CS620c

Recursion, methods calling themselves

Joe Duffin

Eolas Building Room 1.45

Email: Joseph.Duffin@nuim.ie

Recursion

- When a method contains a reference to itself (a call to itself) this is called ***recursion***.
- Methods which have this feature are then called ***recursive*** methods. There are two categories of recursion.
  - ***Direct*** recursion, where the method contains a reference to itself directly.
  - ***Indirect*** recursion where the method has a reference (call to) other methods that eventually have a reference to (call to) the original method. E.g. **method-a** calls **method-b** and then **method-b** calls **method-a**
- We will focus only on ***Direct*** recursion.

Recursion : further details

Any program that can be implemented recursively can be implemented iteratively (with loops). Although some times a program written iteratively can be difficulty to follow compared to the recursive implementation.

A recursive computation solves a problem **by using the solution of the same problem with simpler values. (recursive step)**

For recursion to terminate, there must be special cases for the simplest values. **(base or anchor case)**

Using Recursion in Java

- Java uses a **stack** as part of its internal workings (remember what happens when we call a method? a **stack frame** is created and existed for the duration of the method, remember the return statement or the ending brace "}" terminates a method).

- Java makes use of this **stack** when we write recursive functions (and, as you know for all methods)

- Java creates a **stack frame** representing a call to the method for each and every call of that method (remember the stack frame has its own local variables and formal variables for storage, remember what we said about its ***life time***).

Recursion : (Basic problem definitions)

The definition of the recursive methods that we will see in this lesson consists of two parts.

1) An **anchor** or **base case** in which the method value is specified for one or more known values of the input parameters.

2) A **recursive step or inductive** in which the action to be taken for the current value of the parameter is defined in terms of previously defined values.

Shortly we will have a look at two functions, the **factorial** function and the **sigma** function that we saw previously. In these lessons we wrote these functions as **iterative** methods. (method which use loops to implement the algorithms)

Recursive Functions (they return values)

- We will use some simple recursive functions to see how recursion works

- Factorial(n) = n * Factorial (n-1) (*for n>1*) **// recursive step**
  - Factorial(1) = 1 **// base case or anchor**
  - Factorial(0) = 1

- Sigma (n) = n + Sigma (n-1) (*for n>0*) **// recursive step**
  - Sigma (0) = 0 **// base case or anchor**

Other examples include, Fibonacci series, Towers of Hanoi, searching, sorting and many other areas

Recap: The Factorial Function

- What is a “Factorial” function: Our factorial function is one that takes a positive integer value **n** and multiplies all the number from 1 up to **n** to get a result which is termed the factorial of **n**. **By definition Factorial of zero is equal to 1.**
    - **n=0 then Factorial(0) =1**
    - n=1 then Factorial(1) = 1
    - n=2 then Factorial(2) = 2 * 1
    - n=4 then Factorial(4) = 4 * 3 * 2 * 1

```
// iterative implementation of Factorial
int result = 1;
for (int i=1; i<=n; i++){

      result = result * i;
}
```

Want to know more? visit: http://mathforum.org/dr.math/faq/faq.0factorial.html

The **Iterative** factorial method

```
public static int factorial(int n) {

   int result = 1;

   for (int i=1; i<=n; i++){

      result = result * i;
   }

    return result;
}
```

Iterative factorial method means the factorial method using loops

Using the factorial method

- Now, the rest of my Java program can use this factorial() method.

– Such as the main() method or in any other methods we write

```
public static void main(){
...

 int x = factorial(6);


..
}
```

What happens when we call the method factorial?

1 A memory resource called a **stack frame on the run time stack** is created for the call to the method : `int result = factorial(5);`

2 Memory space is set aside within this **stack frame** for the formal parameter **int n**

3 The **return point or address** is stored in the **stack frame**. This is the point at which the program continues to execute after the method is finished running.

4 The value in the actual parameter (**5**) is **copied** into the formal parameters **n**.

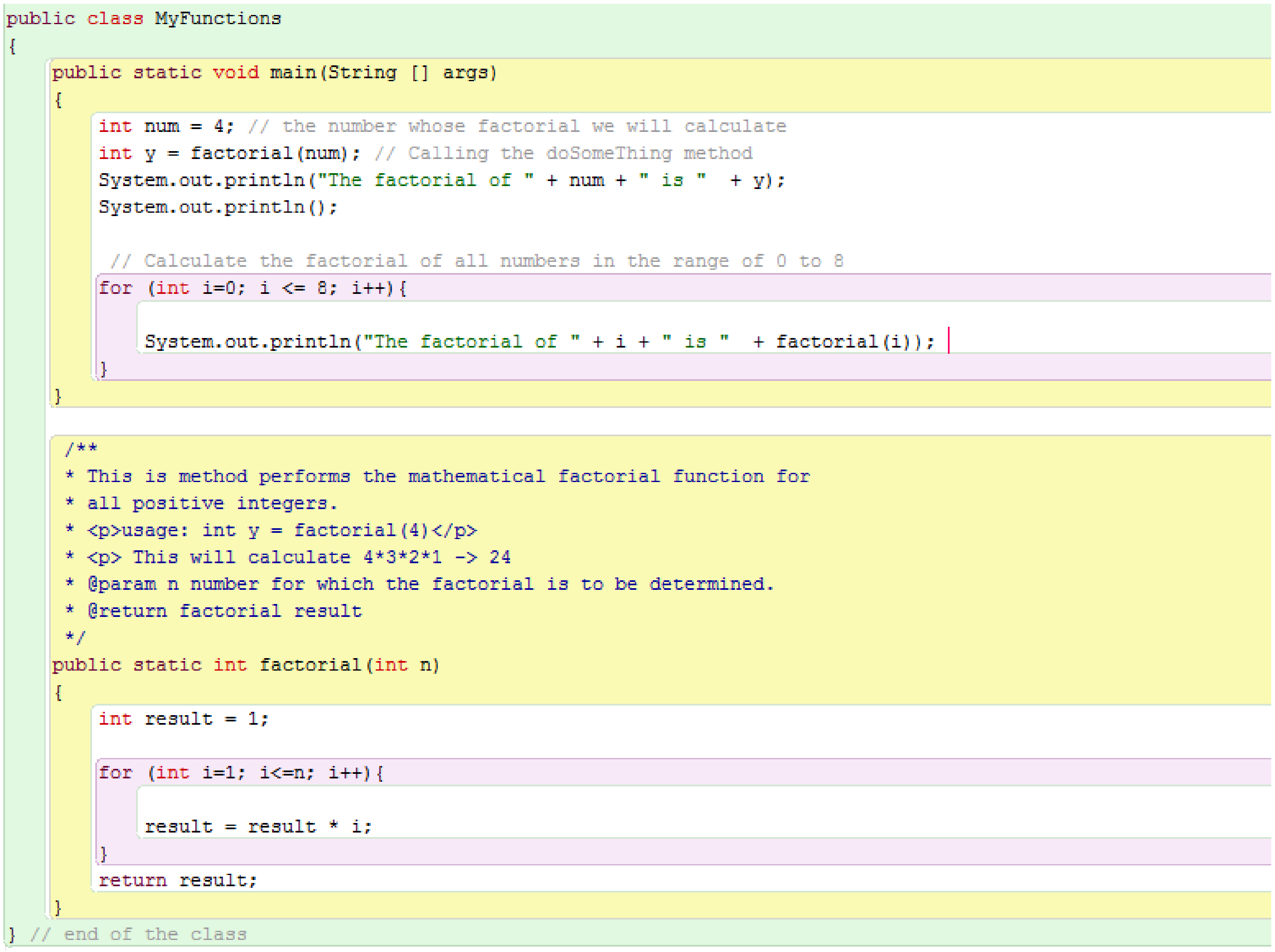
5 Space is set aside for the local variable **int result** and it is assigned the value **1.**

