Міністерство освіти і науки України
Національний технічний університет України „КПІ”
Факультет інформатики та обчислювальної техніки

Кафедра автоматизованих систем обробки
інформації та управління

ЗВІТ

до лабораторної роботи з C#

з дисципліни “Основи технологій програмування”

Виконав студент *ІП-61 Денисенко Марк Олександрович*

(№ групи, прізвище, ім’я, по батькові)

Прийняв *доц. Ліщук К. І.*

(посада, прізвище, ім’я, по батькові)

Київ 2018

Постановка задачи

Варіант 5

При виконанні комп’ютерого практикуму слід реалізувати наступні задачі:

a)Перезавантажити віртуальний метод bool Equals (object obj), таким чином, щоб об’єкти були рівними, якщо рівні всі дані об’єктів. Для кожного з класів самостійно визначити, які атрибути використовуються для порівняння;

b)Визначити операції == та != . При цьому врахувати, що визначення операцій повинно бути погоджено з перезавантаженим методом Equals, тобто критерії, за якими перевіряється рівність об'єктів в методі Equals, повинні використовуватися і при перевірці рівності об'єктів в операціях == та !=;

c)Перевизначити віртуальний метод int GetHashCode(). Класи базової бібліотеки, що викликають метод GetHashCode() з призначеного користувальницького типу, припускають, що рівним об'єктів відповідають рівні значення хеш-кодів. Тому в разі, коли під рівністю об'єктів розуміється збіг даних (а не посилань), реалізація методу GetHashCode() повинна для об'єктів з однаковими даними повертати рівні значення хеш-кодів.

d)Визначити метод object DeepCopy() для створення повної копії об'єкта. Визначені в деяких класах базової бібліотеки методи Clone() та Copy() створюють обмежену (shallow) копію об'єкта - при копіюванні об'єкта копії створюються тільки для полів структурних типів, для полів, на які посилаються типи, копіюються тільки посилання. В результаті в обмеженій копії об'єкта поля-посилання вказують на ті ж об'єкти, що і в вихідному об'єкті. Метод DeepCopy() повинен створити повні копії всіх об'єктів, посилання на які містять поля типу. Після створення повна копія не залежить від вихідного об'єкта - зміна будь-якого поля або властивості вихідного об'єкта не повинно призводити до зміни копії. При реалізації методу DeepCopy() в класі, який має поле типу System.Collections.ArrayList, слід мати на увазі, що визначені в класі ArrayList конструктор ArrayList (ICollection) і метод Clone() при створенні копії колекції, що складається з елементів, на які посилаються типів, копіюють тільки посилання. Метод DeepCop() повинен створити як копії елементів колекції ArrayList, так і повні копії об'єктів, на які посилаються елементи колекції. Для типів, що містять колекції, реалізація методу DeepCopy() спрощується, якщо в типах елементів колекцій також визначити метод DeepCopy().

e)Перезавантажити віртуальний метод string ToString() для формування строки з інформацією про всі елементи списку

f) Визначити клас, котрий містить типізовану колекцію та котрий за допомогою подій інформує про зміни в колекції. Колекція складається з об'єктів силочних типів. Колекція змінюється при видаленні/додаванні елементів або при зміні одного з вхідних в колекцію посилань, наприклад, коли одному з посилань присвоюється нове значення. В цьому випадку у відповідних методах або властивості класу кидаються події. При зміні даних об'єктів, посилання на які входять в колекцію, значення самих посилань не змінюються.

Цей тип змін не породжує подій. Для подій, що сповіщають про зміни в колекції, визначається свій делегат. Події реєструються в спеціальних класах-слухачах.

