**№ 6 Структуры, перечисления, классы контейнеры и контроллеры**

**Курносенко Софья, 7 группа 1 подгруппа 2 курс**

|  |  |
| --- | --- |
| **Вариант 8** | Создать **Программу передач**.  Найти все фильмы, снятые в определенный год*, подсчитать продолжительность программы по времени, число рекламных роликов*. |

**Задания**

**1) К предыдущей лабораторной работе (л.р. 5) добавьте к существующим классам перечисление и структуру.**

**► Перечисления enum**

Кроме примитивных типов данных в C# есть такой тип как enum или перечисление. Перечисления представляют набор логически связанных констант.

**Перечисление** (enumeration) — это определяемый пользователем целочисленный тип. Когда вы объявляете перечисление, то специфицируете набор допустимых значений, которые могут принимать экземпляры перечислений. Но мало того — этим значениям еще должны быть присвоены имена, понятные для пользователей. Если где-то в коде попытаться присвоить экземпляру перечисления значение, не входящее в список допустимых, компилятор выдаст ошибку.

Определение перечисления может в долговременной перспективе сэкономить массу времени и избавить от головной боли. Существует, по крайней мере, три явных **выгоды** от применения перечислений вместо простых целых:

◦ Как упоминалось, перечисления облегчают сопровождение кода, гарантируя, что переменным будут присваиваться только легитимные, ожидаемые значения.

◦ Перечисления делают код яснее, позволяя обращаться к целым значениям, называя их осмысленными именами вместо малопонятных "магических" чисел.

◦ Перечисления облегчают ввод исходного кода. Когда вы собираетесь присвоить значение экземпляру перечислимого типа, то интегрированная среда Visual Studio с помощью средства IntelliSense отображает всплывающий список с допустимыми значениями, что позволяет сэкономить несколько нажатий клавиш и напомнить о возможном выборе значений.

Объявление перечисления происходит с помощью оператора **enum**. Далее идет название перечисления, после которого указывается тип перечисления - он обязательно должен представлять целочисленный тип (byte, int, short, long). Если тип явным образом не указан, то по умолчанию используется тип int. Затем идет список элементов перечисления через запятую:

enum Days

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

enum Time : byte

{

Morning,

Afternoon,

Evening,

Night

}

Каждая символически обозначаемая константа в перечислении имеет целое значение. Тем не менее, неявные преобразования перечислимого типа во встроенные целочисленные типы и обратно в C# не определены, а значит, в подобных случаях требуется явное приведение типов. Кроме того, приведение типов требуется при преобразовании двух перечислимых типов.

В этих примерах каждому элементу перечисления присваивается целочисленное значение, причем первый элемент будет иметь значение 0, второй - 1 и так далее. Мы можем также явным образом указать значения элементов, либо указав значение первого элемента:

enum Operation

{

Add = 1, // каждый следующий элемент по умолчанию увеличивается на единицу

Subtract, // этот элемент равен 2

Multiply, // равен 3

Divide // равен 4

}

Но можно и для всех элементов явным образом указать значения:

enum Operation

{

Add = 2,

Subtract = 4,

Multiply = 8,

Divide = 16

}

При этом константы перечисления могут иметь одинаковые значения, либо даже можно присваивать одной константе значение другой константы:

enum Color

{

White = 1,

Black = 2,

Green = 2,

Blue = White // Blue = 1

}

Каждое перечисление фактически определяет новый тип данных. Затем в программе мы можем определить переменную этого типа и использовать ее:

enum Operation

{

Add = 1,

Subtract,

Multiply,

Divide

}

class Program

{

static void Main(string[] args)

{

Operation op;

op = Operation.Add;

Console.WriteLine(op); // Add

Console.ReadLine();

}

}

В программе мы можем присвоить значение этой переменной. При этом в качестве ее значения должна выступать одна из констант, определенных в данном перечислении. То есть, несмотря на то, что каждая константа сопоставляется с определенным числом, мы не можем присвоить ей числовое значение, например, Operation op = 1;. И также если мы будем выводить на консоль значение этой переменной, то мы получим их константы, а не числовые значения. Если же необходимо получить числовое значение, то следует выполнить приведение к числовому типу:

