Recursion – Walkthrough

Recursion

The programs we have discussed generally are structured as methods that call one another in a hierarchical manner

For some problems, it is useful to have a method actually call itself

A recursive method is a method that calls itself either directly or indirectly through another method

Recursion is an important topic discussed at length in higher-level computer science courses

Recursive problem-solving approaches have a number of elements in common

A recursive method is called to solve a problem

The method actually knows how to solve only the simplest case(s), or base case(s)

If the method is called with a base case, the method returns a result

If the method is called with a more complex problem, the method divides the problem into two conceptual pieces - a piece that the method knows how to perform (base case) and a piece that the method does not know how to perform

To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version of it

The method invokes (calls) a fresh copy of itself to work on the smaller problem - this is referred to as a recursive call, or a recursion step

The recursion step also normally includes the keyword return, because its result will be combined with the portion of the problem that the method knew how to solve

Such a combination will form a result that will be passed back to the original caller

The recursion step executes while the original call to the method is still “open”

(i.e., it has not finished executing)

The recursion step can result in many more recursive calls, as the method divides each new subproblem into two conceptual pieces

Each time the method calls itself with a slightly simpler version of the original problem, the sequence of smaller and smaller problems must converge on the base case, so the recursion can eventually terminate

At that point, the method recognizes the base case and returns a result to the previous copy of the method

A sequence of returns ensues up the line until the original method call returns the final result to the caller

As an example of these concepts, let us write a recursive program to perform a popular mathematical calculation

The factorial of a nonnegative integer n, written n! (and pronounced “n factorial”), is the product

**n \* ( n – 1 ) \* (n - 2 ) \* … \* 1**

with 1! equal to 1, and 0! defined as 1

For example, 5! is the product 5 \* 4 \* 3 \* 2 \* 1, which is equal to 120

The factorial of an integer number greater than or equal to 0 can be calculated iteratively (non - recursively) using a for control structure as follows:

factorial = 1;

**for ( int counter = number; counter >= 1; counter- - )**

**factorial \*= counter;**

We arrive at a recursive definition of the factorial method with the following relationship:

**n! = n \* ( n – 1 )!**

For example, 5! is clearly equal to 5 \* 4!, as shown by the following:

**5! = 5 \* 4 \* 3 \* 2 \* 1**

**5! = 5 \* (4 \* 3 \* 2 \* 1)**

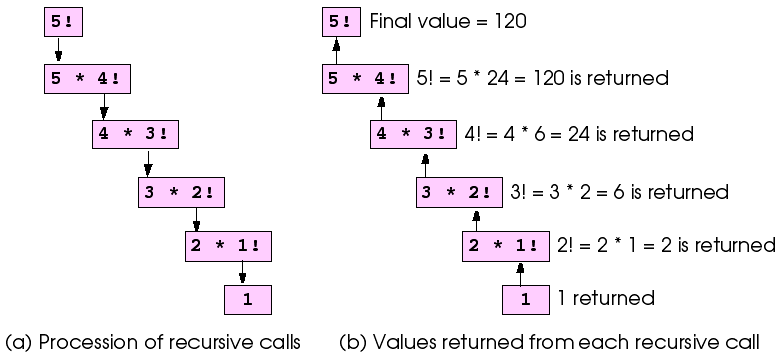
**5! = 5 \* (4!)**

A recursive evaluation of 5! would proceed as shown in the diagram below

Part (a) of the diagram shows how the succession of recursive calls proceeds until 1! is evaluated to be 1, which terminates the recursion

Each rectangle represents a method call

The diagram shows the values returned from each recursive call to its caller until the final value is calculated and returned



