**Отчёт по лабораторной работе №3 по предмету «Базовые компоненты интернет-технологий»**

Подготовила: Гладова Анастасия

Группа: ИУ5-31

МГТУ имени Н.Э.Баумана

2017 г.

**Задание:** разработать программу, реализующую работу с коллекциями.

1. Программа должна быть разработана в виде консольного приложения на языке C#.

2. Создать объекты классов «Прямоугольник», «Квадрат», «Круг».

3. Для реализации возможности сортировки геометрических фигур для класса «Геометрическая фигура» добавить реализацию интерфейса IComparable. Сортировка производится по площади фигуры.

4. Создать коллекцию класса ArrayList. Сохранить объекты в коллекцию. Отсортировать коллекцию. Вывести в цикле содержимое коллекции.

5. Создать коллекцию класса List<Figure>. Сохранить объекты в коллекцию. Отсортировать коллекцию. Вывести в цикле содержимое коллекции.

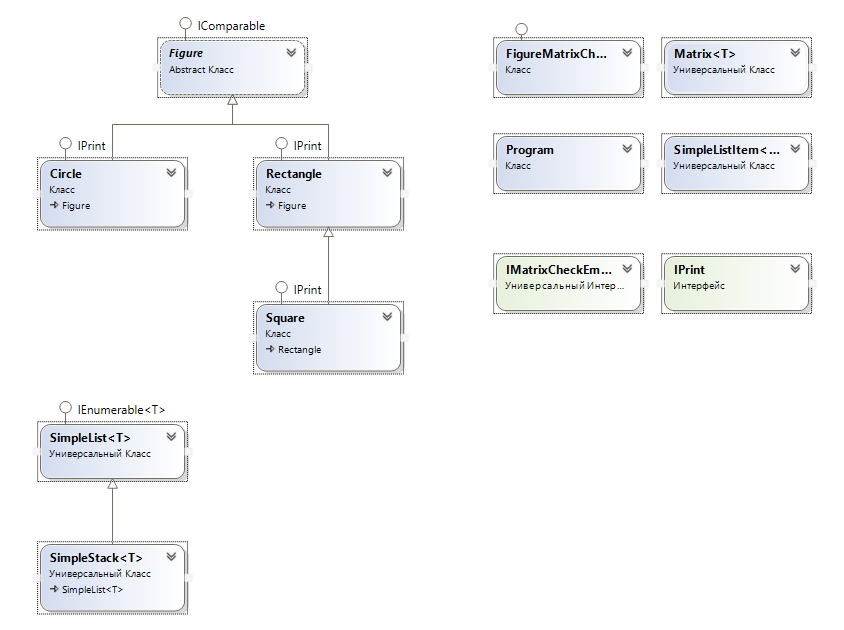
6. Модифицировать класс разреженной матрицы (проект SparseMatrix) для работы с тремя измерениями – x,y,z. Вывод элементов в методе ToString() осуществлять в том виде, который Вы считаете наиболее удобным. Разработать пример использования разреженной матрицы для геометрических фигур.

7. Реализовать класс «SimpleStack» на основе односвязного списка. Класс SimpleStack наследуется от класса SimpleList (проект SimpleListProject). Необходимо добавить в класс методы:

• public void Push(T element) – добавление в стек;

• public T Pop() – чтение с удалением из стека.

8. Пример работы класса SimpleStack реализовать на основе геометрических фигур.

**Диаграмма классов:**

**Текст программы:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Collections;

//Разработать программу, реализующую работу с коллекциями.

//1. Программа должна быть разработана в виде консольного приложения на языке C#.

//2. Создать объекты классов «Прямоугольник», «Квадрат», «Круг».

//3. Для реализации возможности сортировки геометрических фигур для класса «Геометрическая фигура» добавить реализацию интерфейса IComparable.

//Сортировка производится по площади фигуры.

//4. Создать коллекцию класса ArrayList. Сохранить объекты в коллекцию. Отсортировать коллекцию. Вывести в цикле содержимое коллекции.

//5. Создать коллекцию класса List<Figure>. Сохранить объекты в коллекцию. Отсортировать коллекцию. Вывести в цикле содержимое коллекции.

//6. Модифицировать класс разреженной матрицы (проект SparseMatrix) для работы с тремя измерениями – x,y,z.

//Вывод элементов в методе ToString() осуществлять в том виде, который Вы считаете наиболее удобным.