6 The code statements, which in this case is the for loop in the body of the method is now executed. The first thing that happens in this case is a local variable is declared **int i** whose scope is only for the duration of the for loop. (See scope rules earlier)

7 When the loop is finished executing, the variable **result** will have the calculated factorial result and this value will be **returned** to the point at which the method factorial was called (**variable x** is given the returned value). **The flow of control of the program returns to this point using the return address stored in the stack frame.**

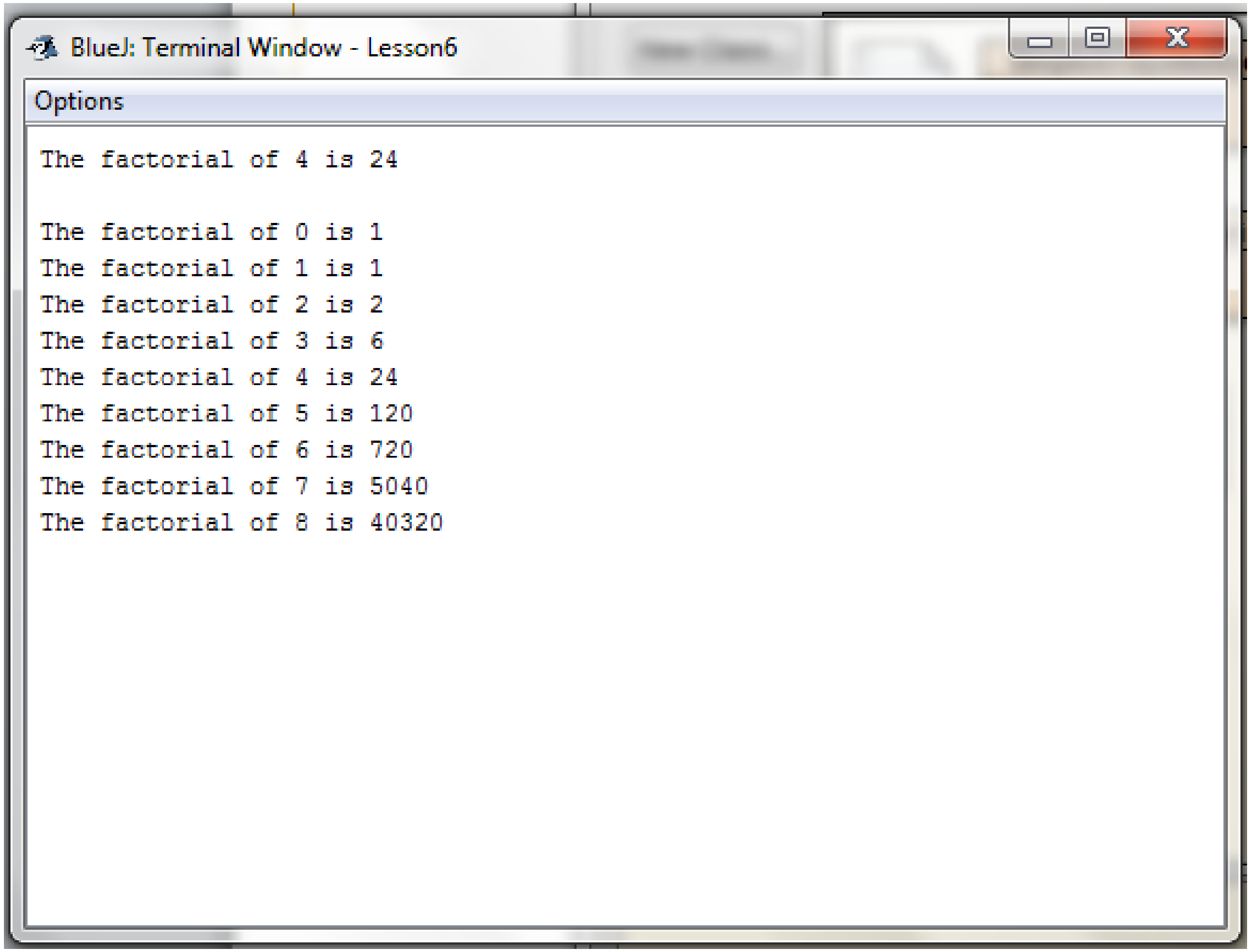
8 Finally the stack frame is "destroyed" or returned to the system to be reused. The Formal parameters int n as well as the local variable int result are no longer available in your program

The factorial method defined in the class MyFunctions



```java
public class MyFunctions
{
    public static void main(String [] args)
    {
        int num = 4; // the number whose factorial we will calculate
        int y = factorial(num); // Calling the doSomeThing method
        System.out.println("The factorial of " + num + " is "  + y);
        System.out.println();

         // Calculate the factorial of all numbers in the range of 0 to 8
        for (int i=0; i <= 8; i++){

            System.out.println("The factorial of " + i + " is "  + factorial(i));
        }
    }

    /**
     * This is method performs the mathematical factorial function for
     * all positive integers.
     * <p>usage: int y = factorial(4)</p>
     * <p> This will calculate 4*3*2*1 -> 24
     * @param n number for which the factorial is to be determined.
     * @return factorial result
     */
    public static int factorial(int n)
    {
        int result = 1;

        for (int i=1; i<=n; i++){

            result = result * i;
        }
        return result;
    }
} // end of the class
```

Output of main method calling the factorial method in a loop



```
The factorial of 4 is 24

The factorial of 0 is 1
The factorial of 1 is 1
The factorial of 2 is 2
The factorial of 3 is 6
The factorial of 4 is 24
The factorial of 5 is 120
The factorial of 6 is 720
The factorial of 7 is 5040
The factorial of 8 is 40320
```

Recursive myFactorial

- The structure for factorial is defined as follows, remember you have a base case and a recursive call. **(let’s break this down)**

```
public static int myFactorial(int x){
        if (x <= 1) { // base case
              return 1;}
        else
             return x * myFactorial(x-1);
    }
```

This is the recursive call

Base case: factorial(0) & factorial (1)

- ALWAYS start by writing the base or termination condition. When the method does **NOT** call itself!
- Remember 0! and 1! are both equal to 1

```
public static int myFactorial(int x){
        if (x <= 1) { // base case
            return 1;
        }
      ................
}
```

