CS 4: Lecture 16
Wednesday, March 15, 2006

THE STACK AND THE HEAP

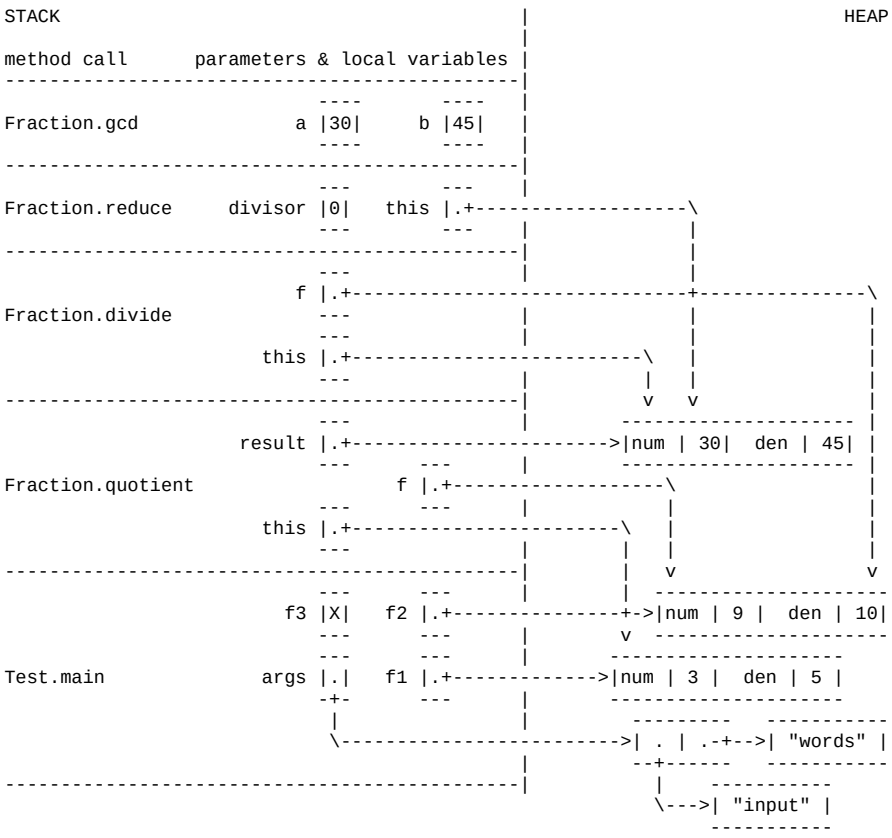
Java stores stuff in two separate pools of memory: the stack and the heap.

The _heap_ stores all objects, including all arrays, and all class variables (i.e. those declared "static").

The _stack_ stores all local variables, including all parameters.

When a method is called, Java creates a _stack_frame_ (also known as an _activation_record_) that stores the parameters and local variables for that method. One method can call another, which can call another, and so on, so the JVM maintains an internal _stack_ of stack frames, with the most recent method call on top, and "main" at the bottom. When a method finishes executing, its stack frame is erased from the top of the stack, and its local variables are erased forever.

Here's a snapshot of memory while Java is executing the "Test.main" method below. The stack frames are on the left. Everything on the right half of the page is in the heap. Read the stack from bottom to top, because that's the order in which the stack frames were created.




```
public class Test {
  public static void main(String[] args) {
    Fraction f1 = new Fraction(3, 5);
    Fraction f2 = new Fraction(9, 10);
    Fraction f3 = f1.quotient(f2);
  }
}
```

The stack is called a "stack" because it acts like a stack of trays at a cafeteria. You can only (conveniently) add a tray to the top of the stack or remove a tray from the top of the stack. A method call _pushes_ a new stack frame onto the top of the stack. When a method finishes executing, its stack frame is _popped_ from the top of the stack.

The method that is currently executing (at any given point in time) is the one whose stack frame is on top. All the other stack frames represent methods that are waiting for the methods above them to return before they can continue executing.

Each stack frame also contains a "program counter" that tells Java what line of code is currently executing in that method. Ever wonder how Java knows where to continue executing when it returns from a method? The stack frame for Fraction.reduce (pictured above) contains a program counter that points to the spot in the "reduce" method where Java should continue execution after the "gcd" method is done. (Your Java code cannot examine the program counters directly, though.)

You'll notice that three of the methods have "this" as a local variable, but "main" and "gcd" do not. That's because "main" and "gcd" are static methods, and the other methods are not.

Remember the "scope" of a variable? Observe that there are two local variables named "f", and three named "this", all in existence at the same time. If you write "f" or "this" in a method, how do you know which "f" or "this" you're referring to? Answer: at any given time, only the local variables in the TOP stack frame are in scope. So a method can only access its own local variables. (This will be important to understand when we learn about _recursion_ a few lectures from now.)

The java.lang library has a method "Thread.dumpStack" that prints a list of the methods on the stack (but it doesn't print their local variables). This method can be convenient for debugging--for instance, when you're trying to figure out which method called another method with illegal parameters that made it crash.

Parameter Passing

Let's reprise parameter passing, using stack frames. Recall that parameters are "passed by value"--they are copied. The copies reside in the stack frame for the method. A method can change its parameters (in its stack frame), but the changes are not visible outside the method call. The original values are not changed.

However, you saw in lab that when a parameter is a reference to an object, the reference is copied, but the object is not; the original object is shared. A method can modify an object that one of its parameters points to, and the change will be visible everywhere.

In this example, the method doNothing sets its parameter to 2, but it has no effect on the value of the calling method's variable a:

```
method:                          | STACK (just before the method returns)
                                 |
  static void doNothing(int x) { |      -----
    x = 2;                       |    x | 2 |
  }                              |      -----     stack frame for doNothing
                                 |----------------------------------------
method call:                     |
                                 |      -----
  int a = 1;                     |    a | 1 |
  doNothing(a);                  |      -----     stack frame for main
```

This example shows what the stack and heap look like while constructing an object:

```
method:                          | STACK              | HEAP
                                 |              IntBox|
class IntBox {                   |      -----         |
  int i;                         |    n | 2 |         |
  IntBox(int n) {                |      -----         |   IntBox object
    i = n;                       |      -----         |   ------
  }                              | this | .-+---------->|i |2|
}                                |      -----         |   ------
                                 |--------------------|      ^
method call:                     |      -----         |      |
                                 |    b | .-+----------------/
  IntBox b = new IntBox(2);      |      -----     main|
```

Note that the figure above actually shows fields at two different points in time, because b isn't made to point to the IntBox object until the constructor terminates, at which point its stack frame is erased.

Here's an example that shows how a method can change an object so that the change is visible to the calling method:

```
method:                          | STACK              | HEAP
                                 |               set3|
class IntBox {                   |      -----         |
  static void set3(IntBox ib) {  |   ib | .-+----------------\
    ib.i = 3;                    |      -----         |      |
  }                              |                    |      |
                                 |--------------------|      v
method call:                     |      -----         |    ------
                                 |    b | .-+---------->|i |3|
  set3(b);                       |      -----     main|    ------
```

Here's an example of a common programming error, where a method tries and fails to make a change that is visible to the calling method:

```
method:                          | STACK              | HEAP
                                 |            badSet4|
class IntBox {                   |      -----         |    ------
  static void badSet4(IntBox ib) { |   ib | .-+---------->|i |4|
    ib = new IntBox(4);          |      -----         |    ------
  }                              |                    |
                                 |--------------------|
method call:                     |      -----         |    ------
                                 |    b | .-+---------->|i |3|
  badSet4(b);                    |      -----     main|    ------
```