**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

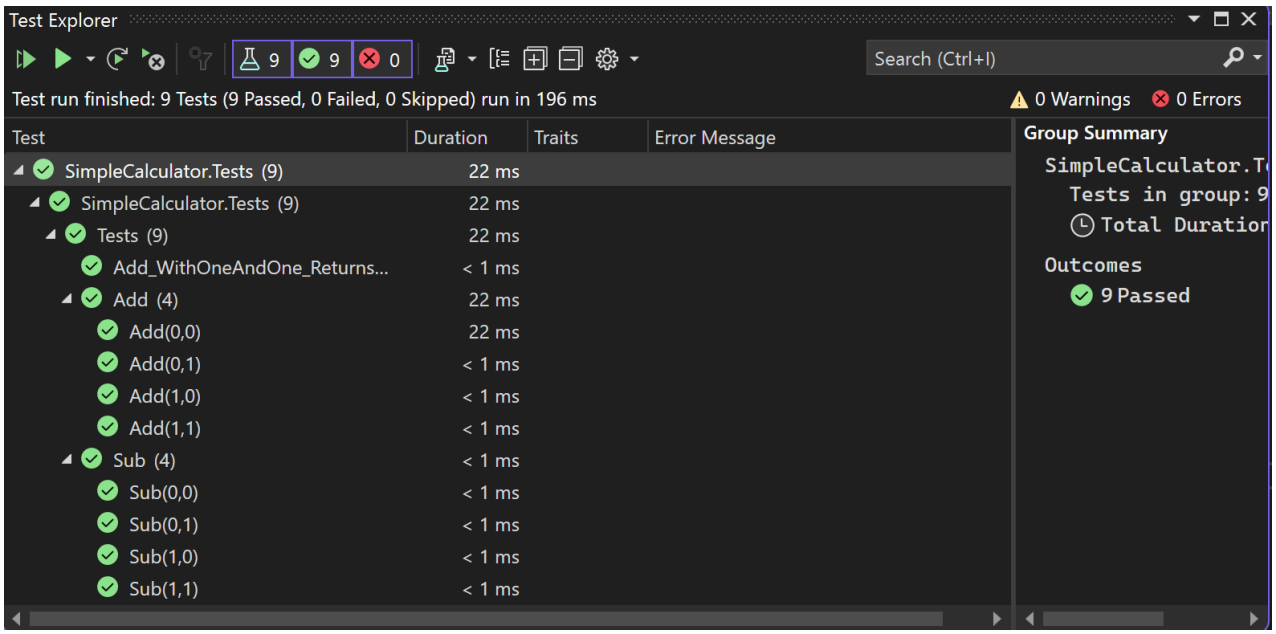
5.Remember to remove all comment lines with "TODO" becouse Sonar will cause an issue when you initiate the task check in AutoCode