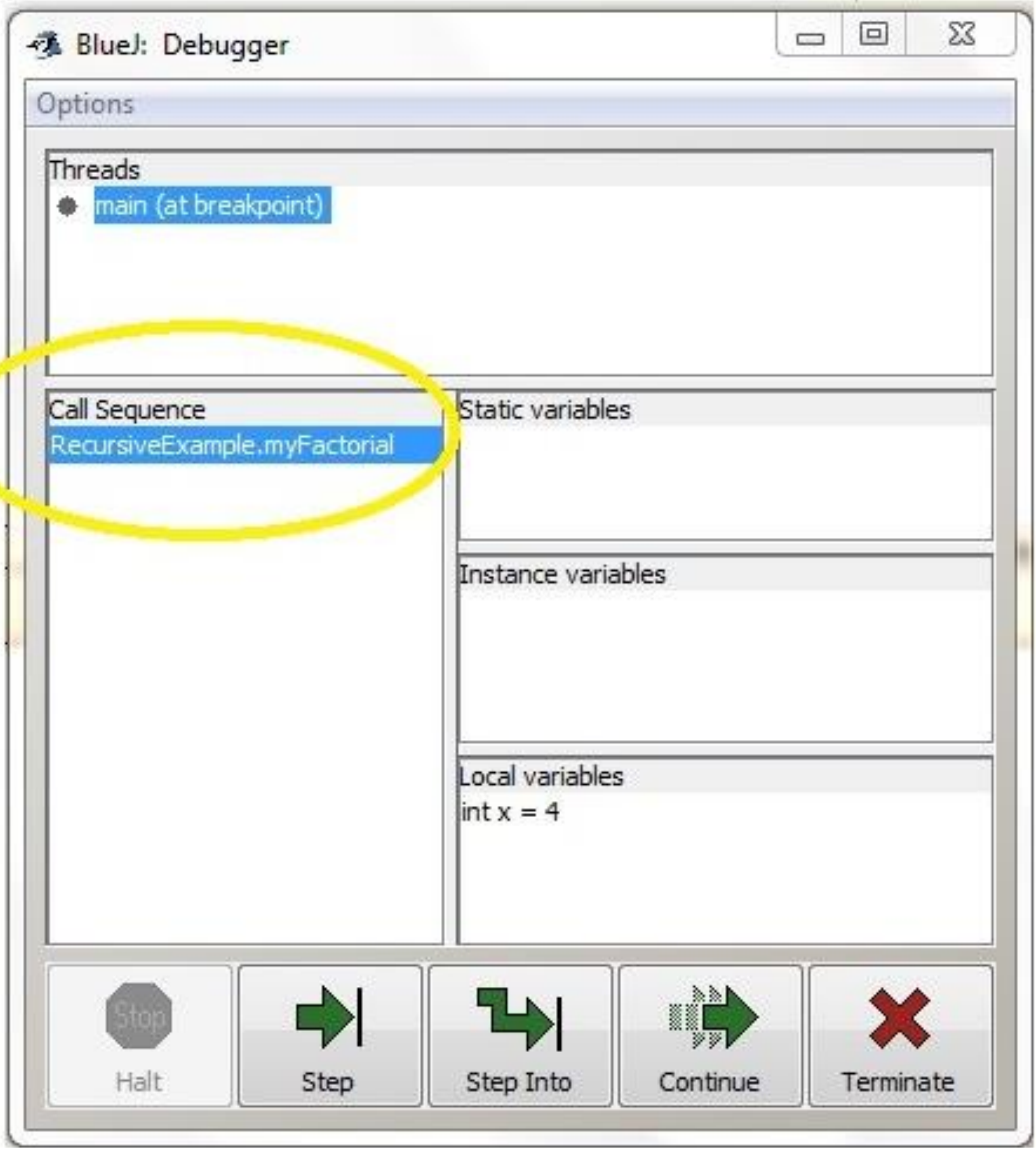
Recursive step: factorial(n)

- Make sure the termination condition works before proceeding.
- Now we can extend this by including the recursive call.

```
public static int myFactorial(int x){

      if (x <= 1) { // base case 0! 1!
          return 1;
      }
      else{ // recursive step
         return (x * myFactorial(x-1));
      }
    }
```

Recursion and the Stack

- We can see the calls to `myFactorial` added to the Stack!
- At one point this stack has 4 calls
  - From myFactorial(4)
- Luckily that is Java's problem and not ours.


BlueJ: Debugger

Options

Threads
● main (at breakpoint)

Call Sequence
RecursiveExample.myFactorial

Static variables

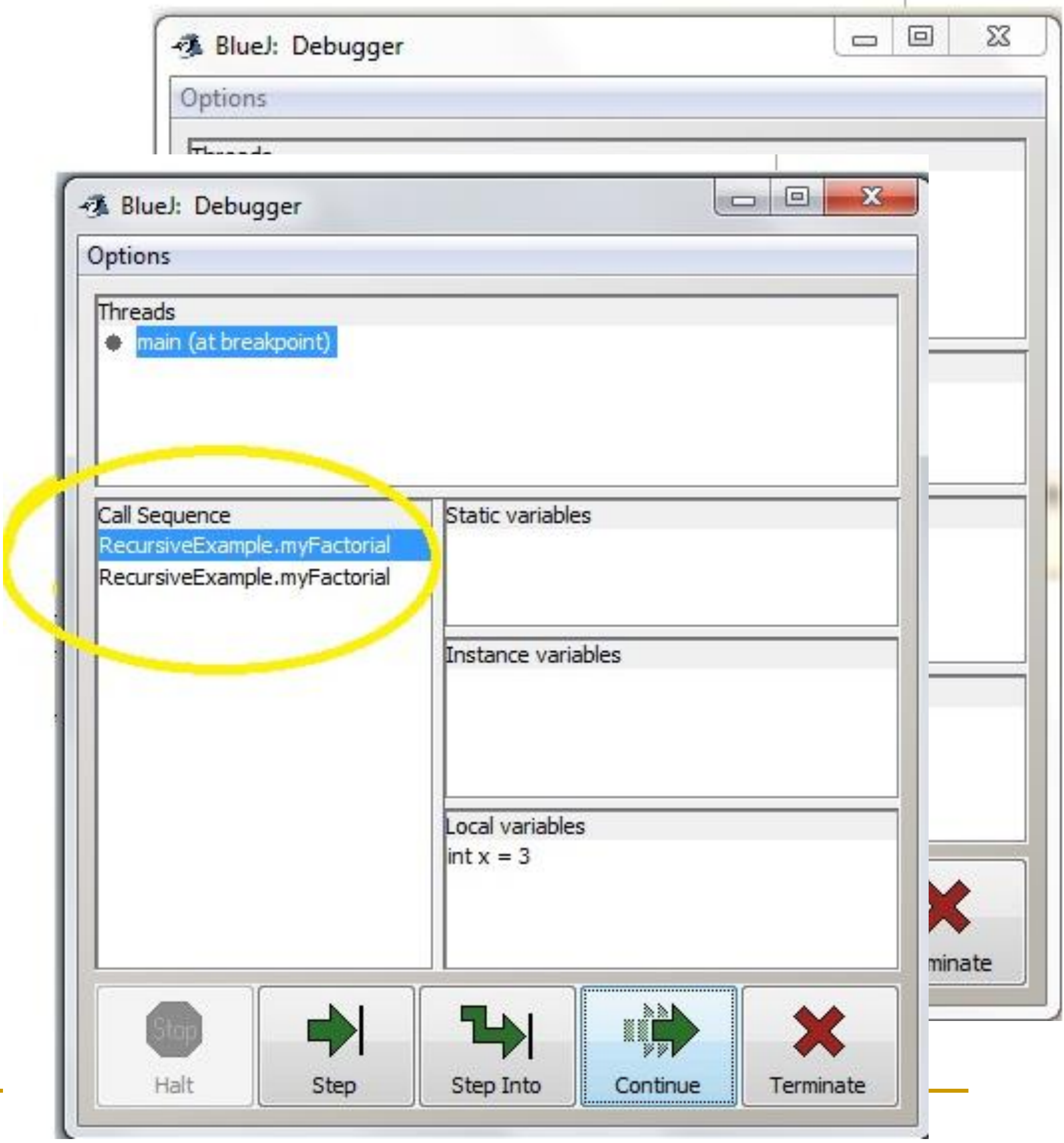
Instance variables

Local variables
int x = 4

Recursion and the Stack

- Now the **stack** holds TWO "instances of stack frames" of myFactorial
  - Waiting to finish

You can consider the **stack** as being the Call Sequence pane in BlueJ.



