Prasun Dewan[1]

16. Recursion

Loops are one mechanism for making a program execute a statement a variable number of times. Recursion offers an alternative mechanism, considered by many to be more elegant and intuitive. It is the primary mechanism for repeating execution in some languages. The repetition is achieved, not by using some special new kind statement, but by simply making a method call.

Developing a Recursive Solution

To illustrate the nature of recursion, consider again the function, `factorial`, which takes as an argument a positive integer parameter, `n`, and returns the product of the first `n` positive integers. We saw earlier several ways of solving this problem using iteration (loops). Here we will develop a recursive solution to it.

Before we look at the steps of the function, let us look at what this function computes for different values of `n`.

```
factorial(0) = 1[2]
factorial(1) = 1                = 1 * factorial(0)
factorial(2) = 2 * 1            = 2 * factorial(1)
factorial(3) = 3 * 2 * 1        = 3 * factorial(2)
```

Based on the pattern in these examples, we can say:

```
factorial(n) = 0                    if n == 0
factorial(n) = n * factorial(n - 1) if n > 0
```

What we have above, in fact, is a precise definition of the problem we must solve. As it turns out, it also gives us an algorithm for solving the problem:

```
public static int factorial(int n) {
      if (n == 0)
            return 1;
      if (n > 0)
            return n*factorial(n-1);
}
```

[2] It is not clear what the product of the first 0 positive integers is. The convention is to assume it is 1.

If we were to try and compile this method, the compiler would complain, saying that the function does not return a value. This is because we have not covered all possible values of `n`. We are assuming here that `n` is a positive integer. Though that is the expected value, Java will not prevent a negative integer to be passed as an actual argument. Therefore, we must specify what value should be returned in this case. Let us return the factorial of the absolute value of `n` for negative values of `n`:

```
public static int factorial(int n) {
      if (n == 0)
            return 1;
      else if (n < 0)
            return factorial(-n);
      else
            return n*factorial(n-1);
}
```

A method such as `factorial` that calls itself is called a *recursive method*.

Notice how compact our recursive solution is. It can be considered more elegant than the iterative solution because the algorithm for the problem is also the definition of the problem! As a result, it is easy to convince ourselves that our solution meets the requirements laid out by the definition, and we do not need to risk the off-by-one errors of loop iteration.

The key to using recursion is to identify a definition that meets the following two requirements:

1) *Recursive Reduction Step(s)*: It defines the result of solving a larger problem in terms of the results of one or more smaller problems. A problem is considered smaller than another problem if we can solve the former without having to solve the latter. In our example, the problem of computing the factorial of positive n is reduced to the problem of computing the factorial of a smaller number, n-1; and the problem of computing the factorial of a negative integer is reduced to the problem of computing the factorial of its positive, absolute value.
2) *Base Terminating Case(s):* It has terminating condition(s) indicating how the base case(s), that is, the smallest size problem(s), can be solved. In the example, it tells us that `factorial(0) = 1`. In general, there can be more than one base case, for instance one for negative numbers and another for 0.

Once we define the problem in this way, we can use recursion to solve it. The general form of a recursive algorithm is:

```
if (base case 1)
      return solution for base case 1
else if (base case 2)
      return solution for base case 2
…
else if (base case n)
      return solution for  base case n
else if (recursive case 1)
        do some preprocessing
       recurse on reduced problem
       do some postprocessing
…
else if (recursive case m)
       do some preprocessing
       recurse on reduced problem
       do some postprocessing
```

Stacking/Unstacking Recursive Calls

Before we look at other recursive problems, let us try to better understand `factorial` by tracing its execution. Assume a main method of invokes:

```
factorial(2)
```

When this call is made, we know the value of the formal parameter of the method, but not its return value. This situation can be expressed as:

Invocation	n	return value
factorial(2)	2	?

