Unit 4b: Recursion

1 Introduction

Recursion is a nice technique for solving problems which have a naturally repetitive streak to them. Suitable for a problem where you are repeatedly performing actions on 'smaller' and 'smaller' 'objects'.

A good analogy is the Russian doll where unscrewing one reveals another similar doll which can be unscrewed etc. until the smallest one is discovered. Another analogy is that of an onion which has smaller and smaller layers until you reach the core.

Each of these two analogies has in common two things: we repeatedly uncover 'smaller' versions of the original and there is some ending point. This is what recursion is all about.

In software development, this idea is implemented by having a function/method calling a smaller version of itself which calls a 'smaller' version still and so on. Remember, there must be some stopping point or get out clause i.e. there needs to be a way of stopping an infinite set of calls by providing an alternative which doesn't call another version of itself.

Recursion provides a more functional alternative to iteration with no *mutable state*, e.g. no count or “running total” variables.

Some “pure” functional languages, e.g. Haskell, don’t provide while/for loops, need recursion to iterate.

2 Powers Example

Consider the expression 3^4. There is an obvious way of calculating this using iteration. Let's look at it in a little more detail. Essentially the calculation is:

$$3 * 3 * 3 * 3$$

However, we can also write this as:

$$3 * (3 * 3 * 3) \text{ i.e. } 3 * 3^3$$

or as:

$$3 * (3 * (3 * 3)) \text{ i.e. } 3 * (3 * 3^2)$$

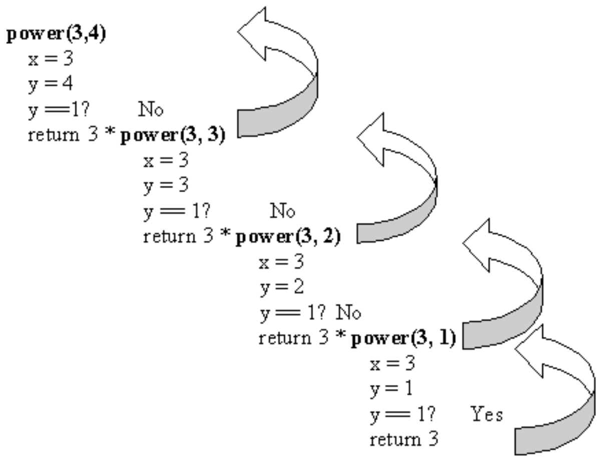
or as:

$$3 * (3 * (3 * (3))) \text{ i.e. } 3 * (3 * (3 * 3^1))$$

What does all this mean? Well, it means that we are solving the problem by considering smaller and smaller versions of the multiplication. Initially, we have to solve 3 * 3 * 3 * 3 i.e. 3^4. We have decomposed this in to 3 multiplied by the solution of 3 * 3 * 3 i.e. 3^3. This latter problem we have decomposed into 3 multiplied by 3 * 3 i.e. $3^2$and so on. This is a recursive situation. Things may become clearer if we see the code which solves this recursively:

```
public static int power(int x, int y) {
    if (y == 1)
        return x;
    else
        return x * power(x, y-1);
}
```

The first call to this function would **power(3,4)** which would result in the parameters taking the values **x = 3**, **y = 4**.



The escape clause is when the power (y) becomes 1 in which case the chain of function calls winds its way back to the top i.e. power(3, 1) returns the value 3 which is used in the calculation 3 * power(3, 1) to equal 9 so that power(3, 2) returns 9 which is fed into the calculation 3 * power(3, 2) i.e. 27 which is returned to the calculation 3 * power(3, 3) i.e. 81 which is the value returned to the calling program.

