7 Factorial – transition to the Sudoku exercise

The factorial can be computed with a simple recursive function:

```
// pre: n >= 0
// post: result = n!
long fac(int n) {
   if (n==0) {
      return 1;
   } else {
      return n * fac(n-1);
   }
}
```

Suppose we want to write a recursive factorial function without parameters or return value. This may look a bit silly, but it shows the concept of *backtracking* in a simple fashion and prepares for the Sudoku exercise.

So we write a method `void facn()` that operates on two instance variables. The variable `n` takes the role of the parameter and stores the integer on which the current call of `facn` has to operate and the variable `result` stores the intermediate and final results of the calls.

In the factorial with parameter, we can write for the recursive call `fac(n-1)`. In the parameterless case, we first subtract 1 from `n` and then call `facn`. After that, the result can be calculated. Note that we need the original value of `n` for this. To make this clearer, we have added a local constant `orig_n` that stores the value of `n` at the beginning of the call and use this constant.

Note: in assertions and postconditions we can use the expression `old(someVariable)` to refer to the value a variable had at the start of the current method call. The constant `orig_n` has the same purpose and furthermore allows to use this value in the program, not only in assertions.

1. After the recursive call, `n` has to be reset to its original value, otherwise the next call of `facn` would use a wrong value of `n` (in fact, the postcondition would not hold). Comment out the line that resets `n` and see what happens. Explain.
2. To further test your understanding of the program, modify it in such a manner that the intermediate values during the recursion are output as well, i.e., 0! = 1, 1! = 1, 2! = 2, 3! = 6 etc. should be output as well. Furthermore, add assertions between all statements in the else-part. One such assertion is already given.

```
1  // Parameterless version of factorial
2
3  import java.util.*;
4
5  class FactorialNoParams {
6      Scanner sc = new Scanner(System.in);
7
8      int n;
9      int result;
10
11     void facn() {
12         final int orig_n = n;
13         if (n==0){
14             result=1;
15         } else {
16             n=n-1;
17             assert n == orig_n - 1;
18             facn();
19             result = orig_n * result;
20             n = orig_n; // set n back to original value
```

```
21          }
22      }
23
24      void demo() {
25          System.out.println("provide number to compute factorial for");
26          n = sc.nextInt();
27          facn();
28          System.out.println(n + "! = " + result);
29      }
30
31      public static void main(String[] args) {
32          new FactorialNoParams().demo();
33      }
34  }
```

8 Recursion – Sudoku

We are going to write a solver for sudoku puzzles. A sudoku puzzle is a grid of 3 by 3 boxes, where each box contains a grid of 3 by 3 squares, so 9 by 9 squares in total. Some squares already contain a number. The problem is to fill in the rest of the squares with the numbers 1 to 9, in such a way that every row and every column and every box contains each number exactly once.

We build our solution step by step. It is not required to stick to these steps, but it may help.

1. Write a class Sudoku, containing an instance variable grid of type `int[][]`. This represents the puzzle and all intermediate steps. Fill the grid with an initial puzzle. Use the value 0 to represent an empty square. An easy way to fill the grid in Java is as follows:

   ```
   grid = new int[][] {
       { 0, 6, 0,  0, 0, 1,  0, 9, 4 },
       { 3, 0, 0,  0, 0, 7,  1, 0, 0 },
       { 0, 0, 0,  0, 9, 0,  0, 0, 0 },
       { 7, 0, 6,  5, 0, 0,  2, 0, 9 },
       { 0, 3, 0,  0, 2, 0,  0, 6, 0 },
       { 9, 0, 2,  0, 0, 6,  3, 0, 1 },
       { 0, 0, 0,  0, 5, 0,  0, 0, 0 },
       { 0, 0, 7,  3, 0, 0,  0, 0, 2 },
       { 4, 1, 0,  7, 0, 0,  0, 8, 0 },
   };
   ```

   Here, we used a puzzle of medium difficulty (for humans) found on the web. Write a method that prints the grid, e.g., as follows.

   ```
   +-----------------+
   |  6  |    1|  9 4|
   |3    |    7|1    |
   |     |  9  |     |
   -------------------
   |7   6|5    |2   9|
   |  3  |  2  |  6  |
   |9   2|    6|3   1|
   -------------------
   |     |  5  |     |
   |    7|3    |    2|
   |4 1  |7    |  8  |
   +-----------------+
   ```

Test it.

2. In the process of solving we will look for an empty square, fill in a number and see if it fits, or in other words, if it has no conflict, where conflict means: the same number occurs somewhere else in the same row, column, or box. Write a method `boolean givesConflict(int x, int y, int n)` that determines whether filling in the number n in the square with coordinates x en y will give a *conflict*. It is probably a good idea to separate further into three methods givesRowConflict, givesColConflict, and givesBoxConflict. (The last one is a bit more complicated than the first two.) Test these methods.

3. Add instance variables `xempty` and `yempty` that will contain the coordinates of an empty square. Write a method that tries to find an empty square and, if it succeeds, stores its coordinates into these two variables. If it can not find an empty square, it signals this (e.g., have it return a boolean, or write special values into xempty and/or yempty. Test this method.

4. Now write a recursive method `boolean tryToSolve()` that tries to solve the current grid. It returns true it it succeeds and false if it cant find a solution.

   Our strategy is "brute force": we look for an empty square, fill it with a number and see if it fits (i.e., there are no conflicts with other numbers. If it doesnt fit, we try the next number, etc. When we have a number that fits, we fill it in and continue by trying to solve the smaller problem of solving the grid with the extra filled in number.

   Here, you have to decide about "grid management". One possibility is to add the grid as a parameter to `tryToSolve` and make every recursive call on a *new* grid, copied from the previous one and provided with the extra filled in number. This is relatively expensive, however, since copying takes time and all these grids have to be stored. In this approach some methods need a grid parameter (such as `tryToSolve` and `givesConflict`.

   An alternative is to you keep only one grid in an instance variable, shared by all recursive calls. When a recursive call returns, you have to re-establish the state of the grid before the call. This technique is called *backtracking*.

   When you test this method, it may be helpful to add a statement to the method that prints the current grid. To avoid an output flood, start with a grid that is almost solved. Later you can remove the print statement or, even better, have it print the grid only now and then, e.g., after a row is finished, or every 50 calls of solve. To help you with testing, here is a correct Sudoku (not the solution of the puzzle given above).

```
+-----------------+
|1 2 3|4 5 6|7 8 9|
|4 5 6|7 8 9|1 2 3|
|7 8 9|1 2 3|4 5 6|
-------------------
|2 1 4|3 6 5|8 9 7|
|3 6 5|8 9 7|2 1 4|
|8 9 7|2 1 4|3 6 5|
-------------------
|5 3 1|6 4 2|9 7 8|
|6 4 2|9 7 8|5 3 1|
|9 7 8|5 3 1|6 4 2|
+-----------------+
```

Kees Huizing – 13 October 2009