Java By Abstraction - Companion Notes
Topic 2 - Application Programming Interface (APIs) PART 2

CSE 1720, Winter 2010, Version 1.0, Prepared by: M. Baljko

this is an alternative version that does not depend on prior knowledge about the primitive data types; this version incorporates information from sec 3.3.1.

Question:
What happens in memory when you launch the VM and tell it to invoke a particular application?

Let's say we are trying to run the application called `RectangleExample`, which is found in the file called `RectangleExample.java` and has been successfully compiled into bytecode that is contained in the file `RectangleExample.class`

```
import type.lib.Rectangle3;

public class RectangleExample {

     public static void main(String[] args) {
          Rectangle3 r;
          r = new Rectangle3();
     }

}
```

Answer:
In Part 1, we discussed how the VM loads the class definition of `RectangleExample` and the class definitions of all classes upon which `RectangleExample` depends. Last we left off, the VM was about to invoke the bytecode that corresponds to the first line of code:

```
          Rectangle3 r;
```

This line of code corresponds to a variable declaration.

<u>Act II, con't</u>

VM: ... (continuing)... I will invoke the first line of the main method of `RectangleExample`. The first line is a variable declaration. SM, I need you to arrange to have a variable called `r` set up.

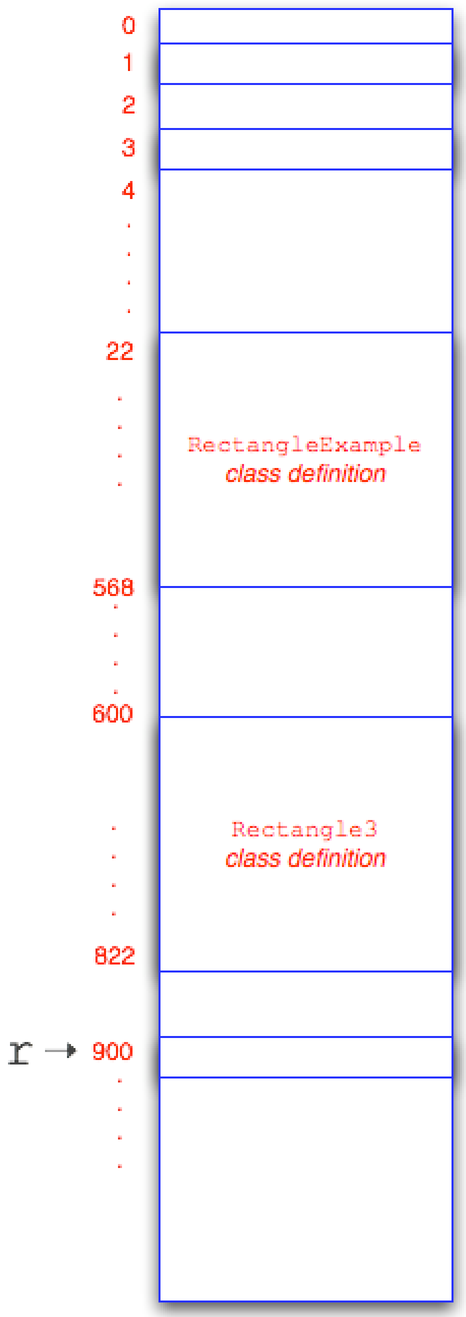
SM: OK, I will work on it, hang on a sec
(SM turns to the MM)
Hello MM. Please arrange for me to have a place in memory for the variable `r`. I know you are not a specialist in matters of representation, so let me tell you how much space this will need. The variable will need 4 blocks of memory.

[this is not a made-up number – non-primitive variables are used to store addresses and these addresses take up 4 block of memory]

MM: (turns away from SM. looks for and locates a place in memory at location 900 that has 4 blocks free. The MM marks the memory blocks from 900-903 as not being available for use, since they are already being used for something)
(MM turns back to SM) Done. The location is 900.

SM: thinks to itself.... hmm, let me update my symbol table.
(SM turns back to VM) ok Done.

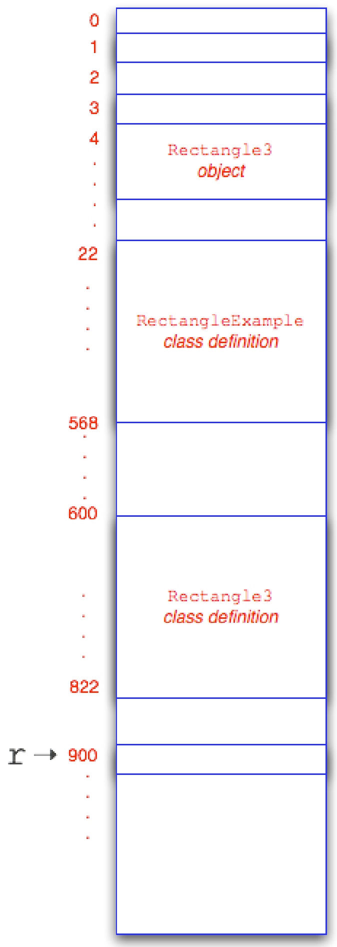
Here's what memory looks like:



Act III
Now the VM shifts its attention to the next line of code

VM: Now I will invoke the second line of the main method of `RectangleExample`. I see that this line is an *assignment statement* (see sec 1.3 of the textbook). When this statement is executed, I will perform the following three steps: (1) evaluate the right-hand-side (RHS), (2) ensure that the RHS and LHS are compatible, and (3) store the value of the RHS in the memory block of the LHS.
So I need to evaluate the RHS. I see that this is an *instantiation* – the main method is using the services of `Rectangle3` to create an object. There are no passed parameters, so I don't need to do anything further (if there were parameters, then I would need to evaluate them as well). So I get an instance of `Rectangle3` created in memory (making use of the SM to find the constructor `Rectangle3` that is contained within the class definition of `Rectangle3` and also making use of the SM to get the object put into memory at some location, which SM keeps track of). The "value" of the RHS is the memory location of the object, which the SM has reported to be 4 or (`00000000 00000000 00000000 00000100`) in binary.