Operation op;

op = Operation.Multiply;

Console.WriteLine((int)op); // 3

Также стоит отметить, что перечисление необязательно определять внутри класса, можно и вне класса, но в пределах пространства имен:

enum Operation

{

Add = 1,

Subtract,

Multiply,

Divide

}

class Program

{

static void Main(string[] args)

{

Operation op;

op = Operation.Add;

Console.WriteLine(op); // Add

Console.ReadLine();

}

}

Зачастую переменная перечисления выступает в качестве хранилища состояния, в зависимости от которого производятся некоторые действия. Так, рассмотрим применение перечисления на более реальном примере:

class Program

{

enum Operation

{

Add = 1,

Subtract,

Multiply,

Divide

}

static void MathOp(double x, double y, Operation op)

{

double result = 0.0;

switch (op)

{

case Operation.Add:

result = x + y;

break;

case Operation.Subtract:

result = x - y;

break;

case Operation.Multiply:

result = x \* y;

break;

case Operation.Divide:

result = x / y;

break;

}

Console.WriteLine("Результат операции равен {0}", result);

}

static void Main(string[] args)

{

// Тип операции задаем с помощью константы Operation.Add, которая равна 1

MathOp(10, 5, Operation.Add);

// Тип операции задаем с помощью константы Operation.Multiply, которая равна 3

MathOp(11, 5, Operation.Multiply);

Console.ReadLine();

}

}

Здесь у нас имеется перечисление Operation, которое представляет арифметические операции. Также у нас определен метод MathOp, который в качестве параметров принимает два числа и тип операции. В основном методе Main мы два раза вызываем процедуру MathOp, передав в нее два числа и тип операции.

## ► Структуры

## Как вам должно быть уже известно, классы относятся к ссылочным типам данных. Это означает, что объекты конкретного класса доступны по ссылке, в отличие от значений простых типов, доступных непосредственно. Но иногда прямой доступ к объектам как к значениям простых типов оказывается полезно иметь, например, ради повышения эффективности программы. Ведь каждый доступ к объектам (даже самым мелким) по ссылке связан с дополнительными издержками на расход вычислительных ресурсов и оперативной памяти.

## Для разрешения подобных затруднений в C# предусмотрена структура, которая подобна классу, но *относится к типу значения*, а не к ссылочному типу данных. Т.е. структуры отличаются от классов тем, как они сохраняются в памяти и как к ним осуществляется доступ (классы — это ссылочные типы, размещаемые в куче, структуры — типы значений, размещаемые в стеке), а также некоторыми свойствами (например, структуры не поддерживают наследование). Из соображений производительности вы будете использовать структуры для небольших типов данных. Однако в отношении синтаксиса структуры очень похожи на классы.

## Наряду с классами структуры представляют еще один способ создания собственных типов данных в C#. Более того многие примитивные типы, например, int, double и т.д., по сути являются структурами.

Например, определим структуру, которая представляет человека:

struct User

{

public string name;

public int age;

public void DisplayInfo()

{

Console.WriteLine($"Name: {name} Age: {age}");

}

}

Как и классы, структуры могут хранить состояние в виде переменных и определять поведение в виде методов. Так, в данном случае определены две переменные - name и age для хранения соответственно имени и возраста человека и метод DisplayInfo для вывода информации о человеке.

Используем эту структуру в программе:

using System;

namespace HelloApp

{

struct User

{

public string name;

public int age;

public void DisplayInfo()

{

Console.WriteLine($"Name: {name} Age: {age}");

}

}

class Program

{

static void Main(string[] args)

{

User tom;

tom.name = "Tom";

tom.age = 34;

tom.DisplayInfo();

Console.ReadKey();

}

}

}

В данном случае создается объект tom. У него устанавливаются значения глобальных переменных, и затем выводится информация о нем.

