**Московский государственный технический**

**университет им. Н.Э. Баумана**

Факультет «Информатика и системы управления»

Кафедра ИУ5 «Системы обработки информации и управления»

Курс «Базовые компоненты интернет-технологий»

Отчет по лабораторной работе №3

«Работа с коллекциями»

|  |  |  |
| --- | --- | --- |
| Выполнил: |  | Проверил: |
| студент группы ИУ5-33 |  |  |
| Миронова Александра |  |  |
| Подпись и дата:  30.10.20 |  | Подпись и дата: |

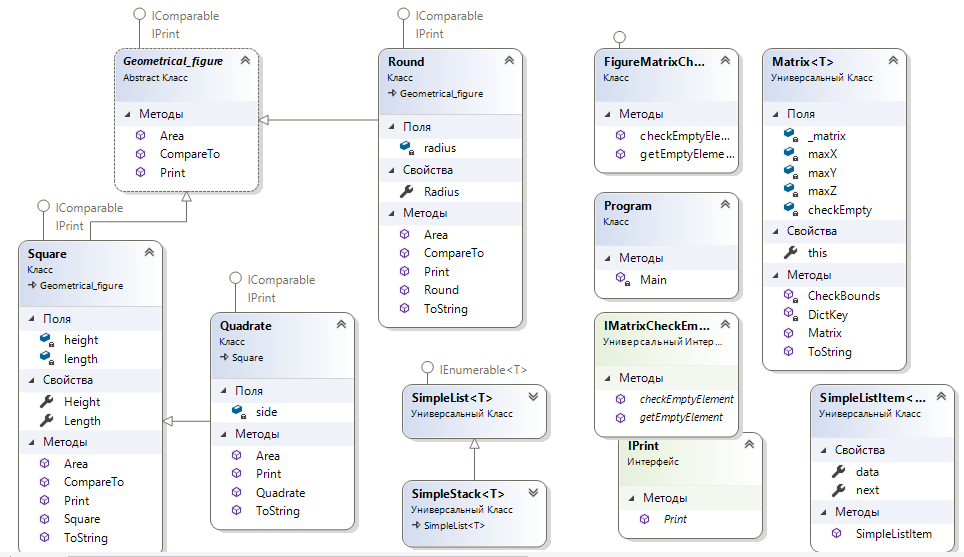
Москва, 2020 г.

**Задание:**

Разработать программу, реализующую работу с классами.

1. Программа должна быть разработана в виде консольного приложения на языке C#.
2. Абстрактный класс «Геометрическая фигура» содержит виртуальный метод для вычисления площади фигуры.
3. Класс «Прямоугольник» наследуется от «Геометрическая фигура». Ширина и высота объявляются как свойства (property). Класс должен содержать конструктор по параметрам «ширина» и «высота».
4. Класс «Квадрат» наследуется от «Прямоугольник». Класс должен содержать конструктор по длине стороны.
5. Класс «Круг» наследуется от «Геометрическая фигура». Радиус объявляется как свойство (property). Класс должен содержать конструктор по параметру «радиус».
6. Для классов «Прямоугольник», «Квадрат», «Круг» переопределить виртуальный метод Object.ToString(), который возвращает в виде строки основные параметры фигуры и ее площадь.
7. Разработать интерфейс IPrint. Интерфейс содержит метод Print(), который не принимает параметров и возвращает void. Для классов «Прямоугольник», «Квадрат», «Круг» реализовать наследование от интерфейса IPrint. Переопределяемый метод Print() выводит на консоль информацию, возвращаемую переопределенным методом ToString().

