Memory Diagram Examples

Question C: Memory Diagram *<16 marks>*

Consider the following app:

```
import java.lang.Math;

public class MyApp {

  public static void main(String[] args) {
    double val;
    val = Math.PI;
  }

}
```

C–1. [2 marks] Was the import statement on the first line necessary? Explain why or why not.

the import statement was not strictly necessary; the app would have compiled even if the import statement had been omitted.

the javac compiler automatically assumes an import of the package `java.lang`. This holds for `java.lang` only and not any other package.

C–2. [2 marks] When you are asked to draw a memory diagram, what is the diagram actually representing, in terms of the real-world. Be specific.

the memory diagram is a visualization of the heap space that is allocated to the java virtual machine (JVM) at run time.

The heap space is an actual portion of working memory of the computer on which the JVM is running (working memory is the RAM part of the computer, the memory that is volatile —e.g., when you turn the power off, the contents are lost. It is not the hard drive, which is a storage device).

If you look at more detailed documentation, such as the wikipedia entry for the JVM, you will see a lot of additional detail (like different areas within the heap for different types of entities, such as objects vs class definitions). This is interesting, but for the purposes of this course we will not cover that level of detail.

C–3. [2 marks] In the case of the app above, how does the virtual machine know that the class `MyApp` will require the services of the class `Math` at run time?

when MyApp is compiled, the resulting byte code will contain the specification of which classes, if any, are required during run time. The class `Math` will be listed there.

The byte code for the class `Math`, in turn, will contain a specification of all of the classes which *it* needs at run time.

For any class that is needed at run-time, the byte code must be available to the JVM (i.e., it must be able to find the corresponding `*.class` file on the hard drive. This is one of the reasons the JVM has the *class path*, so the JVM knows where to look to find the classes that are required during run time. The *class path* is different than the *build path*. The *build path* is used by the compiler. The compiler uses the *build path* in order to know where to look in order to find the classes that are required in order to complete the compilation process.

C–4. [2 marks] Give a general overview of what happens in the JVM upon invocation of an app.

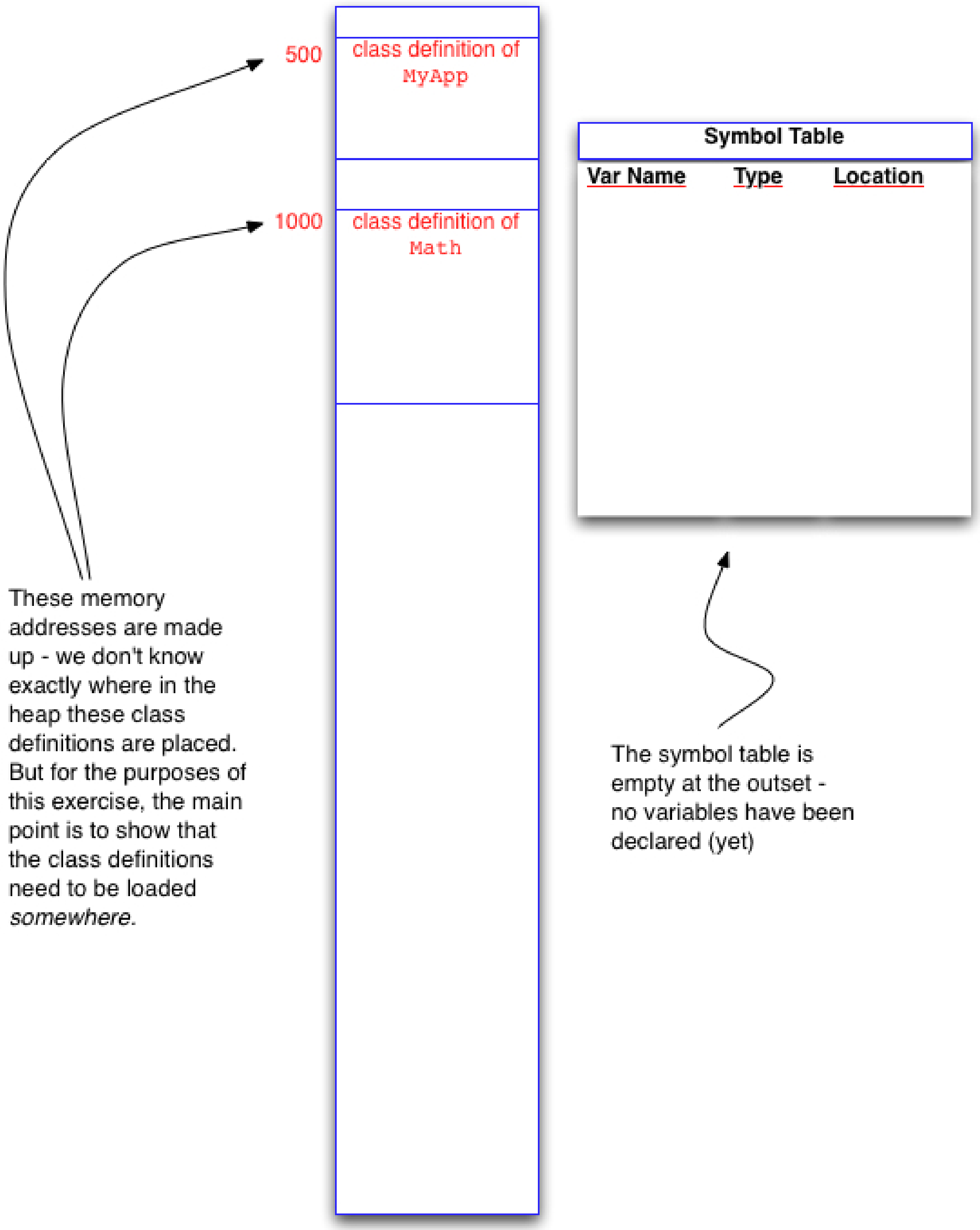
First, the class loader loads the class definition (the bytecode) of the class that contains the main method `MyApp`. Next it loads any and all class definitions (byte code) of classes that are used by the app (strictly speaking, some classes are loaded *on demand*, but this complicates things and we will assume for the purposes of this course that all of the required classes are loaded at the outset). At this point, the job of the class loader is done.

