TWOCALC AND THREECALC

1) Add a **`class`** called **`BaseCalc`** with the **`virtual int Calculate()`** method. This method, by default, throws a **`NotImplementedException.`**

2) Add another **class** called **`TwoCalc`** deriving the **`BaseCalc` class**.

   a) **`TwoCalc`** has a **constructor** that initializes two **integer fields**, ***_a*** and ***_b***, which are both **private** and **read-only**.

   b) The **`Calculate() overridden`** method will calculate and return $a + 2 \times b$. So, if you have {1, 2}, the result of this method is 1+2 $\times 2 = 5$.

   c) The **`ToString()`** method will only indicate both numbers. **`ToString()`** method will return "1, 2" representing the numbers you've stored.

3) Add a class called **`ThreeCalc`** that extends the **`TwoCalc`** class.

   a) **`ThreeCalc`** has a constructor that initializes ***a*** and ***b*** by calling the base constructor and then initializes an additional integer field, ***_c***, which is private and read-only.

   b) Override the **`Calculate() overridden`** method to calculate $a + 2 \times b + 3 \times c$, which is essentially **`TwoCalc.Calculate(a, b) + 3 * c`**. So if you have {1, 3, 5}, the result of this method is $TwoCalc(1, 3) + 3 \times 5 = 1 \times 1 + 2 \times 3 + 3 \times 5 = 22$.

   c) The **`ToString()`** method will return a string representing all three integers. For example, The **`ToString()`** method will return "1, 2, 3" if these are your numbers.

4) Add a class called **`ListCalc`** extending the **`BaseCalc`** class.

   a) It has a **constructor** that takes an **array** of **`BaseCalc`** to initialize a **read-only**, **protected `List<BaseCalc>` field**, named ***Calculations***.

   b) **`ListCalc`** has a constructor that takes an array of integers with **`params`** keyword.

      i) This constructor will throw an **`ArgumentException`** if the size of the array is 0 or 1.

      ii) This constructor will throw an **`ArgumentNullException`** if the array is null.

      iii) This constructor will initialize a **`List<BaseCalc>`** objects of **`ThreeCalc`** instances with at most two **`TwoCalc`** instances at the beginning and the end of the list. Please note that the **`TwoCalc`** instance must start from the beginning of the list. See examples.

CONSTRUCTOR'S INPUT	**List<BaseCalc>**
NULL	**`ArgumentNullException`**
{} OR {1}	**`ArgumentException`**
{1, 2}	{TwoCalc(1, 2)}
{1, 2, 3}	{ThreeCalc(1, 2, 3)}
{1, 2, 3, 4}	{TwoCalc(1, 2), TwoCalc(3, 4)}
{1, 2, 3, 4, 5}	{TwoCalc(1, 2), ThreeCalc(3, 4, 5)}
{1, 2, 3, 4, 5, 6}	{ThreeCalc(1, 2, 3), ThreeCalc(4, 5, 6)}
{1, 2, 3, 4, 5, 6, 7}	{TwoCalc(1, 2), ThreeCalc(3, 4, 5), TwoCalc(6, 7)}
{1, 2, 3, 4, 5, 6, 7, 8}	{TwoCalc(1, 2), ThreeCalc(3, 4, 5), ThreeCalc(6, 7, 8)}
{1, 2, 3, 4, 5, 6, 7, 8, 9}	{ThreeCalc(1, 2, 3), ThreeCalc(4, 5, 6), ThreeCalc(7, 8, 9)}
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}	{TwoCalc(1, 2), ThreeCalc(3, 4, 5), ThreeCalc(6, 7, 8), TwoCalc(9, 10)}

   c) The **`Calculate()`** method simply sums up all of the results from the **`List<BaseCalc>`** collection's **`Calculate()`** method.

   d) The **`ToString()`** method will simply return all of the numbers you've inputted by combining the results from the **`List<BaseCalc>`** collection's **`ToString()`** methods, separate each of them with semicolons (;). This way, you can see the comma-separated numbers for two calc and three calc, which are separated again with ;.