**Диаграмма классов:**

****

**Текст программы:**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Collections

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Program is done by Mironova Alexandra Romanovna group IU5-33b");

Console.WriteLine("\n");

Console.WriteLine("Square");

Square MySquare = new Square(5, 4);

MySquare.Print();

Console.WriteLine("\n");

Console.WriteLine("Quadrate");

Quadrate MyQuadrate = new Quadrate(3);

MyQuadrate.Print();

Console.WriteLine("\n");

Console.WriteLine("Round");

Round MyRound = new Round(4);

MyRound.Print();

//-------------------------

Console.WriteLine("Compare Round area size & Square area size");

if (MyRound.CompareTo(MySquare) > 0)

Console.WriteLine("Round area size > Square area size");

else if (MyRound.CompareTo(MySquare) == 0)

Console.WriteLine("Round area size = Square area size");

else Console.WriteLine("Round area size < Square area size");

Console.WriteLine("\n");

//-------------------------

ArrayList FigureList1 = new ArrayList();

FigureList1.Add(MySquare);

FigureList1.Add(MyQuadrate);

FigureList1.Add(MyRound);

Console.WriteLine("Figures before sorting");

foreach (var i in FigureList1)

((Geometrical\_figure)i).Print();

FigureList1.Sort();

Console.WriteLine("Figures after sorting");

foreach (var i in FigureList1)

((Geometrical\_figure)i).Print();

Console.WriteLine("\n");

//-------------------------

List<Geometrical\_figure> FigureList2 = new List<Geometrical\_figure>();

FigureList2.Add(MySquare);

FigureList2.Add(MyQuadrate);

FigureList2.Add(MyRound);

FigureList2.Sort();

foreach (var i in FigureList2)

((Geometrical\_figure)i).Print();

//-------------------------

Console.WriteLine("\nMatrix");

Matrix<Geometrical\_figure> matrix = new Matrix<Geometrical\_figure>(3, 3, 1, new FigureMatrixCheckEmpty());

matrix[0, 0, 0] = MySquare;

matrix[1, 1, 0] = MyQuadrate;

matrix[2, 2, 0] = MyRound;

Console.WriteLine(matrix.ToString());

//-------------------------

SimpleStack<Geometrical\_figure> stack = new SimpleStack<Geometrical\_figure>();

Console.WriteLine("New stack");

while (stack.Count > 0)

{

Geometrical\_figure f = stack.Pop();

Console.WriteLine(f);

}

Console.WriteLine("\nAdding figures");

Console.WriteLine("1 Square");

stack.Push(MySquare);

Console.WriteLine("2 Quadrate");

stack.Push(MyQuadrate);

Console.WriteLine("3 Round");

stack.Push(MyRound);

Console.WriteLine("\nReading figures from stack");

while (stack.Count > 0)

{

Geometrical\_figure f = stack.Pop();

Console.WriteLine(f);

}

Console.ReadKey();

}

}

////////////интерфейсы//////////////-

interface IPrint

{

void Print();

}

public interface IMatrixCheckEmpty<T>

{ /// Возвращает пустой элемент

T getEmptyElement();

/// Проверка что элемент является пустым

bool checkEmptyElement(T element);

}

///////////////классы////////////////-

abstract class Geometrical\_figure : IComparable, IPrint

{

public virtual double Area()

{

return 0;

}

public int CompareTo(object obj)

{

Geometrical\_figure figure = (Geometrical\_figure)obj;

if (this.Area() < figure.Area())

return -1;

else if (this.Area() == figure.Area())

return 0;

else return 1;

}

public void Print()

{

Console.WriteLine(this.ToString());

}

}

////////////////////////////////////-

class Square : Geometrical\_figure, IComparable, IPrint

{

public Square(double length, double height)

{

this.length = length;

this.height = height;

}

private double length;

private double height;

override public double Area()

{

return length \* height;

}

public override string ToString()

{

return "Height: " + height.ToString() + " Length: " + length.ToString() + " Square area = " + this.Area().ToString();

}

public void Print()

{

Console.WriteLine(this.ToString());

}

public int CompareTo(object obj)

{

Geometrical\_figure figure = (Geometrical\_figure)obj;

if (this.Area() < figure.Area())

return -1;

else if (this.Area() == figure.Area())

return 0;

else return 1;

}

public double Length

{

get

{

return length;

}

}

public double Height

{

get

{

return height;

}

/\* set

{

this.height = value;

}\*/

}

}

////////////////////////////////////-

class Quadrate : Square, IComparable, IPrint

{

public Quadrate(double side) : base(side, side)

{

this.side = side;