Next, the JVM will look to see what byte code instructions it needs to invoke. In the case of the example here, this byte code will be found in the byte code that corresponds to the main method of `MyApp`. The JVM will invoke the byte code corresponding to the first statement of the main method. Then it will invoke the byte code corresponding to the second line of the main method. And so on... until there are no further statements to be invoked. Then the execution portion of the JVM will complete.

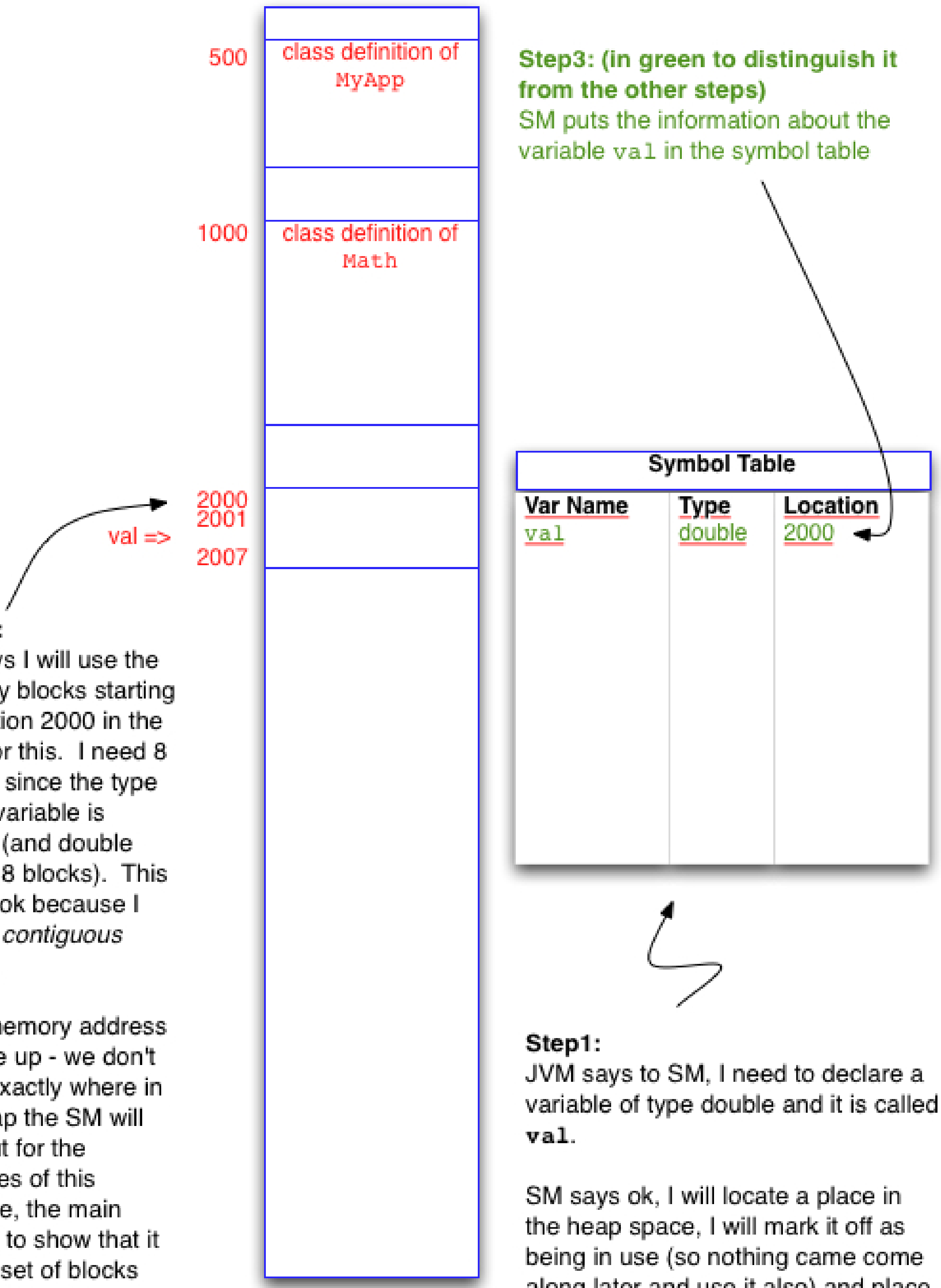
If, during a statement, there is some value that is derived, then the JVM makes use of a special area of workspace to temporarily hold the value. If there are variable declarations during run time, the JVM uses the symbol manager (SM) to keep track of them.

Last, there is a component that cleans up and shuts everything down. Then the JVM terminates.

C–5. **[2 marks]** Draw a memory diagram to show the contents of memory upon invocation of the program and up to **but not including line 6** of the example.


500
class definition of MyApp
1000
class definition of Math
Symbol Table
Var Name
Type
Location
These memory addresses are made up - we don't know exactly where in the heap these class definitions are placed. But for the purposes of this exercise, the main point is to show that the class definitions need to be loaded *somewhere*.
The symbol table is empty at the outset - no variables have been declared (yet)

C–6. [2 marks] Draw a memory diagram to show the contents of memory upon invocation of the program and up to **and including line 6** of the example.



Step3: (in green to distinguish it from the other steps)
SM puts the information about the variable `val` in the symbol table

