Московский государственный технический университет им. Н.Э. Баумана.

Факультет «Информатика и управление»

Кафедра ИУ5. Курс «Базовые компоненты интернет-технологий»

Отчет по лабораторной работе №3 «Разработка программы, реализующей работу с коллекциями»

Выполнил:
студент группы ИУ5-34Б
Сергеев Илья

Проверил:
преподаватель каф. ИУ5
Гапанюк Ю. Е.

Подпись и дата:

Подпись и дата:

Москва, 2018 г.

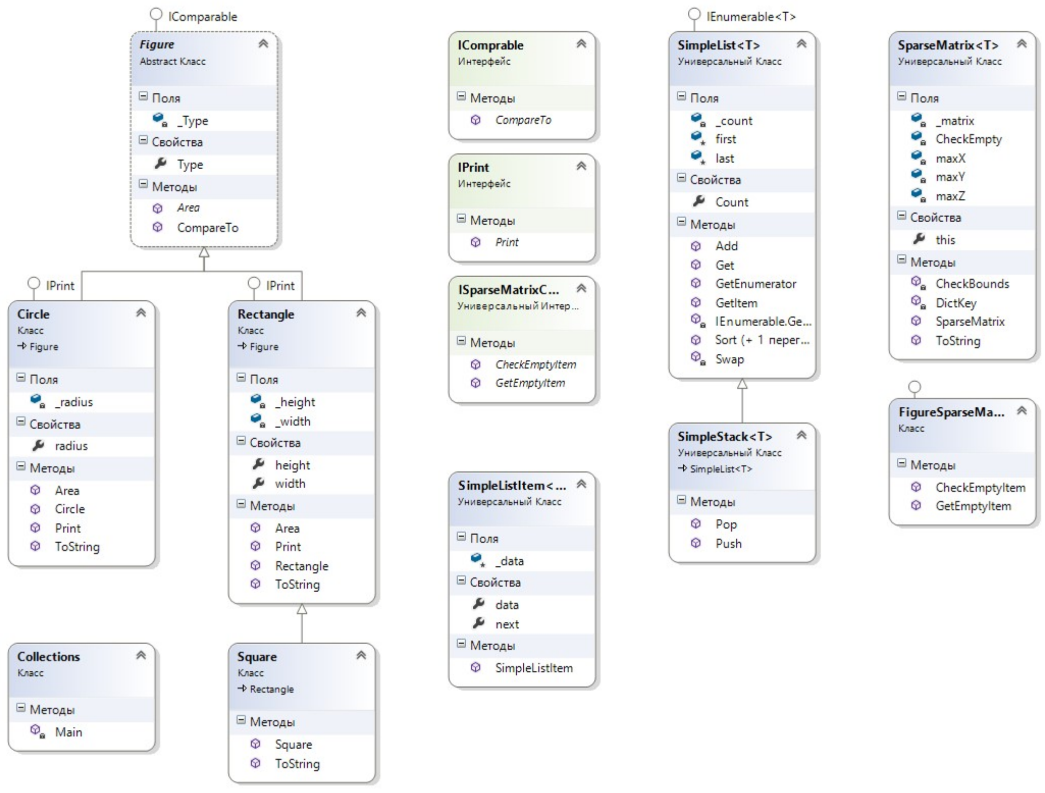
<u>Описание задания</u>

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 (разобранного в пособии). 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.

<u>Диаграмма классов</u>

Диаграмма классов генерируется автоматически в среде Visual Studio:



Текст программы (листинг)

