|  |
| --- |
| Министерство образования Республики Беларусь  УО «Полоцкий государственный университет им. Ефросинии Полоцкой» |

|  |  |
| --- | --- |
|  | Факультет информационных технологий  Кафедра технологий программирования |

|  |
| --- |
| ОТЧЕТ |
| **к лабораторной работе №2**  **Дисциплина**: «Программирование на платформе .NET» |

|  |  |
| --- | --- |
| Выполнил: | Студент группы 19-ИТ-2  Гилимович А.Н. |

|  |  |
| --- | --- |
| Проверила: | Преподователь  Виноградова А.Д. |

|  |
| --- |
| Полоцк, 2022 г. |

**ТИПЫ ДАННЫХ, ОПРЕДЕЛЯЕМЫЕ ПОЛЬЗОВАТЕЛЕМ.**

**НАСЛЕДОВАНИЕ. ОБРАБОТКА ИСКЛЮЧЕНИЙ В С#.**

**Вариант 1**

**Цель работы:** познакомиться с пользовательскими типами данных в языке С#: структура и перечисление; ознакомиться со структурой класса, его созданием и использованием, описанием членов класса: полей, свойств, инициализации объектов класса с помощью конструкторов; изучить механизм создания иерархий классов в С# и применение интерфейсов при наследовании; изучить механизм генерации и обработки исключений.

**Задание 1**

Проработать примеры программ 1–8, данные в теоретических сведениях. Создать на их основе программы. Получить результаты работы программ и уметь их объяснить. Внести их в отчет по работе с комментариями.

**Задание 2**

Для заданной структуры данных разработать абстрактный класс и класс-наследник. В классе реализовать несколько конструкторов. Создать методы, работающие с полями класса. Часть из них должны быть виртуальными. Добавить методы-свойства и индексаторы.

**Задание 3**

Разработать интерфейсные классы, добавляющие некоторые методы в класспотомок. Изучить причины возникновения коллизий имен при наследовании и способы их устранения.

**Задание 4**

Разработать классы исключительных ситуаций и применить их для обработки, возникающих исключений.

**Задание 5**

Написать демонстрационную программу.

**Описание данных пользовательских типов**

СЛУЖАЩИЙ: имя, возраст, рабочий стаж, должности.

**Ход работы**

Изучив теоритический материал, преступил к выполнению заданий.

В ходе методичеких указаний встречались примеры, которые нужно проработать.

Суть первого примера состоялась показать использование перечислений и структур в программах на C#.

**Листинг 1** – Использование перечислений и структур в программах на C#

enum EmpType : byte {

Manager = 10, Grunt = 1,

Contractor = 100, VP = 9

}

struct Employee {

public EmpType title; // Поле – перечисление

public string name;

public short deptID;

}

class StructTester {

public static int Main(string[] args) {

Employee fred; //Создание структурной переменной

fred.deptID = 40;

fred.name = "Fred";

fred.title = EmpType.Grunt;

return 0;

}}

Здесь показано как создавать и испольщовать перечисления и структуры.

Дальше рассмотрим класс Person, у которого пять полей: fam, status, salary, age, health, характеризующих соответственно фамилию, статус, зарплату, возраст и здоровье персоны. Для каждого из этих полей может быть своя стратегия доступа.

**Листинг 2** – Свойства класса на C#

public class Person {

string fam="", status="", health="";

int age=0, salary=0;

public string Fam { //стратегия: Read,Write-once

set {if (fam == "") fam = value;}

get {return(fam);}

}

public string Status { //стратегия: Read-only

get {return(status);}

}

public int Age { //стратегия: Read,Write

set {

age = value;

if(age < 7) status ="ребенок";

else if(age <17) status ="школьник";

else if (age < 22) status = "студент";

else status = "служащий";

}

get {return(age);}

}

public int Salary { //стратегия: Write-only

set {salary = value;}

}

}

public void TestPersonProps(){

Person pers1 = new Person();

pers1.Fam = "Петров"; pers1.Age = 21; pers1.Salary = 1000;