### Конструкторы структуры

Как и класс, структура может определять конструкторы. Но в отличие от класса нам не обязательно вызывать конструктор для создания объекта структуры:

User tom;

Однако если мы таким образом создаем объект структуры, то обязательно надо проинициализировать все поля (глобальные переменные) структуры перед получением их значений или перед вызовом методов структуры. То есть, например, в следующем случае мы получим ошибку, так как обращение к полям и методам происходит до присвоения им начальных значений:

User tom;

int x = tom.age; // Ошибка

tom.DisplayInfo(); // Ошибка

Также мы можем использовать для создания структуры конструктор без параметров, который есть в структуре по умолчанию и при вызове которого полям структуры будет присвоено значение по умолчанию (например, для числовых типов это число 0):

User tom = new User();

tom.DisplayInfo(); // Name: Age: 0

Обратите внимание, что при использовании конструктора по умолчанию нам не надо явным образом инициализировать поля структуры.

Также мы можем определить свои конструкторы. Например, изменим структуру User:

using System;

namespace HelloApp

{

struct User

{

public string name;

public int age;

public User(string name, int age)

{

this.name = name;

this.age = age;

}

public void DisplayInfo()

{

Console.WriteLine($"Name: {name} Age: {age}");

}

}

class Program

{

static void Main(string[] args)

{

User tom = new User("Tom", 34);

tom.DisplayInfo();

User bob = new User();

bob.DisplayInfo();

Console.ReadKey();

}

}

}

Важно учитывать, что если мы определяем конструктор в структуре, то он должен инициализировать все поля структуры, как в данном случае устанавливаются значения для переменных name и age.

Так же, как и для класса, можно использовать инициализатор для создания структуры:

User person = new User { name = "Sam", age = 31 };

Но в отличие от класса нельзя инициализировать поля структуры напрямую при их объявлении, например, следующим образом:

struct User

{

public string name = "Sam"; // ! Ошибка

public int age = 23; // ! Ошибка

public void DisplayInfo()

{

Console.WriteLine($"Name: {name} Age: {age}");

}

}

Когда одна структура присваивается другой, создается копия ее объекта. В этом заключается одно из главных отличий структуры от класса. Когда ссылка на один класс присваивается ссылке на другой класс, в итоге ссылка в левой части оператора присваивания указывает на тот же самый объект, что и ссылка в правой его части. А когда переменная одной структуры присваивается переменной другой структуры, создается копия объекта структуры из правой части оператора присваивания.

**Назначение структур**

В связи с изложенным выше возникает резонный вопрос: зачем в C# включена структура, если она обладает более скромными возможностями, чем класс? Ответ на этот вопрос заключается в повышении эффективности и производительности программ. Структуры относятся к типам значений, и поэтому ими можно оперировать непосредственно, а не по ссылке. Следовательно, для работы со структурой вообще не требуется переменная ссылочного типа, а это означает в ряде случаев существенную экономию оперативной памяти.

Более того, работа со структурой не приводит к ухудшению производительности, столь характерному для обращения к объекту класса. Ведь доступ к структуре осуществляется непосредственно, а к объектам — по ссылке, поскольку классы относятся к данным ссылочного типа. Косвенный характер доступа к объектам подразумевает дополнительные издержки вычислительных ресурсов на каждый такой доступ, тогда как обращение к структурам не влечет за собой подобные издержки. И вообще, если нужно просто сохранить группу связанных вместе данных, не требующих наследования и обращения по ссылке, то с точки зрения производительности для них лучше выбрать структуру.

**!!!Обобщение написанного выше и некоторые дополнения**

При проектировании типа структуры вы будете располагать теми же возможностями, что и для типа [класса](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/class), за исключением следующих аспектов:

* Вы не можете объявить конструктор без параметров. Каждый тип структуры уже предоставляет неявный конструктор без параметров, который создает [значение по умолчанию](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/default-values) типа.

**Примечание**

Начиная с C# версии 10 вы можете объявить конструктор без параметров в типе структуры. Дополнительные сведения см. в разделе [**Конструкторы без параметров и инициализаторы полей**](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/struct#parameterless-constructors-and-field-initializers).

* Вы не можете инициализировать поле экземпляра или свойство при его объявлении. Однако можно инициализировать [статическое](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/static) поле или поле [константы](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/const) или статическое свойство при его объявлении.