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:**

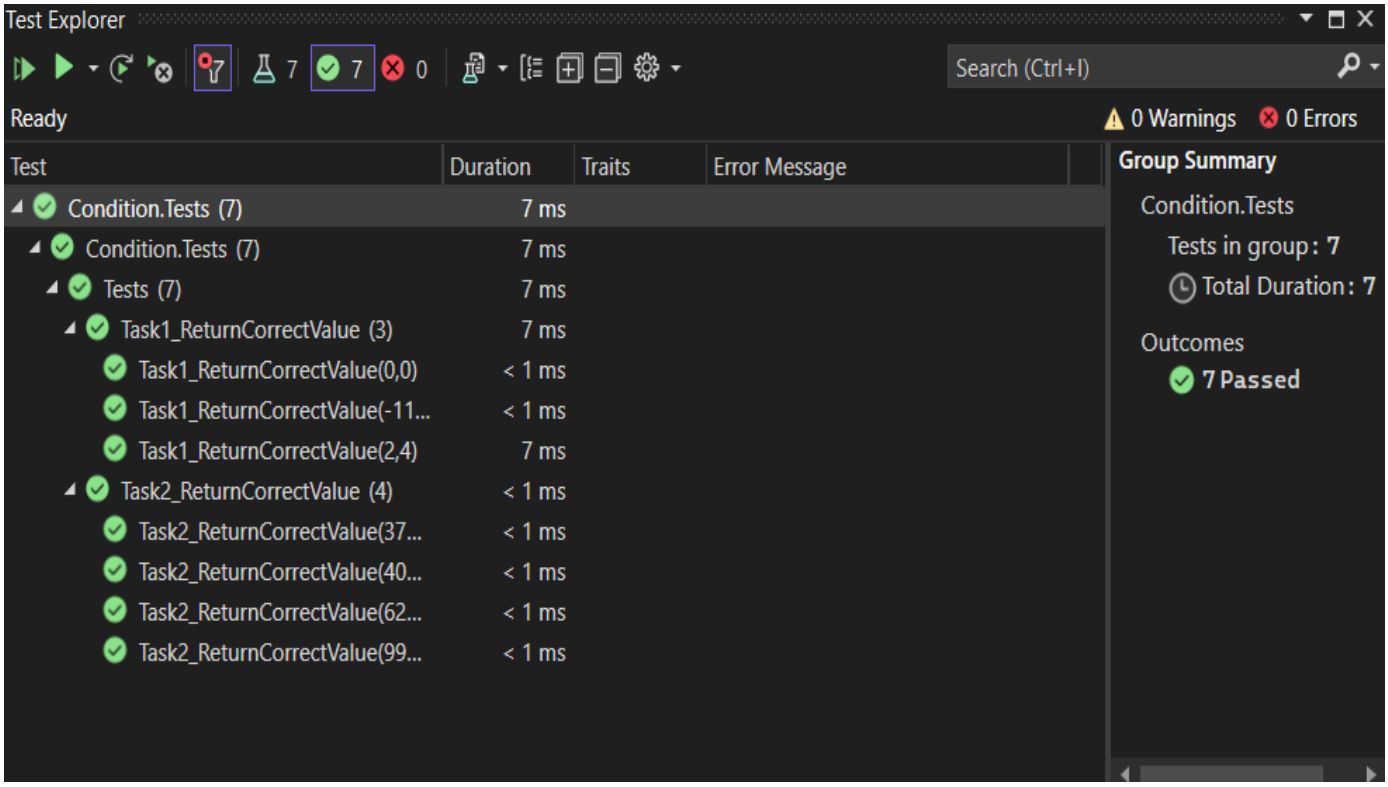
TASK1
For a given integer *n* calculate the value which is equal to:
1.squared number, if its value is strictly positive;
2.modulus of a number, if its value is strictly negative;
3.zero, if the integer n is zero.

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:**


Test Explorer
7
7
0
Search (Ctrl+I)
Ready
0 Warnings
0 Errors
Test
Duration
Traits
Error Message
Condition.Tests (7) 7 ms
Condition.Tests (7) 7 ms
Tests (7) 7 ms
Task1_ReturnCorrectValue (3) 7 ms
Task1_ReturnCorrectValue(0,0) < 1 ms
Task1_ReturnCorrectValue(-11... < 1 ms
Task1_ReturnCorrectValue(2,4) 7 ms
Task2_ReturnCorrectValue (4) < 1 ms
Task2_ReturnCorrectValue(37... < 1 ms
Task2_ReturnCorrectValue(40... < 1 ms
Task2_ReturnCorrectValue(62... < 1 ms
Task2_ReturnCorrectValue(99... < 1 ms
Group Summary
Condition.Tests
Tests in group: 7
Total Duration: 7
Outcomes
7 Passed