}

private double side;

override public double Area()

{

return side \* side;

}

override public string ToString()

{

return "Side: " + side.ToString() + " Quadrate area = " + this.Area().ToString();

}

public new void Print()

{

Console.WriteLine(this.ToString());

}

}

////////////////////////////////////-

class Round : Geometrical\_figure, IComparable, IPrint

{

public Round(double radius)

{

this.radius = radius;

}

private double radius;

override public double Area()

{

return radius \* radius \* Math.PI;

}

override public string ToString()

{

return "Radius: " + radius.ToString() + " Round area = " + this.Area().ToString();

}

public new void Print()

{

Console.WriteLine(this.ToString());

}

public new int CompareTo(object obj)

{

Geometrical\_figure figure = (Geometrical\_figure)obj;

if (this.Area() < figure.Area())

return -1;

else if (this.Area() == figure.Area())

return 0;

else return 1;

}

public double Radius

{

get

{

return radius;

}

}

}

////////////////////////////////////-

public class Matrix<T>

{/// Словарь для хранения значений

Dictionary<string, T> \_matrix = new Dictionary<string, T>();

/// Количество элементов по горизонтали/вертикали (максимальное количество столбцов)

int maxX;

int maxY;

int maxZ = 0;

/// Реализация интерфейса для проверки пустого элемента

IMatrixCheckEmpty<T> сheckEmpty;

/// Конструктор

public Matrix(int px, int py, int pz, IMatrixCheckEmpty<T> сheckEmptyParam)

{

this.maxX = px;

this.maxY = py;

this.maxZ = pz;

this.сheckEmpty = сheckEmptyParam;

}

/// Индексатор для доступа к данных

public T this[int x, int y, int z]

{

set ///аксессор индексатора (запись элемента в матрицу)

{

CheckBounds(x, y,z);

string key = DictKey(x, y,z); this.\_matrix.Add(key, value);

}

get /// аксессор индексатора (вывод элемента матрицы)

{

CheckBounds(x, y,z);

string key = DictKey(x, y,z);

if (this.\_matrix.ContainsKey(key))

{

return this.\_matrix[key];

}

else

{

return this.сheckEmpty.getEmptyElement();

}

}

}

/// Проверка границ

void CheckBounds(int x, int y, int z)

{

if (x < 0 || x >= this.maxX)

{

throw new ArgumentOutOfRangeException("x", "x=" + x + " выходит за границы");

}

if (y < 0 || y >= this.maxY)

{

throw new ArgumentOutOfRangeException("y", "y=" + y + " выходит за границы");

}

if (z < 0 || z >= this.maxZ)

{

throw new ArgumentOutOfRangeException("z", "z=" + z + " выходит за границы");

}

}

/// Формирование ключа

string DictKey(int x, int y, int z)

{

return x.ToString() + "\_" + y.ToString() +"\_" + z.ToString();

}

/// Приведение к строке <returns></returns>

public override string ToString()

{

StringBuilder b = new StringBuilder();

for (int k = 0; k < this.maxZ; k++)

{

b.Append("[");

for (int j = 0; j < this.maxY; j++)

{

b.Append("[");

for (int i = 0; i < this.maxX; i++)

{ //Добавление разделителя-табуляции

if (i > 0)

{

b.Append("\t");

} //Если текущий элемент не пустой

if (!this.сheckEmpty.checkEmptyElement(this[i, j,k]))

{ //Добавить приведенный к строке текущий элемент

b.Append(this[i, j,k].ToString());

}

else

{ //Иначе добавить признак пустого значения

b.Append(" - ");

}

}

b.Append("]\n");

}

b.Append("]\n");

}

return b.ToString();

}

}

////////////////////////////////////-

class FigureMatrixCheckEmpty : IMatrixCheckEmpty<Geometrical\_figure>

{ /// В качестве пустого элемента возвращается null

public Geometrical\_figure getEmptyElement()

{

return null;

}

/// Проверка что переданный параметр равен null

public bool checkEmptyElement(Geometrical\_figure element)

{

bool Result = false;

if (element == null)

