**EXPERIMENT-1**

**AIM:WRITE A PROGRAM TO IMPLEMENT SIMPLE CALCULATOR.**

**DESCRIPTION:**

1.Download the skeleton code from cloned repository. The skeleton repository contains one or more projects for inserting the target code and unit tests for task self-checking before submition in Autocode

2.Open downloaded solution with Visual Studio

3.Change the skeleton code according to the description and requirements of the task

4.Run downloaded unit tests in Visual Studio until all test will be passed

5.Remember to remove all comment lines with "TODO" becouse Sonar will cause an issue when you initiate the task check in AutoCode

6.Put changed solution into remote repository and initiate checking on Autocode

**CODE:**

using System;

namespace SimpleCalculator

{

public class SimpleCalculator

{

public static int Add(int a, int b)

{

return a + b;

}

public static int Sub(int a, int b)

{

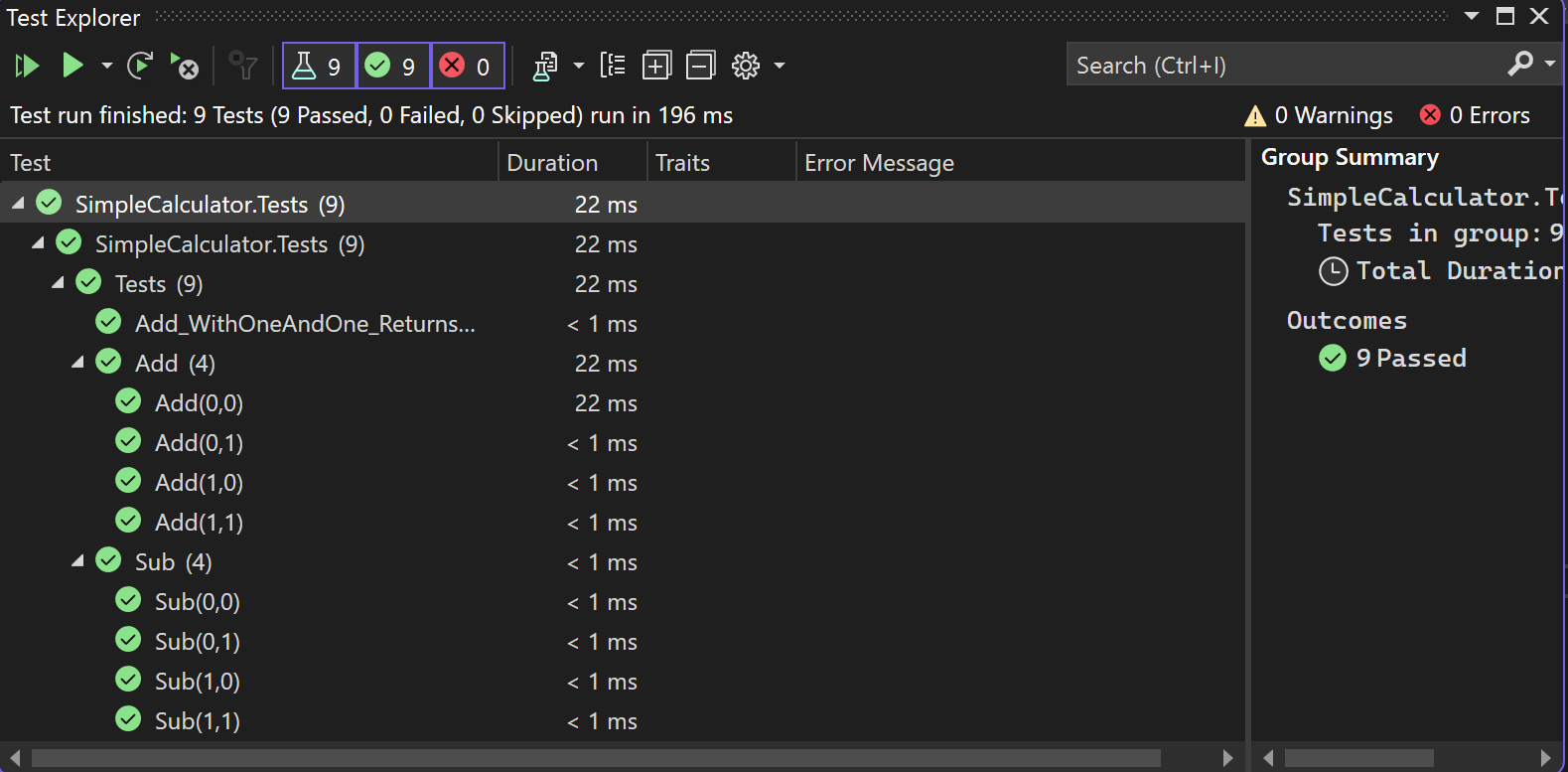
return a - b;