**EXPERIMENT-3**

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

**DESCRIPTION:**

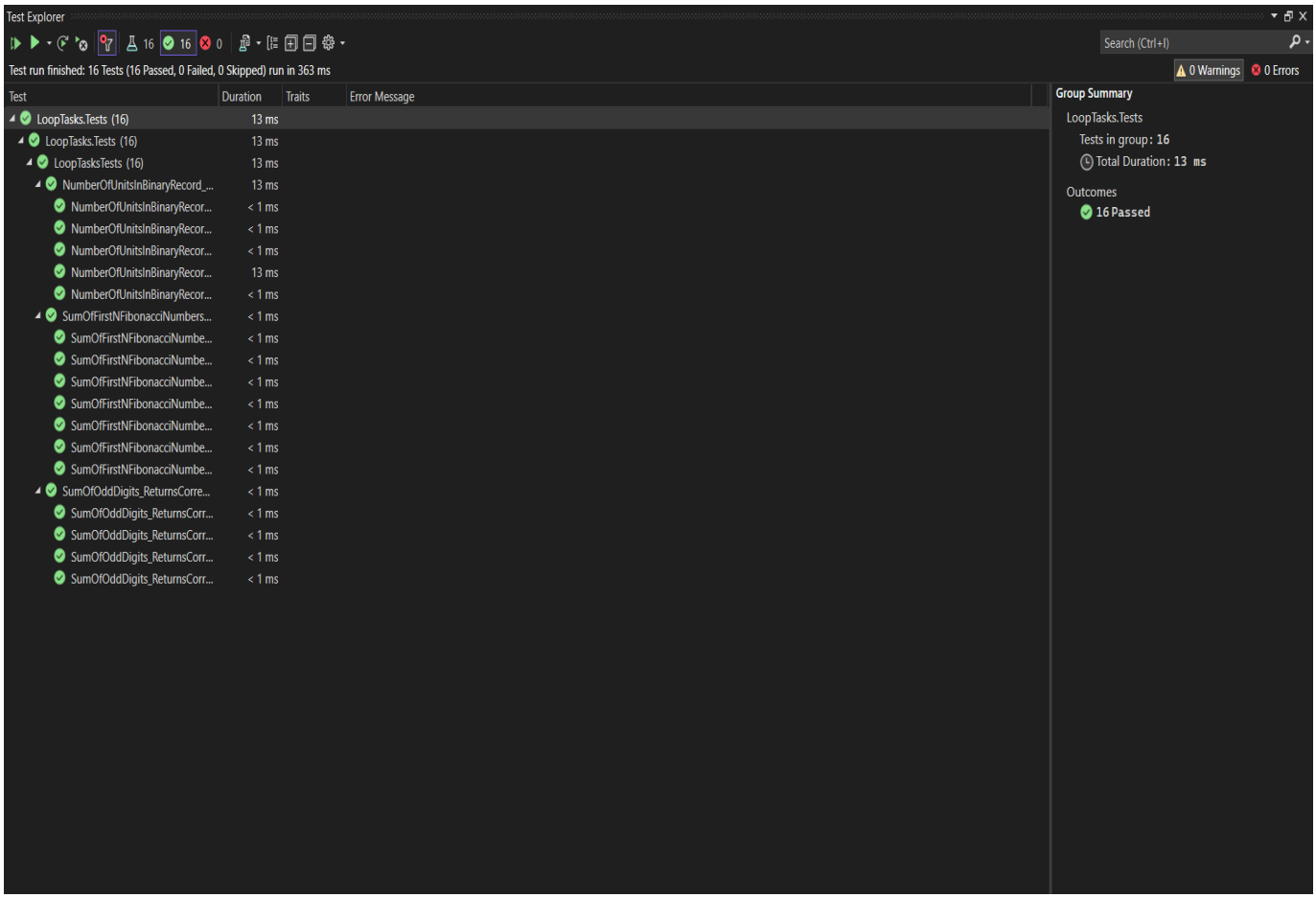
Task 1
For a positive integer *n* calculate the *result* value, which is equal to the sum of the odd numbers in n
Task 2
For a positive integer *n* calculate the result value, which is equal to the sum of the "1" in the binary representation of *n*.
Task3
For a positive integer *n*, calculate the result 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)
      {
         int sumOfOddDigits = 0;
         string numStr = n.ToString();
         foreach (char digit in numStr)
         {
            int currentDigit = digit - '0';
            if (currentDigit % 2 == 1)
            {
               sumOfOddDigits += currentDigit;
            }
         }
         return sumOfOddDigits;