Console.WriteLine ("Фам={0}, возраст={1}, статус={2}", pers1.Fam,

pers1.Age, pers1.Status);

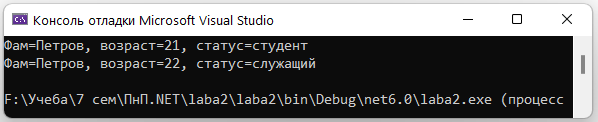
pers1.Fam = "Иванов"; pers1.Age += 1;

Console.WriteLine ("Фам={0}, возраст={1}, статус={2}", pers1.Fam,

pers1.Age, pers1.Status);

}

Вывод этой программы в консоль, следующий:



**Рисунок 1** – Свойства класса на C#

Индексатор обеспечивает доступ к закрытому полю, представляющему массив. Объекты класса индексируются по этому полю. У класса может быть только один индексатор со стандартным именем this.

Добавим в класс Person массив children, задающий детей персоны, сделаем это свойство закрытым, а доступ к нему обеспечит индексатор:

**Листинг 3** – Индексаторы в классе на C#

const int Child\_Max = 10; //максимальное число детей

Person[] children = new Person[Child\_Max];

int count\_children=0; //текущее число детей объекта

public Person this[int i] { //индексатор

get {

if (i>=0 && i< count\_children) return(children[i]);

else return(children[0]);

}

set {

if (i==count\_children && i< Child\_Max) {

children[i] = value; count\_children++;

}

}}

public void TestPersonChildren(){

Person pers1 = new Person(), pers2 = new Person();

pers1.Fam = "Петров"; pers1.Age = 42;

pers1.Salary = 10000; pers1[pers1.Count\_children] = pers2;

pers2.Fam ="Петров"; pers2.Age = 21; pers2.Salary = 1000;

Console.WriteLine ("Фам={0}, возраст={1}, статус={2}", pers1.Fam,

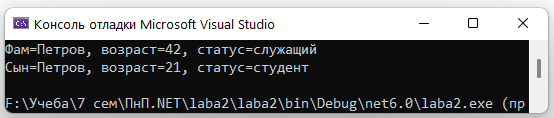
pers1.Age, pers1.Status);

Console.WriteLine ("Сын={0}, возраст={1}, статус={2}",

pers1[0].Fam, pers1[0].Age, pers1[0].Status);

}

После внесения изменений вывод в консоль следующий:



**Рисунок 2** - Индексаторы в классе на C#

Наследование. Повторное использование кода – это одна из главных целей ООП. Класс-потомок наследует все возможности родительского класса – все поля и все методы, открытую и закрытую часть класса. Единственное, что не наследует потомок – это конструкторы родительского класса. Конструкторы потомок должен создавать сам. В C# разрешено только одиночное наследование, т. е. у классапотомка может быть только один родительский класс.

Рассмотрим класс Found, играющий роль родительского клаcса:

**Листинг 4** - Родительский класс Found

public class Found{

protected string name;

protected int credit;

public Found() { }

public Found(string name, int sum) {

this.name = name; credit = sum;

}

public virtual void VirtMethod() { //виртуальный метод

Console.WriteLine ("Отец: " + this.ToString() );

}

//переопределенный метод базового класса Object

public override string ToString() {

return(String.Format("поля: name = {0}, credit = {1}", name, credit));

}

public void NonVirtMethod() {

Console.WriteLine ("Мать: " + this.ToString() );

}

public void Analysis() {

Console.WriteLine ("Простой анализ");

}

public void Work() {

VirtMethod();

NonVirtMethod();

Analysis();

}

}

Класс Found переопределил родительский метод класса Object ToString(), задав собственную реализацию возвращаемой методом строки. На переопределение родительского метода указывает модификатор override.

Создадим класс Derived – потомка класса Found. Потомок может добавить новые свойства – поля класса. Заметьте, потомок не может ни отменить, ни изменить модификаторы или типы полей, наследованных от родителя.

Класс Derived добавляет новое поле protected int debet.

