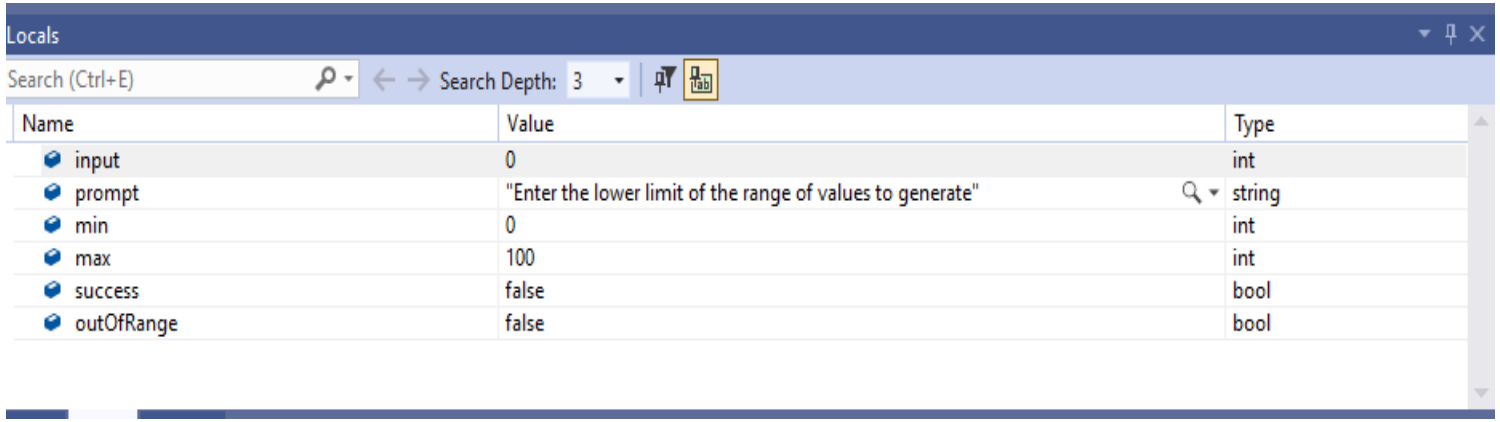
CMPE 1666- Intermediate Programming

Lecture 6 - Recursions

A method calling another method

- You have been given a program Lecture6Program1 on BrightSpace. Download the program and place it in a local folder on your computer.
- Consider the method **GetRange()** that calls **GetValue()** twice to obtain the minimum and maximum values for a range of integers.
- We'll set a number of break points and run the program in Debug mode.
- If we study the call stack (using the Debug tool), we see that when **GetValue()** is called, from GetRange(), our call stack is as below.

The Call Stack



The Call Stack

- Values passed to a method are placed on the system stack for the method to access them.
- The system stack also contains the address of where program control will continue after returning from the method call.
- When a method is called, the return address and arguments are pushed onto the stack.
- When the return statement is executed, the return address is popped off the stack, allowing program execution to continue with the next statement.

What Happens when a method calls itself?

- Our next question is: Can a method call itself?
  - The answer is: Yes.
- How does the call stack work in that case?
  - Return Address for Multiple instances of the same method are pushed onto the stack

Recursion

- Recursion occurs when a method calls itself.
- Recursion is used as form of repetition that does not involve iteration, such as in a loop.
- Any programming problem that can be solved using a loop, can also be solved using recursion.
- Likewise, any problem that can be solved via recursion can be solved using iteration

Recursion

- Iteration is preferred by programmers for most recurring events .
- Recursion is used for instances where the programming solution would be greatly simplified.
- Recursion involves an additional cost in terms of the space used in RAM by each recursive call to a function and in time used by the function call.

Recursion and The Call Stack

- Recursive calls to a method employ a great deal of stack manipulation.
- This overhead can become a performance issue when compared to using iteration.
- Still, there are computing problems where recursion is the preferred solution.

Iterative Version of SumInts

- Let's consider the method given below.
- This method uses iteration to calculate the total of integers from 1 to n (where n>=0)

```
int SumInts(int n) {
      int iSum = 0;
     for (int x=n; x>=0; --x)
            iSum += x;
      return iSum;
}
```

Recursion Example

- We note that For any n>0,

  f(n)=n + (n-1) + (n-2) + (n-3)…….1

- We note that the RHS can be written as: **n + f(n-1)**
- Thus we can rewrite the above formula as: **f(n) = n + f(n-1)**
- Furthermore, we can state that **f(0)=0**
- Hence the function can be defined recursively as below

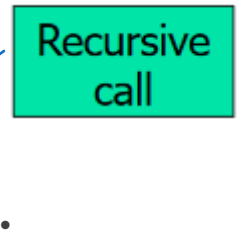
  f(0) = 0

  f(n) = n + f(n-1) Where n is any integer greater than 0.

Recursion Example

- This recursive definition can be expressed in a C# method as follows:

```
int RSum(int n)
 {     if (n < 1)
             return 0;
         else
             return RSum(n-1)+ n;
}
```



Recursion Example

- When writing a recursive method, providing a means for the recursion to halt is of primary importance.
- The **if** control structure ensures that the recursion will halt and return **0** when the value of **n** is reduced to **0**.

Recursion Example

- A recursive call to the same method is required to perform the calculation.
- If **n** has a value greater than **0**, the **if** will fail and **RSum** will be called again with **n** reduced in value, which will eventually cause the recursion to halt.

Recursion Example

- Let's see what happens when the method **RSum( )** is called with a initial value of 5.

```
static void Main(string[] args){
        Console.WriteLine($" {RSum(5)}");
    }
```

- The following slides show how the values 0 to 5 are summed.

Recursion Example

- With the initial call of **RSum(5)** (1st call), the value 5 is placed on the stack and passed to **RSum**.
- Since **n** has the value of 5 and is greater than 0, the else construct will call **RSum(4)** to evaluate the expression **RSum(n-1)+n**.
- This will not be possible because the value of **RSum(n-1)** is not known and causes a recursive call to function RSum.