{

Result = true;

}

return Result;

}

}

////////////////////////////////////-

public class SimpleListItem<T> /// контейнер элемента списка

{

public T data { get; set; } /// Данные

public SimpleListItem<T> next { get; set; } /// Следующий элемент

public SimpleListItem(T param) /// конструктор

{

this.data = param;

}

}

////////////////////////////////////-

public class SimpleList<T> : IEnumerable<T> where T : IComparable

{

protected SimpleListItem<T> first = null; ///Первый элемент списка

protected SimpleListItem<T> last = null; /// Последний элемент списка

/// Количество элементов /// </summary>

public int Count

{ get

{

return \_count;

}

protected set

{

\_count = value;

}

}

int \_count;

public void Add(T element) /// Добавление элемента

{

SimpleListItem<T> newItem = new SimpleListItem<T>(element);

this.Count++;

if (last == null) ///Добавление первого элемента

{

this.first = newItem; this.last = newItem;

}

else ///Добавление следующих элементов

{

//Присоединение элемента к цепочке

this.last.next = newItem;

//Присоединенный элемент считается последним

this.last = newItem;

}

}

public SimpleListItem<T> GetItem(int number) /// Чтение контейнера с заданным номером

{

if ((number < 0) || (number >= this.Count))

{

throw new Exception("Выход за границу индекса"); //Можно создать собственный класс исключения

}

SimpleListItem<T> current = this.first;

int i = 0; //Пропускаем нужное количество элементов

while (i < number)

{ //Переход к следующему элементу

current = current.next;

//Увеличение счетчика

i++;

}

return current;

}

/// Чтение элемента с заданным номером

public T Get(int number)

{

return GetItem(number).data;

}

public IEnumerator<T> GetEnumerator() /// Для перебора коллекции

{

SimpleListItem<T> current = this.first;

while (current != null) //Перебор элементов

{

yield return current.data; //Возврат текущего значения

current = current.next; //Переход к следующему элементу

}

}

//Реализация обобщенного IEnumerator<T> требует реализации необобщенного интерфейса

//Данный метод добавляется автоматически при реализации интерфейса

System.Collections.IEnumerator

System.Collections.IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

public void Sort() /// Cортировка

{

Sort(0, this.Count - 1);

}

/// Алгоритм быстрой сортировки

private void Sort(int low, int high)

{

int i = low;

int j = high; T x = Get((low + high) / 2);

do

{

while (Get(i).CompareTo(x) < 0) ++i;

while (Get(j).CompareTo(x) > 0) --j;

if (i <= j)

{

Swap(i, j);

i++;

j--; }

}

while (i <= j);

if (low < j)

Sort(low, j);

if (i < high)

Sort(i, high);

}

////////////////////////////////////-

/// Вспомогательный метод для обмена элементов при сортировке

private void Swap(int i, int j)

{

SimpleListItem<T> ci = GetItem(i);

SimpleListItem<T> cj = GetItem(j);

T temp = ci.data;

ci.data = cj.data;

cj.data = temp;

}

}

class SimpleStack<T> : SimpleList<T> where T : IComparable

{

public void Push(T element) /// Добавление в стек

{

Add(element); //Добавление в конец списка уже реализовано

}

public T Pop() /// Удаление и чтение из стека

{

T Result = default(T); // default(T) - значение для типа T по умолчанию

if (this.Count == 0) // Если стек пуст, возвращается значение по умолчанию для типа

return Result;

if (this.Count == 1) // Если элемент единственный то из него читаются данные

{

Result = this.first.data;

this.first = null; this.last = null; // обнуляются указатели начала и конца списка

}

else // В списке более одного элемента

{

SimpleListItem<T> newLast = this.GetItem(this.Count - 2); // Поиск предпоследнего элемента

Result = newLast.next.data; // Чтение значения из последнего элемента

this.last = newLast; // предпоследний элемент считается последним

newLast.next = null; // последний элемент удаляется из списка

}

this.Count--; // Уменьшение количества элементов в списке

return Result; // Возврат результата

}

}

}

**Экранные формы с примерами выполнения программы:**