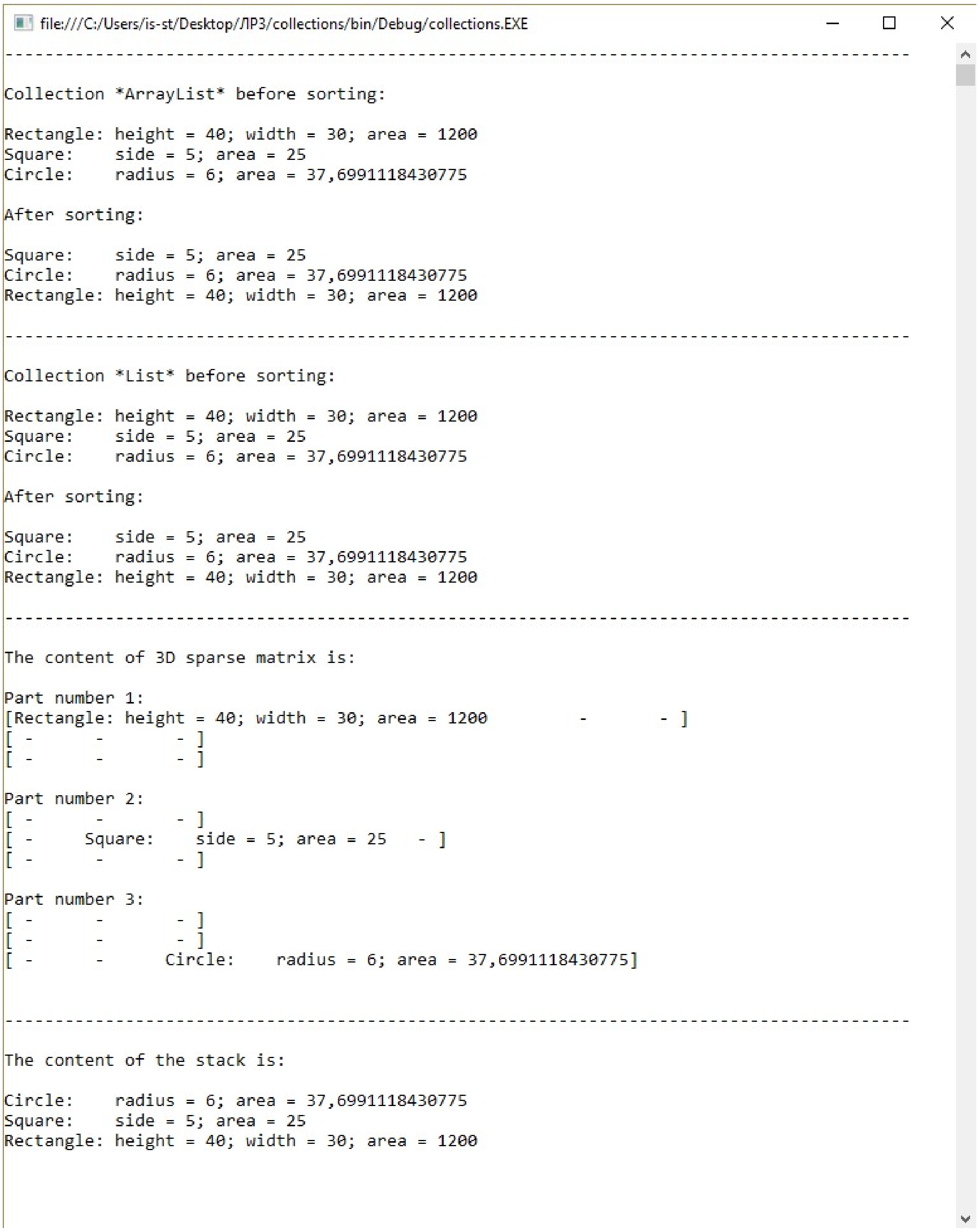
```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;

namespace collections
{
    /// <summary>
    /// Printing interface
    /// </summary>
    interface IPrint
    {
        void Print();
    }

    /// <summary>
    /// Comparing interface for abstract figures
    /// </summary>
    interface IComprable
    {
        int CompareTo(object obj);
    }

    /// <summary>
    /// Geometric figure class
    /// </summary>
    abstract class Figure : IComparable
    {
        private string _Type;

        public string Type
        {
            get { return this._Type; }
            set { this._Type = value; }
        }

        public abstract double Area();

        public int CompareTo(object obj)
        {
            Figure F = (Figure)obj;

            if (this.Area() < F.Area())
                return -1;
            else if (this.Area() == F.Area())
                return 0;
            else
                return 1;
        }
    }

    /// <summary>
    /// Rectangle class
    /// </summary>
    class Rectangle : Figure, IPrint
    {
        private double _height;
        private double _width;

        public double height
        {
            get { return _height; }
            set { _height = value; }
        }

        public double width
        {
            get { return _width; }
            set { _width = value; }
        }

        /// <summary>
        /// Constructs a rectangle with width "w" and height "h"
        /// </summary>
        /// <param name="w"></param>
        /// <param name="h"></param>
        public Rectangle(double w, double h)
        {
            height = h;
            width = w;
            this.Type = "Rectangle";
        }

        /// <summary>
        /// Encalculates an area of the retangle
        /// </summary>
        /// <returns></returns>
        public override double Area()
        {
            return height * width;
        }

        /// <summary>
        /// Converts an information about this rectangle to "String"
        /// </summary>
        /// <returns></returns>
        public override string ToString()
        {
            return this.Type + ": height = " + this.height.ToString() + "; width = " +
this.width.ToString() + "; area = " + this.Area().ToString();
        }

        /// <summary>
        /// Outputs the an information about the rectangle
        /// </summary>
        public void Print()
        {
            Console.WriteLine(this.ToString());
        }
    }

    /// <summary>
    /// Square class
    /// </summary>
    class Square : Rectangle
    {
        /// <summary>
        /// Constructs a square with side "a"
        /// </summary>
        /// <param name="w"></param>
        /// <param name="h"></param>
        public Square(double a) : base(a, a) { Type = "Square"; }

        /// <summary>
        /// Converts an information about this square to "String"
        /// </summary>
        /// <returns></returns>
        public override string ToString()
        {
            return this.Type + ":    side = " + this.height.ToString() + "; area = " +
this.Area().ToString();
        }
    }

    /// <summary>
    /// Circle class
    /// </summary>
    class Circle : Figure, IPrint
    {
        private double _radius;

        public double radius
        {
            get { return _radius; }
            set { _radius = value; }
        }

        /// <summary>
        /// Constructs a circle with radius "r"
        /// </summary>
        /// <param name="w"></param>
        /// <param name="h"></param>
        public Circle(double r)
        {
            radius = r;
            Type = "Circle";
        }

        /// <summary>
        /// Encalculates an area of the circle
        /// </summary>
        /// <returns></returns>
        public override double Area()
        {
            return 2 * System.Math.PI * radius;
        }

        /// <summary>
        /// Converts an information about this circle to "String"
        /// </summary>
        /// <returns></returns>
        public override string ToString()
        {
            return Type + ":    radius = " + this.radius.ToString() + "; area = " +
this.Area().ToString();
        }

        /// <summary>
        /// Outputs the an information about the circle
        /// </summary>
        public void Print()
        {
            Console.WriteLine(this.ToString());
        }
    }

    /// <summary>
    /// Checking if SpaseMatrix is empty iface
    /// </summary>
    /// <typeparam name="T"></typeparam>
    interface ISparseMatrixCheckEmpty<T>
    {
        T GetEmptyItem();
        bool CheckEmptyItem(T item);
    }

    /// <summary>
    /// Sparse matrix class
    /// </summary>
    /// <typeparam name="T"></typeparam>
    class SparseMatrix<T>
    {
        Dictionary<string, T> _matrix = new Dictionary<string, T>(); //dictionary for
accumulating values

        int maxX; //max items on horizontal axis
        int maxY; //max items on vertical axis
        int maxZ; //max items on applicate axis

        ISparseMatrixCheckEmpty<T> CheckEmpty;

        /// <summary>
        /// Constructor with x, y and parameter for checking is matrix is empty
        /// </summary>