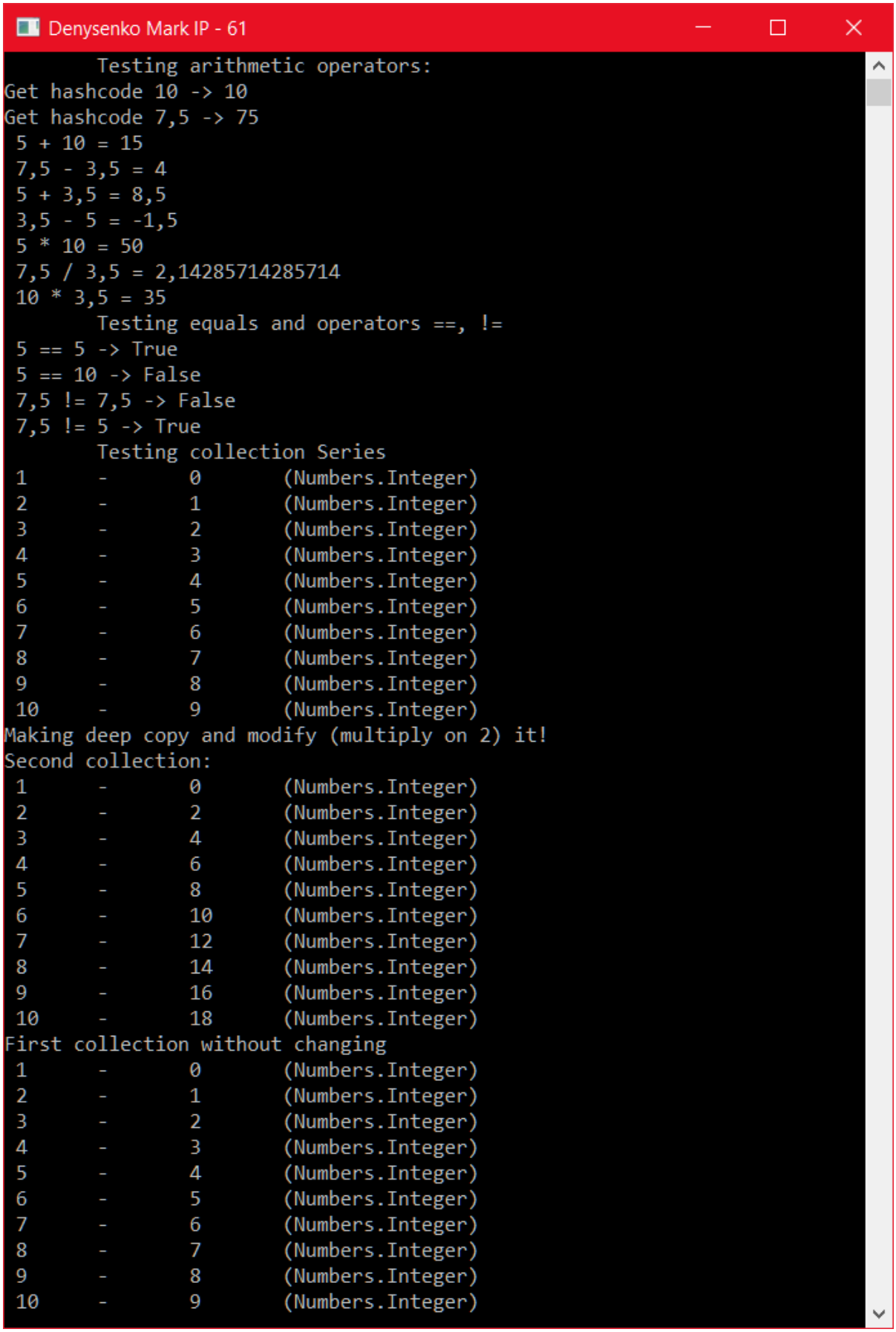
g) Реалізувати обробку помилок, при цьому необхідно перевизначити за допомогою наслідування наступні події: - StackOverflowException - ArrayTypeMismatchException - DivideByZeroException - IndexOutOfRangeException - InvalidCastException - OutOfMemoryException - OverflowException

h) Підготувати демонстраційний приклад, в котрому будуть використані всі розроблені методи

i) Підготувати звіт з результатами виконаної роботи.

