Objectives:

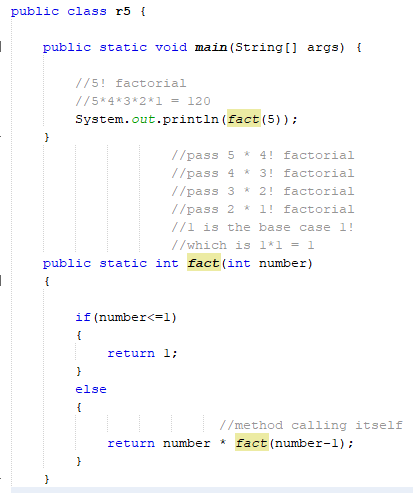
* A recursive method is a method that calls itself.
* Until the method no longer calls itself
* It calls itself over and over again until a **base** **condition** is met that breaks the loop.
* People **use** **recursion** only when it is very complex to write iterative code
  + an **iterative** function is one that loops to repeat some part of the **code**

**There are 1 challenge exercises, each worth 100%**

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

**Project #1**

Retrieving the 5! Factorial



The base condition is if(number<=1)

By taking out the -1 we will get a stack overflow error which means something that will go on for infinity.

**Project #1 (A)**

Use recursion to add all the numbers up to 10.

**Graphical user interface, text, application, email

Description automatically generated**

**Project 1 (B)**

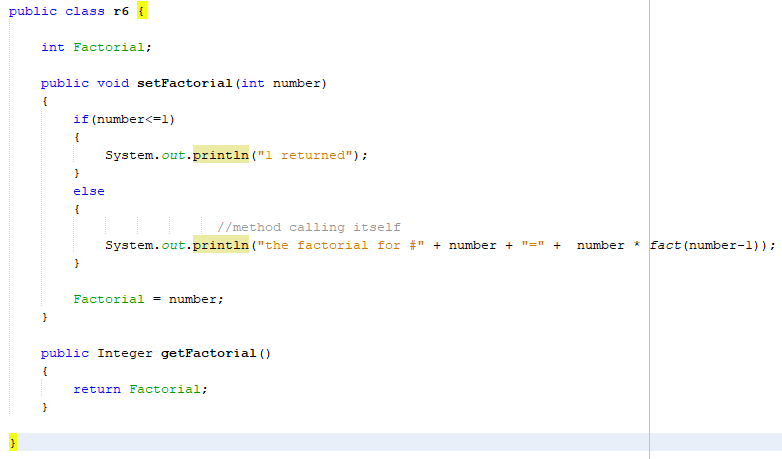
Use recursion to add all the numbers between 5 to 10.

**Graphical user interface

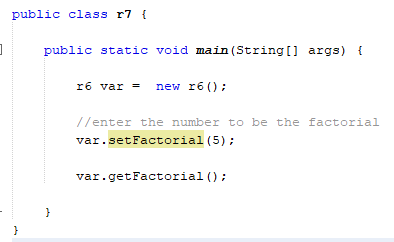
Description automatically generated with medium confidence**

**Project #2**

Creating an Accessor and Mutator methods to enter the factorial



The main method



**Project #3**

Getting the triangular number, they call it the triangular number because it looks like a triangle

If(num==1)

passing 6 as number

1. 6 +? 6+15= (21) FINAL ANSWER
2. 5 +? 5+10= (15) GO UP
3. 4 +? 4+6= (10) GO UP
4. 3 +? 3+3= (6) GO UP
5. 2 +? 2+1= (3) GO UP (previous #)
6. 1 GO UP

{

return 1;

} else

{

(3 +? (6 – 1 = 5));

return (num + getTriNum(num-1));

2 +? mark is after #, 2+f = 3, 3+e, 4+d, 5+c, 6+b

|  |
| --- |
| Are 6 equal to 1? No then go to the else statement and calculate the triangular number |
| Are 5 equal to 1? no |
| Are 4 equal to 1? no |
| Are 3 equal to 1? no |
| Are 2 equal to 1? no |
| Is 1 equals 1? **YES then return 1 and show output** |