        /// <param name="px"></param>
        /// <param name="py"></param>
        /// <param name="CheckEmptyParam"></param>
        public SparseMatrix(int px, int py, int pz, ISparseMatrixCheckEmpty<T>
CheckEmptyParam)
        {
            this.maxX = px;
            this.maxY = py;
            this.maxZ = pz;
            this.CheckEmpty = CheckEmptyParam;
        }

        /// <summary>
        /// Checks bounds for correctness
        /// </summary>
        /// <param name="x"></param>
        /// <param name="y"></param>
        void CheckBounds(int x, int y, int z)
        {
            if (x < 0 || x >= this.maxX)
                throw new ArgumentOutOfRangeException("x = " + x + " is out of
range!\n");
            if (y < 0 || y >= this.maxY)
                throw new ArgumentOutOfRangeException("y = " + y + " is out of
range!\n");
            if (z < 0 || z >= this.maxZ)
                throw new ArgumentOutOfRangeException("z = " + z + " is out of
range!\n");
        }

        /// <summary>
        /// Building a dictionary key
        /// </summary>
        /// <param name="x"></param>
        /// <param name="y"></param>
        /// <returns></returns>
        string DictKey(int x, int y, int z)
        {
            return x.ToString() + "_" + y.ToString() + "_" + z.ToString();
        }

        /// <summary>
        /// Provides access to item by definitely index
        /// </summary>
        /// <param name="x"></param>
        /// <param name="y"></param>
        /// <returns></returns>
        public T this[int x, int y, int z]
        {
            set
            {
                CheckBounds(x, y, z);
                string key = DictKey(x, y, z);
                this._matrix.Add(key, value);
            }
            get
            {
                CheckBounds(x, y, z);
                string key = DictKey(x, y, z);
                if (this._matrix.ContainsKey(key))
                    return this._matrix[key];
                else
                    return this.CheckEmpty.GetEmptyItem();
            }
        }

        /// <summary>
        /// Builds an output string with the matrix data
        /// </summary>
        /// <returns></returns>
        public override string ToString()
        {
            StringBuilder b = new StringBuilder();
            for (int k = 0; k < this.maxZ; k++)
            {
                b.Append("\nPart number " + (k + 1).ToString() + ":\n");
                for (int j = 0; j < this.maxY; j++)
                {
                    b.Append("[");
                    for (int i = 0; i < this.maxX; i++)
                    {
                        if (i > 0)
                            b.Append("\t");
                        if (!this.CheckEmpty.CheckEmptyItem(this[i, j, k]))
                            b.Append(this[i, j, k].ToString());
                        else
                            b.Append(" - ");
                    }
                    b.Append("]\n");
                }
            }
            return b.ToString();
        }
    }

    /// <summary>
    /// Class with realization of iface methods
    /// </summary>
    class FigureSparseMatrixCheckEmpty : ISparseMatrixCheckEmpty<Figure>
    {
        public Figure GetEmptyItem() { return null; }
        public bool CheckEmptyItem(Figure item) { return item == null; }
    }

    /// <summary>
    /// Simple list item class
    /// </summary>
    /// <typeparam name="T"></typeparam>
    class SimpleListItem<T>
```

```
{
    protected T _data;

    /// <summary>
    /// Item data property
    /// </summary>
    public T data
    {
        get { return this._data; }
        set { this._data = value; }
    }

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

    /// <summary>
    /// Constructor by a value
    /// </summary>
    /// <param name="param"></param>
    public SimpleListItem(T param) { this.data = param; }
}

/// <summary>
/// Simple list class
/// </summary>
/// <typeparam name="T"></typeparam>
class SimpleList<T> : IEnumerable<T> where T : IComparable
{
    protected SimpleListItem<T> first = null;
    protected SimpleListItem<T> last = null;

    int _count;

    /// <summary>
    /// Count property
    /// </summary>
    public int Count
    {
        get { return _count; }
        protected set { _count = value; }
    }

    /// <summary>
    /// Adds "item" to the list
    /// </summary>
    /// <param name="item"></param>
    public void Add(T item)
    {
        SimpleListItem<T> NewItem = new SimpleListItem<T>(item);
        this.Count++;

        if (last == null) //if list is empty
        {
            this.first = NewItem;
            this.last = NewItem;
        }
        else //else
        {
            this.last.next = NewItem;
            this.last = NewItem;
        }
    }

    /// <summary>
    /// Returns item by number (index)
    /// </summary>
    /// <param name="num"></param>
    /// <returns></returns>
    public SimpleListItem<T> GetItem(int num)
    {
        if ((num < 0 || num >= this.Count)) //if idex is incorrect
            throw new Exception("Out of range!");

        SimpleListItem<T> cur = this.first;
        int i = 0;
        while (i < num)
        {
