

CSE 1710

Lectures 11, 12

Memory Diagrams, Input Validation

Background Material

- Selected readings from JBA concerning memory diagrams
 - sec 1.2.3
 - sec 3.3.1
 - sec 4.2.1, 4.2.2, 4.2.3

What are memory diagrams?

- a visualization of the heap space that is allocated to the java virtual machine (JVM) at run time
- the heap space is a portion of working memory used by the JVM for dynamic memory allocation
 - Key aspects of dynamic memory allocation
 - allocate memory to the Java program as the program needs it
 - free memory for re-use when it is no longer needed

JVM basics

- when an app is compiled, the resulting byte code will contain the specification of which classes, if any, are required during run time.
 - the byte code for those classes, in turn, will indicate the class files upon which they depend
 - thus, an app may depend on many classes, either directly or indirectly
- when an app is invoked by the JVM, all of the classes upon which the app depends must be available to the JVM
 - the JVM must be able to find the corresponding *.class file on the hard drive.
 - the class path says where to look

JVM basics

- when an app is compiled, the resulting byte code will contain the specification of which classes, if any, are required during run time.
 - the byte code for those classes, in turn, will indicate the class files upon which they depend
 - thus, an app may depend on many classes, either directly or indirectly
- when an app is invoked by the JVM, all of the classes upon which the app depends must be available to the JVM
 - the JVM must be able to find the corresponding *.class file on the hard drive.
 - the class path says where to look

JVM basics

1. the class loader

- loads the class definition that contains the main method
- loads any and all class definitions (byte code) of classes that are used by the app*

2. bytecode execution

- execute the byte code that corresponds to the first statement of the main method.
- then the byte code corresponding to the second line of the main method.
- And so on... until there are no further statements to be invoked.

3. Tidy shut down.

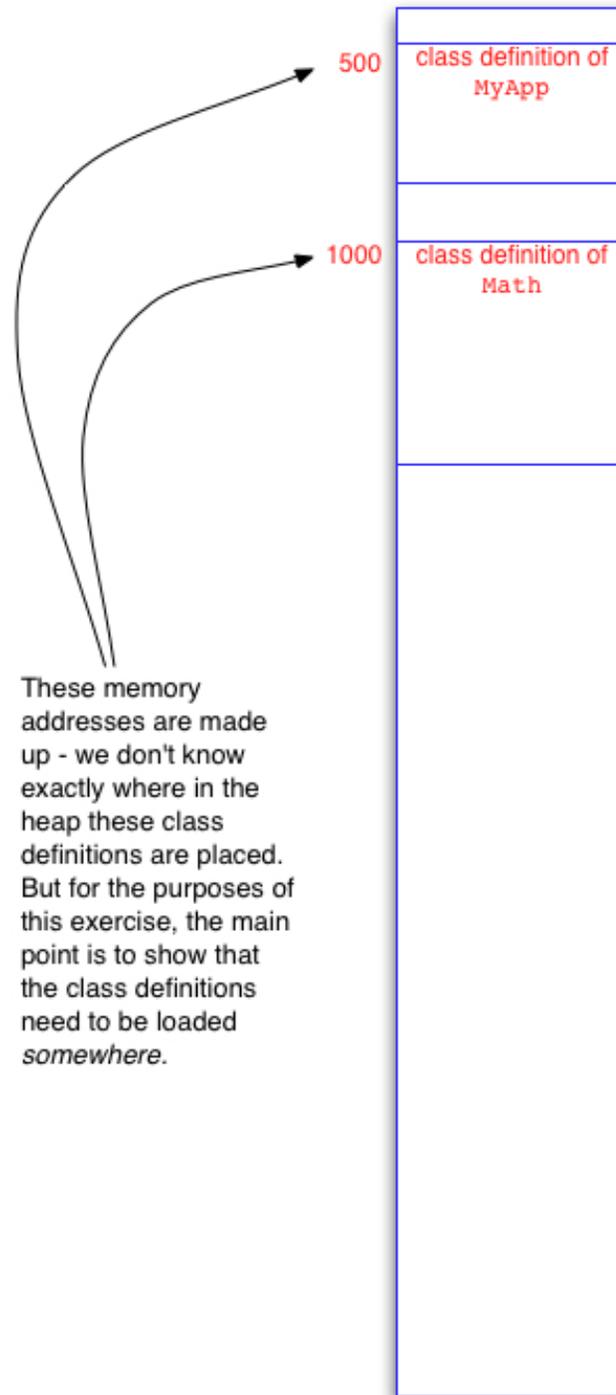
*strictly speaking, some classes are loaded on demand, but this complicates things and we will assume for the purposes of this course that all of the required classes are loaded at the outset).

