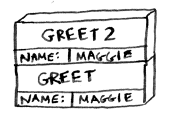
**//---------------------------------------------------------- Recursion -----------------------------------------------------------//**

// Programming technique in which *a method makes a call to itself from within its own method body*.  
// Used to solve complicated **combinatorial problems**, e.g. **generating permutations** and **variations** and simulating **nested loops**.  
// When in the body of a method there is a call to the same method, we say that the method is **directly recursive**..  
// If method A calls method B, B calls method C, and C calls method A we call the methods A, B and C **indirectly recursive** or **mutually recursive**.  
// When using recursion, we have to be totally sure that after a certain count of steps we get a concrete result. For this reason we should have one or more cases in which the solution could be found directly, without a recursive call. These cases are called **bottom of recursion**.  
// If a recursive method has no base case, i.e. bottom, it will become **infinite** and the result will be **StackOverflowException**.  
// Recursive calls can consume much **more resources** (CPU time and memory). On each recursive call in the stack new memory is set aside for arguments, local variables and returned results. If there are too many recursive calls, a stack overflow could happen because of lack of memory.  
// If by using recursion we reach a simpler, shorter and easier for understanding solution, not causing inefficiency and other side effects, then we can prefer recursive solution. Otherwise, it is better to think of iteration.   
// Avoid recursion, unless you are certain about how it works and what has to happen behind the scenes. Recursion is a great and powerful weapon, with which you can easily shoot yourself in the leg. Use it carefully.  
// Computers use a data structure internally, where if we insert an item, it gets added to top of structure & when we read an item, we only read the topmost item, & it’s taken off this structure. This data strcuture which has only two actions: *push* (insert) and *pop*(remove and read) is called the ***call stack.***//Our computer allocates a box of memory every time we make a function call *- it saves the values for all the variables for that call in memory.*// The next time we make a function call, our computer again allocates a box of memory for this function call – like a stack of boxes.   
// Then when we return from the function call, the box on top of the stack gets popped off.   
// ***Important:*** When *you call a function from another function, the calling function is paused in a partially completed state. All the values of the variables for that calling function are still stored on the call stack (i.e., in memory) separately from those of the called function*. Now when we return from the called function, you’re back to the calling function, and you pick up where you left off.

// In general, to create recursive methods, it is necessary that we *break the task we are trying to solve in* ***subtasks*, for the solution of which we can use the same algorithm (recursively)**. The combination of solutions of all subtasks should lead to the solution of the initial problem. In each recursive call the problem area should be limited so that *at some point the bottom of the recursion is reached*, i.e. breaking of each **subtask must lead eventually to the bottom of the recursion**.  
// **Factorial :** Presence of recurrent dependence is not always obvious. Calculate the values of the factorial for the first few integers, it will now become obvious that n! = n(n-1)!

namespace LearningRecursion  
{  
 public class Program  
 {  
 static void Main(string[] args)  
 {  
 Console.Write("Enter a integer for Factorial :");  
 int number = int.Parse(Console.ReadLine());  
 decimal factorial = Factorial(number);  
 Console.WriteLine("The factorial of {0} is {1}.", number, factorial);  
 Console.ReadLine();  
 }  
 static decimal Factorial(int n)  
 {  
 if (n == 0)  
 {  
 return 1;  
 }  
 else  
 {  
 return n \* Factorial(n - 1);  
 }  
 }  
 }  
}