ПИШНИК: ВЕСОМЕ КАКАРО

Глава 13: Перечисления. Структуры. Nullable-Типы

Автор презентации – Сагалов Даниил БПП-196

Значимые Типы в C#

Как уже известно, типы данных в C# делятся на значимые и ссылочные. Помните, что классы – это всегда ссылочные типы.

Но тогда каким образом можно создавать собственные значимые типы в C#? Для этого в языке существуют **структуры (struct)** и **перечисления (enum)**. В отличие от ссылочных типов значимые хранятся в стеке. Благодаря этой особенности выделение и освобождение памяти для значимых типов является более дешёвыми операциями по сравнению с аналогичными для ссылочных типов (размещением в куче и удаление из неё.

Перечисления

Перечисление – значимый тип данных, состоящий из именованных целочисленных констант, создаётся при помощи ключевого слова **enum**. Константы могут быть только одним из указанных типов: **byte**, **sbyte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**. По умолчанию константы имеют тип **Int32** и не могут иметь тип char. Базовым для всех перечислений является **класс System.Enum**, **наследник System.ValueType**.

При объявлении типа перечисления после двоеточия можно явно указать тип значений констант (синтаксис аналогичен наследованию). Затем идёт список именованных констант, перечисленных через запятую. Вы не можете определять функциональные члены в перечислениях напрямую, для этих целей разрешается использовать методы расширения.

```
public enum CompassDirections : byte
{
    North = 0,
    West = 1,
    South = 2,
    East = 3
}
```

По умолчанию нумерация констант начинается с нуля, а каждая последующая константа больше предыдущей на 1. Вы можете явно указывать значения констант, причём при такой схеме последующие не пронумерованные константы всё равно будут иметь значения на 1 больше предыдущих.

```
public enum KakapoStates
{
    Cute = 4,
    Fluffy,
    Kakapopable = 3,
    Adorable,
    Smart
}
```

KakapoStates.Smart = 5

Важно помнить, что значения констант в перечислениях могут повторяться, это никак не отслеживается компилятором. Делать так не рекомендуется.

Работа с Перечислениями

Так как перечисления являются типом, вы можете создавать переменные и поля типов перечислений. Такие поля могут иметь только те значения, которые позволяются значимым типом, взятым за основу перечисления.

И так, Вы можете:

- Присвоить любое число, помещающееся в базовый тип с помощью явного приведения или же через unchecked.
- Константу из списка доступных вариантов в формате <Имя Перечисления>.<Имя Константы>.
- Явно привести константу у её базовому типу значения.

```
public enum Legends : byte
{
    Kodzima = 0,
    ChuckNorris = 1,
    SteveJobs = 2,
    WaltDysney = 3
}

class EnumDemo
{
    static readonly Random randomizer = new Random();

    static void Main()
    {
        // Требует явного приведения к типу перечисления
        Legends legendaryGuy = (Legends)randomizer.Next(4);
        Console.WriteLine(legendaryGuy);
        // При использовании констант используем явное приведение
        legendaryGuy = (Legends)2;
        // Вы также можете использовать любое другое значение через
        // явное приведение, однако оно должно помещаться в тип
        // В таком случае константа всегда будет отображаться численно
        legendaryGuy = (Legends)255;
        // Вы можете работать с переполнением с unchecked
        legendaryGuy = unchecked((Legends)256);
        // Можно использовать <Имя Перечисления>.<Имя Константы>
        legendaryGuy = Legends.SteveJobs;
        // Константу можно привести к её значению
        Console.WriteLine((int)legendaryGuy);
    }
}
```

Неоднозначность Констант

```
enum Legends1 : byte {
    Kodzima = 0,
    ChuckNorris = 1,
    SteveJobs = 0,
    WaltDisney = 1,
}

enum Legends2 : byte {
    Kodzima = 0,
    ChuckNorris = 1,
    SteveJobs = 0,
    WaltDisney = 1,
    JackieChan
}

static readonly Random randomizer = new Random();

static void Main() {
    // Из-за использования бин. поиска в 1 случае
    // Всегда значение - Kodzima, во втором -
    // SteveJobs (зависит от чётности количества констант)
    Legends1 legend1 = 0;
    Legends2 legend2 = 0;
    Console.WriteLine(legend1 + " " + legend2);
    legend1 = Legends1.WaltDisney;
    // При обратном приведении значение всё
    // равно будет ChuckNorris
    Console.WriteLine((Legends1)((int)legend1));
}
```

При появлении констант с одинаковыми значениями, при попытках присваивания чисел всегда будет задаваться одно и то же значение константы. Связано это с тем, что для определения связанной с числом константы используется алгоритм бинарного поиска.

