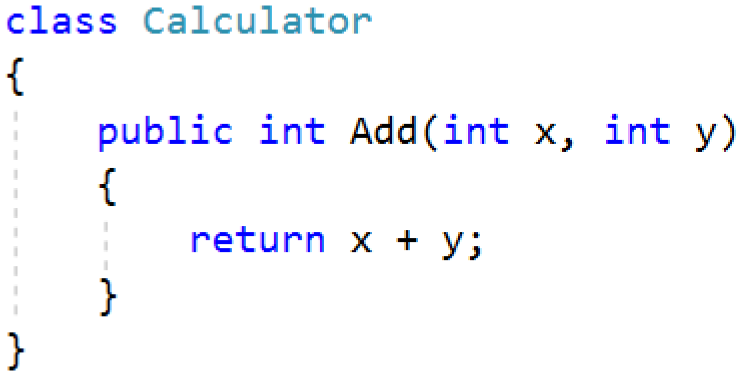
Module 5

Methods

The Syntax of a Method

- The syntax of methods are
  - ReturnValue MethodName(arguments) { MethodBody }

- All methods must exist inside of a class definition – no “global” methods!



```
class Calculator
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

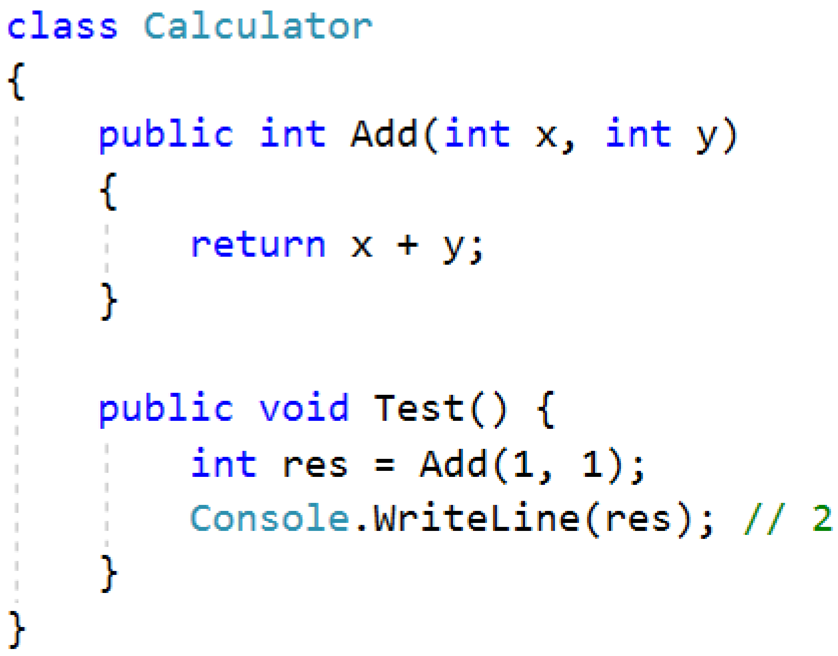
- Main() is a method that you already know
- WriteLine() is a method on the Console class

Local Variables

- Methods can declare local variables
  - Created during method invocation
  - Local to the method (i.e. “private”)
  - Exist only inside method and are destroyed on exit
- Classes can declare member variables
  - These exist for the lifetime of the class
  - Can be used for sharing data
- Local variables take precedence over member variables!

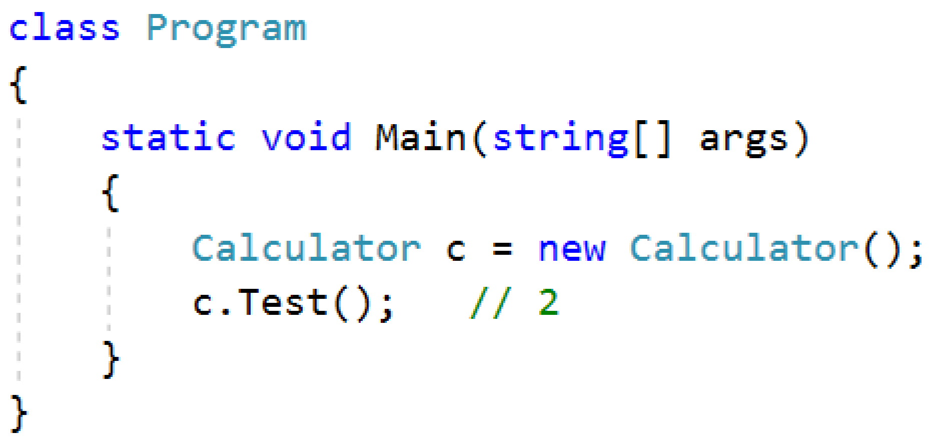
Invoking a Method

- You can invoke a method within the same class
- You can invoke a method within another class
  - must be visible to the outside, i.e. “public”