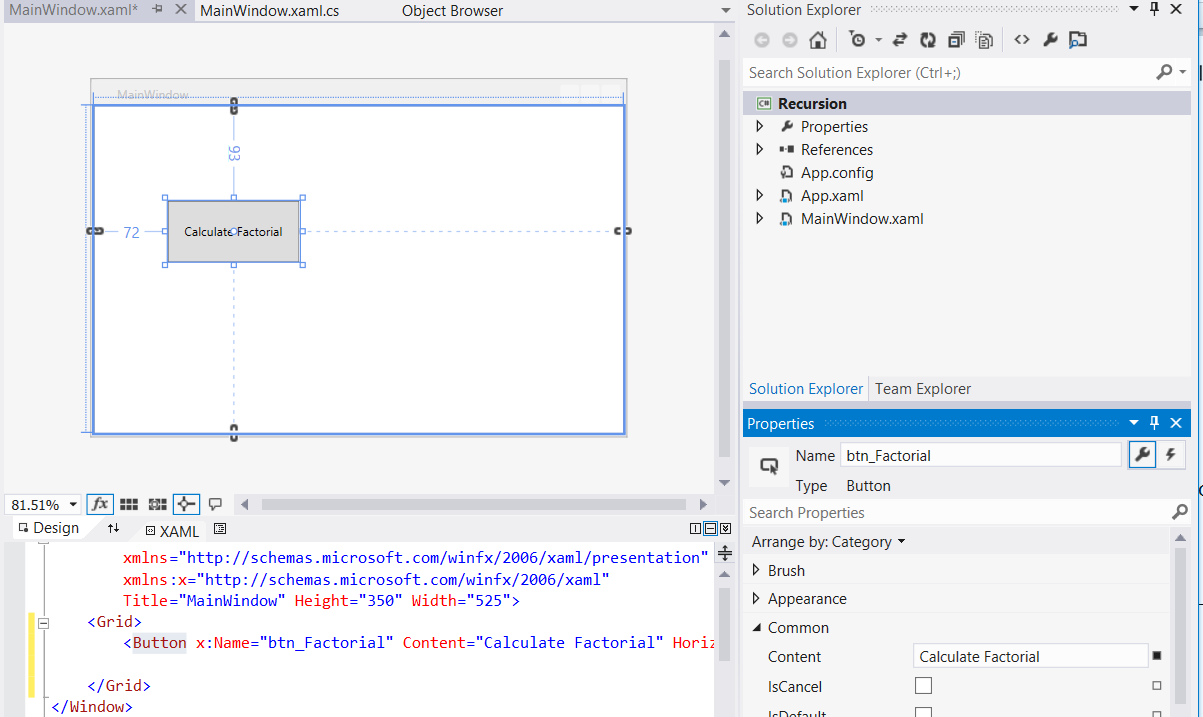
Recursive evaluation of 5!

We are going to build a basic application which has a single button, which when clicked will:

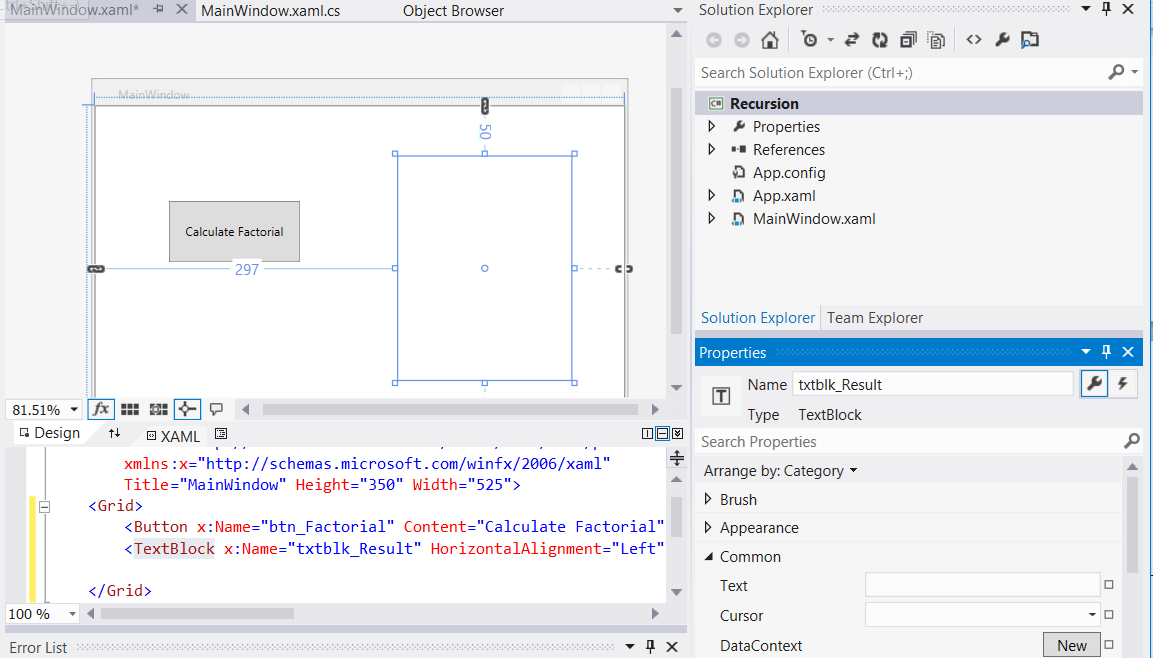
* use recursion to calculate and print the factorials of the integers 0–10

The aim of this tutorial is to show how to call a method recursively.

