**JAVA MIDTERM STUDY GUIDE**

**This Study Guide will go module by module and cover everything from M1-M5**

**By: Puneet Anand**

**For: Rich Smith**

**PART 1 – BASICS (The famous HelloWorld.java)**

The program HelloWorld.java covers everything from this module. Here we learn to print out a basic string and is done as follows:

A screenshot of a computer

Description automatically generated with medium confidence



Here we can see the use of a System.out.println() statement and the use of a main class where the JVM knows to start the program. We can see a class is created called HelloWorld and it is made public, we will get to these return types momentarily.

**\*\*NOTE\*\***: A quick bit of good coding practice can be seen above, the indenting for the System.out.println() shows it is within the main method above it with braces to close and start the method, followed by a bracket to close the class. The class is the furthest out right with everything within the class being indented left. The main method is indented to the left and the print statement is within the method so this is also indented further left within the method itself. All of this follows syntax guidelines but also makes the code much more organized and easier to follow.

**PART 2 – JVM, VARIABLES/VARIABLE TYPES, Casting**

**JVM Memory Footprint**

The JVM (Java Virtual Machine) memory footprint consists of the Stack, Heap and Method as shown in the diagram below.

Text

Description automatically generated

**Method**

All implementations of the JVM in Java programming include a Class Loader which loads bytecode from the .class file to the Method Section of the JVM memory shown above. The Class Loader will first load the class that contains the main method into the Method section of the memory footprint as shown above. Here the JVM starts running the code in the main method.

**Stack**

Text

Description automatically generated

The current stack frame contains the local variables but for each method we must remember that a new current stack frame is made and then destroyed when the method exits as well.

All local variables inside of a method are stored in the current stack frame, when a method ends the stack frame is destroyed, the local variables stored in the destroyed frame are also destroyed.

An important note is that the JVM can only access local variables in the current stack frame.

**Variables**

In Java all variables have a very specific type that the variable belongs to. A variable in java is created as following:



Here int is the variable type, temperature is the variable name, and the variable has a initial value of 20.

**Variable Scope:** How accessible a variable is within the class. An instance variable can be accessed anywhere in the class where it is created. A local variable can only be accessed within the method it is declared.

**Variable Types**

**Local:** variables that are found inside of a method and stored in the current stack frame.

Text

Description automatically generated

Here the local variable that can only be accessed within getRectangleArea is areaRect.

**Instance:** declared at a class level and stored in the object on the heap and are declared at a class level. In the following class, the instance variable is temperature.

Text

Description automatically generated

**Primitive:** begins with a lower- case letter, things like int, char, boolean, double.



**Reference:** stores address of where to find the object on the heap (kind of like pointers in C). Usually starts with an UpperCase instantiation of the object.



**byte:** holds a signed **8-bit** integer value. Ranges from -128 to 127, with a total of 2^**8** = 256 possible integer values. **PRIMITIVE**

short: holds **16-bit** integer value, with a total of 2^**16** = 65536possible integer values. **PRIMITIVE**

**int:** holds **32-bit** integer value with a total of 2^**32** possible integer values. **PRIMITIVE**



**double:** Known as part of the Floating Point Variable set, a double holds a signed **64**-bit value which can hold decimal values. **PRIMITIVE**



**char:** a single character value. 1 byte = 8 bits. **PRIMITIVE**



**String:** a special class that is made up of char type characters**. REFERENCE CAPITAL “S”.** It looks like a primitive but it was just done that way to make it easier to write code. Strings are immutable (cannot be changed, instead when a String is concatenated, JVM makes a brand new string.



**boolean:** Boolean variables answer TRUE and FALSE equations in a sense where they can be used to make comparisons and get a return of either true or false. 2 bytes = 16 bits. **PRIMITIVE**



**Casting**

Timeline, calendar

Description automatically generated

You can’t go double -> int but you can go int -> double. For example:

This works:



This doesn’t:



Casting example:

Text

Description automatically generated

**PART 3 METHOD IN ACTION WITH JVM**

In Module 3 we went over methods in depth. A method is like a function in math/calculus where it needs variables, called parameters in Java, to be passed into the function (method in Java). From the below code we can see examples of methods that print out a “Hello” String statement, then adding a name to the same statement and finally finding the average between 3 double values.

We get an idea of how a return statement works when working with the calculateTheAverage function from assignment1, where it returns the return type mentioned in the method header. For each function below we can see that the method header includes either void or double, which highlights the return type of the function along with the type of the parameter and parameter name found in the parenthesis of the method header as well.

**\*\*NOTE\*\*** Notice how comments are created and used with “//”, comments can help explain the intention of the code, explain variables, add notes to make the code easier to follow and provide tips to the user.

Text

Description automatically generatedGraphical user interface, text, application

Description automatically generated

An example of how the JVM handles this class and method can be seen here:

Table

Description automatically generated

**JVM Rules**

Application

Description automatically generated with low confidence

**Good Coding Practice and camelCase**

Table

Description automatically generated

Above you can find examples of good relevant names for variables but also examples of camelCase, which has the first word start with a lower-case letter and every other word start with a Capital for easy to read and locate variables.

Remember that clean code reads like a book: here is an example of good and bad code:

Text

Description automatically generatedGraphical user interface, text

Description automatically generated

**BAD**  **GOOD**

**PART 4 – Objects, Classes, OOP, Objects on the Heap**

A Class is the blueprint or instructions for creating objects in computer memory (class is on the hard drive but the object is in the memory of the computer). CLASSES ALWAYS HAVE WORDS STARTING WITH CAPITAL LETTERS.

