Java By Abstraction Companion Notes – Chapter 1

CSE 1710, Fall 2013, Version 1.0, Prepared by: M. Baljko

The section 1.2.3 glosses over some aspects of declaration and memory that I think need to be clarified/extended. I've prepared a modified version of section 1.2.3. Some of the additions are shown in bold.

1.2.3. Declaration and Memory (pp.19-20)

Introduction
When we declare a variable, we are asking the compiler **to produce an instruction (in the form of some bytecode) that will, once the compiled program is running, cause the VM to** set aside an area of memory to hold the future values of the variable. Declaration is thus closely related to storage, and in order to develop a deeper understanding of declaration, we need to take a look at computer memory.

Memory can be viewed as a one-dimensional arrangement of cells, each of which is called a memory byte (see Fig 1.10). The total number of bytes is the size of memory in the computer, and since it is typically quite large, it is expressed in multiples of byte: a **kilobyte** (KB) is 1024 bytes, a **megabyte** (MB) is 1024 KB; and a **gigabyte** (GB) is 1024 MB. The bytes are numbered sequentially, starting from 0, and the number assigned to a byte is called its address. What you store in a byte becomes its content, and it is important to distinguish content from address. If you need to store something that does not fit in a byte, use a **memory block**, a group of consecutive bytes.

Bytes can be considered as analogous to seats in a theatre and addresses as tickets (a piece of paper with a seat number printed on it). If you purchase ticket number 6, then you are entitled to sit in seat number 6. The operating system (O/S) is analogous to the box office of the theatre: when the computer is turned on, the O/S is in possession of all tickets, but then it gives them to programs upon request. Specifically, when the Java Virtual Machine (VM) starts up, it requests a large block of tickets (addresses) from the O/S. This is called the VM's **heap space**.

What the Compiler is Doing…
When the compiler encounters a declaration such as:

```
int width;
```

it realizes that, at runtime, the VM will need to make use of 4 bytes of memory to store the value that will be associated with the variable `width`. Given this anticipated need, the compiler will create a series of bytecode instructions that, when invoked by the VM, tells the VM to set aside 4 bytes of memory from the heap space. Moreover, the bytecode instructions need to include the **starting address location** for the memory block, relative to the address `0` (which is the start of the VM's heap space). The compiler keeps track of declarations in the symbol table:

```
Identifier  Type  Block Address
width       int   8
```

In this particular example, the compiler will generate bytecode instructions to tell the VM to set aside 4 bytes of memory from the heap space starting at memory address 8.

Using the symbol table, the compiler can figure out the starting block addresses for subsequent declarations. For instance, suppose the next declaration is this:

```
double height;
```

The compiler needs to add a new entry to its symbol table and it examines the most recent entry. The starting address is 8 and the type is `int`, which means 4 bytes of memory. The compiler calculates that the next appropriate place in memory will be address 12 (address 8 + 4 bytes), resulting in the following:

```
Identifier  Type   Block Address
width       int    8
height      double 12
```

So the compiler will generate bytecode instructions to tell the VM to set aside 8 bytes of memory (since that's the size of a double value) from the heap space starting at memory address 12.

You can now see the pattern. For the next declaration, the entry in the symbol table will have block address 20 (address 12 + 8 bytes).

Another important aspect of compilation is **type checking**. That means every time the compiler encounters a statement that makes use of a variable, it asks a few questions:

- Is this variable previously declared? (e.g., does this have an entry in the symbol table?)
  If no, then COMPILATION ERROR!!! All variables must be defined before their use.
  The source code is rejected by the compiler as unfit and no bytecode is generated.

- Is this variable being used properly? (e.g., is it being used in a context that is appropriate for its type?)
  If no, then COMPILATION ERROR!!!
  The source code is rejected by the compiler as unfit and no bytecode is generated.

Memory Diagrams…
The memory diagram in Fig 1.10 depicts **the bytecode in action**. The variable is written on the left-hand side with an arrow point to the beginning of the allocated memory block, which is shown in blue. If, during runtime, the value of the variable is assigned, then we can update the memory diagram to reflect this.

This interpretation of the declaration statement allows us to give a precise meaning to the term **variable**. So far, we've had a pretty cavalier attitude. We define a **variable** to be the memory block reserved for it. We define a **variable name** to be the symbolic representation of the address of the memory block. In the theatre analogy, a **variable** is a seat, a **variable name** is the ticket for the seat, and the **variable's value** is whoever sits in the seat. (By lifting the armrests in between seats, you can create a "multi-byte" seat).