Recursion Example

main calls RSum(5) ——— 1st Call

n = 5

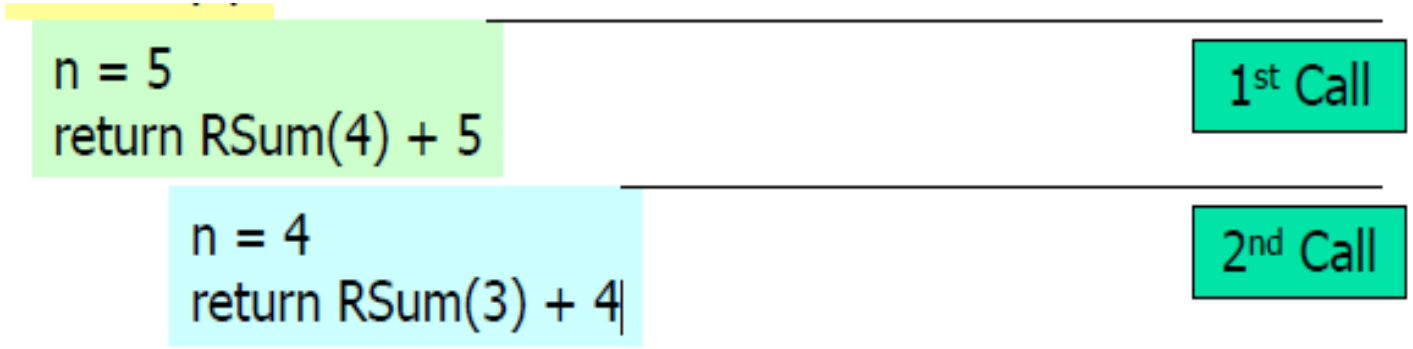
return RSum(4) + 5

Recursion Example

- With the first recursive call (2nd call) of **RSum(4)**, the value 4 is placed on the stack and passed to the second call of **RSum( )**.
- Since n has the value of **4** and is greater than **0**, the else construct will call **RSum(3)** to evaluate the expression **RSum(n-1)+n**.
- This will not be possible because the value of **RSum(n-1)** is not known and causes a recursive call to function RSum.

Recursion Example

- main calls RSum(5)

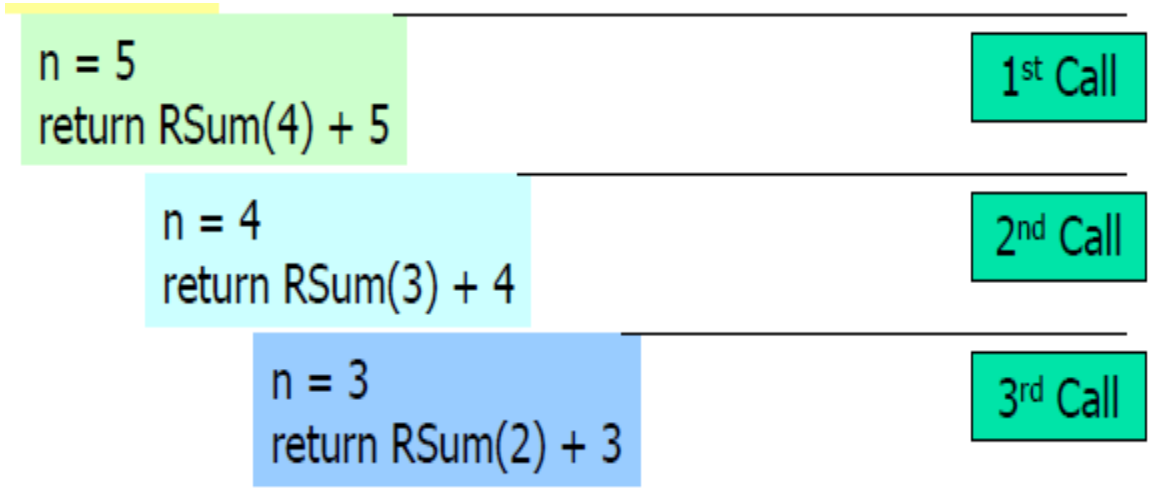


Recursion Example

- With the second recursive call (3rdcall) of **RSum(3)**, the value **3** is placed on the stack and passed to the third instance of **RSum( )**.
- Since **n** has the value of **3** and is greater than **0**, the else construct will call **RSum(2)** to evaluate the expression **RSum(n-1)+n**.
- This will not be possible because the value of **RSum(n-1)** is not known and causes a recursive call to method RSum.

Recursion Example

Main calls RSum(5)

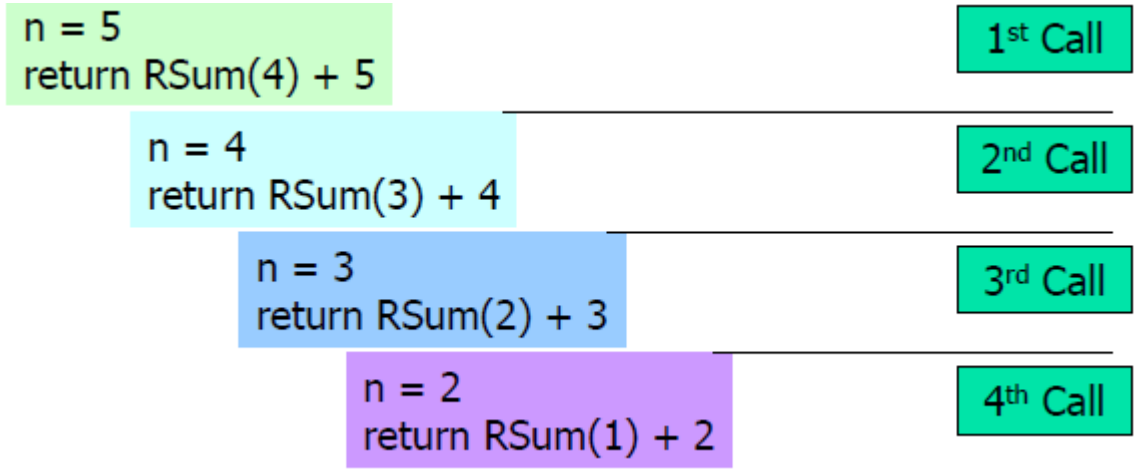


Recursion Example

- With the third recursive call (4thcall) of **RSum(2)**, the value **2** is placed on the stack and passed to the fourth instance of **RSum( )**.
- Since **n** has the value of **2** and is greater than **0**, the else construct will call **RSum(1)** to evaluate the expression **RSum(n-1)+n**.
- This will not be possible because the value of **RSum(n-1)** is not known and causes a recursive call to function **RSum**.