      /// 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:**

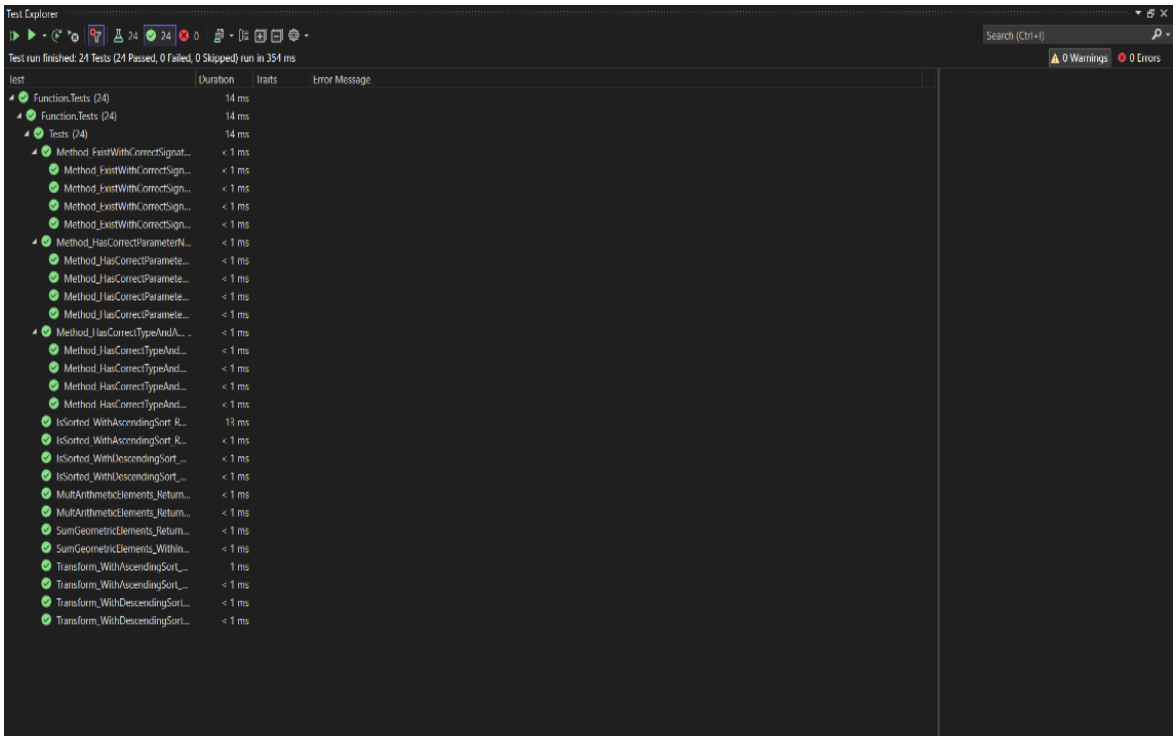
Task 1
Create function IsSorted, determining whether a given array of integer values of arbitrary length is sorted in a given order (the order is set up by enum value SortOrder). Array and sort order are passed by parameters. Function does not change the array
Task 2
Create function Transform, replacing 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 A and B of the rectangle)
- Constructor with two real parameters a and b (parameters specify rectangle sides)
- Constructor with a real parameter a (parameter specify side A of a rectangle, side B is always equal to 5)
- Constructor without parameters (side A of a rectangle equals to 4, side B - 3)
- Method GetSideA, returning value of the side A
- Method GetSideB, returning value of the side B
- 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