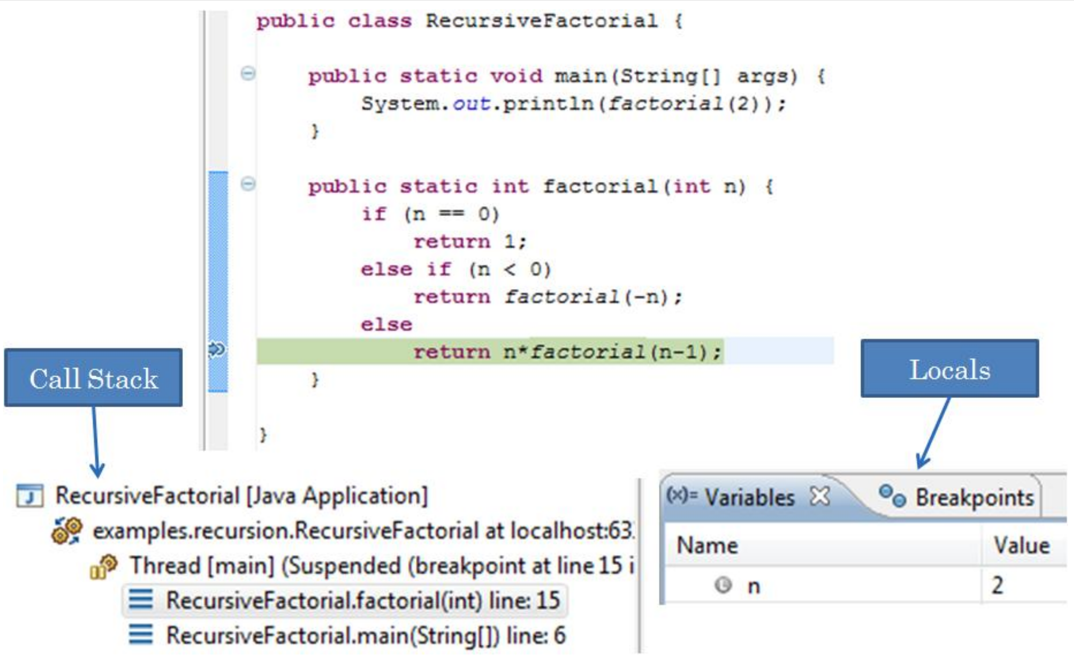
Once the formal parameter is assigned, the method executes the if statement. The if test fails, so the else part is executed, as shown in Figure 1 (top).

The else part contains another call to `factorial`, with the parameter this time being 1. Thus, we now have two executions of the same method running concurrently. Each execution has its own copy of the formal parameter and computes its own copy of the return value. The following table identifies the formal parameters and return values of the two invocations when the second call to `factorial` is made:

Invocation	n	return value
factorial(1)	1	?
factorial(2)	2	?

In a trace like this, it is usual to stack up the calls, that is, put an invoked method above the invoker. The topmost call with an uncomputed return value is the one the computer is currently executing.

The screen dump shown in Figure 2 shows an alternative view of a call stack, showing all the calls active at this point, including the call to the main method made by the interpreter.

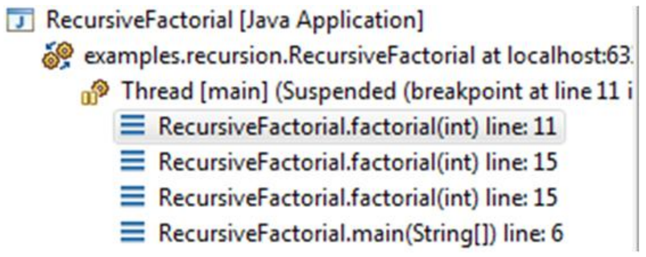


```
public class RecursiveFactorial {

    public static void main(String[] args) {
        System.out.println(factorial(2));
    }

    public static int factorial(int n) {
        if (n == 0)
            return 1;
        else if (n < 0)
            return factorial(-n);
        else
            return n*factorial(n-1);
    }

}
```

```
RecursiveFactorial [Java Application]
  examples.recursion.RecursiveFactorial at localhost:63
    Thread [main] (Suspended (breakpoint at line 15 i
      RecursiveFactorial.factorial(int) line: 15
      RecursiveFactorial.main(String[]) line: 6
```

Name	Value
n	2

Figure 1. Recursive call to factorial



```
RecursiveFactorial [Java Application]
  examples.recursion.RecursiveFactorial at localhost:63
    Thread [main] (Suspended (breakpoint at line 11 i
      RecursiveFactorial.factorial(int) line: 11
      RecursiveFactorial.factorial(int) line: 15
      RecursiveFactorial.factorial(int) line: 15
      RecursiveFactorial.main(String[]) line: 6
```

Figure 2. Call stack

Let us now trace what happens as `factorial(1)` executes its body. Again the test fails, and another call to `factorial` is made, this time with the actual parameter 0:

Invocation	n	return value
factorial(0)	0	?
factorial(1)	1	?
factorial(2)	2	?

When `factorial(0)` executes the if statement, the test finally succeeds, and the function returns 1 and terminates execution.