Recursion Example

- main calls RSum(5)

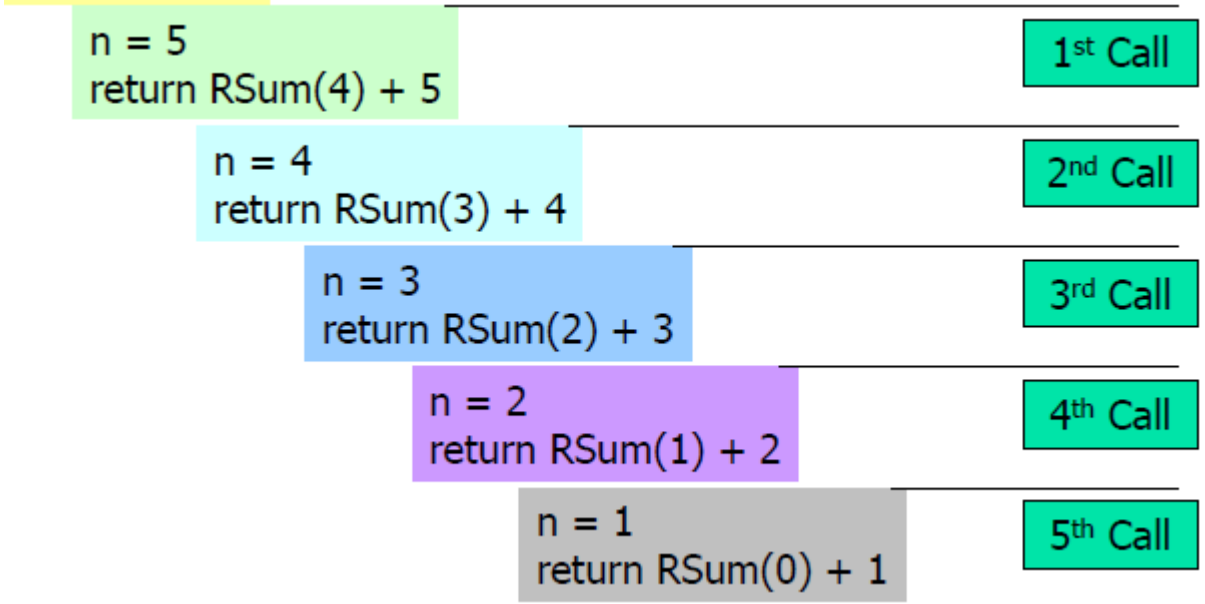


Recursion Example

- With the fourth recursive call (5thcall) of **RSum(1)**, the value 1 is placed on the stack and passed to the fifth instance of **RSum( )**.
- Since n has the value of 1 and is greater than 0, the else construct will call **RSum(0)** to evaluate the expression **RSum(n-1)+n**.
- This will not be possible because the value of **RSum(n-1)** is not known and causes a recursive call to function **RSum**.

Recursion Example

- Main calls RSum(5)

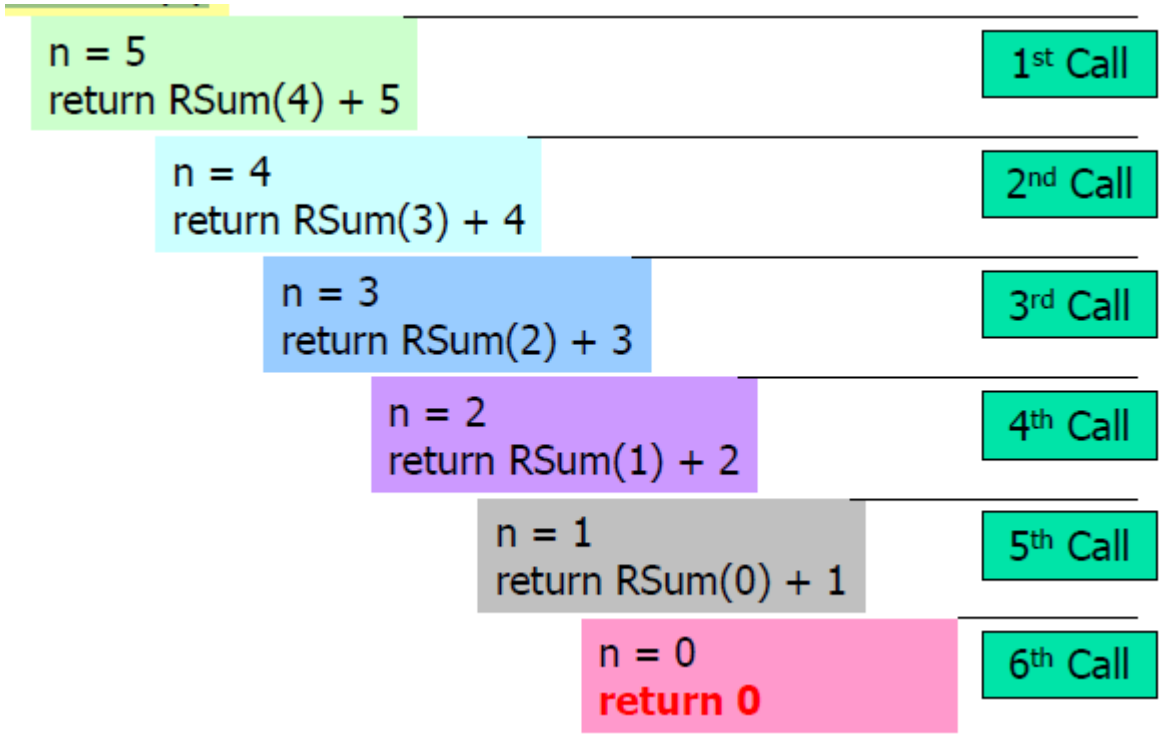


Recursion Example

- With the fifth recursive call (6thcall) of **RSum(0)**, the value **0** is placed on the stack and passed to the sixth instance of **RSum( )**.
- Since **n** has the value of **0**, which is less than **1**, the if construct is true, which will cause the return of **0**.