**Примечание**

Начиная с C# версии 10 вы можете инициализировать поле экземпляра или свойство при его объявлении. Дополнительные сведения см. в разделе [**Конструкторы без параметров и инициализаторы полей**](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/builtin-types/struct#parameterless-constructors-and-field-initializers).

* Конструктор типа структуры должен инициализировать все поля экземпляров данного типа.
* Тип структуры не может наследовать от другого типа класса или структуры и не может быть базовым для класса. Однако тип структуры может реализовывать [интерфейсы](https://docs.microsoft.com/ru-ru/dotnet/csharp/language-reference/keywords/interface).
* Вы не можете объявить [метод завершения](https://docs.microsoft.com/ru-ru/dotnet/csharp/programming-guide/classes-and-structs/finalizers) (деструктор) в типе структуры.

**2) Один из классов сделайте partial и разместите его в разных файлах.**

**3) Определить класс-Контейнер (указан в вариантах жирным шрифтом) для хранения разных типов объектов (в пределах иерархии) в виде списка или массива (использовать абстрактный тип данных). Класс-контейнер должен содержать методы get и set для управления списком/массивом, методы для добавления и удаления объектов в список/массив, метод для вывода списка на консоль.**

**4) Определить управляющий класс-Контроллер, который управляет объектом- Контейнером и реализовать в нем запросы по варианту. При необходимости используйте стандартные интерфейсы (IComparable, ICloneable,….)**

**Вопросы**

**1. Чем класс отличается от структуры?**

◦ Класс – является **ссылочным типом** (reference type), а структуры – **значимым** типом (value type).

◦ Структуры, указываемые в списке параметров метода, передаются **по значению** (то есть копируются), объекты классов — **по ссылке**. Структуру тоже можно передать по ссылке, используя модификаторы out и ref.

◦ Вы не можете использовать **конструктор без параметров** для структуры, потому что компилятор всегда генерирует его сам. При объявлении класса нет проблем создать собственный конструктор без параметров.

◦ Нельзя инициализировать переменные структуры в том месте, где они объявляются. Задать значения переменных структуры по умолчанию можно в *конструкторе с* *параметрами*.

◦ Вследствие того, что структуры являются значимым типом, то можно создать структуру без использования конструктора, например:

using System;

namespace ConsoleApplication1

{

struct Time

{

public int hours, minutes, seconds;

}

class Program

{

static void Main()

{

**Time t;**

t.hours = 7;

Console.WriteLine(t.hours.ToString());

Console.ReadKey();

}

}

}

В таком случае, переменная t создается, но поля не будут инициализированы конструктором.

◦ Структуры не могут быть абстрактными, структуры не имеют деструкторов, структуры не поддерживают наследование

|  |  |  |
| --- | --- | --- |
| Вопрос | Структура | Класс |
| Какого же типа экземпляр объекта? | Значимый (value) тип | Ссылочный (reference) тип |
| Где “живут” экземпляры этих объектов? | Экземпляры структуры называют значениями и “живут” они в стеке (stack). | Экземпляры классов называют объектами и “живут” они в куче (heap). |
| Можно ли создать конструктор по умолчанию? | Нет | Да |
| Если создается свой конструктор, будет ли компилятор генерировать конструктор по умолчанию? | Да | Нет |
| Если в своём конструкторе не будут инициализированы некоторые поля, будут ли они автоматически инициализированы компилятором? | Нет | Да |
| Разрешается ли инициализировать переменные там, где их объявляют? | Нет | Да |

**2. Что может и чего не может быть в структуре?**

**Могут быть** поля, методы, свойства, конструкторы с параметрами.

**Не может быть** конструктора без параметров, деструкторов.

**3. Что такое перечисление? Приведите пример определения и использования перечисления**

(где-то в заданиях--)