Call Sequence
RecursiveExample.myFactorial
RecursiveExample.myFactorial

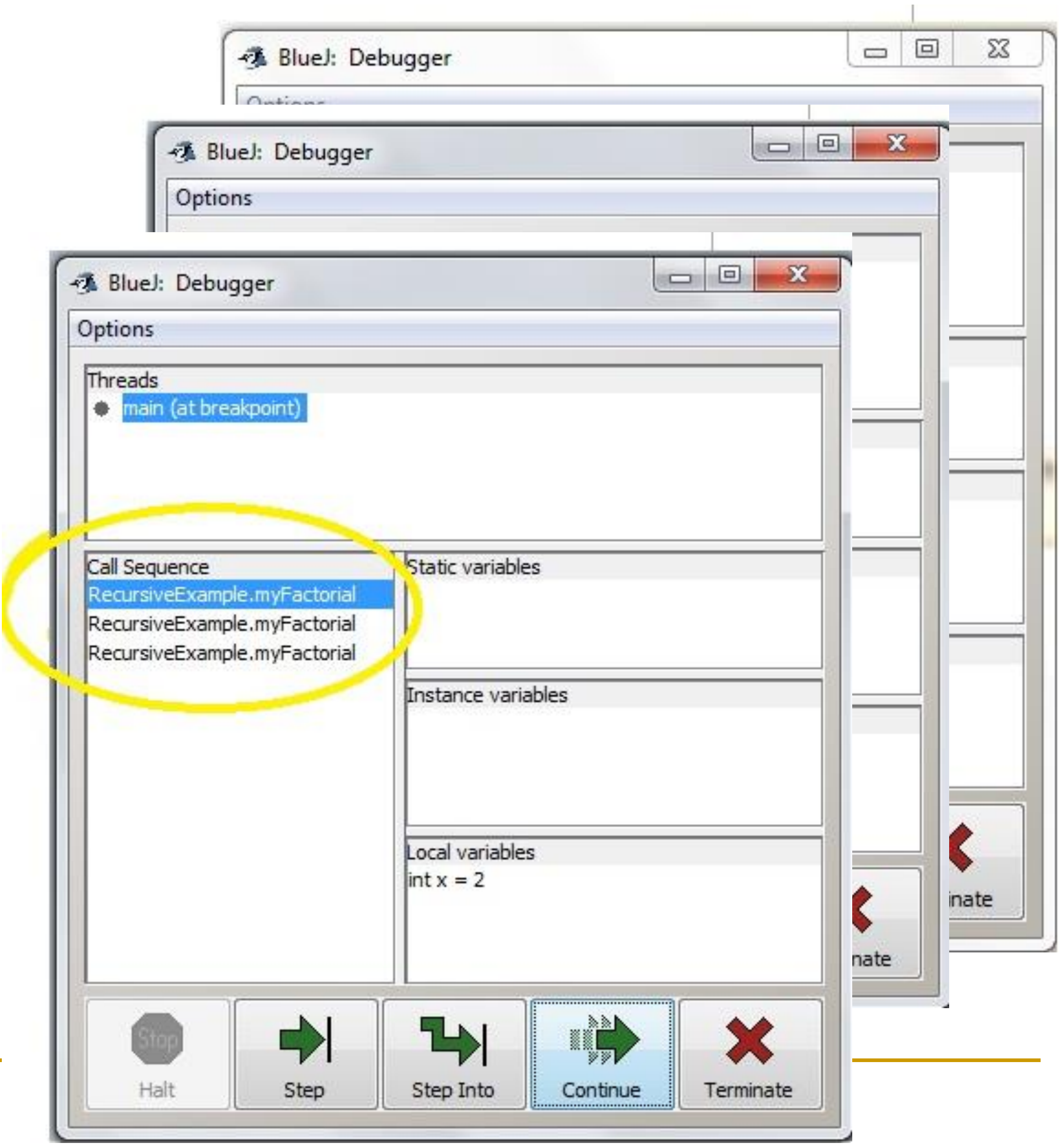
Static variables

Instance variables

Local variables
int x = 3

Recursion and the Stack

- Now the stack holds THREE "stack frame instances" of myFactorial
  - Waiting to finish



BlueJ: Debugger

Options

Threads
main (at breakpoint)

Call Sequence
RecursiveExample.myFactorial
RecursiveExample.myFactorial
RecursiveExample.myFactorial