Recursion Example

- Main calls RSum(5)

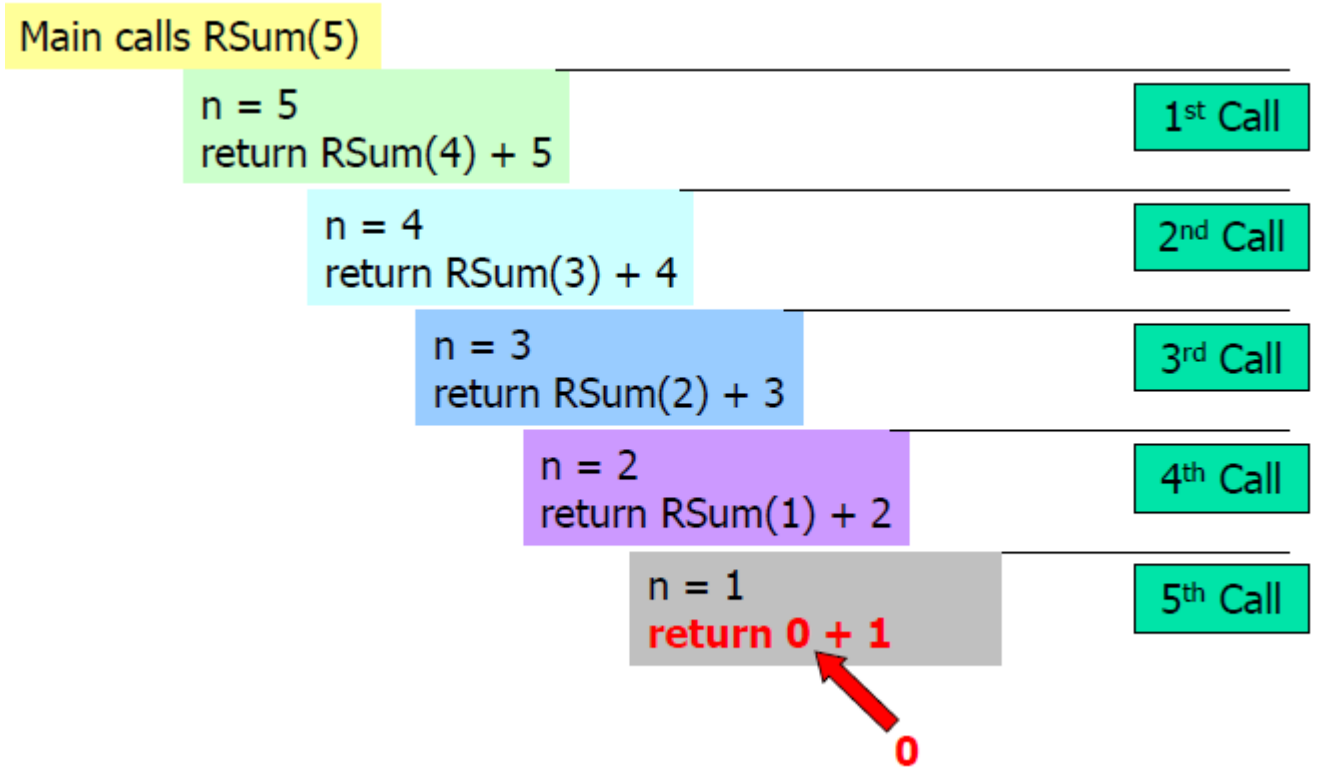


Recursion Example

- The return value of **0** from the sixth call replaces **RSum(0)**.
- The return statement in the 5th call can now be evaluated.
- The sixth function call is removed from the stack, and that instance returns **0**.

Recursion Example

- Main calls RSum(5)