Создать абстрактный класс Number с виртуальными методами, реализующими арифметические операции. На его основе реализовать классы Integer и Real. Создать класс Series (набор), содержащий массив/параметризованную коллекцию объектов этих классов в динамической памяти. Предусмотреть возможность вывода характеристик объектов списка.

Робота програми



Код програми:

1) Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Numbers
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Title = "Denysenko Mark IP - 61";

            ArithmeticTest();

            ColectionTest();

            StackOverflowTest();

            Console.ReadKey();
        }

        #region Test fucntions

        static void ArithmeticTest()
        {
            Console.WriteLine("\tTesting arithmetic operators:");
            Integer n1 = new Integer(5);
            Integer n2 = new Integer(10);

            Real n3 = new Real(7.5);
            Real n4 = new Real(3.5);

            Console.WriteLine($"Get hashcode {n2} -> {n2.GetHashCode()}");
            Console.WriteLine($"Get hashcode {n3} -> {n3.GetHashCode()}");

            Console.WriteLine($" {n1} + {n2} = {n1 + n2}");
            Console.WriteLine($" {n3} - {n4} = {n3 - n4}");
            Console.WriteLine($" {n1} + {n4} = {n1 + n4}");
            Console.WriteLine($" {n4} - {n1} = {n4 - n1}");

            Console.WriteLine($" {n1} * {n2} = {n1 * n2}");
            Console.WriteLine($" {n3} / {n4} = {n3 / n4}");
            Console.WriteLine($" {n2} * {n4} = {n2 * n4}");

            Console.WriteLine("\tTesting equals and operators ==, !=");
            Integer n11 = n1.Clone() as Integer;
            Real n33 = n3.Clone() as Real;

            Console.WriteLine($" {n1} == {n11} -> {n1 == n11}");
            Console.WriteLine($" {n1} == {n2} -> {n1 == n2}");
            Console.WriteLine($" {n3} != {n33} -> {n3 != n33}");
            Console.WriteLine($" {n3} != {n1} -> {n3 != n1}");
        }

        static void ColectionTest()
        {
            Console.WriteLine("\tTesting collection Series");
            Series<Integer> testSeries = new Series<Integer>();

            for (int i = 0; i < 10; i++)
```

```
            {
                testSeries.Add(new Integer(i));
            }

            testSeries.ShowInfo();

            Console.WriteLine("Making deep copy and modify (multiply on 2) it!");

            Series<Integer> testSeries2 = testSeries.DeepCopy();

            for(int i = 0; i < testSeries2.Count; i++)
            {
                testSeries2[i].Number = testSeries2[i].Number * 2;
            }
            Console.WriteLine("Second collection: ");
            testSeries2.ShowInfo();

            Console.WriteLine("First collection without changing");
            testSeries.ShowInfo();

            Console.WriteLine("\tTesting Exceptions");
            try
            {
                testSeries.Delete(11);
            }
            catch(Exceptions.MyIndexOutOfRangeException ex)
            {
                Console.WriteLine("Testing index out of range! " + ex.Message);
            }
            catch(Exception ex)
            {
                Console.WriteLine("Some another exeption! " + ex.Message);
            }
        }

        static void StackOverflowTest()
        {
            try
            {
                Recursive();
            }
            catch (Exceptions.MyStackOverflowException ex)
            {
                Console.WriteLine(ex.Message);
            }
        }

        static void Recursive(int counter = 0)
        {
            counter++;
            if (counter < 100)                      // deep for recursion
            {
                Recursive(counter);
            }
            else
            {
                throw new Exceptions.MyStackOverflowException("My stackoverflow test exception!
Deep: " + counter);
            }
        }

        #endregion
    }
}
```

2) Number.cs

```
using System;

namespace Numbers
{
    abstract class Number : ICloneable
    {
        public abstract int ToInt();
        public abstract double ToDouble();

        public object Clone()
        {
            return this.MemberwiseClone();
        }