Example

```
1 import java.lang.Math;
2
3 public class MyApp {
4
5     public static void main(String[] args) {
6         double val;
7         val = Math.PI;
8     }
9
10 }
```

Memory Diagrams

up to but not including line 6



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*.

Symbol Table		
<u>Var Name</u>	<u>Type</u>	<u>Location</u>

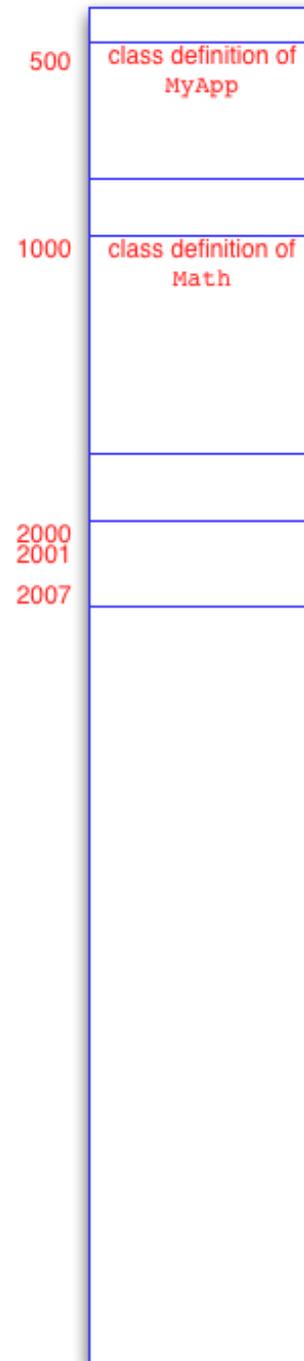
The symbol table is empty at the outset - no variables have been declared (yet)

Memory Diagrams

up to and including line 6 (variable declaration)

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*.)



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
<code>val</code>	<code>double</code>	<code>2000</code>

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

Memory Diagrams

up to and including line 7 (variable assignment)

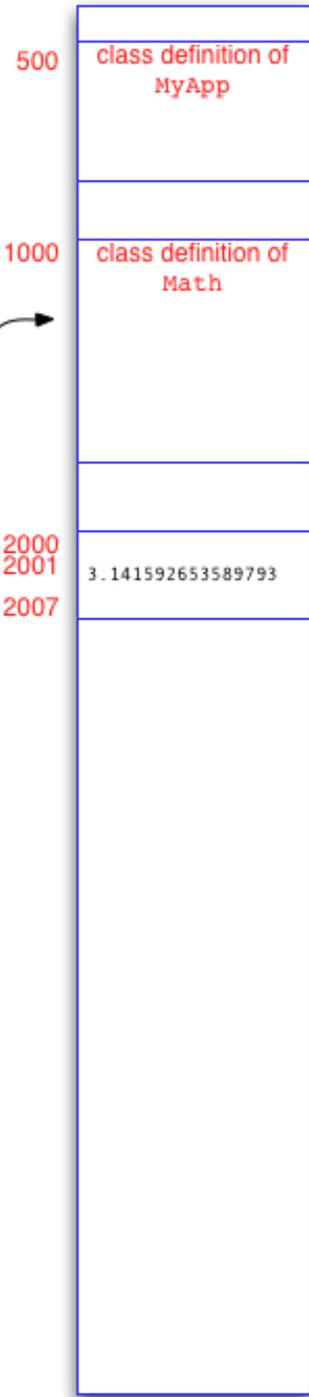
Step 1:
Evaluate the RHS of the assignment statement. This entails obtaining the value that is stored by `Math.PI`

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
<code>val</code>	<code>double</code>	<code>2000</code>

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.

