Java By Abstraction - Companion Notes

Alternative Version of Case Study 2.3.4 "Meet the Managers" PART 1

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

This is an alternative version that, unlike Case Study 2.3.4 in the textbook, 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:
There are two issues:

1. keeping track of the memory blocks (which are in use and which are available to be used)
2. keeping track of the identifiers that refer to the memory block that are in use

To take care of the first issue, we have this thing called the Memory Manager (MM). Its job is to keep track of all the contents of memory and distinguish between blocks that are in use and block that are available to be used (Refresh your memory about the nature of memory by looking at IMD 1.4). The MM does not need to have expertise in how to represent things in those blocks (as we shall soon see, this expertise is *delegated* to the Symbol Manager). The MM talks to the system's Operating System (OS) to actually write representations to memory.

To take care of the second issue, we have this thing called the Symbol Manager (SM). The SM needs to have expertise in how to represent things in the blocks (i.e., how to use binary representations for a wide variety of things, such as integers, decimal numbers, characters, Strings, various kinds of objects, etc). The SM also needs to keep track of the *symbol table*.

Aside: recall the *symbol table* from sec 1.2.3 (p.18). It keeps track of which identifiers correspond to which memory addresses. Recall a memory address refers to a particular block of memory. Each entry in the table has three items: the identifier, the data type of the thing that the identifier refers to, and the block address where the thing is stored (in the case of things that require more than one block to be stored, the block address is the block where the thing starts to be stored).

As part of filling out the symbol table, the SM will talk to the MM to ensure that the representations get written to memory in the way that corresponds to the entry in the table.

Act I

When the VM starts up, the VM asks the SM to load the class `RectangleExample` into memory. `RectangleExample` needs to be in memory so that the VM can invoke each of the lines in it.

Here's how this take place, in the form of a narrative:

VM: Hi SM, could you load the class `RectangleExample` for me?

SM: OK, I will work on it, hang on a sec
(SM turns to the MM; the VM doesn't need to notice this, nor does the VM care who the SM talks to next, just as long as the VM's request gets fulfilled)

SM: Hello MM. Please arrange for me to have a place in memory for `RectangleExample`. I know you are not a specialist in matters of representation (nor should you be – it's not your job!), so let me tell you how much space this will need. It will be 546 memory blocks *[ed --- this is a made-up number]*

MM: (turns away from SM. looks for and locates a place in memory that happens to be at location 22 that is big enough to hold the class, puts the binary representation of the class there and marks the 546 memory blocks starting at block 22 as not being available for use, since they are now being used for `RectangleExample`.)
(MM turns back to SM) Done. The location is 22.

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

VM: OK, thanks. Now I will start invoking application `RectangleExample`. Stay tuned SM because I will need you again shortly.

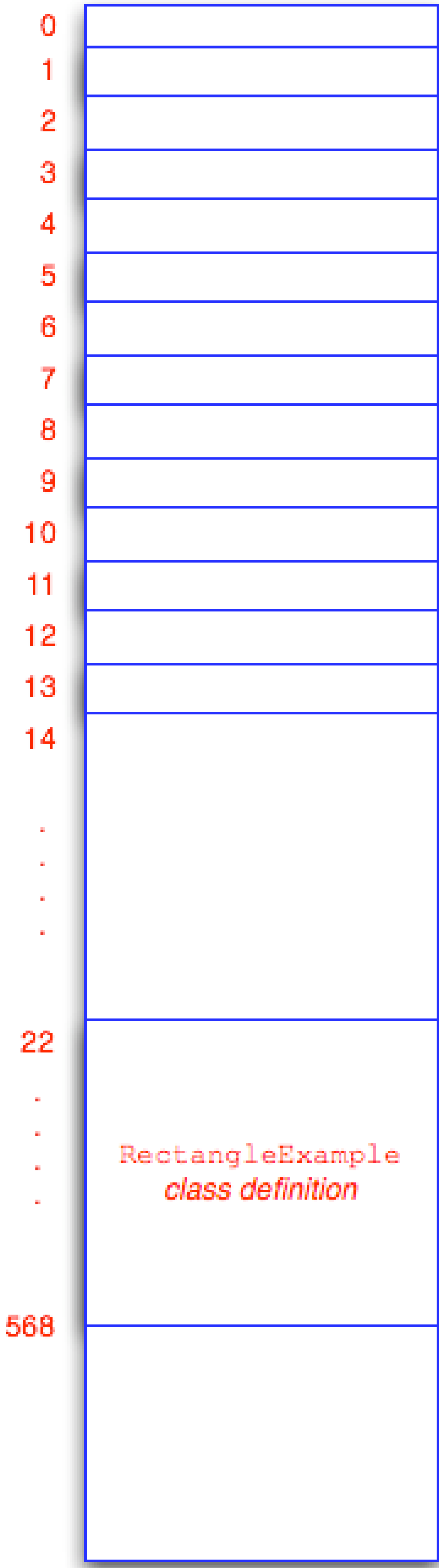
SM: (*thinks to self...*) What do you think I am? I'm not your bitch.