Класс должен позаботиться о создании собственных конструкторов, поскольку, как правило, добавляет собственные поля, о которых родитель ничего не может знать. Если не задать конструкторы класса, то будет добавлен конструктор по умолчанию, инициализирующий все поля значениями по умолчанию. Но это редкая ситуация. Чаще всего класс создает собственные конструкторы и, как правило, не один, задавая разные варианты инициализации полей.

**Листинг 5** - Класс наследник с переопределенными методами родителя

public class Derived:Found {

protected int debet;

public Derived() {}

public Derived(String name, int cred, int deb): Found (name,cred){

debet = deb;

}

public void DerivedMethod(){ //новый метод потомка

Console.WriteLine("Это метод класса Derived");

}

new public void Analysis(){ //сокрытие метода родителя

base.Analysis();

Console.WriteLine("Сложный анализ");

}

public void Analysis(int level){ // перегрузка метода

base.Analysis();

Console.WriteLine("Анализ глубины {0}", level);

}

public override String ToString(){ //переопределение метода

return(String.Format("поля: name = {0}, credit = {1},debet

={2}",name, credit, debet));

}

// переопределение метода родителя

public override void VirtMethod() {

Console.WriteLine ("Сын: " + this.ToString() );

}

}

При множественном наследовании интерфейсов может фозникнуть проблемма коллизии имен.

Она возникает, когда два или более интерфейса имеют методы с одинаковыми именами и сигнатурой. Здесь возможны две стратегии – склеивание методов и переименование.

Стратегия склеивания применяется тогда, когда класс – наследник интерфейсов полагает, что разные интерфейсы задают один и тот же метод, единая реализация которого и должна быть обеспечена наследником. В этом случае наследник строит единственную общедоступную реализацию.

Другая стратегия исходит из того, что методы разных интерфейсов должны быть реализованы по-разному. В этом случае необходимо переименовать конфликтующие методы. Для этого достаточно реализовать методы разных интерфейсов как закрытые, а затем открыть их с переименованием.

Пример двух интерфейсов, имеющих методы с одинаковой сигнатурой, и класса – наследника этих интерфейсов, применяющего разные стратегии реализации для конфликтующих методов:

**Листинг 6** - Коллизии имен в интерфейсах

public interface IProps{

void Prop1(string s);

void Prop2 (string name, int val);

void Prop3();

}

public interface IPropsOne{

void Prop1(string s);

void Prop2 (int val);

void Prop3();

}

public class ClainTwo:IProps,IPropsOne {

// склеивание методов двух интерфейсов

public void Prop1 (string s) { Console.WriteLine(s); }

// перегрузка методов двух интерфейсов

public void Prop2(string s, int x) { Console.WriteLine(s + x); }

public void Prop2 (int x) { Console.WriteLine(x); }

// private реализация и переименование методов 2-х интерфейсов

void IProps.Prop3() {

Console.WriteLine("Метод 3 интерфейса 1");

}

void IPropsOne.Prop3() {

Console.WriteLine("Метод 3 интерфейса 2");

}

public void Prop3FromInterface1() { ((IProps)this).Prop3(); }

public void Prop3FromInterface2() { ((IPropsOne)this).Prop3(); }

}

public void TestTwoInterfaces(){

ClainTwo claintwo = new ClainTwo();

claintwo.Prop1("Склейка свойства двух интерфейсов");

claintwo.Prop2("перегрузка .: ",99);

claintwo.Prop2(9999);

claintwo.Prop3FromInterface1();

claintwo.Prop3FromInterface2();

Console.WriteLine("Интерфейсный объект вызывает методы 1-го ин-

терфейса!");

IProps ip1 = (IProps)claintwo;

ip1.Prop1("интерфейс IProps: свойство 1");

ip1.Prop3();

Console.WriteLine("Интерфейсный объект вызывает методы 2-го ин-

терфейса!");