**4. Перечислите и поясните стандартные интерфейсы .Net?**

|  |  |  |  |
| --- | --- | --- | --- |
| Интерфейс | Методы | Назначение | Примечание |
| |  | | --- | | ICloneable | | |  | | --- | | objectClone() | | |  | | --- | | Клонирование объектов (поверхностное или глубокое) | |  |
| |  | | --- | | lEnumerable | | |  | | --- | | (IEnumerator) GetEnumerator(); | | |  | | --- | | перебор элементов необобщенной коллекции | | |  | | --- | | Основа для большинства | |
| |  | | --- | | IEnumerator | | |  | | --- | | Current  boolMoveNext()  voidReset() | | |  | | --- | | перебор по необобщенной коллекции | | |  | | --- | | можем перебирать объекты в цикле foreach | |
| |  | | --- | | IComparable | | |  | | --- | | intCompareTo( object obj) | | |  | | --- | | Сравнение объектов  Для выяснения порядка | |  |
| |  | | --- | | IComparer | | |  | | --- | | intCompare(object o1,  object o2); | | |  | | --- | | Сравнение объектов | | |  | | --- | | будут иметь больший приоритет | |
| |  | | --- | | IDisposable | | |  | | --- | | voidDispose() | | |  | | --- | | механизм для освобождения управляемых ресурсов | |  |
| IEquatable<T> | bool Equals(T other); | Сравнение объектов на равенство |  |
| IStructuralEquatable | bool Equals(object, IEqualityComparer); intGetHashCode(…); | Проверка на равенство по значению |  |

**5. Как используется интерфейс IComparable?**

## Сортировка объектов. Интерфейс IComparable

Большинство встроенных в .NET классов коллекций и массивы поддерживают сортировку. С помощью одного метода, который, как правило, называется Sort() можно сразу отсортировать по возрастанию весь набор данных. Например:

int[] numbers = new int[] { 97, 45, 32, 65, 83, 23, 15 };

Array.Sort(numbers);

foreach (int n in numbers)

Console.WriteLine(n);

Однако метод Sort по умолчанию работает только для наборов примитивных типов, как int или string. Для сортировки наборов сложных объектов применяется интерфейс IComparable. Он имеет всего один метод:

public interface IComparable

{

int CompareTo(object o);

}

Метод CompareTo предназначен для сравнения текущего объекта с объектом, который передается в качестве параметра object o. На выходе он возвращает целое число, которое может иметь одно из трех значений:

* Меньше нуля. Значит, текущий объект должен находиться перед объектом, который передается в качестве параметра
* Равен нулю. Значит, оба объекта равны
* Больше нуля. Значит, текущий объект должен находиться после объекта, передаваемого в качестве параметра

Например, имеется класс Person:

class Person : IComparable

{

public string Name { get; set; }

public int Age { get; set; }

public int CompareTo(object o)

{

Person p = o as Person;

if (p != null)

return this.Name.CompareTo(p.Name);

else

throw new Exception("Невозможно сравнить два объекта");

}

}

Здесь в качестве критерия сравнения выбрано свойство Name объекта Person. Поэтому при сравнении здесь фактически идет сравнение значения свойства Name текущего объекта и свойства Name объекта, переданного через параметр. Если вдруг объект не удастся привести к типу Person, то выбрасывается исключение.

Применение:

Person p1 = new Person { Name = "Bill", Age = 34 };

Person p2 = new Person { Name = "Tom", Age = 23 };

Person p3 = new Person { Name = "Alice", Age = 21 };

Person[] people = new Person[] { p1, p2, p3 };

Array.Sort(people);

foreach (Person p in people)

{

Console.WriteLine($"{p.Name} - {p.Age}");

}

Интерфейс IComparable имеет обобщенную версию, поэтому мы могли бы сократить и упростить его применение в классе Person:

class Person : IComparable<Person>

{

public string Name { get; set; }

public int Age { get; set; }

public int CompareTo(Person p)

{

return this.Name.CompareTo(p.Name);

}

}

**Я такое написала:**

using System;

namespace asitis

{

class Num : IComparable

{

public int n;

public Num(int n)

{

this.n = n;

}

public int CompareTo(object obj)

{

Num number = obj as Num;

if (number != null)

{

return this.n.CompareTo(number.n);

}

else

{

throw new Exception("Невозможно сравнить два объекта");

}

}

}

class Program

{

static void Main(string[] args)

{

Num a = new Num(34);

Num b = new Num(3);

Num c = new Num(-7);