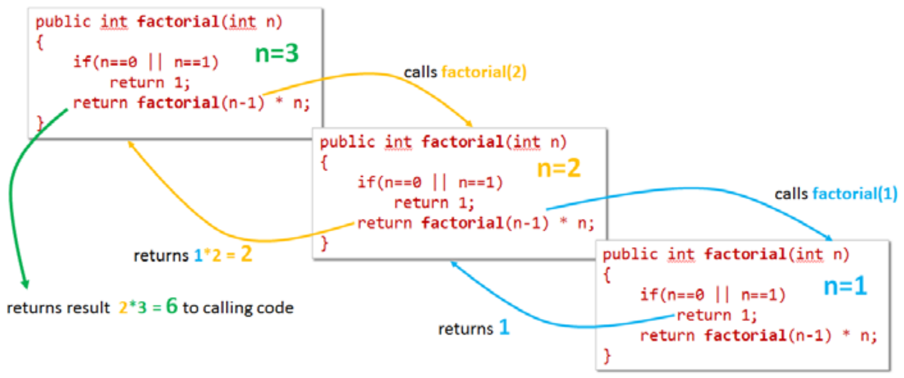
There is a close correlation between iteration and recursion and most programs can be solved either way. Recursion tends to be something developers like or tolerate. However, there are some situations where it is a positively beneficial way of looking at things.

3 Factorial Example

Let’s revisit the factorial example:

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

What happens if we call factorial(3)?



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

There is a downside – the Java example here will crash if *n* becomes large enough (about 15000 when I tried it), and not because the numbers get large in the case of factorials.

Every function call creates a *stack frame* in stack memory to hold parameters/local variables; the stack frame needs to be kept in memory until the function completes.

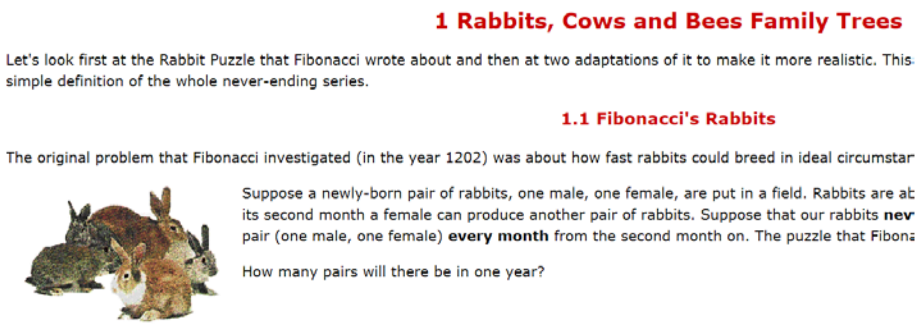
In this example, by the time the call to *factorial(1)* is made there will be 3 stack frames in memory; *factorial(15000)* would try to create 15000 stack frames, each of which takes up memory – can lead to *stack overflow*.

We can avoid this with a technique called *tail recursion* which is typically supported in functional languages including Scala – this involves a slight change to the design of code to allow compiler to perform an optimisation, see later.

4 Fibonacci Example

Let's look at another example to reinforce the point. Fibonacci numbers are a set of numbers with a regular pattern which appear in a variety of guises in nature (and other places). See the following web site for an explanation of their discovery and uses:

http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fib.html.



1 Rabbits, Cows and Bees Family Trees

Let's look first at the Rabbit Puzzle that Fibonacci wrote about and then at two adaptations of it to make it more realistic. This simple definition of the whole never-ending series.

1.1 Fibonacci's Rabbits

The original problem that Fibonacci investigated (in the year 1202) was about how fast rabbits could breed in ideal circumstar

Suppose a newly-born pair of rabbits, one male, one female, are put in a field. Rabbits are ab its second month a female can produce another pair of rabbits. Suppose that our rabbits **nev** pair (one male, one female) **every month** from the second month on. The puzzle that Fibona

How many pairs will there be in one year?

"The original problem that Fibonacci investigated (in the year 1202) was about how fast rabbits could breed in ideal circumstances.

Suppose a newly-born pair of rabbits, one male, one female, are put in a field. Rabbits are able to mate at the age of one month so that at the end of its second month a female can produce another pair of rabbits. Suppose that our rabbits **never die** and that the female **always** produces one new pair (one male, one female) **every month** from the second month on. The puzzle that Fibonacci posed was...

How many pairs will there be in one year?"

Start:	1 pair
End of month 1:	1 pair
End of month 2:	2 pairs
End of month 3:	3 pairs
End of month 4:	5 pairs
etc.	

This produces a sequence which looks like this:

1 2 3 5 8 13 21 34 55 89 144 233

The key to this series is that the next number is the sum of the last two numbers i.e.

Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n - 2)

This is an ideal situation for recursion.

```
public static int fibonacci(int n) {
    if (n <= 2)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```