Invocation	n	return value
factorial(0)	0	1
factorial(1)	1	?
factorial(2)	2	?

At this point control transfers back to `factorial(1)`, which finishes execution of the statement:

```
return n*factorial(n-1);
```

that is, multiplies the value returned by `factorial(0)` with its copy of `n`, which is 1, returns the result, and terminates:

Invocation	n	return value
factorial(0)	0	1
factorial(1)	1	1
factorial(2)	2	?

Control transfers now to `factorial(2)`, which multiplies the return value of `factorial(1)` with its value of `n`, returns the result, and terminates.

Invocation	n	return value
factorial(0)	0	1
factorial(1)	1	1
factorial(2)	2	2

This return value is received by `main`, which prints it, giving us the output:

```
2
```

Thus, as this trace of the recursive function shows, successive calls to reduced versions of the problem get stacked until a base case is reached, after which they get unstacked as reduced versions successively return their results to their callers. Each stacked call get its own copies of the formal parameters and return value.

Recursion Pitfalls

We must be careful to have a terminating condition in a recursive program. Consider the following definition of `factorial`, which does not have a terminating condition:

```
public static int factorial(int n) {
        return n*factorial(n-1);
}
```

Let us trace again the execution of:

```
factorial(2)
```

It computes:

```
2*factorial(1)
```

`factorial(1)` computes

```
     1* factorial(0)
```

`factorial(0)` computes

```
      0*factorial(-1)
```

`factorial(-1)` computes

```
    -1*factorial(-2)
```

and so on. Thus, we make an infinite number of calls to `factorial`.

We must also be careful to make the recursive step compute a reduced version of the problem. Let us say we write the factorial function as follows:

```
public static int factorial(int n) {
      if (n == 0)
            return 1;
      else if (n < 0)
            return factorial(-n);
      else
            return factorial(n+1)/n+1;
}
```

Here, the second recursive step solves `factorial(n)` in terms of a harder problem: `factorial (n+1)`. Consider again the call:

```
factorial(2)
```

It computes:

```
factorial(3)/3
```

`factorial(3)` computes:

```
     factorial(4)/4
```

`factorial(4)` computes

```
      factorial(5)/5
```

and so on. Thus, we make an infinite number of calls to `factorial` again. Even though we have a terminating condition, we never reach it since we do not reduce the problem.

In summary, if we do not have a base case or reduce the problem, we can end up making an infinite number of calls. When this happens, the computer gives an error message saying there has been a stack overflow. Each time a function is called, space must be created for its arguments and its return value from an area of memory called the *stack*. (It is so called because it stacks an invoked method's data over the data of the calling method, as we showed in our tables above). Thus, a non-terminating sequence of recursive calls uses up the complete stack – hence the message of stack overflow.

Recursive Function with two Parameters

The recursive function above took one parameter. Let us consider one with two parameters. Suppose we are to write a `power` function that takes as arguments, a base and a positive exponent, and raises the base to the power of the exponent, that is, computes:

$\texttt{base}^{\texttt{exponent}}$

For instance:

```
power(2,3) == 8
```

When we have two parameters, we need to know which parameter(s) to recurse on, that is, which parameter(s) to reduce in the recursive call(s). Let us try first the `base`:

```
power(0, 0) = 1
power(0, exponent) = 0
power(1, exponent) = 1
power(2, exponent) = 2*2* … 2 (exponent times)
power(3, exponent) = 3*3* … 3 (exponent times)
```

There seems to be no relation between:

```
power(base, exponent)
```

and

```
power(base-1, exponent)
```

Let us try again, this time recursing on the `exponent` parameter:

```
power(base, 0) = 1
power(base, 1) = base * 1     = base * power(base, 0)
power(base, 2) = base * base  = base * power(base, 1)
```

We can use the pattern above to give the general solution:

```
power(base, exponent) = 1                                  if exp <= 0
power(base, exponent) = base * power(base, exponent-1)     otherwise
```

Our recursive function, again, follows straightforwardly:

```
public static int power(int base, int exponent) {
      if (exponent <= 0)
            return 1;
      else
            return base * power(base, exponent-1);
}
```

In this example, this two-parameter function recurses on only one of its parameters. It is possible to write recursive functions that recurse on both parameters, as you see in an exercise. In general, when a recursive method has multiple parameters, we must carefully choose the parameters whose size we will reduce in the recursive step.