Method 1: number = 6 : 6 + getTriNum(6-1)) > 5

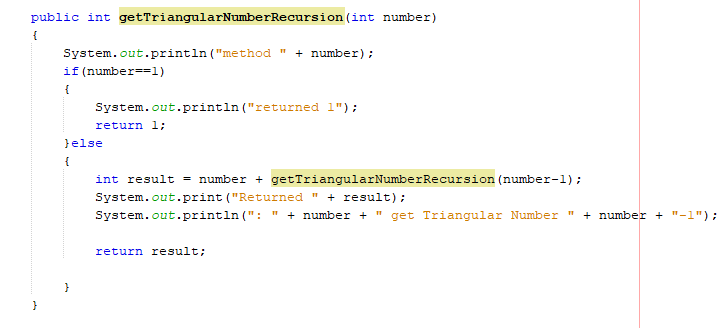
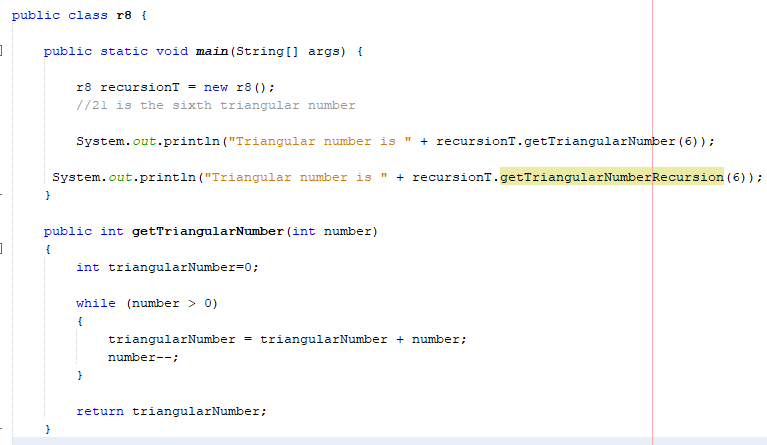
Method 2: number = 5 : 5+ getTriNum(5-1)) > 4

Method 3: number = 4: 4 + getTriNum(4-1)) > 3

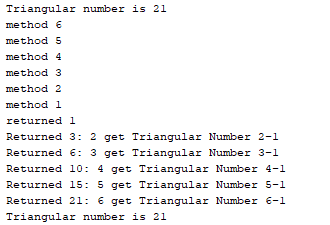
Method 4: number = 3: 3 + getTriNum(3-1)) > 2

Method 5: number 2 : 2 + getTriNum(2-1)) > 1

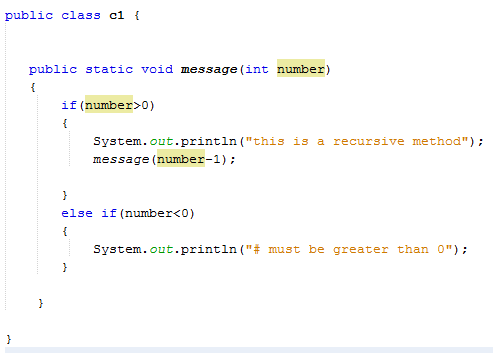
**The return or base value is 1, after it reaches the base value, we go back up**

****

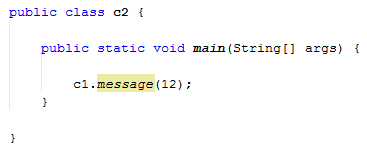
**Output, 2+1 = 3, 3+3 = 6, 4+6=10, 5+10 = 15, 6+15 = 21, the sixth triangular number is 21**

****

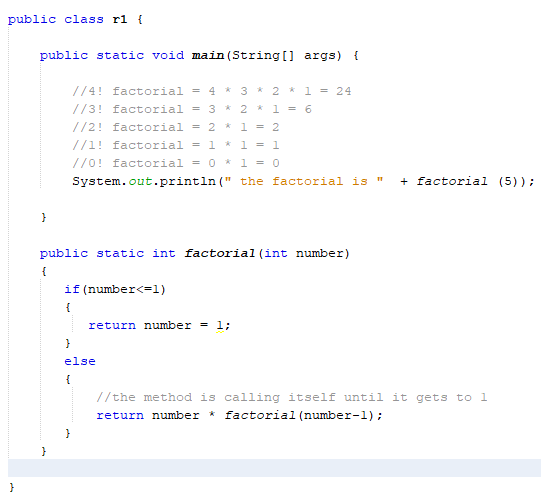
**Project #4**



Note: For example, we compute factorial n if we know the factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.