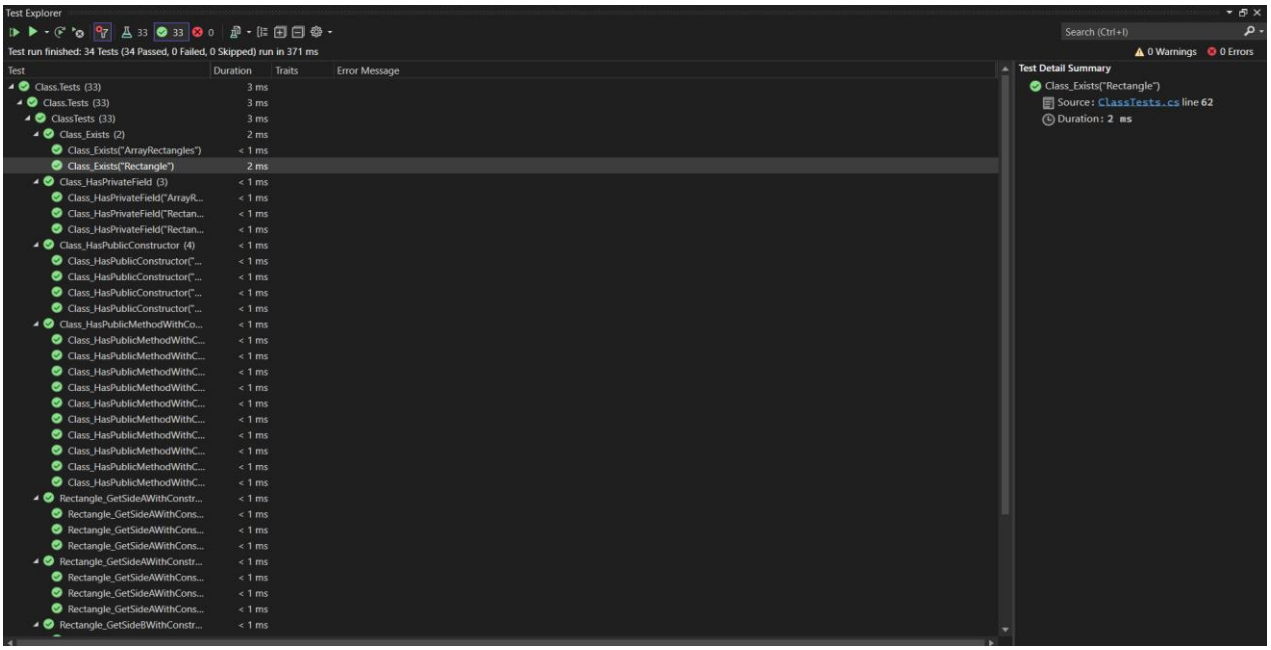
**CODE:**

```
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.

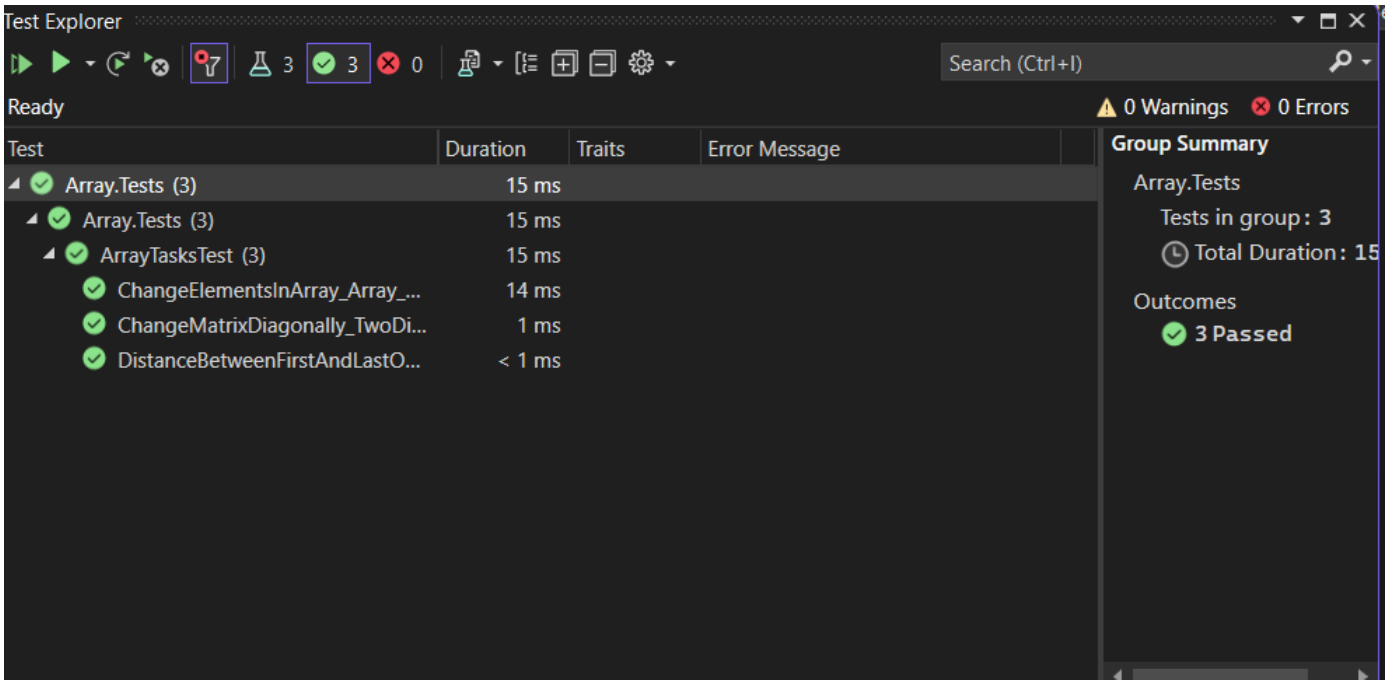
Task 3
In a predetermined two-dimensional integer array (square matrix) *matrix* insert 0 into elements to the left side of the main diagonal, and 1 into elements to the right side of the diagonal.

**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:**



Interfaces code :

BaseDeposit.cs