Recursive Procedures

Like functions, procedures can be recursive. Consider the problem of saying "hello world" n times. We can write a recursive definition of this problem:

```
greet(0):   do nothing
greet(1):   print "hello world"; greet(0);
greet(2):   print "hello world"; greet(1);
```

The general pattern, thus, is:

```
greet(n):  do nothing                          if n <= 0
greet(n):  print "hello world"; greet(n-1);     if n > 0
```

Our recursive procedure, then, is:

```
public static void greet (int n) {
      if (n > 0) {
            System.out.println("hello world");
            greet(n-1);
      }
}
```

A recursive function is a mechanism for creating a *recursive compound expression* – it evaluates part of a compound expression and calls itself to evaluate the remainder. For instance, `factorial(n)`, computes a compound expression that multiplies `n` with the result returned by `factorial(n-1)`. In contrast, a recursive procedure is a mechanism for creating a *recursive compound statement* – it performs part of the steps in a statement list, and calls itself recursively to perform the other steps. For instance, `greet(n)` executes a compound statement that prints one greeting and calls `greet(n-1)` to print the remaining greetings. Recursive functions are perhaps more intuitive because, from our study of mathematics, we are used to recursive expressions. Like recursive functions, recursive procedures are also very useful, often leading to more succinct implementations than loops.

Sentinel-based List Recursion

Like loops, we can use recursion for processing lists that are terminated by sentinels. Consider again the problem of multiplying a list of positive integers terminated by a negative sentinel. Let us look at some example cases:

```
multiplyList() = 1                  if remaining input is -1
multiplyList() = 30*1               if remaining input is 30 -1
multiplyList() = 2*30*1             if remaining input is 2 30 -1
```

Our recursive definition would be:

```
multiplyList() = 1                         if  next input value is < 0
multiplyList() = readNextInputValue() * multiplyList()   otherwise
```

Here the base step returns 1 because the list of remaining items is empty. The reduction step removes one item from a non-empty list and multiplies it to the product of the items in the remaining list.

The recursive function, thus, is:

```
public static int multiplyList() {
      int nextValue = Console.readInt();
      if (nextValue < 0)
            // no more items in the list
            return 1;
      else
            // multiply nextValue to product of remaining items
            return nextValue * multiplyList();
}
```

Contrast this solution with the ones we have seen before. The previous cases are examples of *number-based* recursion: the base cases correspond to small values of one or more numbers, and reducing the problem amounts to reducing the values of these numbers. On the other hand, this is an example of list-based recursion: the base case corresponds to small-sized lists, and reducing the problem involves reducing the size of the lists. We will see further examples of list-based recursion later. Each call to `Console.readInt` reduces the size of the list by one. The number at the front of the list in a variable so that it can be used in the multiplication. Each recursive call get is own copy of this variable, since it is local to the method.

This solution is somewhat confusing because the list is not an explicit parameter of the function. Instead, it is implicitly formed by the series of values input by the user, and implicitly reduced by reading the next value. `multiplyList` is an impure function, returning different values depending on how much of the input has been read. For instance, if the remaining input is:

```
2 30 -1
```

the call:

```
        multiplyList()
```

returns 60; but if the remaining input is:

```
        30 -1
```

the same call returns 30.

To better understand this form of list-based recursion, let us trace the call to `multiplyList` when the input is:

```
        2 30 -1
```

Since there is no explicit parameter, we will instead trace the implicit parameter – the list of remaining input items. The following table shows the value of this parameter when the first call to `multiplyList` is made:

Invocation	Remaining input	return value
multiplyList()	2 30 -1	???

The method reads the first item, the number 2, of the list of remaining input values, and makes another call to `multiplyList`. The list of remaining input values is different for this call, having one less item – the value read by the previous call, as shown below:

Invocation	Remaining input	return value
multiplyList()	30 -1	???
multiplyList()	2 30 -1	???

Again, the method removes the first item, this time the number 30, from the input list, and makes another call to `multiplyList`:

Invocation	Remaining input	return value
multiplyList()	-1	???
multiplyList()	30 -1	???
multiplyList()	2 30 -1	???

The input seen by the third call is the base case, hence it simply returns 1:

Invocation	Remaining input	return value
multiplyList()	-1	1
multiplyList()	30 -1	???
multiplyList()	2 30 -1	???

The second call returns the value returned by the third call multiplied by 30:

Invocation	Remaining input	return value
multiplyList()	-1	1
multiplyList()	30 -1	30
multiplyList()	2 30 -1	???