//Разработать пример использования разреженной матрицы для геометрических фигур.

//7. Реализовать класс «SimpleStack» на основе односвязного списка.

//Класс SimpleStack наследуется от класса SimpleList (проект SimpleListProject).

//Необходимо добавить в класс методы:

//• public void Push(T element) – добавление в стек;

//• public T Pop() – чтение с удалением из стека.

//8. Пример работы класса SimpleStack реализовать на основе геометрических фигур.

//Коллекция в программировании — программный объект, содержащий в себе, тем или иным образом,

//набор значений одного или различных типов, и позволяющий обращаться к этим значениям.

//Назначение коллекции — служить хранилищем объектов и обеспечивать доступ к ним.

namespace bkit\_lab3

{

class Program

{

static void Main(string[] args)

{

//Создание объектов классов фигур:

Rectangle r = new Rectangle(4, 6);

Square s = new Square(2);

Circle c = new Circle(5);

Console.WriteLine("ArrayList: ");

ArrayList al = new ArrayList();

al.Add(r);

al.Add(s);

al.Add(c);

foreach (var x in al) Console.WriteLine(x);

Console.WriteLine("ArrayList после сортировки: ");

al.Sort();

foreach (var x in al) Console.WriteLine(x);

Console.WriteLine("List <Figure>: ");

//Добавление в список:

List<Figure> f = new List<Figure>();

f.Add(r);

f.Add(s);

f.Add(c);

foreach (var x in f) Console.WriteLine(x);

Console.WriteLine("List <Figure> после сортировки: ");

f.Sort();

foreach (var x in f) Console.WriteLine(x);

//Cоздание и вывод в консоль объекта класса «Разреженная матрица»:

Console.WriteLine("Матрица: ");

Matrix<Figure> matr = new Matrix<Figure>(3, 3, new FigureMatrixCheckEmpty());

matr[0, 0] = r;

matr[1, 1] = s;

matr[2, 2] = c;

Console.WriteLine(matr.ToString());

//Использование класса SimpleList:

Console.WriteLine("Список");

SimpleList<Figure> list = new SimpleList<Figure>();

list.Add(c);

list.Add(r);

list.Add(s);

foreach (var x in list) Console.WriteLine(x);

//сортировка

list.Sort();

Console.WriteLine("Список после сортировки:");

foreach (var x in list) Console.WriteLine(x);

//Использование класса SimpleStack:

Console.WriteLine("\nСтек: ");

SimpleStack<Figure> stack = new SimpleStack<Figure>();

//добавление данных в стек

stack.Push(r);

stack.Push(s);

stack.Push(c);

//чтение данных из стека

while (stack.Count > 0)

{

Figure ff = stack.Pop();

Console.WriteLine(ff);

}

Console.ReadLine();

}

}

interface IPrint

{

void Print();

}

/// <summary>

/// Класс Геометрическая фигура

/// </summary>

abstract class Figure : IComparable

{

/// <summary>

/// Тип фигуры

/// </summary>

public string Type

{

get

{

return this.\_Type;

}

protected set

{

this.\_Type = value;

}

}

string \_Type;

/// <summary>

/// Вычисление площади

/// </summary>

/// <returns></returns>

public abstract double Area();

/// <summary>

/// Переопределение метода Object

/// </summary>

/// <returns></returns>

public override string ToString()

{

return this.Type + " площадью " + this.Area().ToString();

}

/// <summary>

/// Сравнение элементов (для сортировки)

/// this - левый параметр сравнения

/// </summary>

/// <param name="obj">правый параметр сравнения</param>

/// <returns>

/// -1 - если левый параметр меньше правого

/// 0 - параметры равны

/// 1 - правый параметр меньше левого

/// </returns>

public int CompareTo(object obj)

{

//Приведение параметра к типу "фигура

Figure p = (Figure)obj;

//Сравнение

if (this.Area() < p.Area()) return -1;

else if (this.Area() == p.Area()) return 0;

else return 1; //(this.Area() > p.Area())

}

}

/// <summary>

/// Класс Прямоугольник

/// </summary>

class Rectangle : Figure, IPrint

{

double height;

double width;

/// <summary>

/// Основной конструктор

/// </summary>

public Rectangle(double ph, double pw)

{

this.height = ph;

this.width = pw;

this.Type = "Прямоугольник";

}

/// <summary>

/// Вычисление площади

/// </summary>

public override double Area()