```
class Calculator
{
    public int Add(int x, int y)
    {
        return x + y;
    }

    public void Test() {
        int res = Add(1, 1);
        Console.WriteLine(res); // 2
    }
}
```

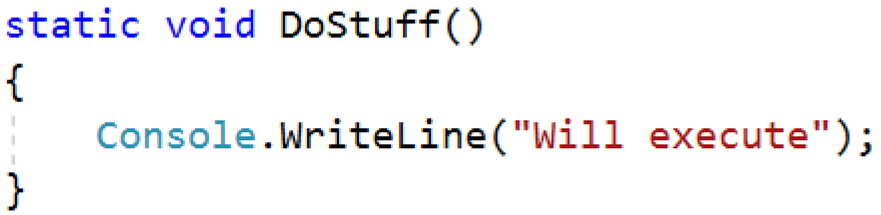


```
class Program
{
    static void Main(string[] args)
    {
        Calculator c = new Calculator();
        c.Test();   // 2
    }
}
```

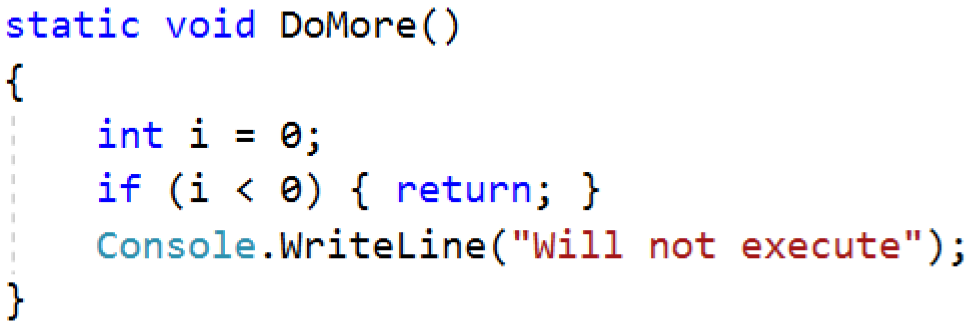
- You can invoke methods, which in turns invokes other methods etc. etc.
  - Call Stack Window in Visual Studio

Returning from a Method

- **The method returns**
  - When the method body has finished executing
  - When a return statement is executed



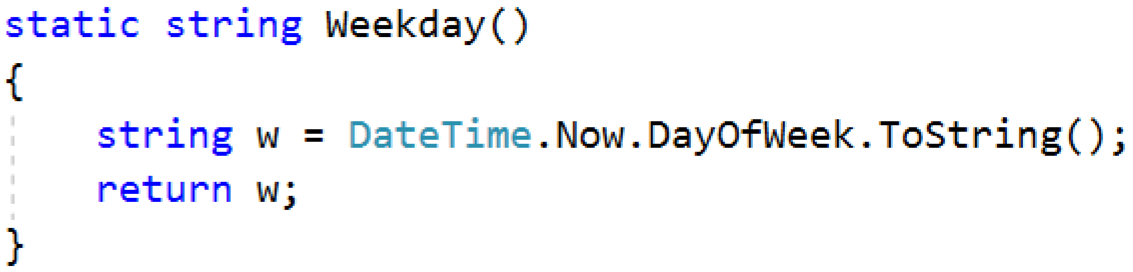
```
static void DoStuff()
{
    Console.WriteLine("Will execute");
}
```



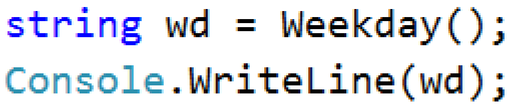
```
static void DoMore()
{
    int i = 0;
    if (i < 0) { return; }
    Console.WriteLine("Will not execute");
}
```

Returning Values from Methods

- Methods can return values if declared with a specific return type (i.e. not void)