```
using System;

namespace Interfaces
{
   public class BaseDeposit : Deposit
   {
      public BaseDeposit(decimal amount, int period) : base(amount, period) { }

      public override decimal Income()
      {
         decimal income = Amount;
         for (int i = 0; i < Period; i++)
         {
            income += income * 0.05m;
         }
         return Math.Round(income - Amount, 2);
      }
   }
}
```

Client.cs

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Interfaces
{
   public class Client : IEnumerable<Deposit>
   {
      private Deposit[] deposits;

      public Client()
      {
         deposits = new Deposit[10];
      }

      public bool AddDeposit(Deposit deposit)
      {
         for (int i = 0; i < deposits.Length; i++)
         {
            if (deposits[i] == null)
            {
               deposits[i] = deposit;
               return true;
            }
         }
         return false;
```

```
    }

    public decimal TotalIncome()
    {
        decimal totalIncome = 0;
        foreach (var deposit in deposits)
        {
            if (deposit != null)
            {
                totalIncome += deposit.Income();
            }
        }
        return totalIncome;
    }

    public decimal MaxIncome()
    {
        decimal maxIncome = 0;
        foreach (var deposit in deposits)
        {
            if (deposit != null)
            {
                maxIncome = Math.Max(maxIncome, deposit.Income());
            }
        }
        return maxIncome;
    }

    public decimal GetIncomeByNumber(int number)
    {
        if (number >= 1 && number <= deposits.Length)
        {
            var deposit = deposits[number - 1];
            if (deposit != null)
            {
                return deposit.Income();
            }
        }
        return 0;
    }

    public void SortDeposits()
    {
        deposits = deposits.Where(deposit => deposit != null)
                           .OrderByDescending(deposit => deposit.TotalAmount())
                           .ToArray();
    }

    public int CountPossibleToProlongDeposit()
    {
        return deposits.Count(deposit => deposit is IProlongable prolongable && prolongable.CanToProlong());
    }

    public IEnumerator<Deposit> GetEnumerator()
    {
        return ((IEnumerable<Deposit>)deposits).GetEnumerator();
    }
```

```
        IEnumerator IEnumerable.GetEnumerator()
        {
            return GetEnumerator();
        }
    }
}
```