            cur = cur.next;
            i++;
        }
        return cur;
    }

    /// <summary>
    /// Returns data of item by number (index)
    /// </summary>
    /// <param name="num"></param>
    /// <returns></returns>
    public T Get(int num) { return GetItem(num).data; }

    /// <summary>
    /// Returns numerator
    /// </summary>
    /// <returns></returns>
    public IEnumerator<T> GetEnumerator()
    {
        SimpleListItem<T> cur = this.first;

        while (cur != null)
        {
            yield return cur.data;
            cur = cur.next;
        }
    }

    /// <summary>
    /// Sorts whole list
    /// </summary>
    public void Sort() { Sort(0, this.Count - 1); }

    /// <summary>
    /// Quick sorting
    /// </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>
        /// Swaps wo items in the list
        /// </summary>
        /// <param name="i"></param>
        /// <param name="j"></param>
        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;
        }

        /// <summary>
        /// Realisation of ENumerator iface
        /// </summary>
        /// <returns></returns>
        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }
    }

    /// <summary>
    /// Simple stack class
    /// </summary>
    /// <typeparam name="T"></typeparam>
    class SimpleStack<T> : SimpleList<T> where T : IComparable
    {
        /// <summary>
        /// Pushes "item" to the stack
        /// </summary>
        /// <param name="item"></param>
        public void Push(T item) { Add(item); }

        /// <summary>
        /// Removes item from the stack and returns it
        /// </summary>
        /// <returns></returns>
        public T Pop()
        {
            T res = default(T); //default value for the following type
            if (this.Count == 0) //if the stack is empty
                return res; //returns default value for the following type
            if (this.Count == 1)
            {
                res = this.first.data;
                this.first = null;
                this.last = null;
            }
            else
            {
                SimpleListItem<T> NewLast = this.GetItem(this.Count - 2);
                res = NewLast.next.data;
                this.last = NewLast;
                NewLast.next = null;
            }
            this.Count--;
            return res;
        }
    }

    class Collections
    {
        static void Main(string[] args)
        {
            const string delim = "-----------------------------------------------------
------------------------------------";

            /* ---2--- */
            Rectangle r1 = new Rectangle(30, 40); //created new Recatangle
            Square s1 = new Square(5); //created new Square
            Circle c1 = new Circle(6); //created new Circle

            /* ---4--- */
            ArrayList al = new ArrayList(); //declaring new *ArrayList* collection
            al.Add(r1); //rectangle added
            al.Add(s1); //square added
            al.Add(c1); //circle added
            Console.WriteLine(delim + "\n\nCollection *ArrayList* before sorting:\n");
            foreach (var x in al)
                Console.WriteLine(x); //output content
            al.Sort(); //sorting a collection
            Console.WriteLine("\nAfter sorting:\n"); //output sorted content
            foreach (var x in al)
                Console.WriteLine(x);
            Console.WriteLine('\n' + delim + "\n");

            /* ---5--- */
            List<Figure> list = new List<Figure>(); //declaring new *List* collection
            list.Add(r1); //rectangle added
            list.Add(s1); //square added
            list.Add(c1); //circle added
            Console.WriteLine("Collection *List* before sorting:\n");
            foreach (var x in list)
                Console.WriteLine(x); //output content
            list.Sort(); //sorting a collection
            Console.WriteLine("\nAfter sorting:\n"); //output sorted content
            foreach (var x in list)
                Console.WriteLine(x);
            Console.WriteLine('\n' + delim + "\n");

            /* ---6--- */
            Console.WriteLine("The content of 3D sparse matrix is:");
            SparseMatrix<Figure> matrix = new SparseMatrix<Figure>(3, 3, 3, new
FigureSparseMatrixCheckEmpty()); //declaring new 3D sparse matrix
            matrix[0, 0, 0] = r1; //rectangle added
            matrix[1, 1, 1] = s1; //square added
            matrix[2, 2, 2] = c1; //circle added
            Console.WriteLine(matrix.ToString()); //output this matrix
            Console.WriteLine('\n' + delim + "\n");

            /* ---8--- */
            Console.WriteLine("The content of the stack is:\n");
            SimpleStack<Figure> stack = new SimpleStack<Figure>(); //declaring a new
simple stack
            stack.Push(r1); //rectangle added
            stack.Push(s1); //square added
            stack.Push(c1); //circle added
            while (stack.Count > 0) //while stack isn't empty
            {
                Figure f = stack.Pop(); //removing item
                Console.WriteLine(f); //output item
            }
```

```
                Console.ReadKey(); //delay for the user
            }
        }
}
```