}

}

}

**OUTPUT:**



**EXPERIMENT-2**

**AIM:WRITE A PROGRAM TO IMPLEMENT CONDITION STATEMENTS.**

**DESCRIPTION:**

TASK1

For a given integer *n* calculate the value which is equal to:

1.squared number, if its value is strictly positive;

2.modulus of a number, if its value is strictly negative;

3.zero, if the integer n is zero.

Task 2

4.Find the maximum integer, that can be obtained by numbers of an arbitrary three-digit positive integer *n* permutation (100<=n<=999).

**CODE:**

using System;

using System.Collections.Generic;

namespace Condition

{

public static class Condition

{

public static int Task1(int n)

{

if (n > 0)

{

return n \* n;

}

else if (n < 0)

{

return Math.Abs(n);

}

else

{

return 0;

}

}

public static int Task2(int n)

{

string numberString = n.ToString();

char[] charArray = numberString.ToCharArray();

Array.Sort(charArray);

Array.Reverse(charArray);

int result = int.Parse(new string(charArray));

return result;

}

}

}

**OUTPUT:**

**EXPERIMENT-3**

**AIM:WRITE A PROGRAM TO IMPLEMENT LOOP TASKS.**

**DESCRIPTION:**

Task 1

For a positive integer *n* calculate the *result* value, which is equal to the sum of the odd numbers in n

Task 2

For a positive integer *n* calculate the result value, which is equal to the sum of the “1” in the binary representation of *n*.

Task3

For a positive integer *n*, calculate the result value equal to the sum of the first *n* Fibonacci numbers Note: Fibonacci numbers are a series of numbers in which each next number is equal to the sum of the two preceding ones: 0, 1, 1, 2, 3, 5, 8, 13... (F0=0, F1=F2=1, then F(n)=F(n-1)+F(n-2) for n>2)

**CODE:**

using System;

namespace LoopTasks

{

public static class LoopTasks

{

/// Task 1

public static int SumOfOddDigits(int n)

{

//this method should return the sum of the odd digits of n.

// TODO: delete code line below, write down your solution

int count = 0;

while (n > 0)

{

count += n & 1; // If the least significant bit is set (i.e., 1), increment the count

n >>= 1; // Right shift n by 1 to check the next bit

}

return count;

}

/// Task 2

public static int NumberOfUnitsInBinaryRecord(int n)

{

//this method should return the number of units in the binary notation of n.

// TODO: delete code line below, write down your solution

int count = 0;

while (n > 0)

{

count += n & 1; // If the least significant bit is set (i.e., 1), increment the count

n >>= 1; // Right shift n by 1 to check the next bit

}

return count;

}

/// Task 3

public static int SumOfFirstNFibonacciNumbers(int n)

{

//this method should return the sum of the first n Fibonacci numbers.

