30 Recursive Functions

1 Recursive functions

Methods may employ a technique called *recursion*. We will explain it by writing a function to compute factorials. The *factorial* of a natural number *n* is defined as $1\times2\times3\times \ldots \times n$. We write *n*! to denote the factorial of *n*. For example, $3! = 1\times2\times3 = 6$, and $5! = 1\times2\times3\times4\times5 = 120$. As an aside we mention that *n*! is the number of ways of laying out *n* balls in a line where all the balls have different colours. The idea of 0! may not be intuitive, but we allow it nonetheless and define it to be 1. The defining properties of factorial are:

(i) $0! = 1$
(ii) $n! = n\times(n\text{-}1)!$ for $n>0$

For example, property (ii) with $n=5$ states that 5! equals $5\times4!$, which you will easily convince yourself is true. Properties (i) and (ii) are called *defining properties* because it turns out that they are sufficient to calculate *n*! for any natural n (assuming we know how to do subtraction and multiplication). To compute 3! for example:

3!
= "property (ii) with $n=3$"
$3\times2!$
= "property (ii) with $n=2$"
$3\times(2\times1!)$

= "property (ii) with n=1"
$3\times(2\times(1\times 0!))$
= "property (i)"
$3\times(2\times(1\times 1))$
= "multiply out"
6

Property (ii) is called a *recursive property*. By this is meant that the operation being defined on the left-hand side (here, factorial) occurs also on the right-hand side. A recursive property might at first sight seem to be circular, but this need not be so. All is well if the operation on the right hand side is applied to a smaller term than on the left hand side. That is indeed the case in (ii) which has factorial n-1 on the right-hand side, and factorial n on the left. The importance of the term being smaller is that it is tending towards a term (here, 0!) for which the operation is defined without recursion. This ensures that we can "unwind" the recursive property as often as it takes to arrive at a non-recursive case (also called a *base* case). The computation of 3! above illustrates this: we applied property (ii) in turn for 3!, 2!, and 1!, after which we arrived at 0! for which property (i) applies. After that it only remained to multiply out.

In summary, properties (i) and (ii) are defining properties of factorial because (a) there is a rule covering every natural argument (in fact, one for 0 and one for positives), and (b) in the recursive property (ii), the factorial on the right-hand side is applied to a smaller term than that on the left. All this might be no more than interesting mathematics but for the fact that such defining properties can be translated directly into Java methods. The following is a method to compute factorials:

```
static int fac(int n) { // factorial n,  n>=0
    if (n==0) return 1;
    else return (n*fac(n-1));
}
```

We say that `fac()` is *recursive* because its body contains an invocation of `fac()`. Let us check that `fac()` really does faithfully encode the defining properties of factorial. We see immediately from an inspection of its body that it satisfies the following:

(a) `fac(0) = 1`
(b) `fac(n) = n*fac(n-1)` when `n>0`

– there is no intelligence being applied in deducing (a) and (b), just a mechanical inspection of the text of `fac()`. Properties (a) and (b) are evidently faithful representations in code of (i) and (ii), respectively – just mentally replace `fac(0)` with 0!, `fac(n)` with n!, and `fac(n-1)` with (n-1)!. That's it! Having satisfied ourselves at the outset that properties (i) and (ii) suffice to compute n!, and now having checked that `fac()` faithfully encodes (i) and (ii), we can relax. The machine will do the rest (we will see how it does so shortly).

Example 1: the first few factorials

The following program prints a table of the factorials of 0, 1, 2, ... 9.

```
class Factorials {

    static int fac(int n) { // factorial n, n>=0
        if (n==0) return 1;
        else return n*fac(n-1);
    }

    public static void main(String[] args) {
        for (int k=0; k<10; k++) {
            System.out.println(k + "! = " + fac(k));
        }
    }
}
```

Example 2: celebrity numbers

A natural number is called a *celebrity* number if it is equal to the sum of the factorials of its decimal digits. 1 and 2 are trivial celebrity numbers (1!=1 and 2!=2) but a more interesting one is 145 (1!+4!+5! = 1+24+120 = 145). The following program finds all celebrity numbers up to one million (in fact celebrity numbers are rare – the program discovers just one more).

```
class Celebrity {

    static int fac(int n) { // factorial n, n>=0
        if (n==0) return 1;
        else return n*fac(n-1);
    }

    static boolean isCelebrity(int n) { // is n a celebrity number, n>0
        int total = 0; // running total of factorials of digits of n
        int highN = n; // the high end of n – its digits remain to be fac'd & added
        while (highN>0) {
            // extract rightmost digit from highN ...
            int digit = highN%10; highN = highN/10;
            // ... and add its factorial to running total
            total = total+fac(digit);
        }
        return (total==n);
    }

    public static void main(String[] args) {
        for (int k=1; k<1000000; k++) {
            if (isCelebrity(k))
                System.out.println(k + " is a celebrity number");
        }
```

```
    }
}
```