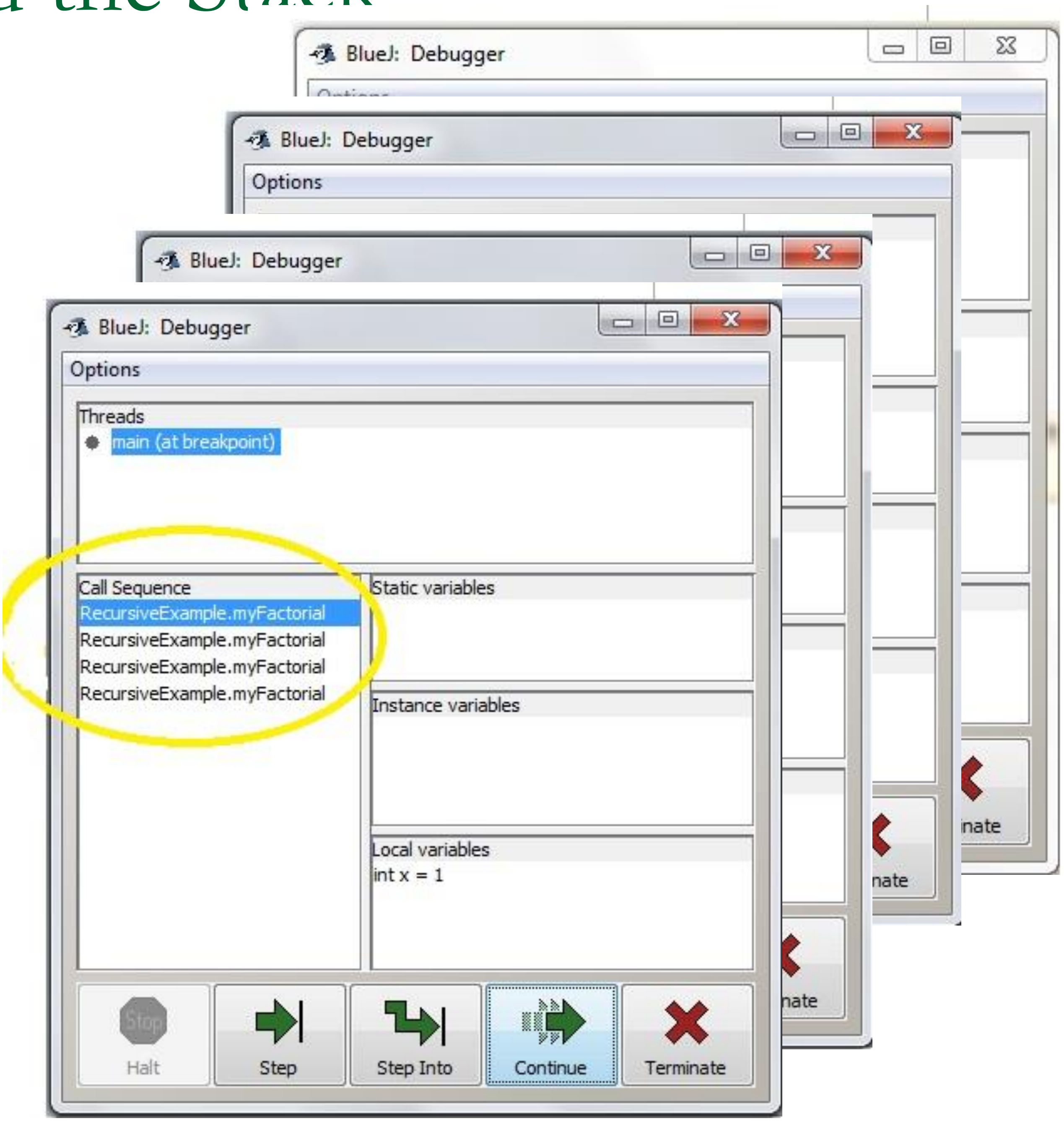
Static variables

Instance variables

Local variables
int x = 2

Recursion and the Stack

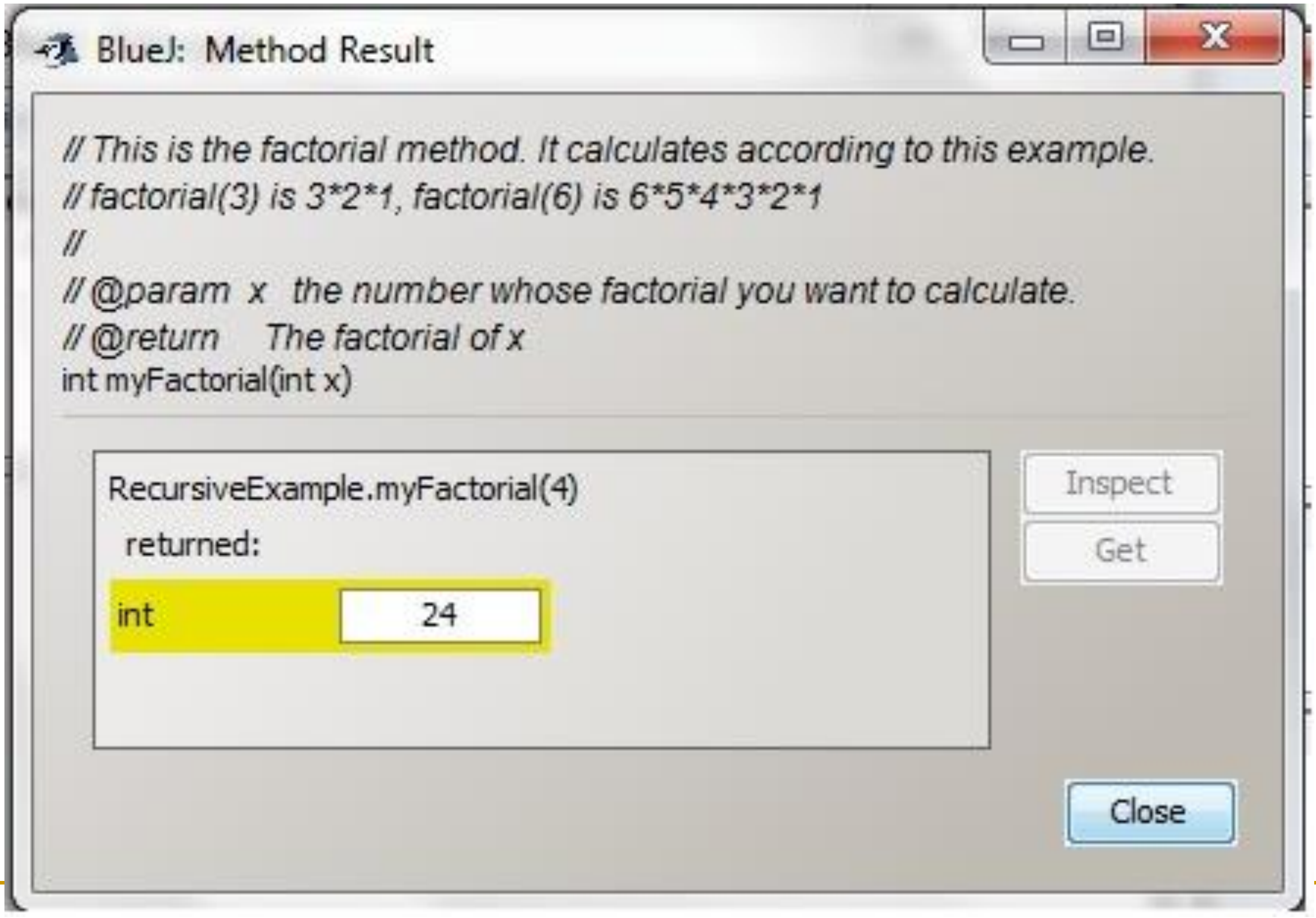
- Now the stack holds FOUR “stack frame instances” of myFactorial
  - Waiting to finish
- Notice that myFactorial finishes when x=1



```
RecursiveExample.myFactorial
RecursiveExample.myFactorial
RecursiveExample.myFactorial
RecursiveExample.myFactorial
```

```
int x = 1
```

BlueJ out put after the **recursive sequence** has finished.



```
// This is the factorial method. It calculates according to this example.
// factorial(3) is 3*2*1, factorial(6) is 6*5*4*3*2*1
//
// @param  x   the number whose factorial you want to calculate.
// @return     The factorial of x
int myFactorial(int x)
```

```
RecursiveExample.myFactorial(4)
 returned:
int        24
```

Calculating factorial(4)

So to calculate recursive **factorial** (4).

```
factorial (4) = 4 * factorial( 3 )
              = 4 * (3 * factorial( 2 ))
              = 4 * (3 * ( 2 * factorial( 1 )))
              = 4 * (3 * ( 2* 1 ))

              = 4 * ( 3 * 2 )
              = 4 * 6
              = 24
```

Write a recursive function named printRecursion (10 minutes)

1. Your recursive method will be named ***printRecursion*** and it will have the following method signature:

```
    public static void printRecursion( int num )
```

1. Your method will take in the positive integer parameter num and will use it to print out the message “Recursion in action” a **number of times determined by the value of num**.

2. Because it is a direct recursive method it will contain a **call to itself** in its body.

3. The non-recursive section of code will do nothing when the value of num is less than 1.

4. Make a call to this method passing it the value of 4. Can you visualise how this program will execute. (look back at recursive factorial example)

The **iterative** Sigma function **(Q8 lab 2)**

- What is the “Sigma” function
  - Sigma(0) = 0   // future base or anchor case
  - Sigma(1) = 1
  - Sigma(2) = 1 + 2
  - Sigma(3) = 1 + 2 + 3
  - Sigma(4) = 1 + 2 + 3 + 4

```
// iterative sigma function
int result = 0;
for (int counter=1; counter<=n; counter++){
       result = result + counter;
}
```

Using sigma

- Now, the rest of my Java program can use this `sigma()` method.
  - Such as the `main()` method or any other methods we write

```
Public static void main(){
  ...
     int result = sigma(num);
}
```

Recursive Sigma(n)

```
public static int mySigma(int x){

     if (x <= 0) { // base case
         return 0;}
     else{          // recursive step

         return (x + mySigma(x-1));
     }
  }
```

This is the recursive call

Base case: dealing with Sigma(0)

- ALWAYS start by writing the termination condition. When the method does **NOT** call itself!
- Sigma(0) is zero.

```
public static int mySigma(int x){
        if (x <= 0) { // base case
            return 0;
         }
           ……………. .
}
```