// TODO: delete code line below, write down your solution

if (n <= 0)

{

throw new ArgumentException("n must be a positive integer.");

}

if (n == 1)

{

return 0; // The first Fibonacci number is 0

}

int firstFibonacci = 0;

int secondFibonacci = 1;

int sum = firstFibonacci + secondFibonacci;

for (int i = 3; i <= n; i++)

{

int nextFibonacci = firstFibonacci + secondFibonacci;

sum += nextFibonacci;

firstFibonacci = secondFibonacci;

secondFibonacci = nextFibonacci;

}

return sum;

}

}

}

**OUTPUT:**

**EXPERIMENT-4**

**AIM:WRITE A PROGRAM TO IMPLEMENT FUNCTIONS.**

**DESCRIPTION:**

Task 1

Create function **IsSorted**, determining whether a given **array** of integer values of arbitrary length is sorted in a given **order** (the order is set up by enum value **SortOrder**). Array and sort order are passed by parameters. Function does not change the array

Task 2

Create function **Transform**, replacing the value of each element of an integer array with the sum of this element value and its index, only if the given array is sorted in the given order (the order is set up by enum value SortOrder). Array and sort order are passed by parameters. To check, if the array is sorted, the function IsSorted from the Task 1 is called.

Task 3

Create function MultArithmeticElements, which determines the multiplication of a given number of first n elements of arithmetic progression of real numbers with a given initial element of progression a(1) and progression step t. a(n) is calculated by the formula a(n+1) = a(n) + t.

Task4

Create function SumGeometricElements, determining the sum of the first elements of a decreasing geometric progression of real numbers with a given initial element of a progression a(1)) and a given progression step t, while the last element must be greater than a given alim. an is calculated by the formula a(n+1) = a(n) \* t, 0<t<1.

**CODE:**

using System;

namespace Function

{

public enum SortOrder { Ascending, Descending }

public static class Function

{

public static bool IsSorted(int[] array, SortOrder order)

{

if (order == SortOrder.Ascending)

{

for (int i = 1; i < array.Length; i++)

{

if (array[i] < array[i - 1])

{

return false;

}

}

}

else

{

for (int i = 1; i < array.Length; i++)

{

if (array[i] > array[i - 1])

{

return false;

}

}

}

return true;

}

public static void Transform(int[] array, SortOrder order)

{

if (IsSorted(array, order))

{

for (int i = 0; i < array.Length; i++)

{

array[i] += i;

}

}

}

public static double MultArithmeticElements(double a, double t, int n)

{

double result = 1.0;

double currentElement = a;

for (int i = 0; i < n; i++)

{

result \*= currentElement;

currentElement += t;

}

return result;

}

public static double SumGeometricElements(double a, double t, double alim)

{

double sum = 0.0;

double currentElement = a;

while (currentElement > alim)

{

sum += currentElement;

currentElement \*= t;

}

return sum;

}

}

}

**OUTPUT:**

**EXPERIMENT-5**

**AIM:WRITE A PROGRAM TO IMPLEMENT CLASS CODE.**

**DESCRIPTION:**

Task 1

Develop **Rectangle** and **ArrayRectangles** with a predefined functionality.

On a Low level it is obligatory:

To develop **Rectangle** class with following content:

* 2 closed real fields **sideA** and **sideB** (sides А and В of the rectangle)
* Constructor with two real parameters **a** and **b** (parameters specify rectangle sides)
* Constructor with a real parameter **a** (parameter specify side А of a rectangle, side B is always equal to 5)
* Constructor without parameters (side А of a rectangle equals to 4, side В - 3)
* Method **GetSideA**, returning value of the side А
* Method **GetSideВ**, returning value of the side В
* Method **Area**, calculating and returning the area value
* Method **Perimeter**, calculating and returning the perimeter value
* Method **IsSquare**, checking whether current rectangle is shape square or not. Returns true if the shape is square and false in another case.
* Method **ReplaceSides**, swapping rectangle sides

On Advanced level also needed:

Complete Level Low Assignment

Develop class **ArrayRectangles**, in which declare:

* Private field **rectangle\_array** - array of rectangles
* Constructor creating an empty array of rectangles with length n
* Constructor that receives an arbitrary amount of objects of type **Rectangle** or an array of objects of type **Rectangle**.
* Method **AddRectangle** that adds a rectangle of type Rectangle to the array on the nearest free place and returning true, or returning false, if there is no free space in the array
* Method **NumberMaxArea**, that returns order number (index) of the rectangle with the maximum area value (numeration starts from zero)
* Method **NumberMinPerimeter**, that returns order number(index) of the rectangle with the minimum area value (numeration starts from zero)
* Method **NumberSquare**, that returns the number of squares in the array of rectangles