{

double Res = this.width \* this.height;

return Res;

}

public void Print()

{

Console.WriteLine(this.ToString());

}

}

/// <summary>

/// Класс Квадрат

/// </summary>

class Square : Rectangle, IPrint

{

public Square(double size)

: base(size, size)

{

this.Type = "Квадрат";

}

}

/// <summary>

/// Класс Круг

/// </summary>

class Circle : Figure, IPrint

{

double radius;

/// <summary>

/// Основной конструктор

/// </summary>

public Circle(double pr)

{

this.radius = pr;

this.Type = "Круг";

}

public override double Area()

{

double Result = Math.PI \* this.radius \* this.radius;

return Result;

}

public void Print()

{

Console.WriteLine(this.ToString());

}

}

//стр.150

//Создание нестандартной коллекции без использования стандартных коллекций.

//Пример создания коллекции в том случае, когда использовать стандартные коллекции нецелесообразно.

//В качестве примера создается простой однонаправленный список и в качестве его наследника реализуется класс стека.

//Рассматриваемые классы создаются полностью с нуля без использования стандартных коллекций.

//Классы простого списка

/// <summary>

/// Элемент списка

/// </summary>

public class SimpleListItem<T>

{

/// <summary>

/// Данные

/// </summary>

public T data { get; set; } // Тип-обобщение T – тип данных списка

/// <summary>

/// Следующий элемент

/// </summary>

public SimpleListItem<T> next { get; set; }

///конструктор

public SimpleListItem(T param)

{

this.data = param;

}

}

// Класс SimpleList реализует список:

/// <summary>

/// Список

/// </summary>

public class SimpleList<T> : IEnumerable<T>

where T : IComparable

{

/// <summary>

/// Первый элемент списка

/// </summary>

protected SimpleListItem<T> first = null;

/// <summary>

/// Последний элемент списка

/// </summary>

protected SimpleListItem<T> last = null;

/// <summary>

/// Количество элементов

/// </summary>

public int Count

{

get { return \_count; }

protected set { \_count = value; }

}

int \_count;

/// <summary>

/// Добавление элемента

/// </summary>

/// <param name="element"></param>

public void Add(T element)

{

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

this.Count++;

//Добавление первого элемента

if (last == null)

{

this.first = newItem;

this.last = newItem;

}

//Добавление следующих элементов

else

{

//Присоединение элемента к цепочке

this.last.next = newItem;

//Просоединенный элемент считается последним

this.last = newItem;

}

}

/// <summary>

/// Чтение контейнера с заданным номером

/// </summary>

public SimpleListItem<T> GetItem(int number)

{

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

{

//Можно создать собственный класс исключения

throw new Exception("Выход за границу индекса");

}

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;

}

/// <summary>

/// Чтение элемента с заданным номером

/// </summary>

public T Get(int number)

{

return GetItem(number).data;

}

/// <summary>

/// Для перебора коллекции

/// </summary>

public IEnumerator<T> GetEnumerator()

{

SimpleListItem<T> current = this.first;

//Перебор элементов

while (current != null)

{

//Возврат текущего значения

yield return current.data;

//Переход к следующему элементу

current = current.next;

}

}

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

//Данный метод добавляется автоматически при реализации интерфейса

System.Collections.IEnumerator

System.Collections.IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

/// <summary>

/// Cортировка

/// </summary>

public void Sort()

{

Sort(0, this.Count - 1);

}

/// <summary>

/// Реализация алгоритма быстрой сортировки

/// </summary>

/// <param name="low"></param>

/// <param name="high"></param>

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);

}

/// <summary>

/// Вспомогательный метод для обмена элементов при сортировке

/// </summary>

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;

}

}

// Класс стека наследуется от класса списка:

///<summary>

///Класс стек

///</summary>

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

{

///<summary>

/// Добавление в стек

/// </summary>

public void Push(T element)

{ //Добавление в конец списка уже реализовано

Add(element);

}

/// <summary>

/// Удаление и чтение из стека

/// </summary>

public T Pop()

{

//default(T) - значение для типа T по умолчанию

T Result = default(T);

//Если стек пуст, возвращается значение по умолчанию для типа

if (this.Count == 0) return Result;

//Если элемент единственный

if (this.Count == 1)

{

//то из него читаются данные

Result = this.first.data;

//обнуляются указатели начала и конца списка

this.first = null;

this.last = null;