Recursion vs iteration

Recursion does not give us any extra power in the sense that it does not enable us to write programs that we could not have written without it, but it can be an elegant and simple way to write some methods. It is an alternative to loops. For example, `fac()` can be written iteratively (i.e. using loops rather than recursion) as follows:

```
static int fac(int n) {  // factorial n,  n>=0
    int z = 1;
    for (int i=1; i<=n; i++)
        z = z*i;
    return z;
}
```

Recursion applies equally to static and dynamic methods.

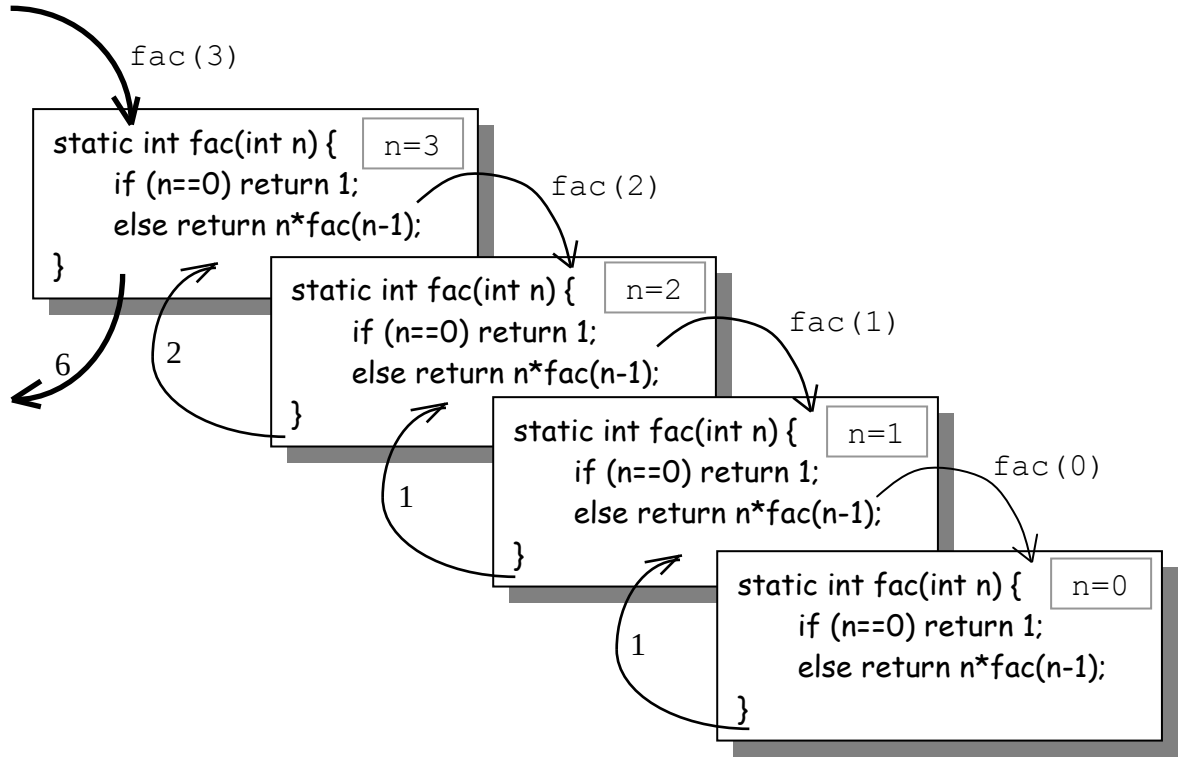
2 Recursion operationally OPTIONAL

How does the machine implement recursion? Examining the body of a recursive method would seem to suggest that the method calls itself, but that is not the way to look at it. A recursive method does not literally call itself, but rather the system makes a clone of the method and calls the clone. This is the key to understanding recursion operationally: each time the execution of a method encounters a recursive call on the method, it makes a clone and calls the clone. The clone is identical in all respects to the parent. If the clone in turn encounters a recursive call (which is very likely) it makes a new clone and calls it. Eventually (if the method is properly written) a clone will compute its result without meeting a recursive invocation, and so without cloning, and the answer will be returned to the parent. The parent can then complete its work and pass the result back to *its* parent, and so on. Once a clone completes its work and returns to its parent, the clone dies and all memory it occupied is released. For example, suppose the statement `System.out.print(fac(3))` is executed. This will cause the system to invoke `fac(3)` whose execution can be pictured in the top box above.