В данном примере для типа Legends1 при присваивании 0 всегда константа будет иметь значение Kodzima, для Legends2 – SteveJobs. Аналогичную ситуацию можно наблюдать после приведения константы WaltDisney к int и обратно: итоговым значением будет ChuckNorris.

Перечисления как Битовые Флаги

Одним из интересных применений перечислений является их использование в качестве битовых флагов для комбинаций вариантов. Для этого перед объявлением типа перечисления вам нужно добавить атрибут **[Flags]**. При использовании данного атрибута ToString( ) выведет через запятую названия установленных флагов.

Для правильной работы константы такого перечисления должны являться битовыми полями (т. е. степенями 2).

Особенности перечислений как битовых флагов и советы по их использованию:

- Логические операции |, & и ^ можно использовать для комбинирования значений битовых флагов.
- Вы можете задавать константы, являющиеся комбинациями флагов (см. пример).
- Побитовое И позволяет быстро определить, установлен ли нужный флаг.
- В случае с битовыми флагами нулевое значение нельзя использовать при проверке через побитовое И. Вместо этого используйте сравнение с 0.
- Не рекомендуется создавать константы-пустышки для «будущих обновлений».
- При написании метода, принимающего значение перечисления в качестве параметра не забудьте прописать валидацию значения, так как перечислениям можно присваивать любые допустимые значения базового типа.
- Настоятельно рекомендуется определять константу со значением 0 для отсутствия состояния по умолчанию/отсутствию состояния, т. к. CLR по стандарту инициализирует перечисление значением 0.

```
[Flags]
public enum ProgramOptions
{
    // Двоичное представление чисел
    None = 0b_0000_0000,
    Debug = 0b_0000_0001,
    Release = 0b_0000_0010,
    BetaTest = 0b_0000_0100,
    x86Version = 0b_0000_1000,
    DefaultVersion = 0b_0001_0000,
    Administrator = 0b_0010_0000,
    // Настройки стандартного запуска включают
    // в себя комбинацию Release + DefaultVersion
    NormalStart = Release | DefaultVersion
}
```

Строковое Представление Перечислений

Метод ToString( ) перечислений можно использовать для получения строк различных форматов:

- G или g – отображает перечисление в виде имени константы (или её значение, если имени не существует). Для перечисления с атрибутом [Flags] выведется комбинация имён флагов через запятую.
- F или f – аналогично G, однако в случае возможности представления в виде совокупности всех элементов через запятую будет напечатано именно данное представление, даже если атрибут [Flags] не указан.
- D или d – отображает значение перечисления в кратчайшем целочисленном представлении.
- X или x – отображает значение перечисления в шестнадцатеричном виде со всеми ведущими нулями.

Задачи

Задание 1

В результате выполнения фрагмента программы:

```
using System;

class Program {
    enum MyEnum {
        a = 1,
        b, c,
        d = 2,
        e = 5,
        f,
        g = b - c
    }
    static void Main() {
        MyEnum m = new MyEnum();
        Console.Write(m.HasFlag((MyEnum)(8 - (int)MyEnum.b)));
    }
}
```

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 2

Выберите допустимые типы для констант перечисления (*укажите все верные ответы*)**:**

1) `int`;
2) `uint`;
3) `double`;
4) `float`;
5) `object`;

Задание 3

В результате выполнения фрагмента программы:

```
using System;

class Program {
    enum MyEnum {
        a = 1,
        b, c,
        d = 2,
        e = 5,
        f,
        g = b - c
    }
    static void Main() {
        Console.Write((int)MyEnum.b + (int)MyEnum.g);
    }
}
```

на экран будет выведено:

Примечание:

*Если возникнет ошибка компиляции, введите: ****

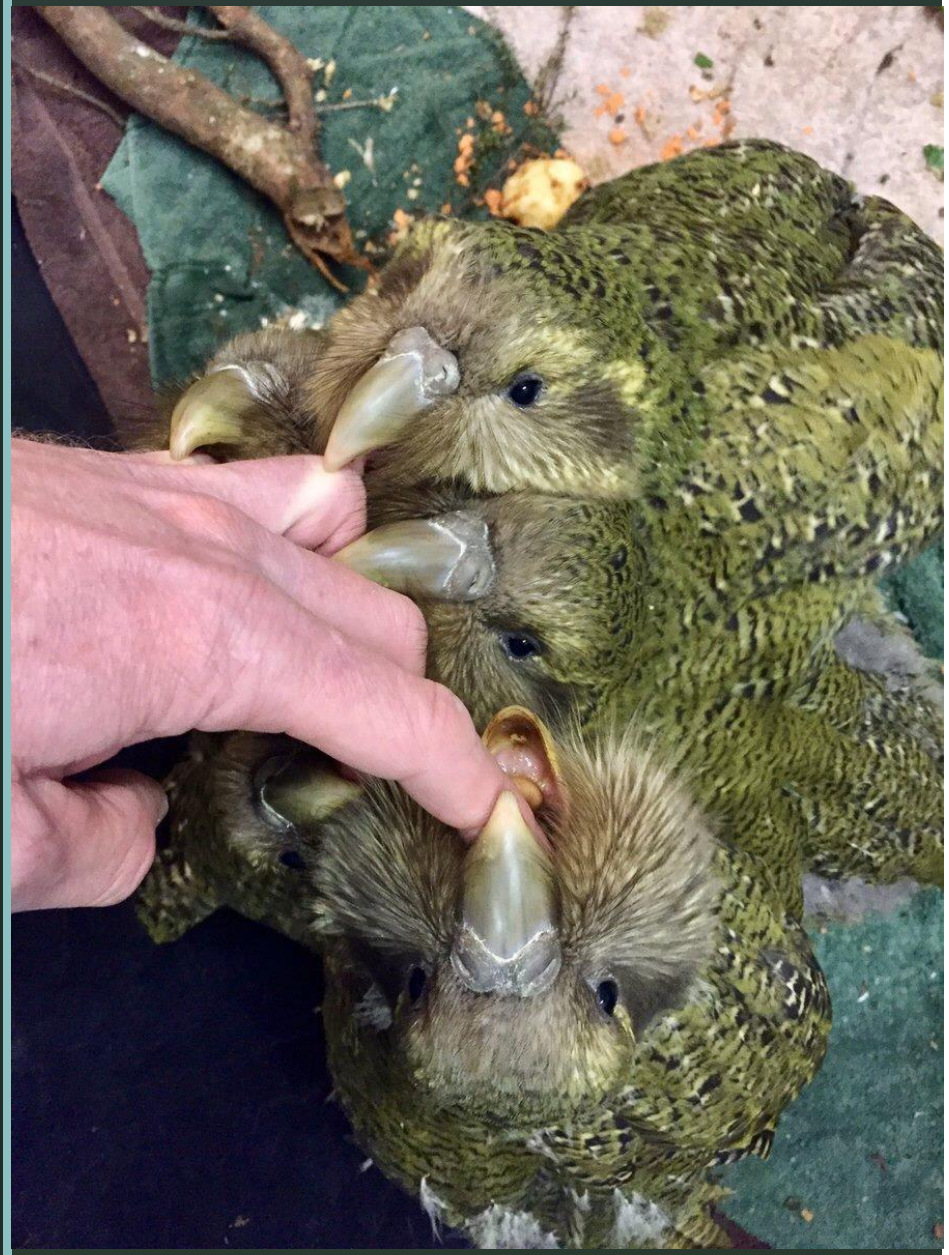
Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---

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

Задание 4

Про перечисление верно (*укажите все верные ответы*)**:**

1) Может быть внутри другого перечисления.
2) Может содержать в себе вещественные константы.
3) Может содержать в себе статические константы.
4) Базовый тип для констант - `int`.
5) Запятая разделяет константы перечисления вместо точки с запятой.