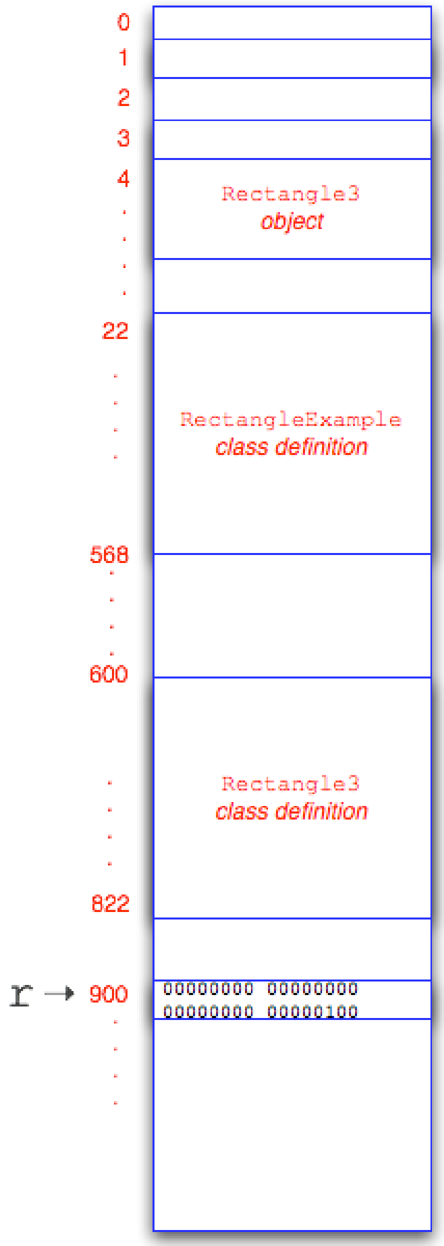
Here's what memory looks like:



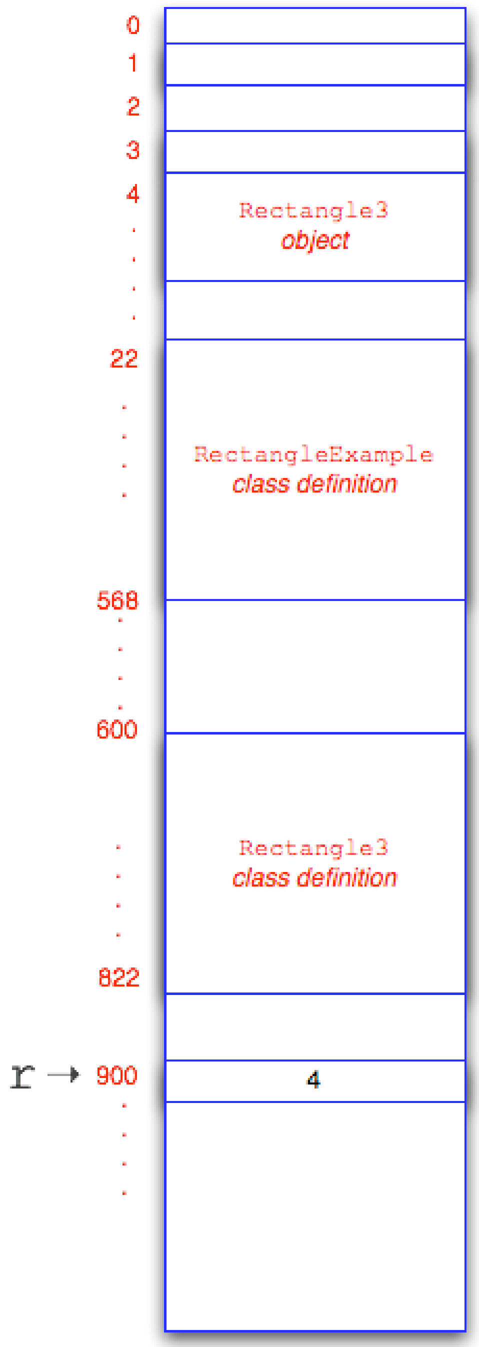
VM: Now I've complete step (1), so I will move onto step (2). Are the types of the the RHS and LHS of this assignment statement compatible? Yes, they are both `Rectangle3`. So this is ok.
Now what about step (3)? I need to ensure that the value of the RHS is stored in the memory block of the LHS.
(turns to SM) SM! hey! you need to assign the value `00000000 00000000 00000000 00000100` to the variable `r`. (**Note!** the VM doesn't actually know where r is stored – the SM does, though)

SM: ok, I know where `r` is stored, it is at memory location 900. So I'll put the value `00000000 00000000 00000000 00000100` there.

Here's what memory looks like:



But....
It can be time-consuming to deal with binary representations, so we'll just use the integer representation for 4 (but you should know that there is no "4" in memory, since memory consists only of 0's and 1's). So given this short-cut, here's what our memory diagram looks like:



<u>Act IV</u>
Now the VM shifts its attention away from the second line of code.
VM: Now I've finished this line of code. I see there's no bytecode left to invoke. Ok, I'll stop now. We'll shut everything down and stop running. (the contents of memory are lost).