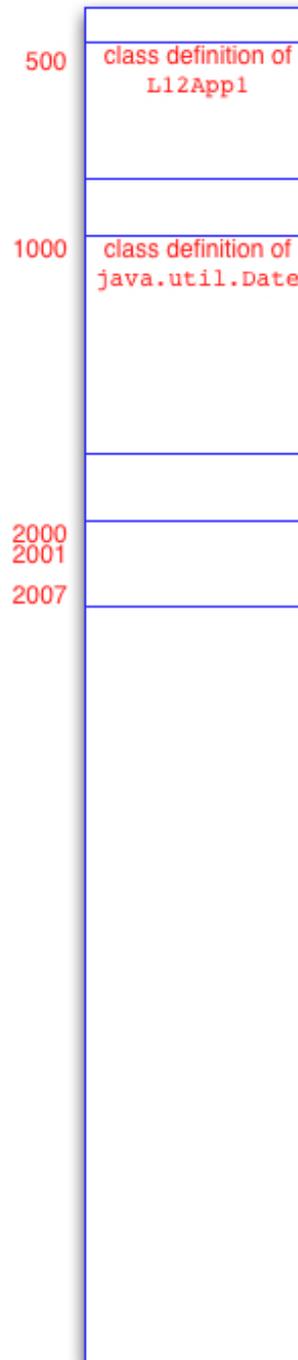
(Actually, the 8 bytes of 0's and 1's get written here, but we'll write the decimal counterpart here for the sake of readability)

Example – object creation

```
1 import java.util.Date;
2
3 public class L12App1 {
4     public static void main(String[] args) {
5         Date d;
6         d = new Date();
7     }
8 }
```


Memory Diagrams

up to and including line 5 (the variable declaration)



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 an object and object references are 8 bytes. This spot is ok because I have 8 contiguous blocks.

(*) assumes 32-bit JVM; there are also 64-bit JVM implementations

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
<u>d</u>	<u>Date</u>	<u>2000</u>

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

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

Memory Diagrams

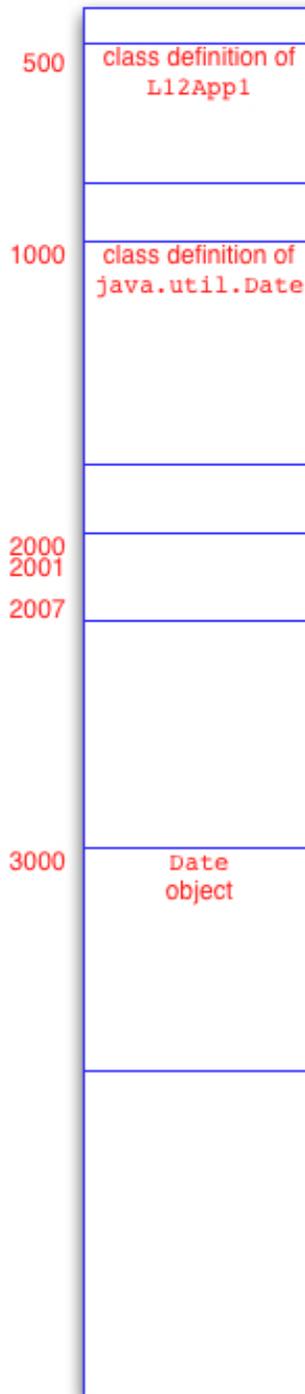
up to and including the RHS of line 6 (object construction)

Step 1:
Evaluate the RHS of the assignment statement. This entails object creation.