Deposit.cs:

```
using System;

namespace Interfaces
{
    public abstract class Deposit : IComparable<Deposit>
    {
        public decimal Amount { get; }
        public int Period { get; }

        public Deposit(decimal depositAmount, int depositPeriod)
        {
            Amount = depositAmount;
            Period = depositPeriod;
        }

        public abstract decimal Income();

        public int CompareTo(Deposit other)
        {
            if (other == null) return 1;
            return TotalAmount().CompareTo(other.TotalAmount());
        }

        public decimal TotalAmount()
        {
            return Amount + Income();
        }
    }
}
```

IProlongable.cs:

```
using System;

namespace Interfaces
{
    public interface IProlongable
    {
        bool CanToProlong();
    }
}
```

LongDeposit.cs:

```
using System;

namespace Interfaces
{
   public class LongDeposit : Deposit, IProlongable
   {
      public LongDeposit(decimal amount, int period) : base(amount, period) { }

      public override decimal Income()
      {
         decimal income = Amount;
         for (int i = 0; i < Period; i++)
         {
            if (i >= 6)
            {
               income += income * 0.15m;
            }
         }
         return Math.Round(income - Amount, 2);
      }

      public bool CanToProlong()
      {
         return Period <= 36; // 3 years or less
      }
   }
}
```

SpecialDeposit.cs:

```
using System;

namespace Interfaces
{
   public class SpecialDeposit : Deposit, IProlongable
   {
      public SpecialDeposit(decimal amount, int period) : base(amount, period) { }

      public override decimal Income()
      {
         decimal income = Amount;
         for (int i = 0; i < Period; i++)
         {
            income += income * (i + 1) / 100.0m;
         }
         return Math.Round(income - Amount, 2);
      }

      public bool CanToProlong()
      {
```

```
        return Amount > 1000;
      }
   }
}
```

_____|||

Short description of the Aggregation task:

This task is about creating a model using c sharp for bank deposits. The application should have the following classes:

Deposit - an abstract class that represents a deposit account. It has properties for the amount and period of the deposit, as well as an abstract method for calculating the income from the deposit.
BaseDeposit, SpecialDeposit, and LongDeposit - inheritor classes of Deposit that implement different interest addition schemes.
Client - a class that represents a bank client. It has a list of deposits and methods for adding, calculating the total income from, and getting the income from a specific deposit.

Aggregation

Deposit.cs:

```
namespace Aggregation
{
   public abstract class Deposit
   {
      public decimal Amount { get; }
      public int Period { get; }

      public Deposit(decimal depositAmount, int depositPeriod)
      {
         Amount = depositAmount;
         Period = depositPeriod;
      }

      public abstract decimal Income();
   }
}
```

BaseDeposit.cs:

```
namespace Aggregation
{
   public class BaseDeposit : Deposit
   {
      public BaseDeposit(decimal amount, int period)
         : base(amount, period)
```

```
    {
    }

    public override decimal Income()
    {
      decimal income = 0;
      decimal currentAmount = Amount;

      for (int i = 0; i < Period; i++)
      {
        income += currentAmount * 0.05m;
        currentAmount += income;
      }

      return income;
    }
  }
}
```

SpecialDeposit.cs:

```
namespace Aggregation
{
  public class SpecialDeposit : Deposit
  {
    public SpecialDeposit(decimal amount, int period)
      : base(amount, period)
    {
    }

    public override decimal Income()
    {
      decimal income = 0;
      decimal currentAmount = Amount;

      for (int i = 0; i < Period; i++)
      {
        income += currentAmount * (i + 1) / 100.0m;
        currentAmount += income;
      }

      return income;
    }
  }
}
```

LongDeposit.cs:

```
namespace Aggregation
{
  public class LongDeposit : Deposit
  {
    public LongDeposit(decimal amount, int period)
      : base(amount, period)
```

```
        {
        }

        public override decimal Income()
        {
            decimal income = 0;
            decimal currentAmount = Amount;

            for (int i = 0; i < Period; i++)
            {
                if (i >= 6)
                {
                    income += currentAmount * 0.15m;
                }
                currentAmount += income;
            }

            return income;
        }
    }
}
```