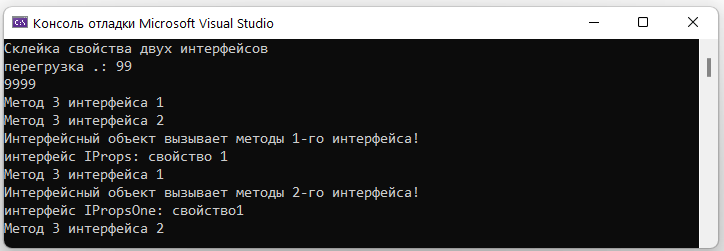
IPropsOne ip2 = (IPropsOne)claintwo;

ip2.Prop1("интерфейс IPropsOne: свойство1");

ip2.Prop3();

}

Вывод в консоль этой программы:



**Рисунок 3** - Коллизии имен в интерфейсах

Приступил к реализации заданий по индивидуальному заданию.

Создал абстраутный класс Worker который сохраняет свойства при создании объекта класса, свойства приватный поэтому для доступа и перезаписи существующийх свойств равлизовал индексаторы.

**Листинг 7** – class Worker

abstract class Worker

{

private string name = "";

private int age = 0, workExperience = 0;

public Worker(string name, int age, int workExperience)

{

this.name = name;

this.age = age;

this.workExperience = workExperience;

}

//методы-свойства и индексаторы

public string Name

{

get { return name; }

set

{

name = value;

}

}

public int Age

{

get { return age; }

set { age = value; }

}

public int WorkExperience

{

get { return workExperience; }

set { workExperience = value; }

}

//виртуальный метод

public virtual void work()

{

Console.WriteLine("{0} работает", this.name);

}

}

Класс абстрактный и используется как общий класс с общими методами для более конкретных классов.

Далее реализовал класс наследник от класса Worker, скласс Programmer. В классе добавляются новые свойства к уже существующим. JobTitle – должность программист без возможности изменения, а так же уровень знаний программиста. Для этих свойст также были созданы индексаторы. Переопределяет метод work() с большим уочнением о работующем.

Был создан интерфейс для реализации методов джуна. Читать документацию и учить английский.

**Листинг 8** – интерфейс IJunior

public interface IJunior

{

void readDocumentation(string doc);

void learnEnglish();

}

Класс Programer реализует эти методы.

**Листинг 9** – класс Programer

class Programer : Worker, IJunior

{

private string jobTitle = "", knowledgeLevel = "";

public Programer(string name, int age, int workExperience, string knowledgeLevel)

: base(name, age, workExperience)

{

this.jobTitle = "programer";

this.knowledgeLevel = knowledgeLevel;

}

//методы-свойства и индексаторы

public string JobTitle

{

get { return jobTitle; }

}

public string KnowledgeLevel

{

get { return knowledgeLevel; }

set { knowledgeLevel = value; }

}

//переопределение виртуального метода

public override void work()

{

Console.WriteLine("{0} {1} работает и он {2}", this.JobTitle, this.Name, this.KnowledgeLevel);

}

void IJunior.readDocumentation(string doc)

{

Console.WriteLine("{0} пошел читать документацию по {1}", this.Name, doc);

}

void IJunior.learnEnglish()

{

Console.WriteLine("{0} учит английский", this.Name);

}

}

Для тестирования работы программы создал объект Programmer и запустил методы объекта jun.

Листинг 10 – Тестирование программы

Programer jun = new Programer("Artyom", 20, 1, "Junior");

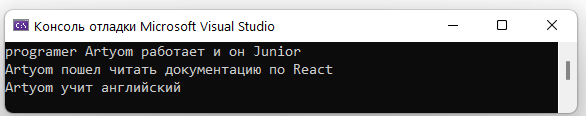
jun.work();

IJunior ij = (IJunior)jun;

ij.readDocumentation("React");

ij.learnEnglish();

Результат тестирования показан на рисунке 4.



**Рисунок 4** – тестирование программы

**Вывод:** познакомились с пользовательскими типами данных в языке С#: структура и перечисление; ознакомились со структурой класса, его созданием и использованием, описанием членов класса: полей, свойств, инициализации объектов класса с помощью конструкторов; изучили механизм создания иерархий классов в С# и применение интерфейсов при наследовании; изучили механизм генерации и обработки исключений