        #region Basic functions (abstract)

        public abstract Number Add(Integer n);
        public abstract Number Add(Real n);
        public abstract Number Subtraction(Integer n);
        public abstract Number Subtraction(Real n);
        public abstract Number Multiply(Integer n);
        public abstract Number Multiply(Real n);
        public abstract Number Division(Integer n);
        public abstract Number Division(Real n);

        #endregion

        public static bool operator == (Number n1, Number n2)
        {
            return n1.Equals(n2);
        }

        public static bool operator != (Number n1, Number n2)
        {
            return !(n1.Equals(n2));
        }
    }
}
```

3) Integer.cs

```
using System;

namespace Numbers
{
    class Integer : Number
    {
        private int number;
        public int Number
        {
            set
            {
                number = value;
            }
            get
            {
                return number;
            }
        }

        public Integer(int num)
        {
            number = num;
        }
```

```
#region Overriding Object methods

public override int GetHashCode()
{
    return number;
}

public override bool Equals(object obj)
{
    if(obj == null || obj.GetType() != GetType())
    {
        return false;
    }

    return number == (obj as Integer).number;
}

public override string ToString()
{
    return number.ToString();
}

#endregion

public override int ToInt()
{
    return number;
}

public override double ToDouble()
{
    return (double)number;
}

#region Realization basic functions

public override Number Add(Integer n)
{
    number += n.number;
    return new Integer(number);
}
public override Number Add(Real n)
{
    number += n.ToInt();
    return new Integer(number);
}
public override Number Subtraction(Integer n)
{
    number -= n.number;
    return new Integer(number);
}
public override Number Subtraction(Real n)
{
    number -= n.ToInt();
    return new Integer(number);
}
public override Number Multiply(Integer n)
{
    number *= n.number;
    return new Integer(number);
}
public override Number Multiply(Real n)
{
    number *= n.ToInt();
```

```
            return new Integer(number);
        }
        public override Number Division(Integer n)
        {
            number /= n.number;
            return new Integer(number);
        }
        public override Number Division(Real n)
        {
            number /= n.ToInt();
            return new Integer(number);
        }

        #endregion

        #region Operators

        public static Integer operator +(Integer n1, Integer n2)
        {
            return new Integer(n1.Number + n2.Number);
        }
        public static Real operator +(Integer n1, Real n2)
        {
            return new Real(n1.number + n2.Number);
        }
        public static Integer operator -(Integer n1, Integer n2)
        {
            return new Integer(n1.Number - n2.Number);
        }
        public static Real operator -(Integer n1, Real n2)
        {
            return new Real(n1.Number - n2.Number);
        }
        public static Integer operator *(Integer n1, Integer n2)
        {
            return new Integer(n1.Number * n2.Number);
        }
        public static Real operator *(Integer n1, Real n2)
        {
            return new Real(n1.Number * n2.Number);
        }
        public static Integer operator /(Integer n1, Integer n2)
        {
            return new Integer(n1.Number / n2.Number);
        }
        public static Real operator /(Integer n1, Real n2)
        {
            return new Real(n1.Number / n2.Number);
        }

        #endregion
    }
}
```

4) Real.cs

```
using System;

namespace Numbers
{
    class Real : Number
    {
        private double number;
        private const double eps = 0.000001;

        public double Number
```

```
        {
            set
            {
                number = value;
            }
            get
            {
                return number;
            }
        }

        public Real(double num)
        {
            number = num;
        }

        #region Overriding Object functions

        public override int GetHashCode()
        {

//number.ToString().Split(CultureInfo.CurrentCulture.NumberFormat.NumberDecimalSeparator.ToArray
<char>());
            return
Convert.ToInt32(number.ToString().Replace(CultureInfo.CurrentCulture.NumberFormat.NumberDecimalS
eparator, string.Empty));
        }

        public override bool Equals(object obj)
        {
            if (obj == null || obj.GetType() != GetType())
            {
                return false;
            }

            return this.GetHashCode() == (obj as Real).GetHashCode();
        }

        public override string ToString()
        {
            return number.ToString();
        }