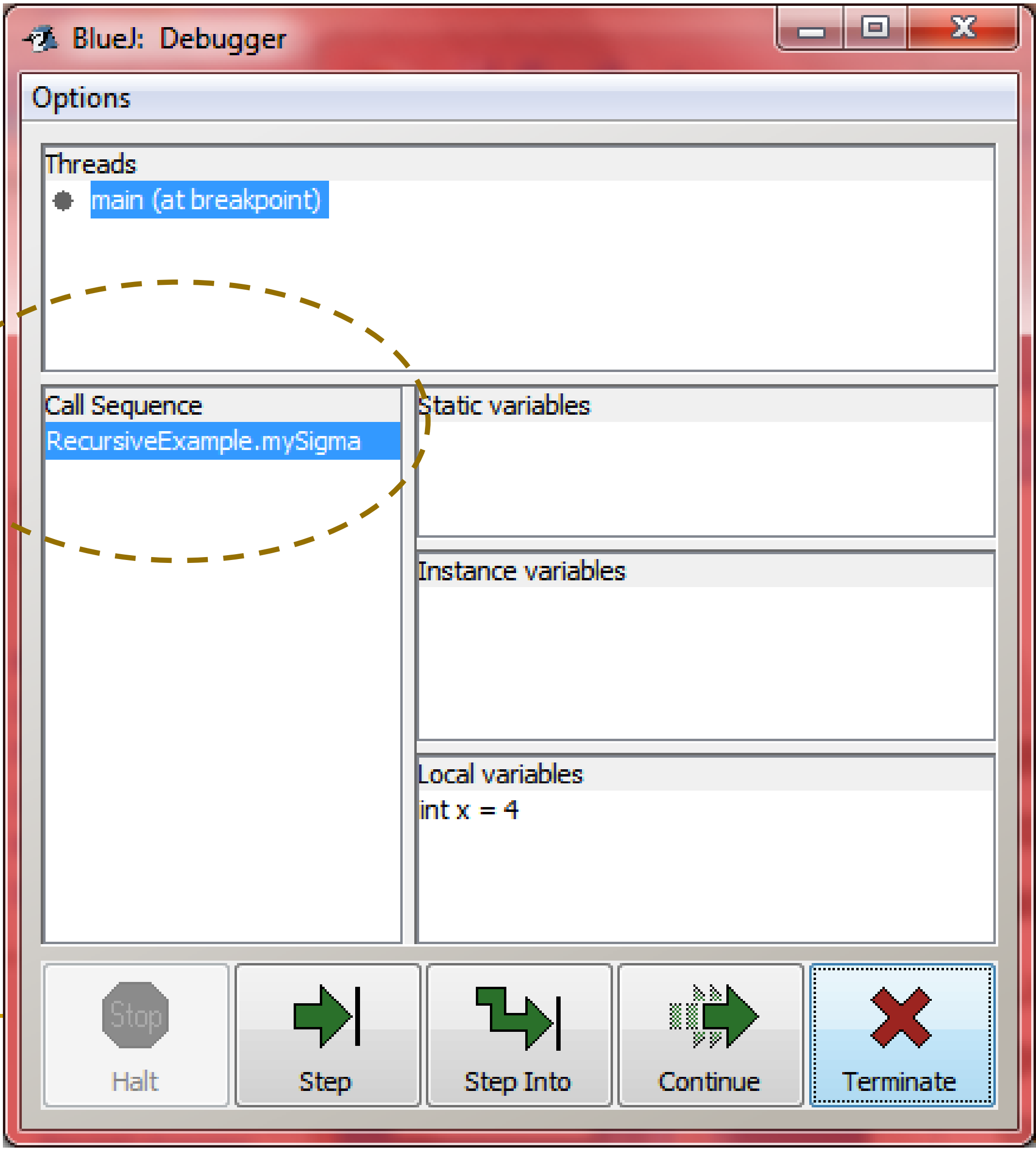
Recursive step: Sigma(n)

- Make sure the termination condition works before proceeding.
- Now we can extend this by including the recursive call.

```
public static int mySigma(int x){

    if (x <= 0) { // base case
        return 0;}
    else{             // recursive step

        return (x + mySigma(x-1));
    }
  }
```

Recursion and the Stack

- We can see the calls to mySigma being added to the Stack!
- At one point this stack has 4 calls
  - From mySigma(4)



BlueJ: Debugger

Options

Threads

main (at breakpoint)

Call Sequence

RecursiveExample.mySigma

Static variables

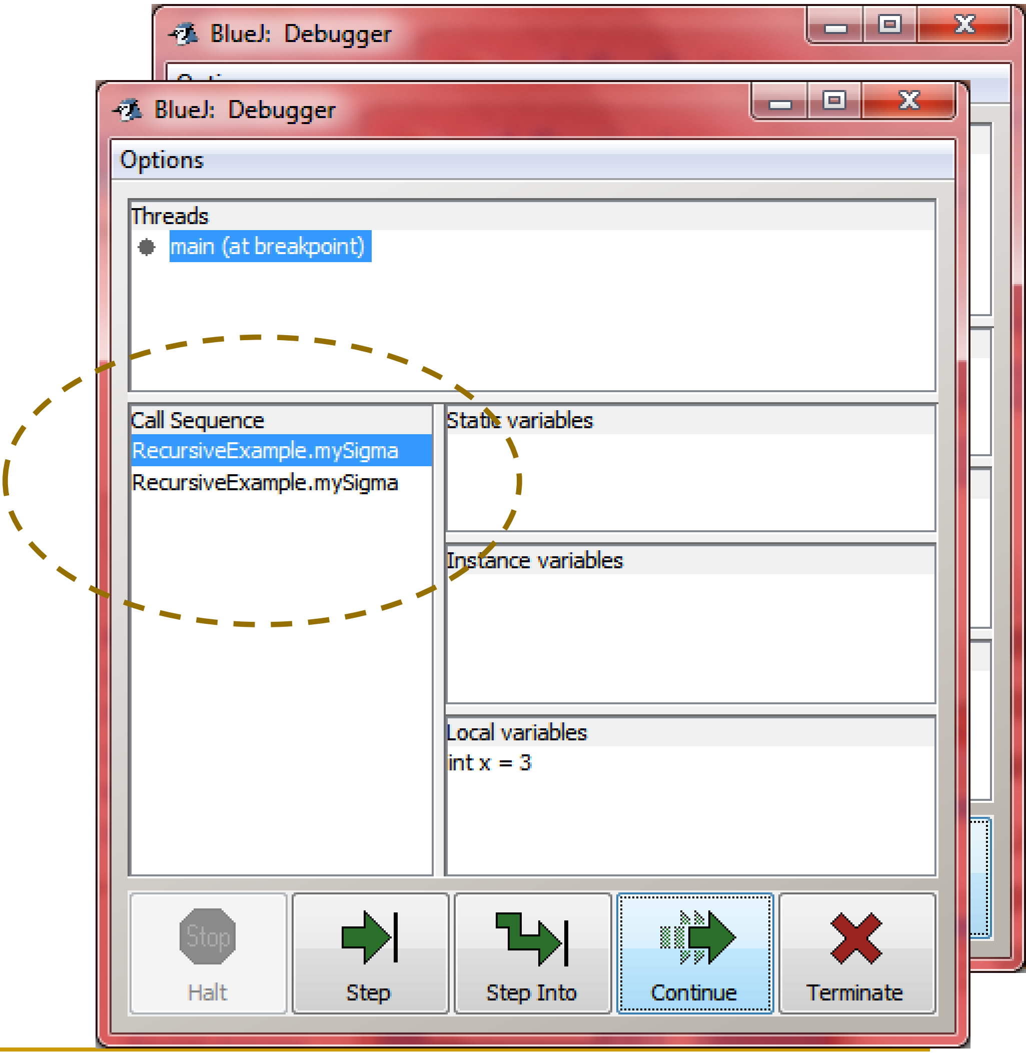
Instance variables

Local variables

int x = 4

Recursion and the Stack

- Now the stack holds TWO "stack frame instances" of mySigma
  - Waiting to finish