```
static string Weekday()
{
    string w = DateTime.Now.DayOfWeek.ToString();
    return w;
}
```



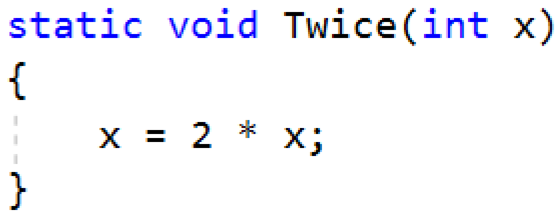
```
string wd = Weekday();
Console.WriteLine(wd);
```

- Values are returned with a return statement
- Must return a value of the specified return type!

- Return value is <u>copied</u> back
- Return value does not have to be used

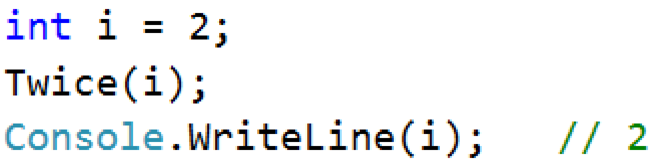
Passing Parameters by Value

- Define *formal* parameters within parentheses in method
  - Supply type and name for each parameter



```
static void Twice(int x)
{
    x = 2 * x;
}
```

- Invoke method by supplying *actual* parameters in parentheses
  - The formal and actual parameter types and count must be compatible

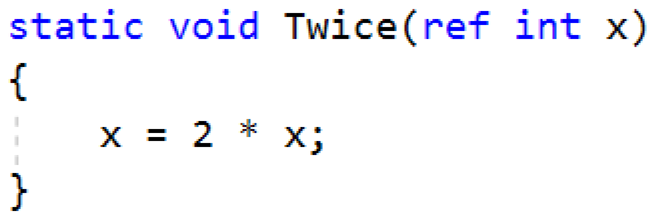


```
int i = 2;
Twice(i);
Console.WriteLine(i);   // 2
```

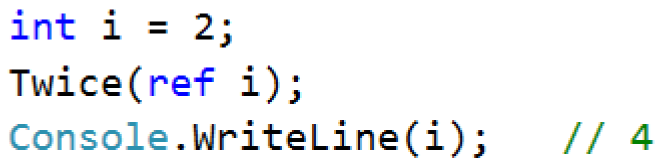
- Parameter values are copied from actual to formal
- **Changes made inside method has no effect outside method!**

The ref Modifier

- Reference parameters are references to memory locations, i.e. aliases for variables
- Use the ref modifier to pass variables by reference