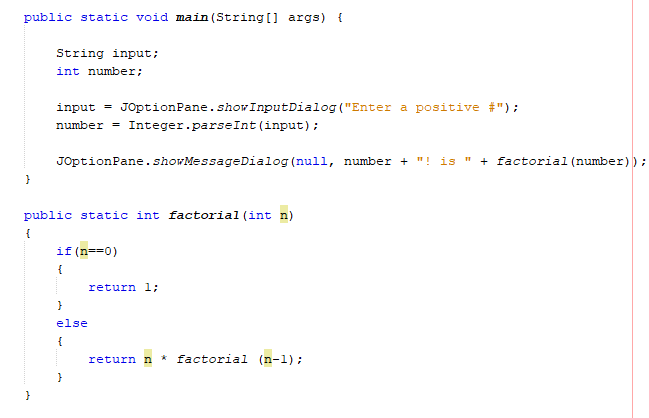


**Project #5**



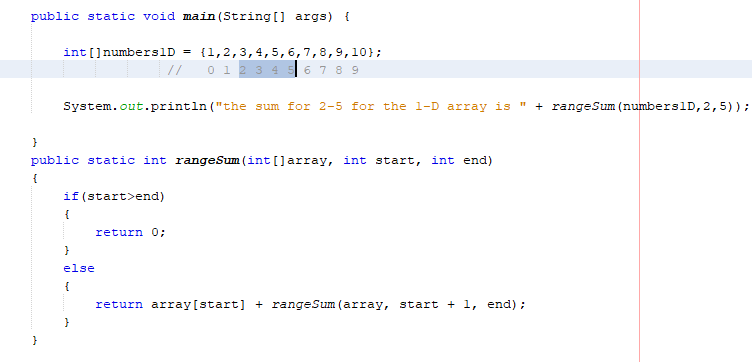
**Project #6**

Using a JOptionPane with recursions



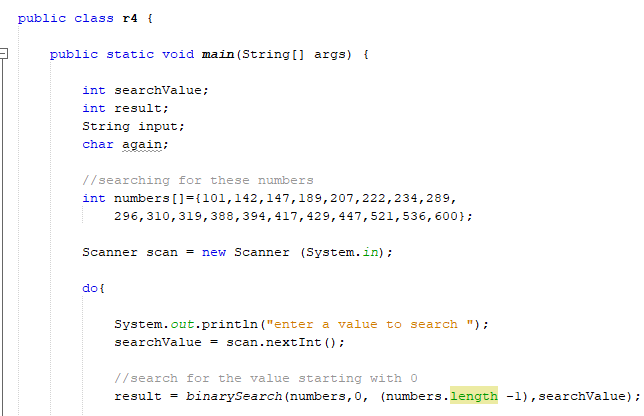
**Project #7**

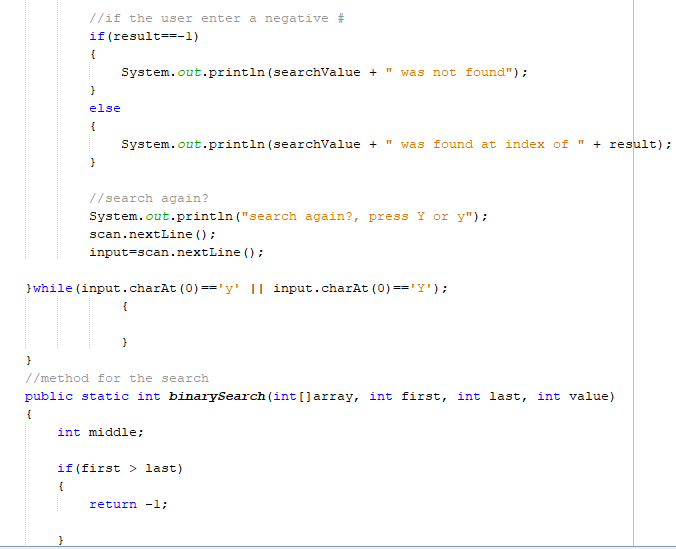
*Summing Arrays with Recursions*

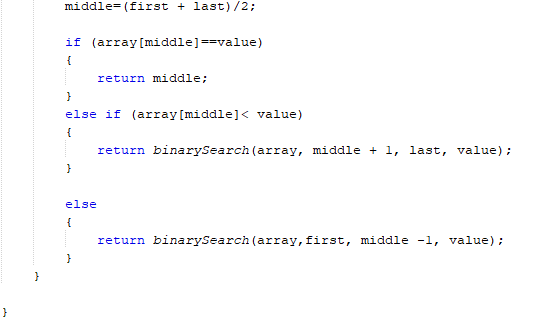


**Project #8**

*Binary Search*

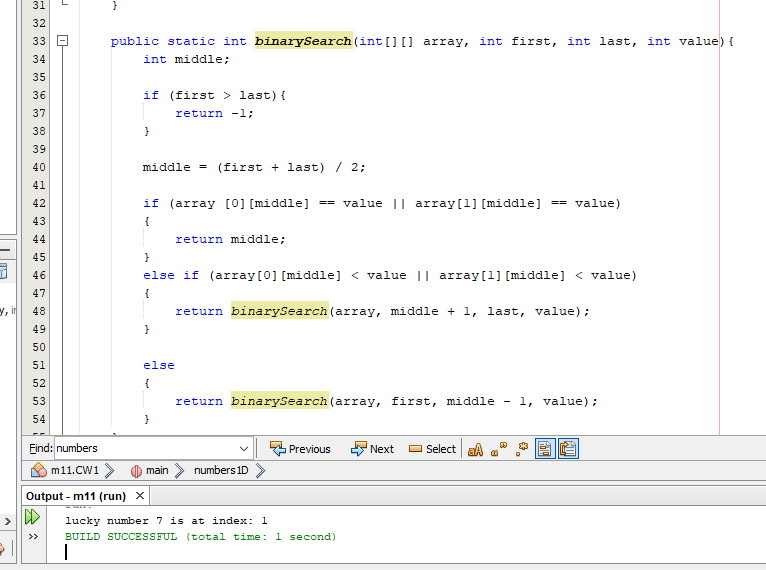






**Challenge Exercise #1:** write a program (using a recursive method) that will use a 1-D and 2-D array, for numbers 1-10 and retrieve the lucky number 7. **(See project #8 (Binary Search) for reference)**

**#1 print screen the output with the code below here.**



**Submit this document to the Module 7 Class Exercise.**