Finally, the first call returns the value returned by the second call multiplied by 2:

Invocation	Remaining input	return value
multiplyList()	-1	1
multiplyList()	30 -1	30
multiplyList()	2 30 -1	60

Thus, each call processes a list with one less item, until we come to the base case of a list containing only –1.

It is easier to understand list-based recursion in functional languages, which tend to provide better support than Java for making lists explicit parameters of functions.

Summary

- Recursion offers an alternative to loops that is considered more elegant and less error-prone, since the solution to the problem is the same as its definition.
- Each recursive method must have a recursive reduction step, which solves the problem in terms of the same problem of a smaller size; and one or more terminating base cases, which solve it for the smallest-size version(s) of the problems.
- Successive calls to reduced versions of the problem get stacked until a base case is reached, after which they get unstacked as reduced versions successively return their results to their callers.
- Each stacked call gets its own copies of the formal parameters and return value from an area of memory called the stack.
- When a recursive method has multiple parameters, we must carefully choose the parameters whose size we will reduce in the recursive step.
- Like functions, procedures can also be recursive. While recursive functions create recursive expressions, recursive procedures create recursive statements.
- Number and list-based problems are particularly amenable for a recursive solution. In the former, the reduced problem involves a smaller number; while in the latter, it involves a smaller list.

Exercises

1) Trace the following two calls. The number of rows created for tracing the calls may be less than the number of actual calls:

   a. f(3)

```
public static int f(int n) {
   if (n <= 1)
         return 0;
   else if (n % 2 == 0)
         return f(n - 1) + n;
   else
         return f(n - 1) - n
```

Invocation	n	return value
f (3)	3	

   b. g(3,2)

```
public static int g(int n1, int n2) {
      if (n1 <= 0)
            return n2;
      else if (n2 <= 0)
            return n1;
      else
            return f(n1 - 1, n2 - 1);
}
```

Invocation	n1	n2	return value
g (3, 2)	3	2	

2) Trace the call to each of the following definitions of the recursive procedure, `p` by showing the output produced by each call. Assume that the user enters the four lines:

```
hello
hello
goodbye
goodbye
```

a.

```
public static void p (){
     if (Console.readString().equals("hello")) {
          System.out.println ("ca va");
          p();
     }
}
```

Invocation	Output produced
p ()	

b.

```
public static void p (){
     if (Console.readString().equals("hello")) {
          p();
          System.out.println ("ca va");
     }
}
```

Invocation	Output produced
p ()	

c.

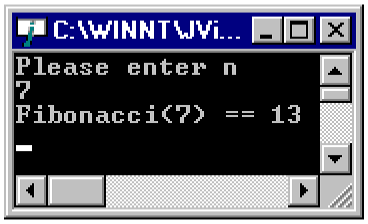
```
public static void p (){
     if (Console.readString().equals("hello")) {
          System.out.println ("ca va");
     }
     p();
}
```

Invocation	Output produced
p ()	

3) In this problem, you will solve a famous mathematical problem. The problem is to compute elements of the Fibonacci sequence. The first two elements of the sequence are 1 and 1 and a subsequent element is the sum of the two previous elements. Thus, the element of the sequence are:

```
1 1 2 3 5 8 13 ....
```

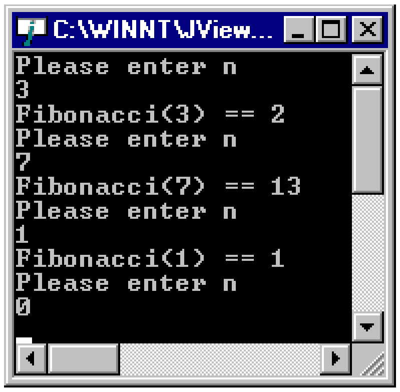
Your program will take as input a number n and output the nth element of the Fibonacci sequence. Thus, if the input is 2, the output should be 1; and if the input is 7, the output should be 13. Do not use a loop; instead write a recursive function.



```
C:\WINNT\JVi...
Please enter n
7
Fibonacci(7) == 13
```

If the user enters zero or a negative value for n, your program can print an arbitrary number.

4) your solution to Q3 by allowing the user to enter a series of values for `n` ending with a 0 or a negative number. Output the value of `Fibonacci(n)` for each user input, as shown below. Do not write a use a loop to process the list; write a recursive procedure instead.



```
C:\WINNT\JView...
Please enter n
3
Fibonacci(3) == 2
Please enter n
7
Fibonacci(7) == 13
Please enter n
1
Fibonacci(1) == 1
Please enter n
0
```