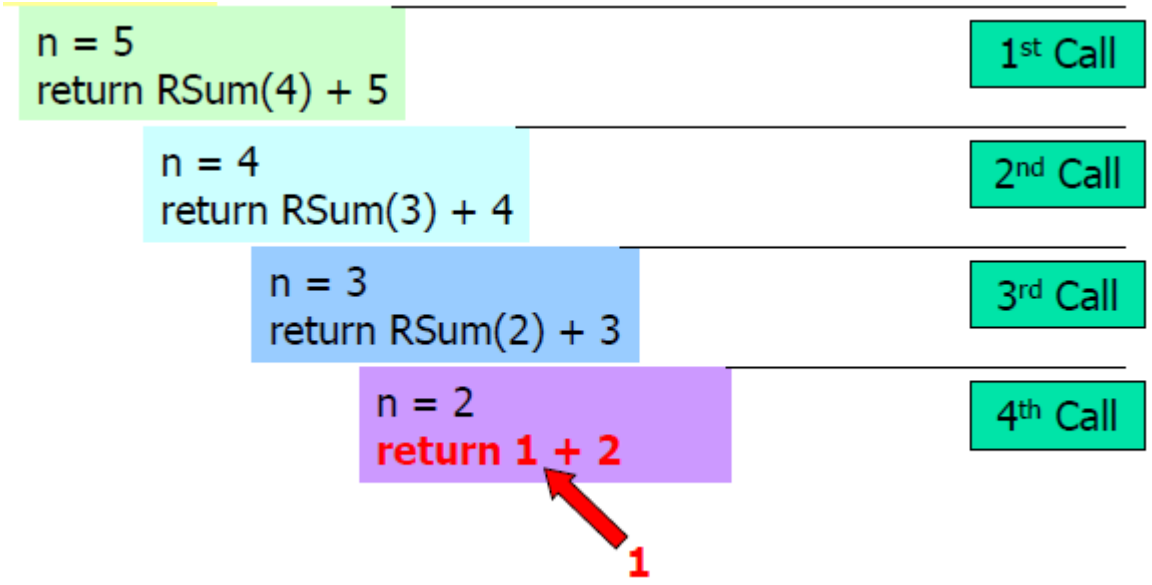


Recursion Example

- The return value of **1** from the fifth call replaces **RSum(1)**.
- The return statement in the 4th call can now be evaluated.
- The fifth method call is removed from the stack, and that instance returns **1**.

Recursion Example

- Main calls RSum(5)

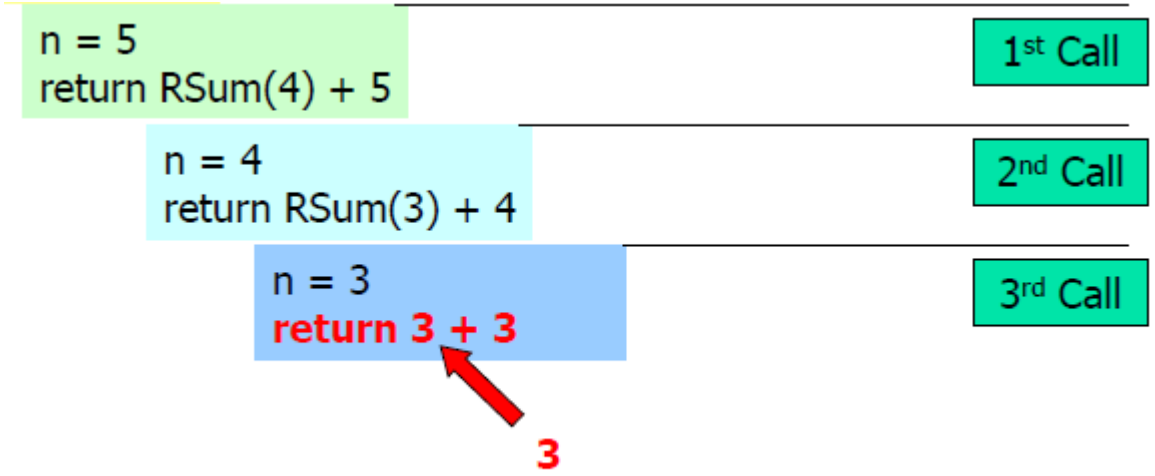


Recursion Example

- The return value of **3** from the fourth call replaces **RSum(2).**
- The return statement in the 3rd call can now be evaluated.
- The fourth function call is removed from the stack, and that instance returns **3**.

Recursion Example

Main calls RSum(5)

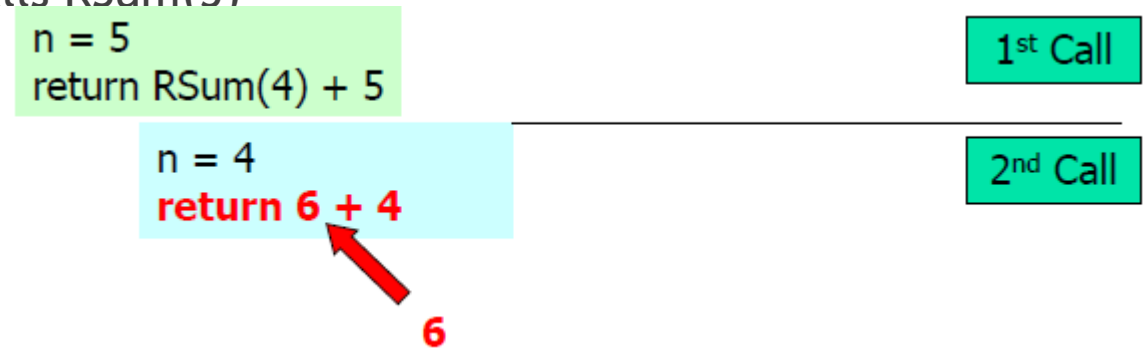


Recursion Example

- The return value of **6** from the third call replaces **RSum(3)**.
- The return statement in the 2nd call can now be evaluated.
- The third function call is removed from the stack, and that instance returns **6**.

Recursion Example

- Main calls RSum(5)

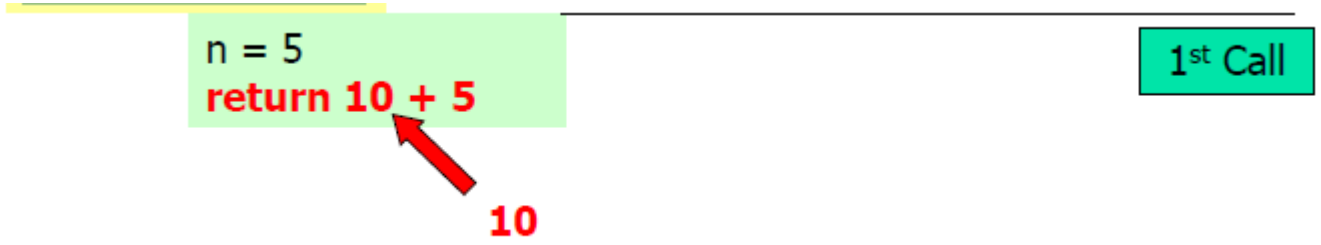


Recursion Example

- The return value of **10** from the 2nd call replaces **RSum(4)**.
- The return statement in the 1st call can now be evaluated.
- The 2nd method call is removed from the stack, and that instance returns **10**.