Ответы

Ответ 1: **False**

Ответ 2: **12**

Ответ 3: **1**

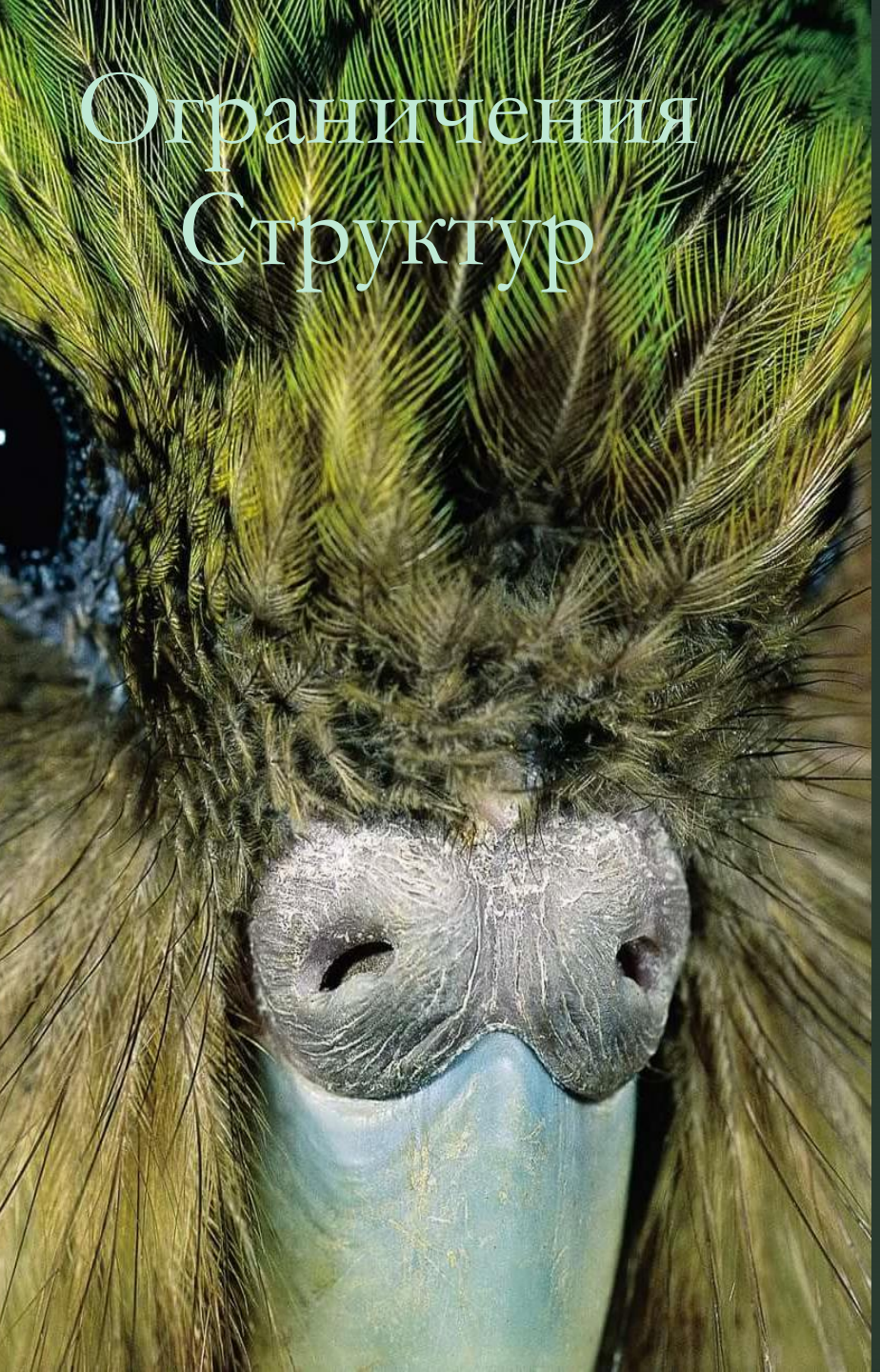
Ответ 4: **45**

Структуры в языке C# – значимые типы, которые могут инкапсулировать данные и функции. Для определения типа структуры используется ключевое слово **struct**. Переменные типов структуры напрямую хранят в себе экземпляры соответствующего типа, при их передаче в методы передаются копии значений (для передачи по ссылке воспользуйтесь ref/out/in). Базовый тип для структур – **System.ValueType**.

Начиная с версии C# 7.2 вы можете помечать структуры модификатором **readonly**. Это позволит вам определять неизменяемые структуры, т. к. для readonly-структур действует правило: все поля и свойства (включая автоматически реализуемые) тоже должны быть помечены как readonly.

Структуры рекомендуется использовать для небольших типов, ориентированных на данные с минимумом поведения. Спецификация языка C# советует определять структуры вместо классов, если экземпляры типа имеют небольшой объём, часто используются кратковременно или на регулярной основе внедряются в другие объекты.