```
static void Twice(ref int x)
{
    x = 2 * x;
}
```



```
int i = 2;
Twice(ref i);
Console.WriteLine(i);   // 4
```

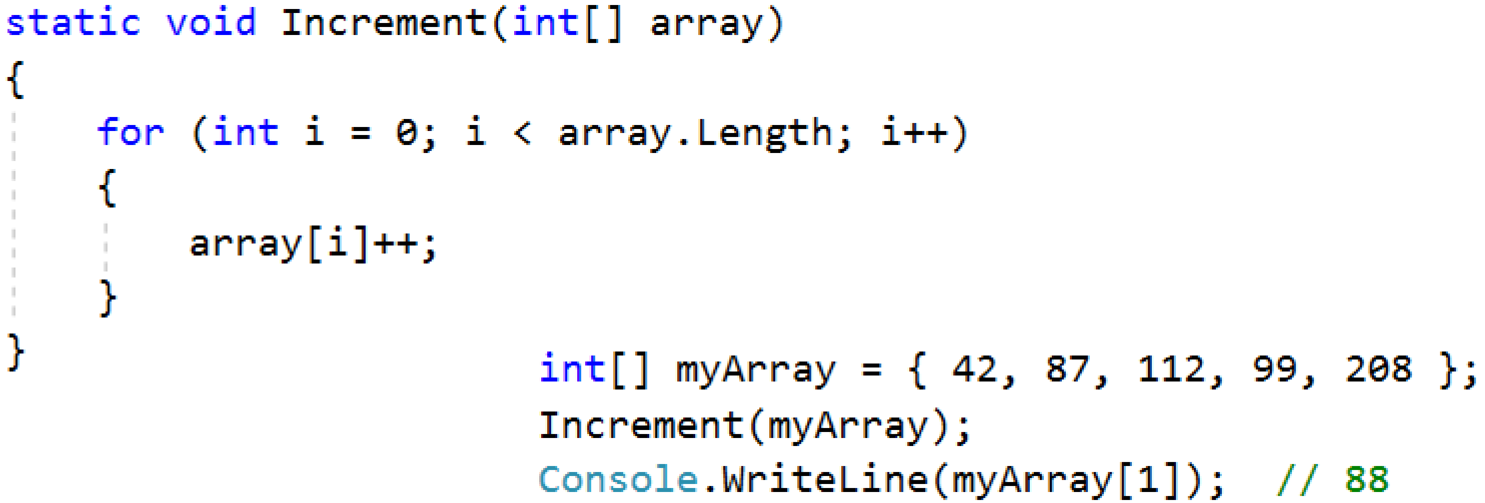
- Also use the ref keyword when invoking the method

- Parameter values are referred (or aliased)
- Changes made inside method has indeed effect outside method!
- Variable must be assigned before call

- Use the out modifier if value is not assigned before call

Methods and Reference Types

- Reference types can of course be passed to methods as well



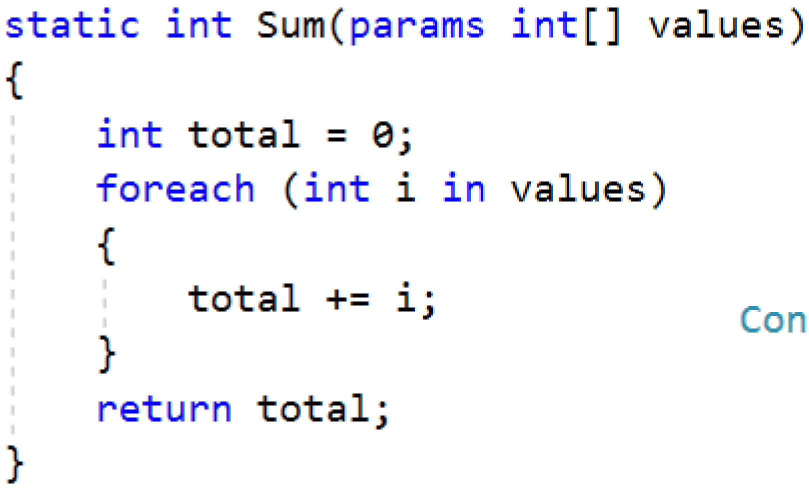
```
static void Increment(int[] array)
{
    for (int i = 0; i < array.Length; i++)
    {
        array[i]++;
    }
}
```

```
int[] myArray = { 42, 87, 112, 99, 208 };
Increment(myArray);
Console.WriteLine(myArray[1]);  // 88
```

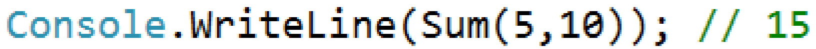
- What do you think happens here?

The params Modifier

- Passing parameter lists of varying length by using the params modifier