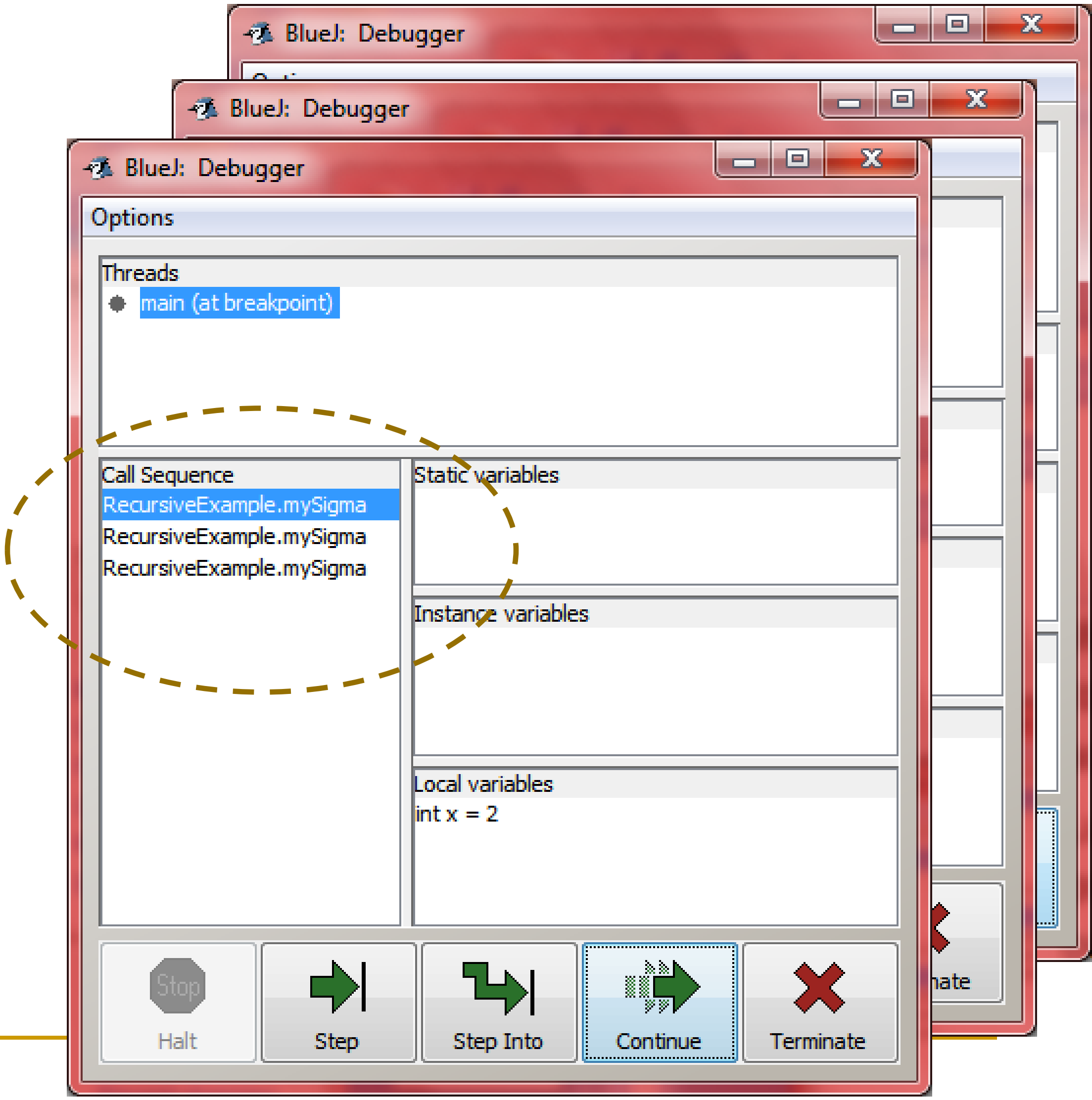


```
RecursiveExample.mySigma
RecursiveExample.mySigma
```

```
int x = 3
```

Recursion and the Stack

- Now the stack holds THREE "stack frame instances" of mySigma
  - Waiting to finish

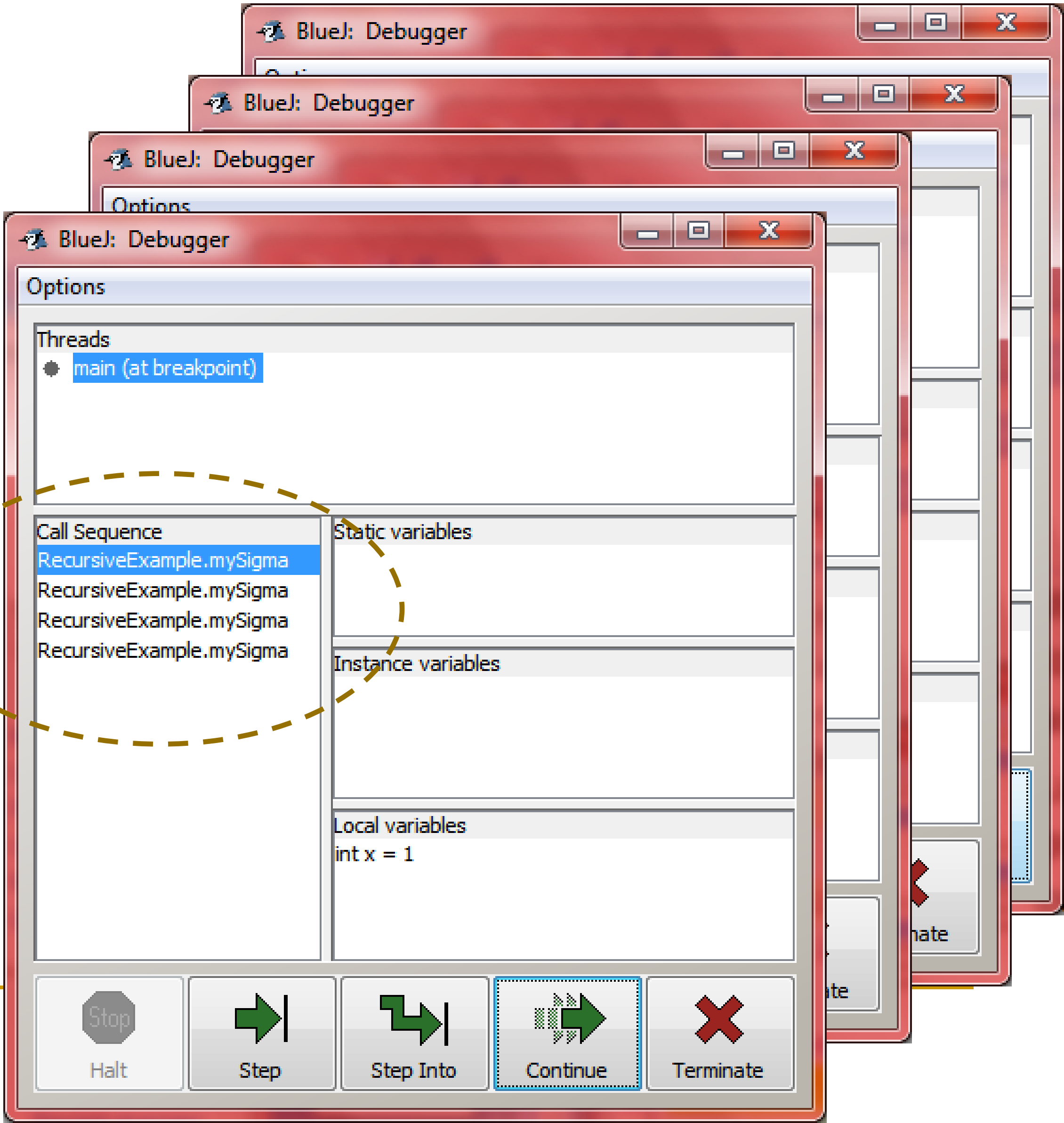


RecursiveExample.mySigma
RecursiveExample.mySigma
RecursiveExample.mySigma

int x = 2

Recursion and the Stack

- Now the stack holds FOUR “stack fram instances” of mySigma
  - Waiting to finish
- Notice that mySigma finishes when x=1



Threads
main (at breakpoint)