Recursion Example

- Main calls RSum(5)

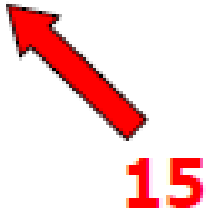


Recursion Example

- The initial method call to RSum( ) returns the value of 15 to the Main program.
- Recursion is complete, and the initial instance of the method RSum( ) is removed from the stack.

Recursion Example

RSum(5) returns 15



15

Recursion Types

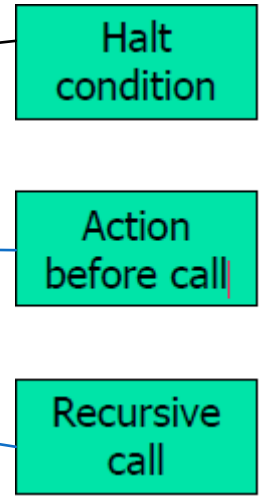
- Determined by when the processing takes place in the function in relation to when the recursive call is made.
- There are three types of recursion:
  - Pre-order
  - In-order
  - Post-order

PreOrder Recursion (Also called Head Recursion)

- Performs the method's processing **prior** to the recursive call.
- Let's apply recursion to displaying a string.
  - Pass a reference to the string, and a character position, to each recursive call, adding one to the position value.
  - The endpoint will be: the position value is equal to the string length.

PreOrder Recursion

```
static private void PreOrder(string sString, int iPosition)
{
int length=sString.Length;
 if (iPosition !=length)
{    Console.Write(sString[iPosition]);
      PreOrder(sString, iPosition + 1);
}
}
```



PreOrder Recursion

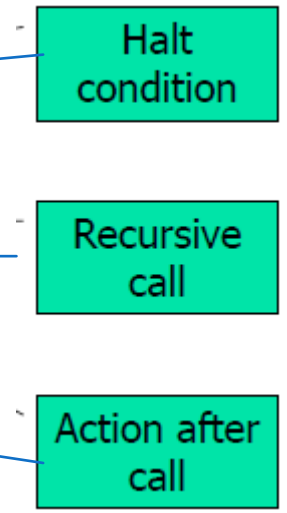
- Call PreOrder("Meat.", 0);
- Displays…
- Meat.

PostOrder Recursion (Also called Tail Recursion)

- Perform the method's processing **after** the recursive call.
- Let's apply recursion to displaying a string backwards.
- Pass a reference to the string, and character position value, to each recursive call, adding one to the character position value.
- The endpoint will be: when the position value reaches the length of the string.

PostOrder Recursion

```
static void void PostOrder(string sString, int iPosition)
{
int length=sString.Length;
    if(iPosition !=length)
{     PostOrder(sString, iPosition + 1);
       Console.Write( sString[iPosition]);
}
}
```



PostOrder Recursion

- Call PostOrder(“Meat", 0);
- Displays the string backwards…

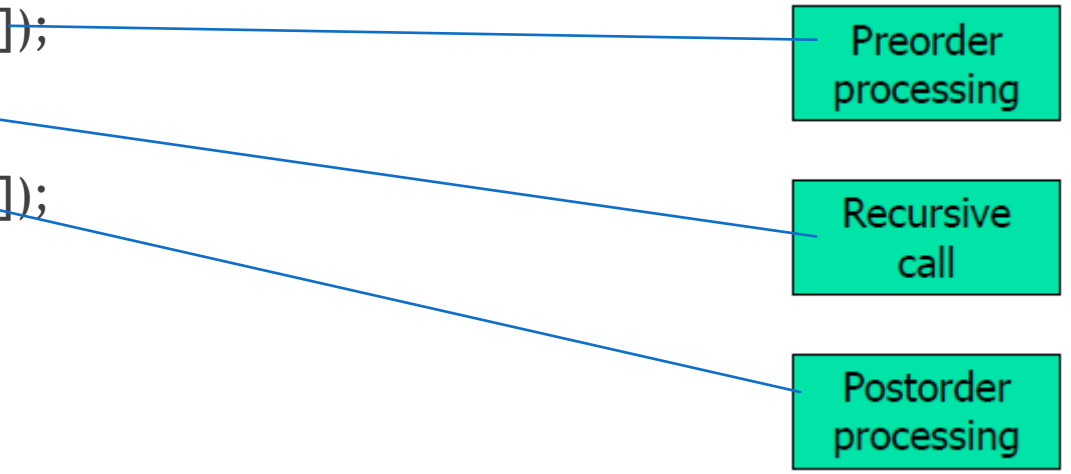
  taeM

InOrder Recursion

- Perform the function's processing **before** and **after** the recursive call.
- Let's apply recursion to displaying a string forward and backwards.
- Pass a reference to the string, and a character position value, to each recursive call, adding one to the character position value.
- The endpoint will be when the character position reaches the string length.

InOrder Recursion

```
static private void InOrder(string sString, int iPosition)
{
int length=sString.Length;
    if(iPosition !=length)
{
        Console.Write( sString[iPosition]);
        InOrder(sString, iPosition + 1);
        Console.Write( sString[iPosition]);
}
}
```



InOrder Recursion

- Call InOrder("meat", 0);
- Displays the string forwards and backwards…

  meattaem

Why Recursion?

- Recursion is a common programming technique applied to searching, traversing data structures, and decision trees for game AI.

Recursion Rules

- Recursive calls must always have an endpoint that will trigger the returns.
- Arguments passed to recursive calls must change to approach the endpoint.
- Other processing performed can be before or after the recursive call.

Designing Recursive Programs

- When designing a recursive algorithm, there are several considerations you should typically bear in mind.
- It is extremely useful in all programming situations to sketch out your algorithm and design on paper before you begin programming.
- With recursion (because it is so difficult to hold in your head), such planning is critical.

Designing Recursive Programs

- The design steps include:
  1. Determine what sort of processing you will need to do.
     - Do you wish to progress from the initial case to the base case (preorder)?
     - Do you need to progress from the base case to the initial case (postorder)?
     - Or must you do some processing on the way down to the base case as well as on the way up (inorder)?