```
static int Sum(params int[] values)
{
    int total = 0;
    foreach (int i in values)
    {
        total += i;
    }
    return total;
}
```



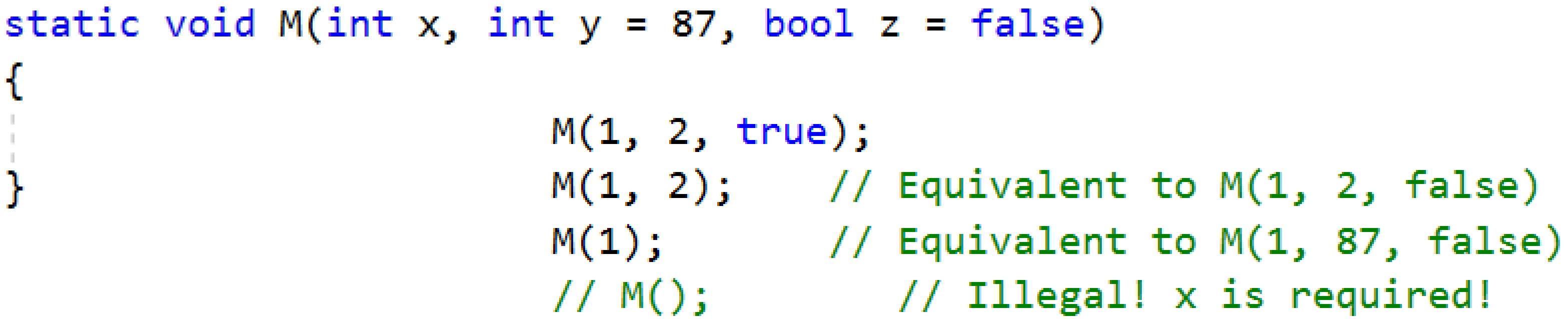
```
Console.WriteLine(Sum(5,10)); // 15
```

- Actual parameters are then passed into the method by value as an array

- Only one params per method

Optional Parameters

- Methods can have optional parameters by specifying their default values



```
static void M(int x, int y = 87, bool z = false)
{
}
```

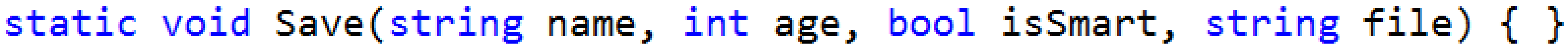
```
M(1, 2, true);
M(1, 2);     // Equivalent to M(1, 2, false)
M(1);        // Equivalent to M(1, 87, false)
// M();         // Illegal! x is required!
```

- Optional parameters can be omitted when invoking the method
- Note: Optional parameters must appear last in parameter list

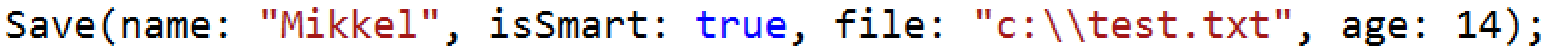
- Default values for optional parameters must be known at compile time!

Named Parameters

- Can pass parameter values using their *names* (as opposed to their *position*)



```
static void Save(string name, int age, bool isSmart, string file) { }
```

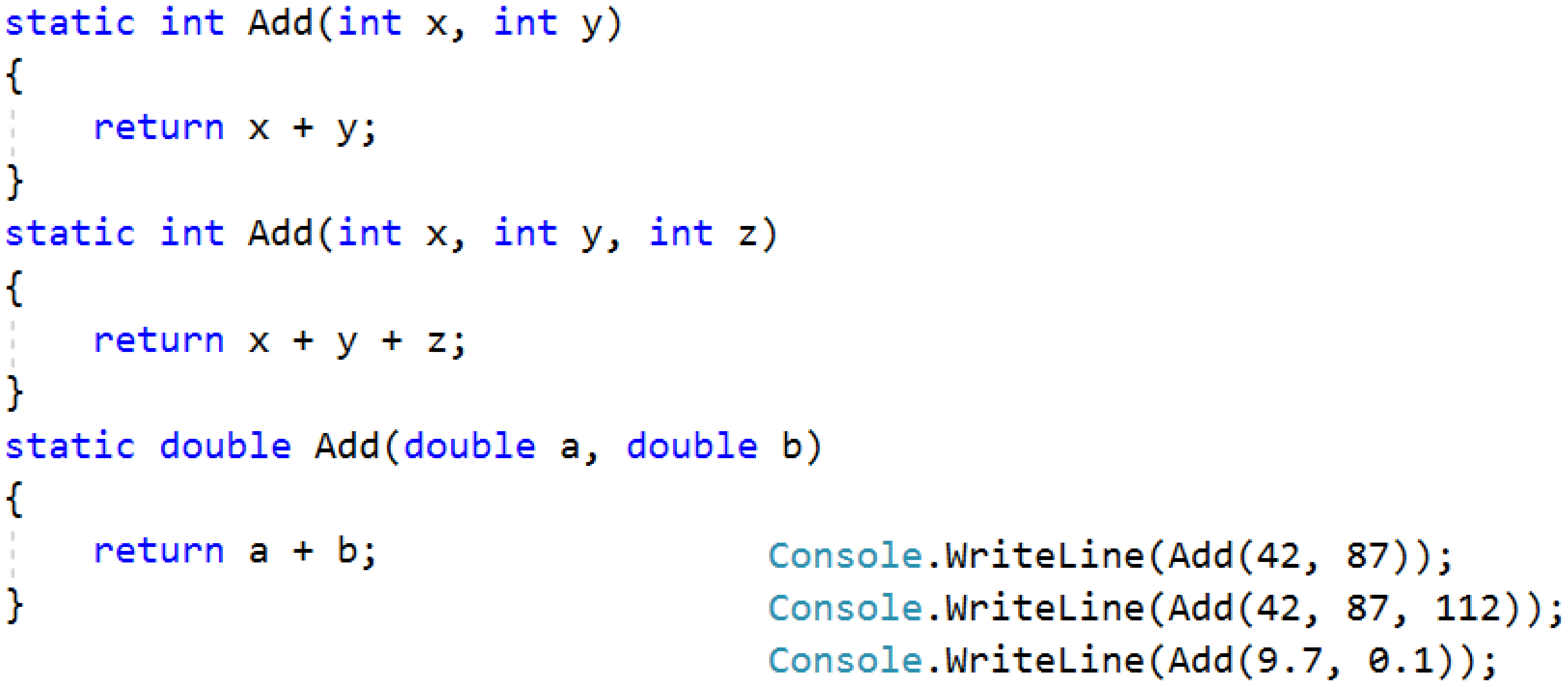


