadPlayerPrefs()
    {
        data.Comment = PlayerPrefs.GetString("Comment", "");
        data.X = PlayerPrefs.GetFloat("X", 0.0f);
        data.Y = PlayerPrefs.GetFloat("Y", 0.0f);
```

```
        data.Z = PlayerPrefs.GetFloat("Z", 0.0f);
        data.RotX = PlayerPrefs.GetFloat("RotX", 0.0f);
        data.RotY = PlayerPrefs.GetFloat("RotY", 0.0f);
        data.RotZ = PlayerPrefs.GetFloat("RotZ", 0.0f);
        data.ScaleX = PlayerPrefs.GetFloat("ScaleX", 1.0f);
        data.ScaleY = PlayerPrefs.GetFloat("ScaleY", 1.0f);
        data.ScaleZ = PlayerPrefs.GetFloat("ScaleZ", 1.0f);
    }
}
```