  2. Determine your state change. How will it be tracked? Through an argument? Through a returned value?

Designing Recursive Programs

3. Determine your base case.
   - Are you guaranteed to get to the base case no matter what your initial condition?
   - Does processing need to be done at the base case, or should it simply return?
4. Check your design. Sketch the call stack as well as the processing status for several iterations and for different initial conditions to see if it is behaving how you planned.

Designing Recursive Programs

5. Implement the algorithm in code and test it using the debugger.
   - Step through several call instances and inspect the call stack (available in Visual Studio as one of the debugger windows).
   - Does the algorithm function as designed?
   - Is the state changing?
   - Will it escape at the base case?

Common Recursive Problems

1. Factorial
2. Palindrome
3. Fibonacci
4. Prime Factors
5. Greatest Common Divisor
6. Evaluating Prefix Expressions
7. The Tower of Hanoi
8. The Flood-Fill Algorithm
9. Tree Algorithms

Factorial

```
static private long factorial(int n) {
        if (n >= 1)
                return n * factorial(n - 1);
        else
                return 1;
        }
```

n!= n*(n-1) *(n-2)*(n-3)……..1

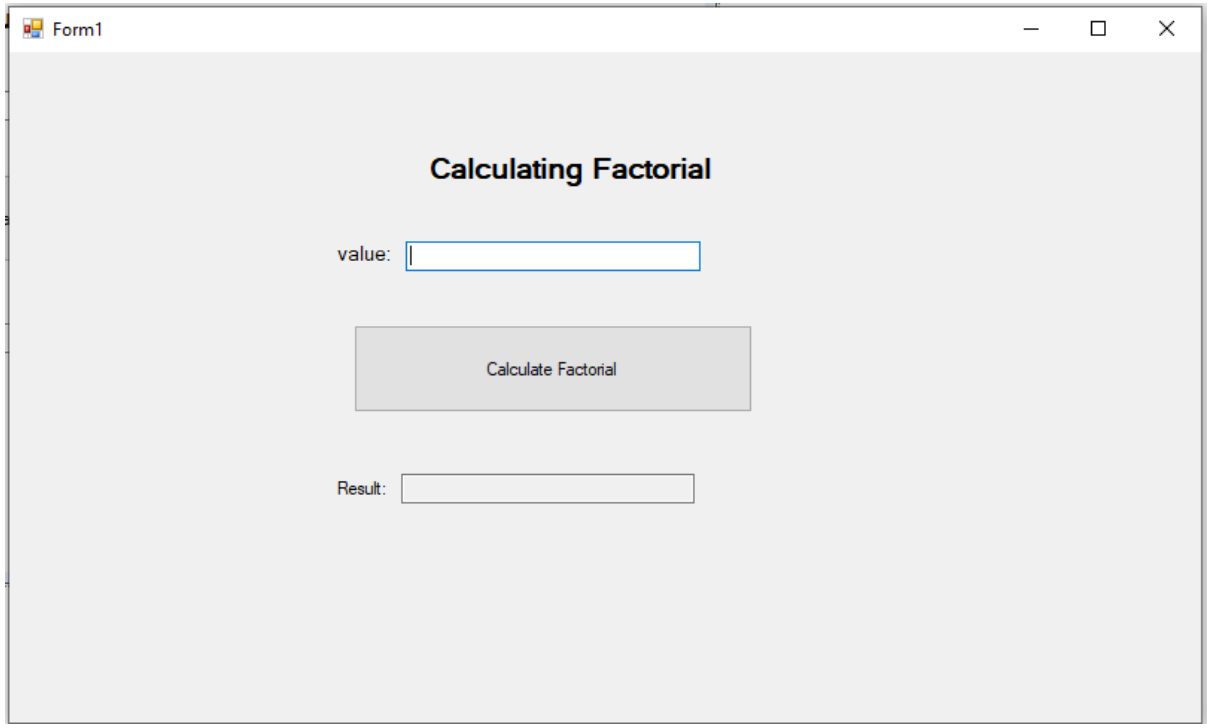
(n-1)!= (n-1)*(n-2)*(n-3)……..1

n!=n*(n-1)!

- For the upcoming exercises, Download the zip files Lecture6Exercise1 and Lecture6Exercise2 from Brightspace.

Lecture6- Exercise1

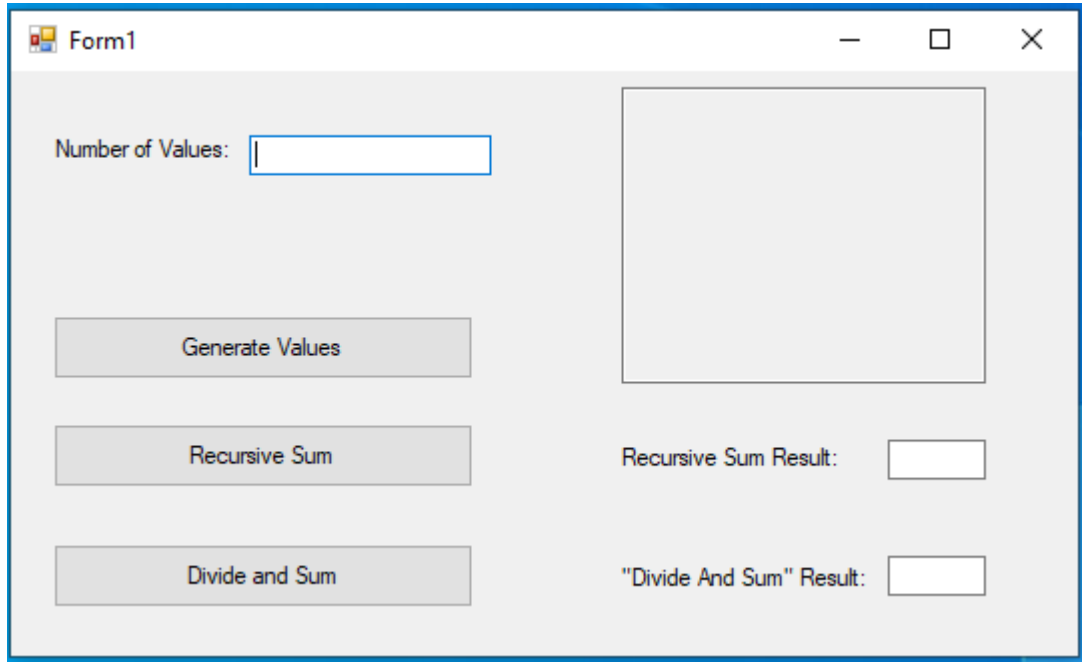
- You have been provided with an application containing the UI shown below.
- Add the code to the click event handler of the button, such that it verifies if the value textbox has a valid integer value, performs the required conversion and calculates the factorial using a recursive method.
- The factorial value is then displayed in the Read-only textbox.