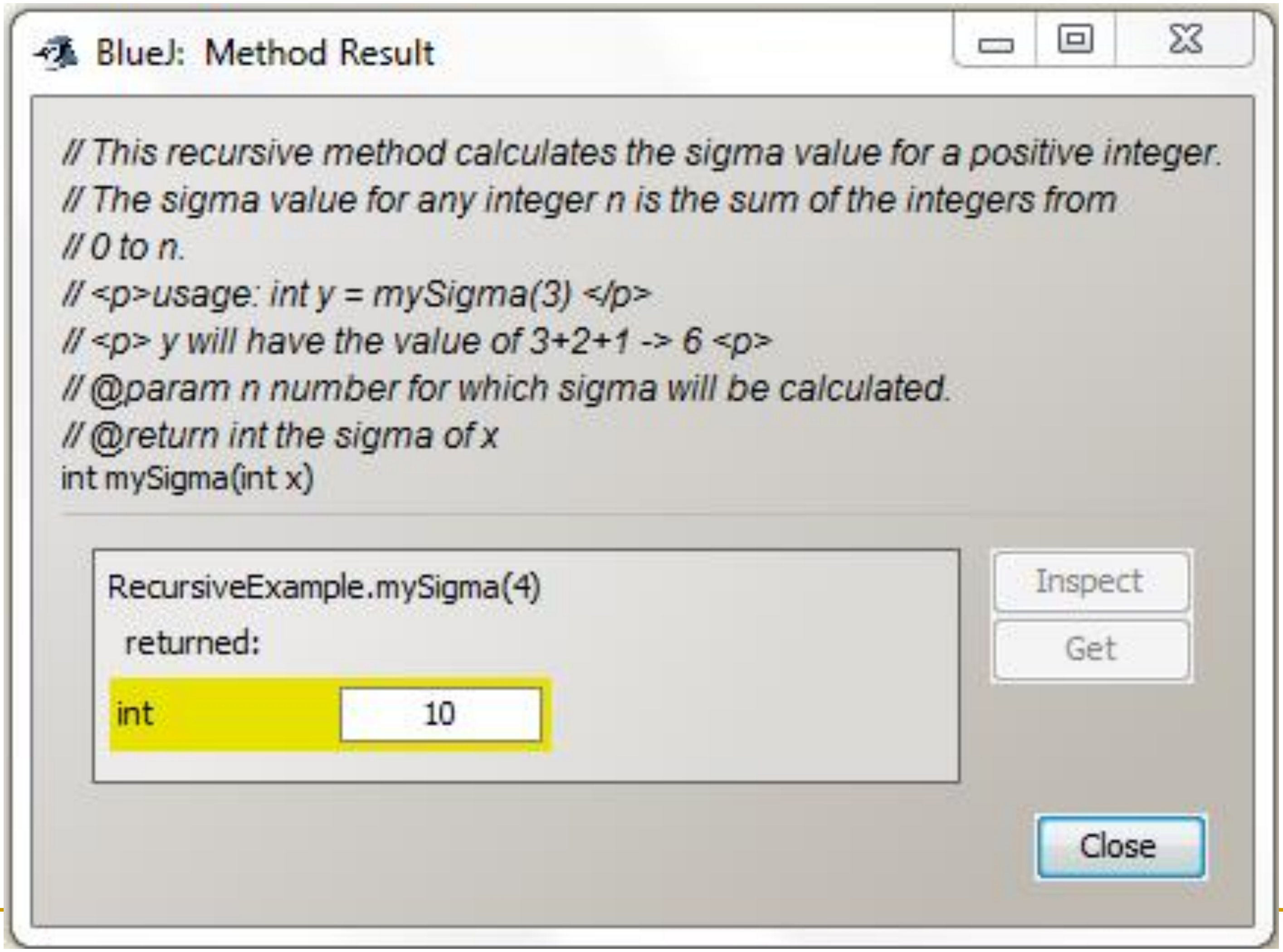
Call Sequence
RecursiveExample.mySigma
RecursiveExample.mySigma
RecursiveExample.mySigma
RecursiveExample.mySigma

Static variables

Instance variables

Local variables
int x = 1

BlueJ output for the mySigma(4)



```
// This recursive method calculates the sigma value for a positive integer.
// The sigma value for any integer n is the sum of the integers from
// 0 to n.
// <p>usage: int y = mySigma(3) </p>
// <p> y will have the value of 3+2+1 -> 6 <p>
// @param n number for which sigma will be calculated.
// @return int the sigma of x
int mySigma(int x)
```

```
RecursiveExample.mySigma(4)
 returned:
int        10
```

Calculating sigma(4)

So to calculate recursive **sigma**(4)

sigma(4) = 4 + sigma(3)

= 4 + (3 + sigma(2))

= 4 + (3 + (2 + sigma(1)))

= 4 + (3 + (2 + (1 + sigma(0)))

= 4 + (3 + (2 + (1 + 0)))

= 4 + (3 + (2 + 1)

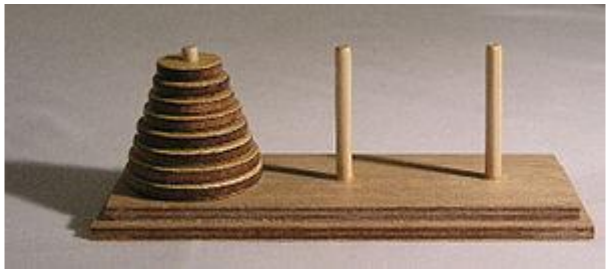
= 4 + (3 + 3)

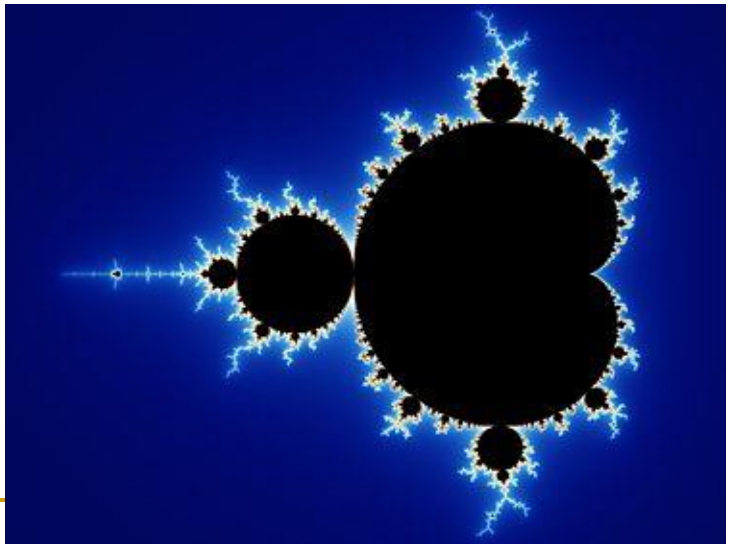
= 4 + 6

= 10

Other naturally recursive problems

- Many other tasks can use recursion
  - searching, sorting, series, natural structures...
  - Fractal graphics are heavily dependent on recursion


Towers of Hanoi


Mandelbrot Set (Fractal)

Recursion : (the negatives)

The Negative aspects of Recursion

Recursion repeatedly invokes the method and as a result this incurs a cost (overhead) in terms of memory and processing time.

Each ***recursive*** call causes another copy of the method (variables) to be created and this set of copies can consume considerable memory space.

By contrast, iteration occurs within a method body so repeated method calls and extra memory assignments are avoided.

Recursion : (the positives)

The Positive aspects of Recursion
Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.

Occasionally, a recursive solution runs much slower than its iterative counterpart. However in many cases the recursive solution is only slightly slower.

In many cases, a recursive solution **is easier to understand** and implement correctly than an iterative solution.

Conclusion

- Recursion is a powerful programming tool.
- The ease of using the principle or recursion depends on the degree to which the problem you are trying to implement can be described recursively.
- Factorial and Sigma are easily described recursively and are easily amenable to recursive implementation.
- There are many and much more complicated examples of recursion where it becomes more obvious why you should use recursion over iteration.