Структуры



Ограничения Структур

При проектировании структур вы столкнётесь со следующими ограничениями:

- В структурах нельзя явно определить конструктор без параметров, он всегда существует неявно и задаёт все поля структуры значениями по умолчанию.
- Конструктор структуры обязан инициализировать все её поля.
- В структурах при объявлении можно инициализировать только статические поля, статические свойства или константы (readonly поля не входят в эту категорию!).
- Структуры не могут наследоваться от каких-либо классов или других структур (от Object структуры наследуются только неявно), однако могут реализовывать сколько угодно интерфейсов. Из этого вытекает, что модификаторы **abstract**, **static** и **sealed** для типов структур недопустимы (**override** применим только для переопределения методов Object, **new** использовать можно).
- Деструкторы в структурах запрещены.

Неполная Инициализация

При работе со структурами можно столкнуться с понятием неполной инициализации. Как мы помним, локальные переменные значимых типов могут использоваться только после их инициализации. При этом экземпляры структур можно создавать и без new, если все поля (и автоматически реализуемые свойства) структуры публичны. Однако в этом случае необходимо явно инициализировать все публичные поля:

Пример:

```
public struct StudentData { public int Id; public int Group; }
class Program {
    static void Main( ) {
        StudentData data;
        data.Group = 196;
        // data.Id = 16; // Раскоментируйте, ошибка компиляции пропадёт (все поля будут инициализированы)
        Console.WriteLine("Group: {0}", data.Group); // OK, данная часть StudentData проинициализирована!
        Console.WriteLine("Group: {0}, Id:{1}", data.Group, data.Id); // Не скомпилируется, Id не инициализирован
    }
}
```

ref struct C# 7.2

Начиная с версии C# 7.2 вы можете определять типы структур с модификатором ref, что делает возможным создание типов структур, которые строго располагаются только в стеке. Таким образом, если Вы пометите структуру как ref, вы никогда не сможете упаковать её или каким-либо другим образом поместить в кучу.

По этой причине ref структуры также не могут реализовывать никаких интерфейсов, т. к. помещение по ссылке типа интерфейса предполагает упаковку.

Важно также понимать, что ref-структуры никогда не могут быть полями внутри классов.

Задачи

Задание 1

Выберите допустимые модификаторы при объявлении структуры (*укажите все верные ответы*)**:**

1) `public;`
2) `abstract;`
3) `partial;`
4) `sealed;`
5) `static;`

Задание 2

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

1) В структуре запрещено объявлять явно конструктор без параметров.
2) При определенных явно параметрических конструкторов в структуре нельзя при создании объекта этой структуры вызвать беспараметрический конструктор.
3) В структурах не допускается использование деструкторов.
4) В структурах можно использовать статический конструктор.
5) В структурах все поля должны быть инициализированы сразу при их объявлении.

Задание 3

Про структуры верно (*укажите все верные ответы*)**:**

1) Все унаследованы от структуры `ValueType`.
2) Не могут иметь вложенных структур.
3) Не могут переопределять методы.
4) Возможно сравнение через == любых двух объектов структур одного типа.
5) Хранятся в стеке.

Задание 4

Верно, что структура может иметь (*укажите все верные ответы*)**:**

1) Защищённое поле.
2) Статический метод.
3) Абстрактный класс.
4) Виртуальное свойство.
5) Публичный индексатор.

Задачи

Задание 5

В результате выполнения фрагмента программы:

```
using System;

struct MyStruct {
    public int X;
    public int Y {
        get => X;
        set => X = value;
    }

    public MyStruct(int Y) {
        this.Y = Y;
    }

    public override string ToString() {
        return $"{X}{Y}";
    }
}

class Program {
    static void Main() {
        MyStruct ms = new MyStruct(100);
        Console.Write(ms.Equals(new MyStruct(100)));
    }
}
```

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 6

В результате выполнения фрагмента программы:

```
using System;

struct MyStruct {
    public int Y;
    MyStruct ms;

    public MyStruct(int Y) {
        this.Y = Y;
        ms.Y = ++Y + Y++;
    }

    public override string ToString() {
        return $"{Y}";
    }
}

class Program {
    static void Main() {
        MyStruct ms = new MyStruct(100);
        Console.Write(ms.Y + new MyStruct(3).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 7

В результате выполнения фрагмента программы:

```
using System;

struct MyStruct {
    public int Y;
    public static MyStruct ms;

    public MyStruct(int Y) {
        this.Y = Y++ + ++Y;
        if (Y > 5) {
            ms = new MyStruct(Y -= 3);
        }
    }
}

class Program {
    static void Main() {
        MyStruct ms = new MyStruct(10);
        Console.Write(ms.Y + new MyStruct(3).Y + MyStruct.ms.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: +++

Задание 8

В результате выполнения фрагмента программы:

```
using System;

struct MyStruct {
    public int Y;
    public MyStruct2 ms2;
    public struct MyStruct2 {
        public int X;
        public MyStruct2(int Y) {
            X = Y++;
        }
    }

    public MyStruct(int Y) {
        ms2 = new MyStruct2(Y -= 3);
        this.Y = Y++ + ++Y / ms2.X;
    }
}

class Program {
    static void Main() {
        MyStruct ms = new MyStruct(10);
        Console.Write(ms.Y + new MyStruct(7).Y + ms.ms2.X);
    }
}
```

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: +++

В результате выполнения фрагмента программы:

```
using System;

struct MyStruct {
    public static int X = 10;
    public int Y;

    public MyStruct(int Y) {
        this.Y = X + Y;
    }

    public static void Meth(int Z) {
        X += new MyStruct(Z).Y;
    }

    public void Meth2(int Z) {
        Y += Z + 10;
    }

    public override string ToString() =>
        $"{X}{Y}";
}

class Program {
    static void Main() {
        MyStruct ms1 = new MyStruct();
        MyStruct ms2 = new MyStruct(7);
        Console.Write(ms1);
        Console.Write(ms2);
        ms1 = ms2;
        Console.Write(ms1);
        Console.Write(ms2);
        MyStruct.Meth(4);
        MyStruct.Meth(2);
        ms1.Meth2(5);
        ms2.Meth2(3);
        Console.Write(ms1);
        Console.Write(ms2);
    }
}
```

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: +++

Задача 9

**большая и какапопистая*

Ответы

Задание	Ответ	Задание	Ответ
1	**13**	6	***
2	**134**	7	**50**
3	**15**	8	**20**
4	**235**	9	**10010171017101750325030**
5	***		

Значимые Nullable-Типы

При работе со значимыми типами вы могли сталкиваться с проблемой, когда вам необходимо использовать значимый тип и при этом иметь возможность присвоить значение null.

Именно для этих целей созданы значимые **nullable-типы**, которые представляют из себя все значения своего базового типа, а также дополнительно значение null. Стоит помнить, что базовый тип при этом не должен быть nullable-типом (нельзя использовать bool??).

В качестве примера применения nullable-типов можно привести bool? – с его помощью вы можете использовать третье значение в качестве неопределённого состояния.

По своей сути nullable-типы представляют собой экземпляры структуры **System.Nullable<T>**. Структура содержит 3 ключевых члена:

- **bool HasValue { get; }** – свойство, возвращает true, если значение структуры не null и false в противном случае.
- **T Value { get; }** – свойство, возвращает значение данного типа, если значение не равно null, иначе выбрасывает **InvalidOperationException**.
- **T GetValueOrDefault ( )** – возвращает Value, если значение HasValue == true или значение базового типа по умолчанию. Имеет перегрузку: **T GetValueOrDefault (T defaultVal)**, позволяющую вернуть конкретное значение базового типа вместо варианта по умолчанию.

```
// Nullable-переменные
bool? nullFlag = null;
int? nullInt = 45;
// Массив nullable дат
DateTime?[] dates = new DateTime?[5];

int myInt1 = 32;
nullInt = myInt1;
// OK, базовый тип неявно приводится к nullable
IComparable example = nullInt;
// Требует явного преобразования, может вызвать исключение
int myInt2 = (int)nullInt;

// nullFlag == null, условие невыполнимо
if (nullFlag is bool flag)
    Console.WriteLine("This line will never appear.");
// Если flag - null, получим значение типа по умолчанию,
// в противном случае - само значение. В данном случае
// присваивание будет эквивалентно nullFlag = false;
nullFlag = nullFlag.GetValueOrDefault();
```

Любой базовый тип можно **неявно** преобразовать к соответствующему nullable-типу, между стандартными значимыми nullable-типами действуют такие же правила преобразований, как и для их базовых (int? можно неявно привести к long? и т. д.).

При этом (от nullable к базовому, не допускающем null) обратное преобразование всегда явное, оно может приводить к **InvalidOperationException**.