**CODE:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Security.Claims;

namespace Class

{

public class Rectangle

{

private double sideA;

private double sideB;

public Rectangle(double a, double b)

{

sideA = a;

sideB = b;

}

public Rectangle(double a)

{

sideA = a;

sideB = 5;

}

public Rectangle()

{

sideA = 4;

sideB = 3;

}

public double GetSideA() => sideA;

public double GetSideB() => sideB;

public double Area() => sideA \* sideB;

public double Perimeter() => 2 \* (sideA + sideB);

public bool IsSquare() => sideA == sideB;

public void ReplaceSides()

{

double temp = sideA;

sideA = sideB;

sideB = temp;

}

}

public class ArrayRectangles

{

private Rectangle[] rectangle\_array;

public ArrayRectangles(int n)

{

rectangle\_array = new Rectangle[n];

}

public ArrayRectangles(params Rectangle[] rectangles)

{

rectangle\_array = rectangles;

}

public bool AddRectangle(Rectangle rectangle)

{

int index = Array.IndexOf(rectangle\_array, null);

if (index != -1)

{

rectangle\_array[index] = rectangle;

return true;

}

return false;

}

public int NumberMaxArea()

{

double maxArea = double.MinValue;

int maxIndex = -1;

for (int i = 0; i < rectangle\_array.Length; i++)

{

if (rectangle\_array[i] != null && rectangle\_array[i].Area() > maxArea)

{

maxArea = rectangle\_array[i].Area();

maxIndex = i;

}

}

return maxIndex;

}

public int NumberMinPerimeter()

{

double minPerimeter = double.MaxValue;

int minIndex = -1;

for (int i = 0; i < rectangle\_array.Length; i++)

{

if (rectangle\_array[i] != null && rectangle\_array[i].Perimeter() < minPerimeter)

{

minPerimeter = rectangle\_array[i].Perimeter();

minIndex = i;

}

}

return minIndex;

}

public int NumberSquare()

{

int count = 0;

foreach (var rectangle in rectangle\_array)

{

if (rectangle != null && rectangle.IsSquare())

{

count++;

}

}

return count;

}

}

}

**OUTPUT:**

**EXPERIMENT-6**

**AIM:WRITE A PROGRAM TO IMPLEMENT ARRAYS.**

**DESCRIPTION:**

Task 1

In a given array of integers nums swap values of the first and the last array elements, the second and the penultimate etc., if the two exchanged values are even

Task 2

In a given array of integers nums calculate integer result value, that is equal to the distance between the first and the last entry of the maximum value in the array.

Task 3

In a predetermined two-dimensional integer array (square matrix) matrix insert 0 into elements to the left side of the main diagonal, and 1 into elements to the right side of the diagonal.

**CODE:**

using System;

namespace ArrayObject

{

public static class ArrayTasks

{

/// Task 1

public static void ChangeElementsInArray(int[] nums)

{

for (int i = 0; i < nums.Length / 2; i++)

{

if (nums[i] % 2 == 0 && nums[nums.Length - 1 - i] % 2 == 0)

{

int temp = nums[i];

nums[i] = nums[nums.Length - 1 - i];

nums[nums.Length - 1 - i] = temp;

}

}

}

/// Task 2

public static int DistanceBetweenFirstAndLastOccurrenceOfMaxValue(int[] nums)

{

int max = int.MinValue;

int maxIndex = -1;

for (int i = 0; i < nums.Length; i++)

{

if (nums[i] > max)

{

max = nums[i];

maxIndex = i;

}

}

if (maxIndex != -1)

{

return Math.Abs(maxIndex - Array.LastIndexOf(nums, max));

}

return 0;

}

/// Task 3

public static void ChangeMatrixDiagonally(int[,] matrix)

{

int n = matrix.GetLength(0);

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (j < i)

{

matrix[i, j] = 0;

}

else if (j > i)

{

matrix[i, j] = 1;

}

}

}

}

}

}

**OUTPUT:**