CS 4: Lecture 17
Monday, March 20, 2006

Recursion

Recursion is the word for when a method calls itself.

Recursion is a natural way to compute a factorial, for example. The factorial of a nonnegative integer n can be defined inductively as

$$n! = \begin{cases} 1, & n = 0 \text{ or } n = 1 \\ n\,(n-1)!, & n > 1 \end{cases}$$

We can write recursive code that echoes this inductive definition.

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

If you call "factorial(0)" or "factorial(1)", the factorial method follows the first branch of the "if" statement, and simply returns 1.

But suppose you call "factorial(4)". The factorial method computes 4! by calling "factorial(3)", multiplying 3! by 4, and returning the product.

Seems odd, doesn't it? How can you depend on "factorial(3)" working when we're in the middle of writing the "factorial" code?

The answer is because computing 3! is easier (takes less computation) than computing 4!.

It's a lot like inductive proofs in mathematics. Recall how induction works.

Theorem: For all $n \geq 1$, $n! \leq n^n$.

Proof: By induction.

Base case: Suppose n = 1. Then n! = 1 and $n^n = 1$, so the theorem holds for n = 1.

Inductive step: Suppose n = j > 1. For the sake of induction, assume the theorem holds for all n _less_ than j. We'll show that it holds for n = j too.

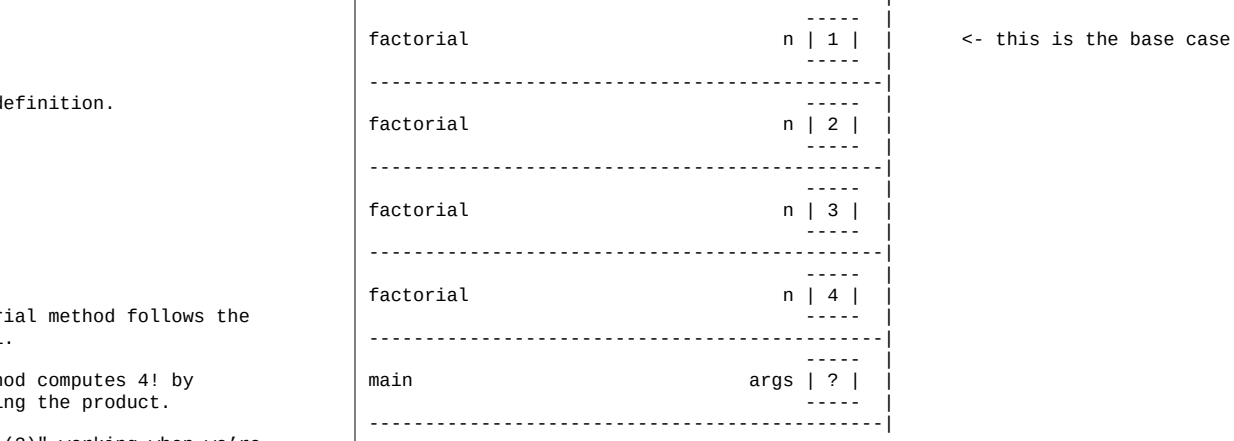
The theorem holds for n = j - 1, so $(j-1)! \leq (j-1)^{j-1} < j^{j-1}$.

Multiplying both sides by j gives $j! \leq j^j$. Therefore, the theorem holds for n = j. QED

The recursive factorial method is a lot like this proof.
- First, the recursive method has a _base_case_, which handles the cases n = 0 and n = 1.
- Second, the recursive method _assumes_ that it itself works correctly for numbers less than n, and uses that fact to compute n!.

The best way to understand recursion, though, is to think of what's happening on the stack. When you call "factorial(4)", a stack frame is created for the factorial method, with the local variable n set to 4.

Then factorial makes the method call "factorial(3)". This puts a second stack frame for factorial on top of the stack, with the local variable n set to 3. Then factorial(3) calls factorial(2) which calls factorial(1), so we have four frames on the stack.



Finally, we've reached the base case; "factorial(1)" returns 1, and does not call "factorial" again.

Then, "factorial(2)" multiplies 2 by 1 and returns 2.

Then, "factorial(3)" multiplies 3 by 2 and returns 6.

Finally, "factorial(4)" multiplies 4 by 6 and returns 24.

The idea of scope is very important with recursion, because there could be hundreds of "factorial" stack frames on the stack at once, and thus hundreds of variables named "n". As always, _only_ the top stack frame is accessible at any given moment. The variable "n" in the factorial method always refers to the n in the top stack frame.

What would happen if we wrote "factorial" like this?

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

This "factorial" would keep calling itself forever--effectively, in an infinite loop--endlessly putting more stack frames on the stack, until the computer runs out of stack space and Java terminates with an error message. For recursion to work, you need to make sure that any call to a recursive method will always reach the base case eventually and stop calling itself.

Most loops can be turned into recursive methods. For example, the following two methods print all the numbers from 1 to n.

```
  public static void count(long n) {
    for (long i = 1; i <= n; i++) {
      System.out.println(i);
    }
  }
```

```
  public static void count(long n) {
    if (n > 0) {
      // Print numbers 1...n - 1
      count(n - 1);
      System.out.println(n);
    }
  }
```

How does the recursive version on the right work? It calls itself recursively to print all the numbers from 1...n - 1, then it prints n. The base case is n <= 0, for which the "count" method does nothing.

What happens if we switch the order of the "println" call and the recursive call?

```
  public static void countDown(long n) {
    if (n > 0) {
      System.out.println(n);
      // Print numbers 1...n - 1
      countDown(n - 1);
    }
  }
```

This procedure prints n _first_, _then_ prints the numbers between 1 and n - 1. The effect is that it prints the numbers in reverse order, from n down to 1.

Rewriting a simple loop (or even the factorial computation) as a recursive call isn't useful in practice, because the code just got longer and slower. However, some loops lend themselves naturally to a recursive expression. Here's a recursive version of the gcd method.

```
  static private long gcd(long a, long b) {
    if (b == 0) {
      return a;
    } else {
      return gcd(b, a % b);
    }
  }
```

The base case occurs when b = 0, because GCD(a, 0) = a. In all other cases, "gcd" calls itself recursively using the formula GCD(a, b) = GCD(b, r) we derived last week, where r is the remainder from dividing a by b. If you call gcd(15, 6), after two recursive calls the stack looks like this.

```
method call      parameters & local variables
-----------------------------------------------|
                              -----     -----  |
gcd                       a | 3 |   b | 0 |  |      <- this is the base case
                              -----     -----  |
-----------------------------------------------|
                              -----     -----  |
gcd                       a | 6 |   b | 3 |  |
                              -----     -----  |
-----------------------------------------------|
                              -----     -----  |
gcd                       a | 15|   b | 6 |  |
                              -----     -----  |
-----------------------------------------------|
```

The top "gcd" on the stack has reached the base case; next, all the "gcd" methods on the stack (from top to bottom) will return 3.

Here's an example of recursion from a computer game called "Dungeon of the Overlord" I wrote as a teen. It included a magic "Wand of Wonder". When you wave the magic wand, it casts a magic spell chosen randomly from a selected subset of all the possible magic spells.

```
  private magicSpell(int spellNumber) {
    // Make magic sound, other preparations....
    switch(spellNumber) {
    ...
    case 38:
      // Wand of wonder.
      int randomSpell = (int) (15 * random.nextDouble());      // Spells 0...14
      magicSpell(randomSpell);
      break;
    case 39:
    ...
    }
  }
```