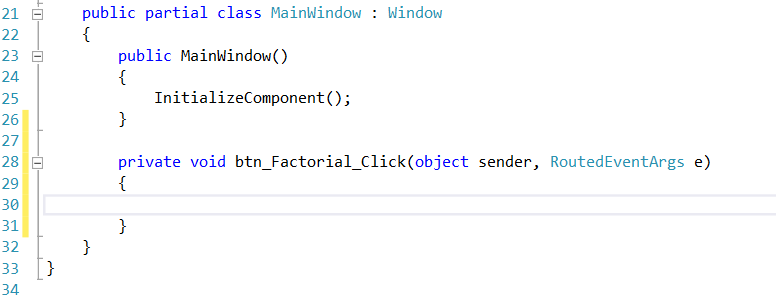
1. Create a new WPF project
2. Drag across a button from the toolbox
   1. set the “Name” property to btn\_Factorial
   2. set the “Content” property to “Calculate Factorial”



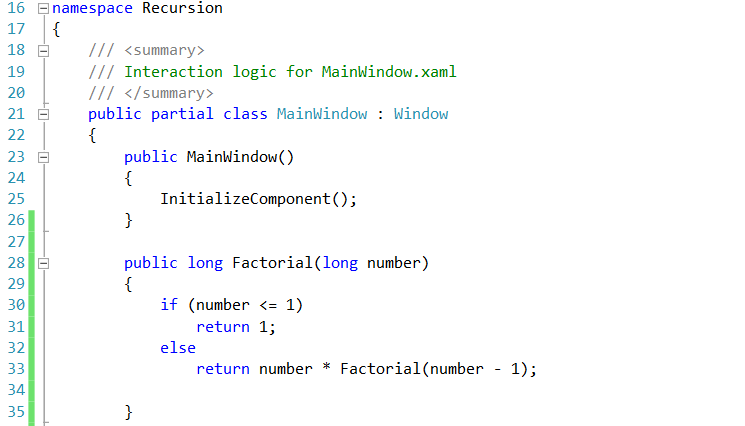
1. Drag across a text block and from the toolbox
   1. set the “Name” property to txtblk\_Result
   2. set the “Text” property to blank

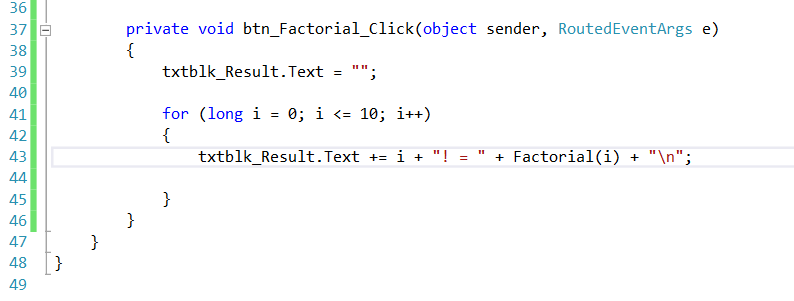


1. Double-click on the “Calculate Factorial” button to automatically create an event handler in the MainWindow.xaml.cs file, which is a method called btn\_Factorial\_Click



1. We are now going to write the code for this event handler and additional “helper” method called “Factorial”. The final code can be viewed in the screenshot below followed by a full explanation





Calculating factorials with a recursive method - code walkthrough

The recursive method Factorial (lines 28 - 35) first determines whether its terminating condition is true (i.e., number is less than or equal to 1)

If number is less than or equal to 1, factorial returns 1, no further recursion is necessary and the method returns

If number is greater than 1, line 33 expresses the problem as the product of number and a recursive call to Factorial, evaluating the factorial of number-1

Note that Factorial(number-1) is a slightly simpler problem than the original calculation Factorial(number)

Method Factorial receives a parameter of type long and returns a result of type long

because factorial values become large quickly

We choose data type long so the program can calculate factorials greater than 20!

Unfortunately, the Factorial method produces large values so quickly, even long does not help us print many more factorial values before the size of even the long variable is exceeded

Factorials of larger numbers require the program to use float and double variables

This points to a weakness in most programming languages, namely, that the languages are not easily extended to handle the unique requirements of various applications

Recursion vs. Iteration

In the previous sections, we studied a method that can be implemented either recursively or iteratively (loop)

In this section, we compare these two approaches and discuss why the programmer might choose one approach over the other

Both iteration and recursion are based on a control structure –

iteration uses a repetition structure (such as for, while or do/while)

and recursion uses a selection structure (such as if, if/else or switch)

Both iteration and recursion involve repetition - iteration explicitly uses a repetition structure and recursion achieves repetition through repeated method calls

Iteration and recursion each involve a termination test - iteration terminates when the loop-continuation condition fails and recursion terminates when a base case is recognized

Iteration with counter-controlled repetition and recursion both gradually approach termination - iteration keeps modifying a counter until the counter assumes a value that makes the loop-continuation condition fail and recursion keeps producing simpler versions of the original problem until a base case is reached

Both iteration and recursion can execute infinitely - an infinite loop occurs with iteration if the loop-continuation test never becomes false and infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on a base case

Recursion has disadvantages as well

It repeatedly invokes the mechanism, and consequently the overhead, of method calls

This can be costly in both processor time and memory space

Each recursive call creates another copy of the method (actually, only the method’s variables); this can consume considerable memory

Iteration normally occurs within a method, so the overhead of repeated method calls and extra memory assignment is omitted