- The first box contains the `fac()` that is invoked as a result of executing `System.out.print(fac(3))`. Parameter `n` in `fac()` is given the initial value 3. Therefore the else-branch of the if-statement is taken resulting in an invocation of `fac(n-1)`, i.e. `fac(2)`.

- This causes a clone to be created – the one in the second box – in which parameter `n` has value 2. This looks exactly like its parent and even has the same name. However, it is a different method. In particular, its parameter `n` is a different variable from parameter `n` in the parent. In the parent, `n` has the value 3 at all times, whereas in the clone `n` has the value

2 at all times. The parent continues to exist while the clone does its job, but in suspended animation. The clone takes the else-branch resulting in an invocation of `fac(1)`.



- This causes another clone to be created – the one in the third box. In this clone parameter `n` has the value 1. The original and two clones now exist simultaneously, but the parent and the first clone are suspended. This newest clone takes the else-branch resulting in an invocation of `fac(0)`.
- This causes the clone in the fourth box to be created. Now the parent and three clones exist. In the latest clone, parameter `n` has the value 0.
- The newest clone immediately returns the value 1 to its invoker (clone 2) and dies, releasing the space it occupied.
- Clone 2 (in which `n` has value 1) now computes `n*1` and returns the result 1 to its caller (clone 1) and dies. The space it occupied is released.
- Clone 1 (in which `n` has value 2) now computes `n*1` and returns the result 2 to its caller (the original) and dies. The space it occupied is released.
- The original (in which `n` has value 3) computes `n*2` and returns the result 6 to the `System.out.println()` statement that invoked it, and so 6 is printed.

3 Ensuring termination

It is easy to write a recursive function that never terminates, for example:

```
static int silly(int n) {
  return silly(n);
```

```
}
```

Actually, if `silly(3)`, say, is invoked, it will eventually terminate abnormally with a message from the system to the effect that it has run out of memory — because each recursive call consumes memory in the machine until there is no free memory left. Of course, we would never deliberately write a non-terminating recursive function, but we may do so inadvertently and it is wise to double-check. A recursive method is guaranteed to terminate if it satisfies the following criteria:

(i) The initial call and every recursive call is applied to arguments that are within the design range of the method.

(ii) The result is computed for the smallest arguments (the *base* cases) without recourse to recursion.

(iii) Every argument of a recursive call is by some measure smaller than the incoming argument. By "smaller" is meant that the argument is at least one step nearer a base case.

As an example, we show that `fac()` meets the three criteria and so we can be satisfied that it terminates. As regards design range, `fac()` is designed on the assumption that the argument `n` satisfies $n \geq 0$.

(i) The argument in the recursive call is `n-1` and so we have to be sure that at the point of call $n-1 \geq 0$. This is indeed so because by assumption $n \geq 0$, and the else-branch of the if-statement is taken ensuring $n > 0$ at the point where the call `fac(n-1)` occurs.

As regards base cases:

(ii) `fac(0)` is computed without recourse to recursion.

For the recursive case, we take as our measure of the argument the value of `n` itself:

(iii) We observe `n-1` is smaller (i.e. closer to the base case 0) than the incoming `n` (which we know is positive at the point of the recursive call).

It follows that `fac(n)` terminates provided `n` satisfies $n \geq 0$. For negative `n`, `fac()` does not terminate, but it was not designed to handle that case.

Banana skin: recursion in a loop

Some beginners apparently get so concerned about ensuring termination that they seek extra insurance by placing recursive calls inside a loop! The following is typical:

```
static int fac(int n) { // factorial of n, n>=0
    if (n==0) return 1;
    else {
        while (n>0) { // Wrong!
            return (n* fac(n-1));
        }
```

```
        }
    }
```

This is wrong! Loops rarely occur in recursive methods (at least in the simpler ones programmers are likely to meet in everyday programming), and it is exceedingly rare for a recursive call to occur inside the body of a loop (as above). If you find yourself writing a loop in the body of a recursive method, you may well be going down the wrong track, so think again and make sure that that is really what you want to do. Chances are it isn't.