Symbol Table

Var Name	Type	Location
`val`	double	2000

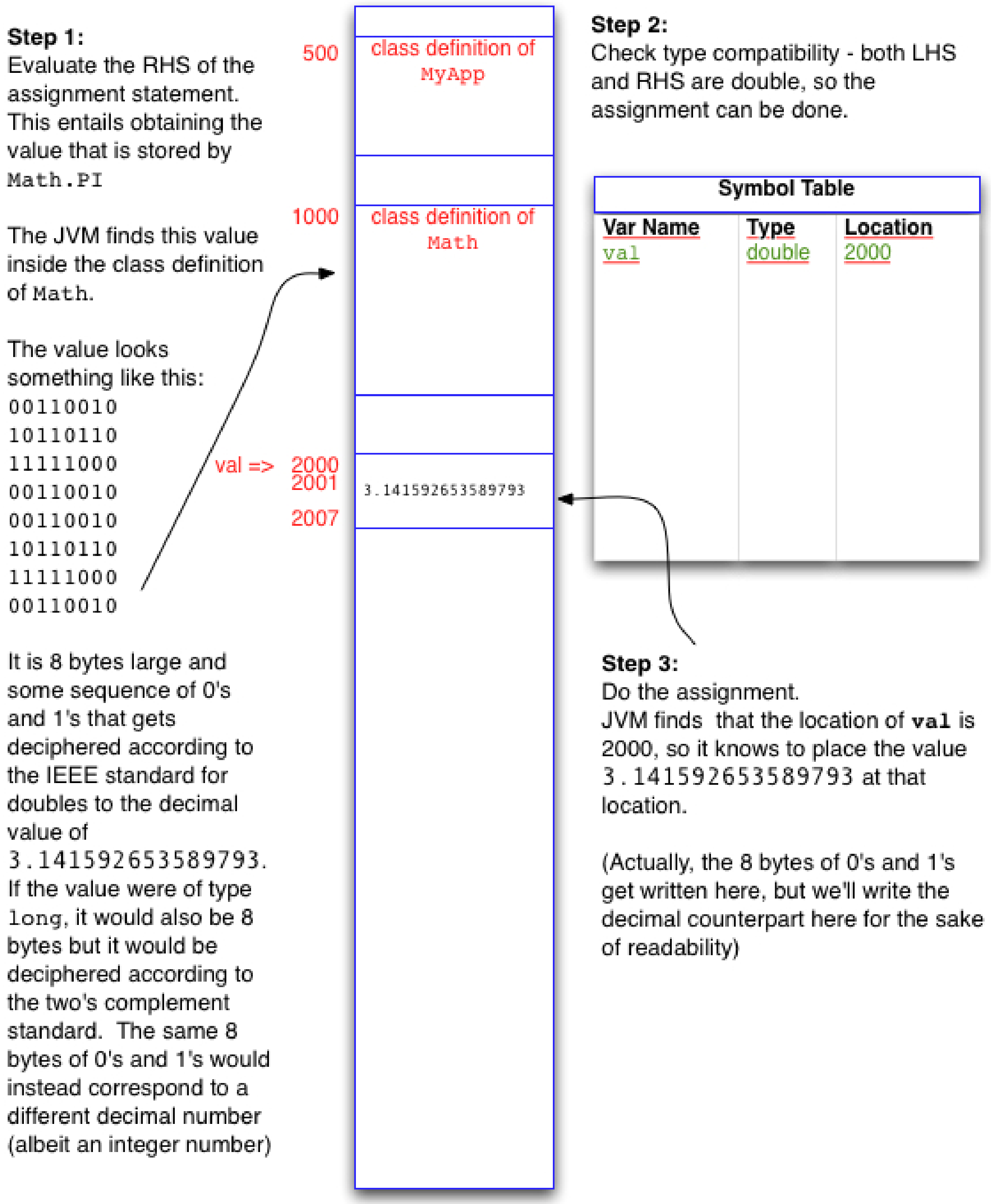
Step 2:
SM says I will use the memory blocks starting at location 2000 in the heap for this. I need 8 blocks, since the type of this variable is double (and double means 8 blocks). This spot is ok because I have 8 *contiguous* blocks.

(This memory address is made up - we don't know exactly where in the heap the SM will use, but for the purposes of this exercise, the main point is to show that it uses a set of blocks *somewhere.)*

Step1:
JVM says to SM, I need to declare a variable of type double and it is called **`val`**.

SM says ok, I will locate a place in the heap space, I will mark it off as being in use (so nothing came come along later and use it also) and place that information in my symbol table

C–7. [2 marks] Draw a memory diagram to show the contents of memory upon invocation of the program and up to **and including line 7** of the example.



Step 1:
Evaluate the RHS of the assignment statement. This entails obtaining the value that is stored by `Math.PI`

The JVM finds this value inside the class definition of `Math`.

The value looks something like this:

```
00110010
10110110
11111000
00110010
00110010
10110110
11111000
00110010
```

It is 8 bytes large and some sequence of 0's and 1's that gets deciphered according to the IEEE standard for doubles to the decimal value of `3.141592653589793`. If the value were of type `long`, it would also be 8 bytes but it would be deciphered according to the two's complement standard. The same 8 bytes of 0's and 1's would instead correspond to a different decimal number (albeit an integer number)

Step 2:
Check type compatibility - both LHS and RHS are double, so the assignment can be done.

Symbol Table

Var Name	Type	Location
`val`	double	2000

Step 3:
Do the assignment.
JVM finds that the location of **`val`** is 2000, so it knows to place the value `3.141592653589793` at that location.

(Actually, the 8 bytes of 0's and 1's get written here, but we'll write the decimal counterpart here for the sake of readability)