Экранные формы с примерами выполнения программы (скриншоты)



file:///C:/Users/is-st/Desktop/ЛР3/collections/bin/Debug/collections.EXE

```
------------------------------------------------------------------------------------------

Collection *ArrayList* before sorting:

Rectangle: height = 40; width = 30; area = 1200
Square:    side = 5; area = 25
Circle:    radius = 6; area = 37,6991118430775

After sorting:

Square:    side = 5; area = 25
Circle:    radius = 6; area = 37,6991118430775
Rectangle: height = 40; width = 30; area = 1200

------------------------------------------------------------------------------------------

Collection *List* before sorting:

Rectangle: height = 40; width = 30; area = 1200
Square:    side = 5; area = 25
Circle:    radius = 6; area = 37,6991118430775

After sorting:

Square:    side = 5; area = 25
Circle:    radius = 6; area = 37,6991118430775
Rectangle: height = 40; width = 30; area = 1200

------------------------------------------------------------------------------------------

The content of 3D sparse matrix is:

Part number 1:
[Rectangle: height = 40; width = 30; area = 1200         -       - ]
[ -      -       - ]
[ -      -       - ]

Part number 2:
[ -      -       - ]
[ -     Square:    side = 5; area = 25    - ]
[ -      -       - ]

Part number 3:
[ -      -       - ]
[ -      -       - ]
[ -      -      Circle:    radius = 6; area = 37,6991118430775]


------------------------------------------------------------------------------------------

The content of the stack is:

Circle:    radius = 6; area = 37,6991118430775
Square:    side = 5; area = 25
Rectangle: height = 40; width = 30; area = 1200
```