}

//В списке более одного элемента

else

{ //Поиск предпоследнего элемента

SimpleListItem<T> newLast = this.GetItem(this.Count - 2);

//Чтение значения из последнего элемента

Result = newLast.next.data;

//предпоследний элемент считается последним

this.last = newLast;

//последний элемент удаляется из списка

newLast.next = null;

}

//Уменьшение количества элементов в списке

this.Count--;

//Возврат результата

return Result;

}

}

public interface IMatrixCheckEmpty<T>

{ /// <summary>

/// Возвращает пустой элемент

/// </summary>

T getEmptyElement();

/// <summary>

/// Проверка что элемент является пустым

/// </summary>

bool checkEmptyElement(T element);

}

//Использование класса «Разреженная матрица».

//Вначале необходимо создать класс, реализующий интерфейс IMatrixCheckEmpty:

class FigureMatrixCheckEmpty : IMatrixCheckEmpty<Figure>

{ /// <summary>

/// В качестве пустого элемента возвращается null

/// </summary>

public Figure getEmptyElement()

{ return null; }

/// <summary>

/// Проверка что переданный параметр равен null

/// </summary>

public bool checkEmptyElement(Figure element)

{

bool Result = false;

if (element == null)

{ Result = true; }

return Result;

}

}

//стр.142

//Создание нестандартной коллекции на основе стандартной коллекции

//В том случае, когда необходимой стандартной коллекции в библиотеке нет, прикладному программисту нужно самостоятельно реализовывать требуемую коллекцию.

//В этом случае возможны два подхода:

//1. Использовать стандартную коллекцию и написать «надстройку» над ней для реализации требуемого поведения.

//Данный способ предпочтителен, так как требует относительно небольших ресурсов при кодировании.

//2. Если предыдущий подход невозможен, то приходится создавать коллекцию полностью с нуля.

//Подход на примере класса «Разреженная матрица».

//Разреженная матрица – тип данных, который достаточно часто используется в информатике.

//Это матрица большой размерности, которая может быть заполнена на 3-5%, остальные ячейки матрицы пусты.

//Использование в этом случае обычной матрицы нецелесообразно, так как потребует выделения очень больших объемов памяти для хранения пустых значений.

//Разреженную матрицу можно реализовать «поверх» стандартной коллекции.

//В приведенном ниже примере разреженная матрица реализована на основе словаря.

//Ключом элемента словаря является комбинация индексов ячейки матрицы по строке и столбцу, значением элемента словаря – значение элемента матрицы.

//Класс «Разреженная матрица» реализован в виде обобщенной коллекции, класс-обобщение T соответствует типу ячейки матрицы.

/// <summary>

/// Класс Разреженная матрица

/// </summary>

public class Matrix<T>

{

/// <summary>

///Словарь для хранения значений

/// </summary>

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

/// <summary>

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

/// </summary>

int maxX;

/// <summary>

/// Количество элементов по вертикали (максимальное количество строк)

/// </summary>

int maxY;

/// <summary>

/// Реализация интерфейса для проверки пустого элемента

/// </summary>

IMatrixCheckEmpty<T> сheckEmpty;

/// <summary>

/// Конструктор

/// </summary>

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

{

this.maxX = px;

this.maxY = py;

this.сheckEmpty = сheckEmptyParam;

}

/// <summary>

/// Индексатор для доступа к данных

/// </summary>

public T this[int x, int y]

{

set

{

CheckBounds(x, y);

string key = DictKey(x, y);

this.\_matrix.Add(key, value);

}

get

{

CheckBounds(x, y);

string key = DictKey(x, y);

if (this.\_matrix.ContainsKey(key)) { return this.\_matrix[key]; }

else { return this.сheckEmpty.getEmptyElement(); }

}

}

/// <summary>

/// Проверка границ

/// </summary>

void CheckBounds(int x, int y)

{

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");

}

}

/// <summary>

/// Формирование ключа

/// </summary>

string DictKey(int x, int y)

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

/// <summary>

/// Приведение к строке

/// </summary>

/// <returns></returns>

public override string ToString()

{

StringBuilder b = new StringBuilder();

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]))

{

//Добавить приведенный к строке текущий элемент

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

}

else

{

//Иначе добавить признак пустого значения

b.Append(" - ");

}

}

b.Append("]\n");

}

return b.ToString();

}

}

}

**Экранные формы с примерами выполнения программы:**