```
Save(name: "Mikkel", isSmart: true, file: "c:\\test.txt", age: 14);
```

- Note: Positional parameters <u>must always appear</u> before any named parameters when invoking methods!
- Named and optional parameters mix perfectly

Overloading Methods

- Methods can be overloaded
  - Same name for multiple methods within a class



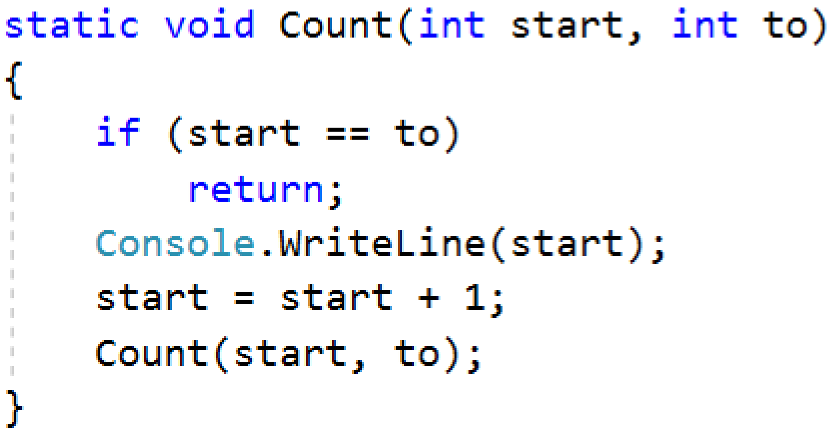
```
static int Add(int x, int y)
{
    return x + y;
}
static int Add(int x, int y, int z)
{
    return x + y + z;
}
static double Add(double a, double b)
{
    return a + b;
}
```

```
Console.WriteLine(Add(42, 87));
Console.WriteLine(Add(42, 87, 112));
Console.WriteLine(Add(9.7, 0.1));
```

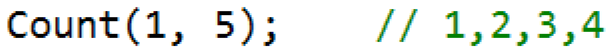
- Compiler chooses correct method to invoke

Recursive Methods

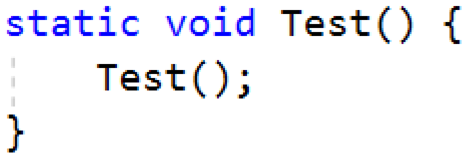
- Methods can call itself either directly or indirectly.
- Such methods are said to be *recursive*
- Perfect for solving inductively defined problems
- Must have terminating base clause
- Use with care!



```
static void Count(int start, int to)
{
    if (start == to)
        return;
    Console.WriteLine(start);
    start = start + 1;
    Count(start, to);
}
```



```
Count(1, 5);    // 1,2,3,4
```



```
static void Test() {
    Test();
}
```