And so the class gets loaded in memory. When the application is running, the SM will ask the MM to store other things in memory; the MM will not accidentally overwrite anything, and the SM will keep track of everything.

Notice this is an example of layered abstraction. The VM and the running application speak with the SM, and the SM speaks with the MM. The VM and the running application never speak with the MM directly.

The VM and the running application are *clients* of the SM, and the SM is a *client* of the MM. Or conversely, the SM provides services to the VM and the running application, and the MM provides services to the SM.

Here’s what memory looks like:


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
22
RectangleExample
class definition
568

Act II

So now the VM is going to start running the application. The VM is getting the byte code instructions one line at a time and is performing the actions that each line specifies. For the sake of explanation, we'll refer to *lines of java code* as being invoked. Technically this isn't correct since it is actually the lines of bytecode being invoked (***not*** the lines of java). But since the bytecode is derived directly from the lines of java code by the compiler, let's just consider this to be a short-cut.

VM: ok, so `RectangleExample` is now loaded into memory. Let's start. Well, the first thing I notice is that this app requires the services of the `Rectangle3` class, so I'd better get that loaded into memory.
(turns to SM)
hey SM, hop to it

SM: whatever

VM: I need `Rectangle3 in` memory. You know the drill – whichever classes the application depends upon, I need to get them loaded straight away.

SM: OK, I will work on it, hang on a sec
(SM turns to the MM; again the VM doesn't need to notice this, nor does the VM care who the SM talks to next, just as long as the VM's request gets fulfilled)

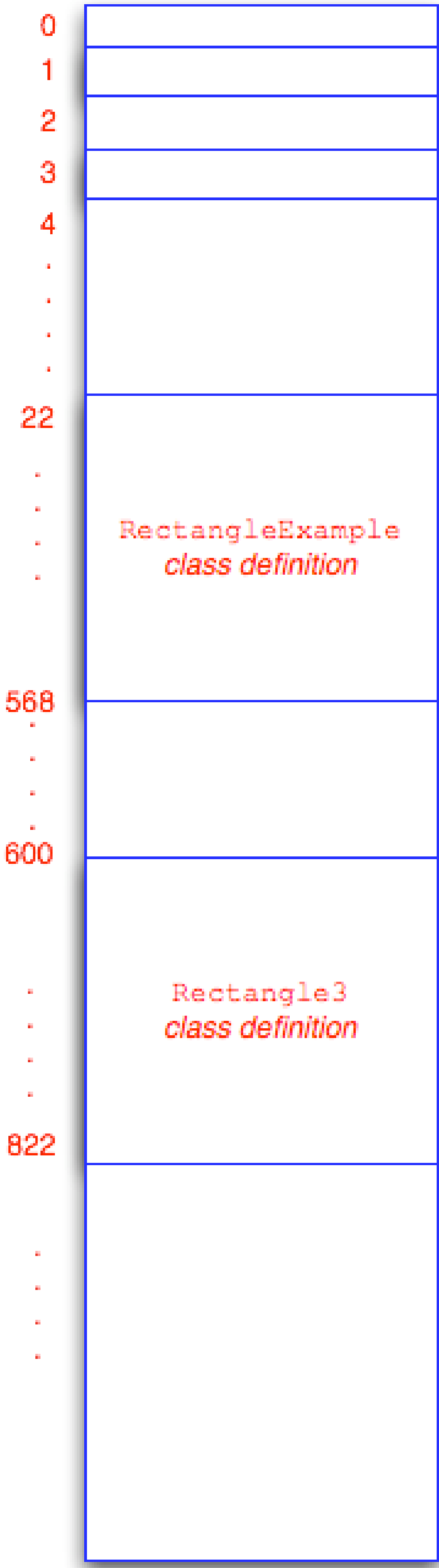
SM: Hello MM. Please arrange for me to have a place in memory for the class definition `Rectangle3`. I know you are not a specialist in matters of representation (yadda yadda), so let me tell you how much space this will need. It will be 222 memory blocks *[ed --- this is another made-up number]*

MM: (turns away from SM. looks for and locates a place in memory that happens to be at location 600 that is big enough to hold the class, puts the binary representation of the class there and marks the 222 memory blocks are not being available for use, since they are now being used for `Rectangle3`)
(MM turns back to SM) Done. The location is 600.

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

VM: OK, thanks. So now this is taken care of, 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.... *... to be continued ...*

here's what memory looks like:


0
1
2
3
4
22
RectangleExample
class definition
568
600
Rectangle3
class definition
822