

CS 61B: Lecture 9
Monday, February 6, 2012

Today's reading: Sierra & Bates pp. 77, 235-239, 258-265, 663.

THE STACK AND THE HEAP

=====

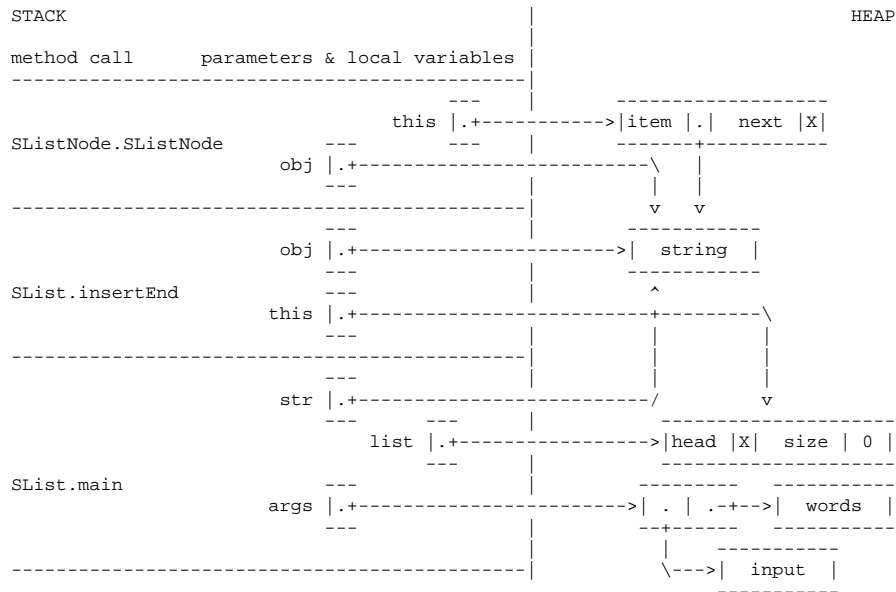
Java stores stuff in two separate pools of memory: the stack and the heap.

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, the Java Virtual Machine 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 "main" at the bottom, and the most recent method call on top.

Here's a snapshot of the stack while Java is executing the `SList.insertEnd` method. 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.



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

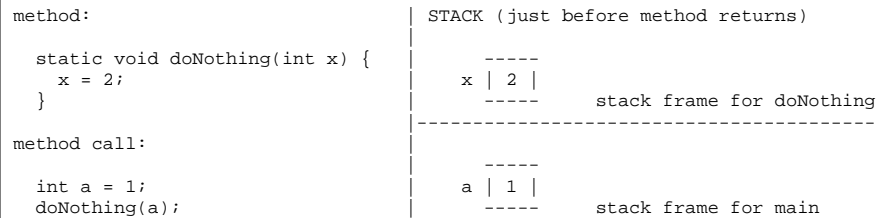
When a method finishes executing, its stack frame is erased from the top of the stack, and its local variables are erased forever.

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

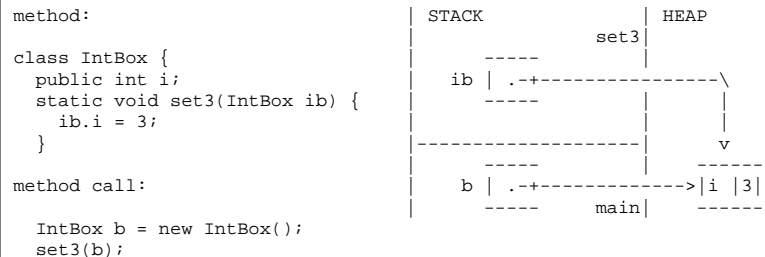
As in Scheme, Java passes all parameters `_by_value_`. This means that the method has `_copies_` of the actual parameters, and cannot change the originals. The copies reside in the method's stack frame for the method. The method can change these copies, but the original values that were copied are not changed.

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`:



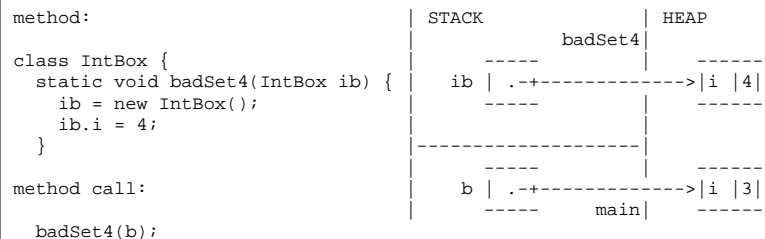
When the method call returns, `a` is still 1. The `doNothing` method, as its name suggests, failed to change the value of `a`, or do anything relevant at all.

However, 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. Here's an example that shows how a method can make a change to an object that is visible to the calling method:



For those of you who are familiar with programming languages that have "pass by reference," the example above is as close as you can get in Java. But it's not "pass by reference." Rather, it's passing a reference by value.

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. (Assume we've just executed the example above, so `b` is set up.)



- (1) Class variables (static fields) are in scope everywhere in the class, except when overridden by a local variable or parameter of the same name.
- (2) Fully qualified class variables ("System.out", rather than "out") are in scope everywhere in the class, and cannot be overridden. If they're public, they're in scope in `_all` classes.
- (3) Instance variables (non-static fields) are in scope in non-static methods of the class, except when overridden.
- (4) Fully qualified instance variables ("kayla.name", "this.i") are in scope everywhere in the class, and cannot be overridden. If they're public, they're in scope in all classes.
- (5) Local variables and parameters are in scope only inside the method that declares them, and only for the topmost stack frame.