При проверке наличия значения вы можете воспользоваться сопоставлением с шаблоном типа при помощи оператора is с базовым типом (см. примеры в левой части слайда).

Преобразуем Значимые Nullable-Типы

Операторы с Нуллификацией

Унарные и бинарные операторы, а так же любые перегруженные операторы, поддерживаемые типом T по умолчанию поддерживаются соответствующим ему типом T?. Пользовательские преобразования базовых типов будут поддерживаться и для их nullable вариантов.

При выполнении любой арифметической/побитовой операции, содержащей null, с nullable-типом результат будет равен null (bool? – исключение, об этом будет на следующем слайде).

Операторы >, >=, <, <= всегда возвращают false, если хотя бы один из операндов равен null.

Оператор == всегда будет возвращать false, кроме случая null == null, противоположным образом будет работать оператор !=.

Операторы при Работе с bool?

Побитовые операторы для bool? работают по следующей логике:

- null & true == true & null == null
- null & false == false & null == false
- null | true == true | null == true
- null | false == false | null == null

Операторы ! и ^ поддерживаются bool? и будут всегда возвращать null, если значение изначально равно null.

Операторы && и || не поддерживают операнды типа bool?.

Упаковка, Распаковка и Определение Nullable

Nullable-типы упаковываются следующим образом:

- Если HasValue возвращает false, будет создана пустая ссылка (null).
- Если HasValue возвращает true, **значение будет упаковано как экземпляр базового типа**.

С этим связана основная проблема, так как при определении типа через **GetType( )** происходит упаковка в Object, из-за чего полученный тип будет неотличим от базового значимого. Аналогичная ситуация происходит с оператором **is**, который вернёт false только в случае, когда nullable-переменная равна null.

Для того, чтобы точно определить nullable-тип потребуется метод:

static Type GetUnderlyingType (Type typeVal), который вернёт соответствующий объект Type, если typeVal является nullable-типом и null в противном случае.

Документация Microsoft рекомендует написать такой метод для проверки:

```
bool IsNullableOfType<T>(T o) {
    var type = typeof(T);
    return Nullable.GetUnderlyingType(type) != null;
}
```

Null-Условные Операторы и Объединение с Null

Операторы **?.** и **?[ ]** возвращают значение null, если член, к которому вы пытаетесь получить доступ является null. Хотя это позволит избежать NullReferenceException, всегда стоит помнить, что данные операторы не смогут обезопасить вас от других возможных исключений (например, IndexOutOfRangeException при обращении за границу массива). Данные операторы выполняются по короткой схеме – если левая часть вернула null, дальнейшие вычисления не производятся.

Правоассоциативный оператор объединения с null **??** возвращает значение своего левого операнда, если оно не равно null, в противном случае вычисляется операнд справа и возвращается его результат (правая часть не вычисляется, если левая не равна null).

Правоассоциативный оператор **??=** (C# 8.0) присваивает левому операнду значение правого только в случае, если левый равен null (правая часть не вычисляется, если левая не равна null). Левый операнд ??= может быть только переменной, свойством или элементом индексатора.

Тип левого операнда операторов ?? и ??= обязан быть типом, допускающим значение null. Начиная с версии C# 8.0, вы можете использовать оператор объединения с null с неограниченными типизирующими параметрами:

```
private static void Display<T>(T a, T backup) => Console.WriteLine(a ?? backup);
```

Ссылочные Nullable-Типы в C# 8.0

Начиная с версии C# 8.0 появляется такое понятие, как **ссылочные nullable-типы**. Как мы знаем, одним из наиболее часто встречающихся и не самых очевидных исключений является NullReferenceException. Поэтому важно уметь анализировать код и определять потенциальные источники данного исключения. Именно с этой целью в C# и были добавлены ссылочные nullable-типы.

Важно понимать, что данная функция предназначена для анализа кода и является опциональной (выключена по умолчанию в проектах). Она не запрещает вам присваивать ссылочным типам значение null, а лишь подсвечивает подобного рода присваивания как предупреждения – **все связанные с ней вещи не меняют принцип работы кода во время исполнения**.

Вы можете включить ссылочные nullable-типы глобально в проекте или указывать необходимые фрагменты кода при помощи препроцессинговых директив **#nullable enable** и **#nullable disable**.

GOODBYE, KAKAPOLANDS!

ДО НОВЫХ ВСТРЕЧ!