Use the services of the Date object to construct a new object.

d =>

The JVM places the object somewhere in the heap space that is appropriate (large enough). This location has some address (in this example it is 3000)



Symbol Table		
Var Name	Type	Location
<u>d</u>	<u>Date</u>	<u>2000</u>

Memory Diagrams

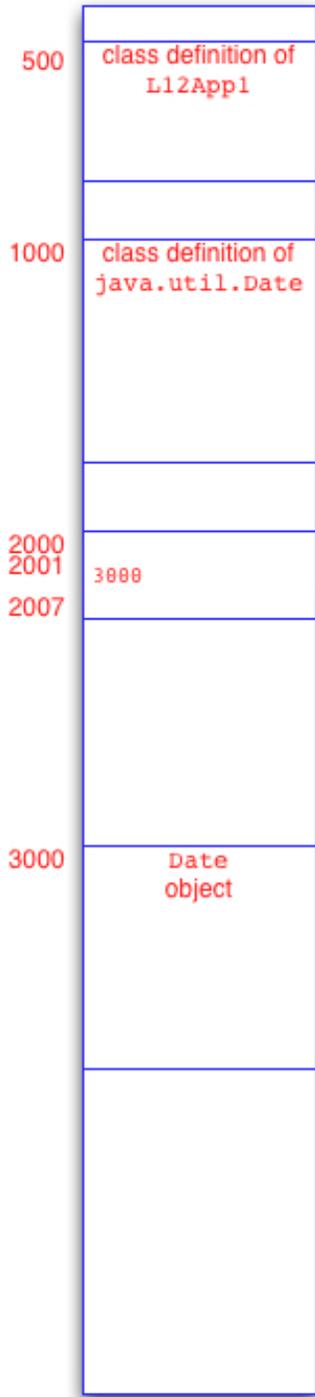
up to and including the entire line 6 (object construction and reference assignment)

Step 1:
Evaluate the RHS of the assignment statement. This entails object creation.

Use the services of the Date object to construct a new object.

d =>

The JVM places the object somewhere in the heap space that is appropriate (large enough). This location has some address (in this example it is 3000)



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

Symbol Table		
Var Name	Type	Location
<u>d</u>	<u>Date</u>	<u>2000</u>

Step 3:
Do the assignment. The JVM knows the location of the Date object is 3000, so it knows to place the value 3000 at the location corresponding to variable d.

Actually, the bytes in location 2000-2007 will be a 64-bit binary representation of the decimal location 3000.

Example – object creation

```
1 import java.util.Date;
2
3 public class L12App2 {
4     public static void main(String[] args) {
5         Date d;
6         d = new Date();
7         Date d1;
8         d1 = new Date();
9         // is d==d1?
10    }
11 }
```

Example – object equality

```
1 import java.util.Date;
2
3 public class L12App3 {
4     public static void main(String[] args) {
5         long now = System.currentTimeMillis();
6         Date d;
7         d = new Date(now);
8         Date d1;
9         d1 = new Date(now);
10        // is d==d1?
11        System.out.println(d==d1);
12        System.out.println(d.equals(d1));
13        //how do you explain this???
14
15    }
16 }
```

Input Validation – Exception-Based Approach

```
boolean cond = amount < 0;  
...  
String msg = "The inputted amount was negative";  
...  
ToolBox.crash(cond, msg);
```

Input Validation – Message-Based Approach

```
boolean cond = amount < 0;
...
String msg = "The inputted amount was negative";
...
if (cond) {
    output.println(msg);
}
else {
    //rest of program
}
```

Input Validation – Friendly Approach

```
boolean cond = amount < 0;
...
String msg = "The inputted amount was negative";
...
for (n=input.nextInt(); n<=0; n=input.nextInt())
{
    output.println(msg);
}
```