        #endregion

        public override int ToInt()
        {
            return (int)number;
        }

        public override double ToDouble()
        {
            return number;
        }

        #region Realization of basics functions

        public override Number Add(Integer n)
        {
            number += n.Number;
            return new Real(number);
        }

        public override Number Add(Real n)
        {
```

```
        number += n.Number;
        return new Real(number);
    }

    public override Number Subtraction(Integer n)
    {
        number -= n.Number;
        return new Real(number);
    }

    public override Number Subtraction(Real n)
    {
        number -= n.Number;
        return new Real(number);
    }

    public override Number Multiply(Integer n)
    {
        number *= n.Number;
        return new Real(number);
    }

    public override Number Multiply(Real n)
    {
        number *= n.Number;
        return new Real(number);
    }

    public override Number Division(Integer n)
    {
        number /= n.Number;
        return new Real(number);
    }

    public override Number Division(Real n)
    {
        number /= n.Number;
        return new Real(number);
    }

    #endregion

    #region Operators

    public static Real operator + (Real n1, Real n2)
    {
        return new Real(n1.Number + n2.Number);
    }
    public static Real operator +(Real n1, Integer n2)
    {
        return new Real(n1.Number + n2.Number);
    }

    public static Real operator - (Real n1, Real n2)
    {
        return new Real(n1.Number - n2.Number);
    }
    public static Real operator -(Real n1, Integer n2)
    {
        return new Real(n1.Number - n2.Number);
    }

    public static Real operator * (Real n1, Real n2)
    {
        return new Real(n1.Number * n2.Number);
```

```
        }
        public static Real operator *(Real n1, Integer n2)
        {
            return new Real(n1.Number * n2.Number);
        }

        public static Real operator / (Real n1, Real n2)
        {
            return new Real(n1.Number / n2.Number);
        }
        public static Real operator /(Real n1, Integer n2)
        {
            return new Real(n1.Number / n2.Number);
        }

        #endregion
    }
}
```

5) Series.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Numbers
{
    class Series<T> where T : Number
    {
        private List<T> numbers;

        public int Count
        {
            get
            {
                return numbers.Count;
            }
        }

        public Series()
        {
            numbers = new List<T>();
        }

        public void Add(T element)
        {
            numbers.Add(element);
        }

        public void Delete(int index)
        {
            if (index > 0 && index < numbers.Count)
            {
                numbers.RemoveAt(index);
            }
            else
            {
                throw new Exceptions.MyIndexOutOfRangeException("Invalid index for deleting!");
            }
        }

        public Series<T> DeepCopy()
        {
            Series<T> clone = new Series<T>();
```

```
            foreach(var item in numbers)
            {
                clone.Add((T)item.Clone());
            }

            return clone;
        }

        public T this[int index]
        {
            get
            {
                if (index >= 0 && index < numbers.Count)
                {
                    return numbers[index];
                }
                else
                {
                    throw new Exceptions.MyIndexOutOfRangeException("Invalid index for
indexator");
                }
            }
            set
            {
                if (index > 0 && index < numbers.Count)
                {
                    if (numbers[index].GetType() == value.GetType())
                    {
                        numbers[index] = value;
                    }
                    else
                    {
                        throw new Exceptions.MyInvalidCastException("Inccorect type to
Series!");
                    }
                }
                else
                {
                    throw new Exceptions.MyIndexOutOfRangeException("Invalid index for
indexator");
                }
            }
        }

        public void ShowInfo()
        {
            int counter = 1;
            foreach(var item in numbers)
            {
                Console.WriteLine($" {counter}\t-\t{item.ToString()}\t({item.GetType()})");
                counter++;
            }
        }
    }
}
```