Num d = new Num(50);

Num e = new Num(3);

Num[] numbers = new Num[] { a, b, c, d, e };

Array.Sort(numbers);

foreach(Num number in numbers)

{

Console.WriteLine(number.n);

}

}

}

}

**6. Как используется интерфейс ICloneable?**

## Копирование объектов. Интерфейс ICloneable

## Поскольку классы представляют ссылочные типы, то это накладывает некоторые ограничения на их использование. В частности:

class Program

{

static void Main(string[] args)

{

Person p1 = new Person { Name = "Tom", Age = 23 };

Person p2 = p1;

p2.Name = "Alice";

Console.WriteLine(p1.Name); // Alice

Console.Read();

}

}

class Person

{

public string Name { get; set; }

public int Age { get; set; }

}

В данном случае объекты p1 и p2 будут указывать на один и тот же объект в памяти, поэтому изменения свойств в переменной p2 затронут также и переменную p1.

Чтобы переменная p2 указывала на новый объект, но со значениями из p1, мы можем применить клонирование с помощью реализации интерфейса **ICloneable**:

public interface ICloneable

{

object Clone();

}

Реализация интерфейса в классе Person могла бы выглядеть следующим образом:

class Person : ICloneable

{

public string Name { get; set; }

public int Age { get; set; }

public object Clone()

{

return new Person { Name = this.Name, Age = this.Age };

}

}

Использование:

class Program

{

static void Main(string[] args)

{

Person p1 = new Person { Name = "Tom", Age = 23 };

Person p2 = (Person)p1.Clone();

p2.Name = "Alice";

Console.WriteLine(p1.Name); // Tom

Console.Read();

}

}

Теперь все нормально копируется, изменения в свойствах p2 не сказываются на свойствах в p1.

Для сокращения кода копирования мы можем использовать специальный метод **MemberwiseClone()**, который возвращает копию объекта:

class Person : ICloneable

{

public string Name { get; set; }

public int Age { get; set; }

public object Clone()

{

return this.MemberwiseClone();

}

}

Этот метод реализует поверхностное (неглубокое) копирование. Однако данного копирования может быть недостаточно. Например, пусть класс Person содержит ссылку на объект Company:

class Person : ICloneable

{

public string Name { get; set; }

public int Age { get; set; }

public Company Work { get; set; }

public object Clone()

{

return this.MemberwiseClone();

}

}

class Company

{

public string Name { get; set; }

}

В этом случае при копировании новая копия будет указывать на тот же объект Company:

Person p1 = new Person { Name = "Tom", Age = 23, Work = new Company { Name = "Microsoft" } };

Person p2 = (Person)p1.Clone();

p2.Work.Name = "Google";

p2.Name = "Alice";

Console.WriteLine(p1.Name); // Tom

Console.WriteLine(p1.Work.Name); // Google - а должно быть Microsoft

Поверхностное копирование работает только для свойств, представляющих примитивные типы, но не для сложных объектов. И в этом случае надо применять **глубокое копирование**:

{

public string Name { get; set; }

public int Age { get; set; }

public Company Work { get; set; }

public object Clone()

{

Company company = new Company { Name = this.Work.Name };

return new Person

{

Name = this.Name,

Age = this.Age,

Work = company

};

}

}

class Company

{

public string Name { get; set; }

}

**7. Что такое полиморфизм? Перечислите его формы. Приведите примеры.**

**Полиморфизм**: «поли» значит «много», а «морфизм» — «изменение» или «вариативность», таким образом, «полиморфизм» — это свойство одних и тех же объектов и методов принимать разные формы.

Перегрузка методов обеспечивает статический полиморфизм, а виртуальный метод – динамический.

**8. Зачем в классе определяют виртуальные методы?**

То, что метод является виртуальным, позволяет переопределить его в классе наследнике.

**9. Как сделать запрет переопределения методов?**

Определить его без модификатора virtual; а если метод с модификатором override (то есть является переопределенным методом в классе-наследнике, а мы хотим использовать этот класс как базовый для какого-то другого, и этому другому запретить переопределение метода), то нужно дописать модификатор sealed.