Client.cs:

```
namespace Aggregation
{
    public class Client
    {
        private Deposit[] deposits;

        public Client()
        {
            deposits = new Deposit[10];
        }

        public bool AddDeposit(Deposit deposit)
        {
            for (int i = 0; i < deposits.Length; i++)
            {
                if (deposits[i] == null)
                {
                    deposits[i] = deposit;
                    return true;
                }
            }
            return false;
        }

        public decimal TotalIncome()
        {
            decimal totalIncome = 0;
            foreach (Deposit deposit in deposits)
            {
                if (deposit != null)
                {
                    totalIncome += deposit.Income();
```

```
            }
        }
        return totalIncome;
    }

    public decimal MaxIncome()
    {
        decimal maxIncome = 0;
        foreach (Deposit deposit in deposits)
        {
            if (deposit != null)
            {
                decimal income = deposit.Income();
                if (income > maxIncome)
                {
                    maxIncome = income;
                }
            }
        }
        return maxIncome;
    }

    public decimal GetIncomeByNumber(int number)
    {
        if (number >= 1 && number <= deposits.Length && deposits[number - 1] != null)
        {
            return deposits[number - 1].Income();
        }
        return 0;
    }
  }
}
```

---

Inheritance

Company.cs:

```
using System;

namespace InheritanceTask
{
  public class Company
  {
    private Employee[] employees;

    public Company(Employee[] employees)
    {
      this.employees = employees;
    }

    public void GiveEverybodyBonus(decimal companyBonus)
    {
```

```
        foreach (Employee employee in employees)
        {
            employee.SetBonus(companyBonus);
        }
    }

    public decimal TotalToPay()
    {
        decimal totalSalary = 0;
        foreach (Employee employee in employees)
        {
            totalSalary += employee.ToPay();
        }
        return totalSalary;
    }

    public string NameMaxSalary()
    {
        decimal maxSalary = 0;
        string employeeName = "";

        foreach (Employee employee in employees)
        {
            decimal salary = employee.ToPay();
            if (salary > maxSalary)
            {
                maxSalary = salary;
                employeeName = employee.Name;
            }
        }

        return employeeName;
    }
  }
}
```

Employee.cs:

```
using System;

namespace InheritanceTask
{
  public class Employee
  {
    private string name;
    private decimal salary;
    private decimal bonus;

    public string Name
    {
      get { return name; }
    }

    public decimal Salary
    {
```

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

    public Employee(string name, decimal salary)
    {
        this.name = name;
        this.salary = salary;
    }

    public virtual void SetBonus(decimal bonus)
    {
        this.bonus = bonus;
    }

    public decimal ToPay()
    {
        return salary + bonus;
    }
  }
}
```

Manager.cs:

```
using System;

namespace InheritanceTask
{
  public class Manager : Employee
  {
    private int quantity;

    public Manager(string name, decimal salary, int clientAmount)
      : base(name, salary)
    {
      quantity = clientAmount;
    }

    public override void SetBonus(decimal bonus)
    {
      if (quantity > 150)
      {
        base.SetBonus(bonus + 1000);
      }
      else if (quantity > 100)
      {
        base.SetBonus(bonus + 500);
      }
      else
      {
        base.SetBonus(bonus);
      }
    }
  }
}
```

SalesPerson.cs:

```
using System;

namespace InheritanceTask
{
   public class SalesPerson : Employee
   {
      private int percent;

      public SalesPerson(string name, decimal salary, int percent)
         : base(name, salary)
      {
         this.percent = percent;
      }

      public override void SetBonus(decimal bonus)
      {
         if (percent > 200)
         {
            base.SetBonus(bonus * 3);
         }
         else if (percent > 100)
         {
            base.SetBonus(bonus * 2);
         }
         else
         {
            base.SetBonus(bonus);
         }
      }
   }
}
```