Lecture6-Exercise2

- You have been provided with an application containing the UI shown in the next slide.
- In the program class, include a method **RecursiveSum()** that has as parameters a List of int type and an index. It recursively calculates the sum of the values in the list, incrementing the index at each recursive call.
- Include a class-level variable of type List<int>
- Initially, the "**Generate Values**" button is enabled and the other buttons are disabled. When the user clicks on the "**Generate Values**" button, the program verifies if the "**Number Of Values**" textbox contains a valid integer between 1 and 50. If value is not valid or not in the required range, a message box pops up with an appropriate message.
- If the value is valid and in the required range the program generates a number of integers as specified (in the textbox) in the range 1..1000, adds then to the class-level List variable and also displays them in the multi-lines textbox. It disables the "**Generate Values**" button and enables the other buttons. When the user clicks the "**Recursive Sum**" button, the program uses the **RecursiveSum()** method to calculate the sum and displays the result in the corresponding textbox.

Lecture 6- Exercise2



Lecture6-Exercise2B

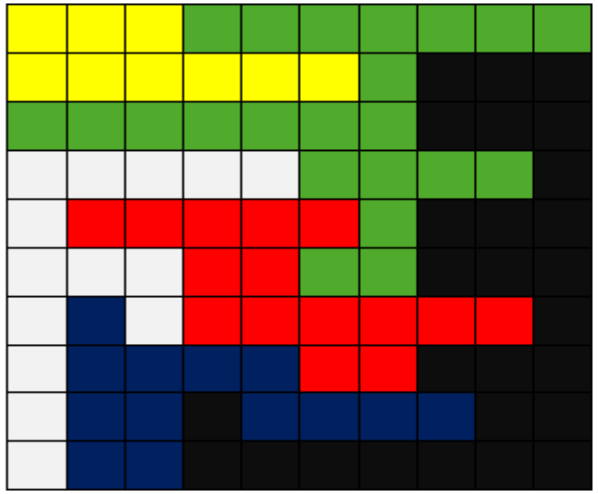
- To the program for exercise 2, add a method **DivideAndSum(),** that works as follows:
  - It has as parameters a List of int type, and 2 variables called **low** and **high**, representing a low and high index in the list.
  - If low==high, it must return the value at that index
  - Otherwise, it sets a new variable mid=(low+high)/2, it recursively calls itself on the 2 parts of the list between low and mid and the part between mid+1 and high and returns the sum of the 2 obtained values.
- Add an event listener for the click event of the "**Divide And Sum**" button, such that it calls the **DivideAndSum** method, passing the class-level list variable, the first and last indices of the list, and it displays the obtained result in the corresponding textbox.

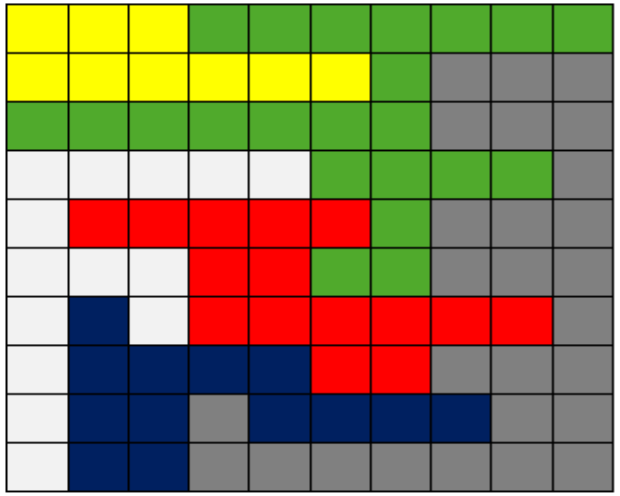
Palindrome

- Method **IsPalindrome()**
- Parameters: String to be processed, low_index, high_index.
- Returns true or false
- Base case: high_index<=low_index -> return true
- Recursive step: Let string be processed be str
- If str[low_index] != str[high_index] return false
- Else: recursive step= call IsPalindrome() passing str, low_index+1, high_index-1

Flood-Fill

- **Flood fill,** also called **seed fill,** is an algorithm that determines the area connected to a given node in a multi-dimensional array. It is used to fill connected, similarly-colored areas with a different color.
- Consider the matrix below. If the starting node =(3,9), the target color= “BLACK”, the replacement color=“GREY”.
- The algorithm looks for all nodes in the matrix that are connected to the start node by a path of the target color and changes them to the replacement color





Flood-Fill

```
def floodfill(x, y, oldColor, newColor):
  # assume surface is a 2D image and surface[x][y] is the color at x, y.
  if surface[x][y] != oldColor: # the base case
                return
  surface[x][y] = newColor
  floodfill(x + 1, y, oldColor, newColor) # right
  floodfill(x - 1, y, oldColor, newColor) # left
  floodfill(x, y + 1, oldColor, newColor) # down
  floodfill(x, y - 1, oldColor, newColor) # up
```