Tail recursion

In a recursive method, a recursive call is called a *tail call* if it is the final action in the method —the method returns immediately after this call. A method that contains a tail call is said to be *tail recursive*.

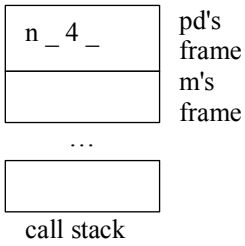
```
/** Print the integers 1..n in
   * descending order.  */
public static void pd(int n) {
   if (n < 1) return;
   System.out.println(n);
   pd(n-1);
}

/** Print the integers 1..n in
   * ascending order.  */
public static void pa(int n) {
   if (n < 1) return;
   pa(n-1);
   System.out.println(n);
}
```

Procedure `pd`, to the right, is tail recursive because the last statement in its body, `pd(n-1)`, is a tail call. In procedure `pa`, the call `pa(n-1)` is not a tail call because something is done after it. Procedure `pa` does not contain a tail call, so it is not tail recursive.

Why this interest in tail calls and tail recursion? We explain. You know that a call on a method is executed as follows.

1. Push a frame for the call on the call stack.
2. Assign the arguments to the parameters (in the frame for the call).
3. Execute the method body —using the local variables and parameters in the frame for the call.
4. Pop the frame for the call from the call stack, and if the method is a function, push the value being returned onto the call stack.

On the right is the call stack in the case that a method `m` executed the call `pd(4)` on procedure `pd`, above. Since `n≥1`, `n` is printed; then it is time to execute `pd(n-1)`. Here is the critical point. Since `pd(n-1)` is a tail call, meaning nothing is done after it, a new frame for the call need not be pushed onto the stack. Instead, use the current one! Change the value of parameter `n` to 3 and execute the method body. When it terminates, the frame for the call will be popped and m will resume.



If tail calls are optimized, a call `pd(1000)` will push only one frame for `pd` onto the call stack, not 1,000! That saves not only the time for 999 method calls but also space on the call stack for 999 frames.

In functional languages (in which there is essentially no assignment statement and no loops), tail calls are optimized as explained above. Java version 9 does not optimize tail calls, although a later version may do so. In procedural languages like Java, Pascal, Python, and Ruby, check whether tail calls are optimized; it may be declared so in the language specification, or it may be a feature of the compiler being used.

1. Optimizing tail calls yourself.

It's quite simple to perform the tail-call optimization yourself. We show you how to do this for a procedure `p` — a method that does not return a value:

1. Make sure the last statement of the procedure body is a return statement `return;` .
2. Enclose the procedure body in a while-loop:

```
tailRecursionLoop: while (true) {
   procedure body
} //end of tailRecursionLoop
```

3. Replace each tail-call (and possibly following return statement) `p(…) return;` into

   Assign arguments of the call `p(…)` to the parameters;
   `continue` `tailRecursionLoop;`

The label `tailRecursionLoop` in the continue-statement is needed only if there are nested loops within the method body, to be sure that the continue statement is associated with the right loop

To the right, we show the optimized version of procedure `pd` given at the top of the page. In this case, the continue-statement is not needed and can be deleted.

```
/** Print the integers 1..n in
   * descending order.  */
public static void pd(int n) {
   tailRecursionLoop: while (true) {
      if (n < 1) return;
      System.out.println(n);
      // pd(n-1);
      n= n-1;
      continue tailRecursionLoop;
   }
}
```

Tail recursion

1. Optimizing tail calls in a function.

To the right is a function that counts the number of 'e's in a String. The first recursive call in it is a tail call. The last one is not, because an addition is done after the call.

We optimize the tail call. Note that the last statement is a return statement. Below to the right, we give the optimization. We added the while-loop around the procedure body and replaced the tail call by (1) the assignment of the argument of the tail call to parameter s and (2) the continue statement.

In the original method, the depth of recursion is the length of String s. In the optimized method, the depth of recursion is 1 + (the number of 'e's in s).

```
/** = number of 'e's in s */
public static int nE(String s)  {
   if (s.length() == 0) return 0;
   if (s.charAt(0) != 'e') {
      return nE(s.substring(1));
   }
   return 1 + nE(s.substring(1));
}
```

```
/** = number of 'e's in s */
public static int nE(String s)  {
   tailRecursionLoop: while (true) {
      if (s.length() == 0) return 0;
      // s has at least char. s = s[0] + s[1..]
      if (s.charAt(0) != 'e') {
         // return nE(s.substring(1));
         s= s.substring(1);
         continue;
      }
      return 1 + nE(s.substring(1));
   } //end tailRecursionLoop
}
```