You will need a reference variable to point at the memory where the object is stored.

A class is the code that will tell the JVM how to build an object, objects on the other hand are instantiated by the JVM into the heap area of memory. The JVM will follow the instructions found in the class file itself.

**JVM** – interprets bytecode into machine code

**Compiler** – turns source code into bytecode

**JRE** – contains JVM and Java API

**Class** – The blueprint for instantiating objects

**Object** – an instantiation of a class into the memory of the computer

**Object Instantiation**: when an object is instantiated, there is generally a variable which points to the place in the heap memory at which the object is saved. In the calling file below, under QuoteGenerator, we can see that the Scanner class is used to instantiate scan, a new Scanner. The variable new points to a place in the heap memory where this instantiation of the object, Scanner, is used. Note that the Scanner class is imported from the Java API through java.util.Scanner;.

Text

Description automatically generated

\*\***NOTE\*\*:** Notice how the string is manipulated in order to create full sentences. /n allows for a new line. We can see that we start with a String ending with a space to account for the space after the final word, then we use the + operator to add on the customer name and other variables to the String itself. By using the + we are able to convert the variable input into a string and simply add it onto the entire sentence. To keep code neat using += will continue to add to the sentence while keeping code neat and easier to read.

Text

Description automatically generated

**Memory: Heap**

Timeline

Description automatically generated with low confidence

**Heap vs Stack Memory**

From this example from class we can see that there are two variables being created, length and rnd.

length holds a primitive variable type, int, and so the value 5 is saved to length which is saved onto the stack in the current stack frame. length is a local, primitive variable.

rnd is created and an object is written to the Heap of type Random. The address of the object is saved to rnd on the Stack. rnd would be a local variable which is also a reference variable.

**Scope**

Scope is the portion of code where a variable exists and is usually only accessible within the code block in which they are written.

**\*\*NOTE\*\*** A code block is all code within two “{ }”. All code inside of a set of squiggles is a code block.

We can have instance variables which have a scope of the entire class, but there can be local variables created within classes as well, but those local variables can only be accessed within the method in which they are created.

**CLASSES IN THE JVM**

Source Code: stuff we write -> saved as a .java file -> send to compiler

Compiler: Compiles the code into an executable (javac.exe), creates ByteCode which is stored in the class itself.

After the compiler has created ByteCode which is stored in the class itself, Java can now instantiate an object of the class that was created, after instantiated the class can be used to perform methods.

When the JVM sees the “new” keyword, it goes to find the ByteCode which holds the instructions on how to create the object out of the class instructions provided by the SourceCode.

Text

Description automatically generated

We did this Person class during class time.

Here we can see that first the JVM looks for bytecode for the Person class into the method area of memory, then the JVM will look for the bytecode for the main method and load that into the Method memory as well.

Next at line 7 we see a main method is created thus a new current stack frame is made too. At line 9 in the current stack frame the int a is made and holds the value 5.

At line 10 the JVM loads a the class byte code into the Method memory (the bytecode will also hold the methods within the class, literally everything in the class. At line 10 again, p1 will be loaded onto the stack with an address that points to the new Person, p1, which is instantiated onto the heap.

Next at line 11 the JVM will travel to the address held in the Stack by p1 and change the name instance variable to “Bart” in the heap. At line 12 the JVM follows the address held at p1 in the Stack and changes the age instance variable to 14 in the heap.

Finally on line 13, the JVM will follow the address at p1 to the sleep method, load the bytecode in the Method memory, since a new method is entered a new current stack frame is created, when sleep method ends, that current stack frame is destroyed, and when the entire main method ends, that current stack frame is also destroyed and the JVM realizes no more stack frames and ends the program.

**PART 5 – User Input, If Statements, Switch, NOT Operator**

**Scanners**

Text

Description automatically generated

Scanners help us get user input in programs. You will have to do: import java.util.Scanner or java.util.\* to just import everything. A scanner needs to be instantiated as an object as the class is used to create an object.

From Module 5, we go through an example together where the input of a scanner is used in various ways. (Code Above)

To begin the first input of the scanner is scanned for any input to find a String value, and then it is saved under quote.customerName.

Next by using scan.NextDouble(), the scanners input is converted from a String into a Double, and saved under quote.lengthOfRoom.

**\*\*NOTE\*\***: To convert the input into an int instead you would use scan.NextInt()

**Boolean Operators**

**== :** checks if the variables or values on both sides of the “==” are the same, returns true or false accordingly

**!=**: checks if the variables or values on both sides of the “!=” are not equal, returns true or false accordingly

**>:** checks if the variable or value on the right side of the “<” is less than the value or variable on the left side. Returns true if it is, false if not.

**<:** checks if the variable or value on the right side of the “<” is greater than the value or variable on the left side. Returns true if it is, false if not.

**>=:** checks if the variable or value on the right side of the “<” is less than or equal to the value or variable on the left side. Returns true if it is, false if not.

**<=:** checks if the variable or value on the right side of the “<” is greater than or equal to the value or variable on the left side. Returns true if it is, false if not.

**&&:** if both sides of the operator are not 0, then this returns true, if not, returns false.

**||:** or operator, if either side of the operator consists of a value that is not 0, then it returns true, else returns false.

**!:** Not operator, basically flips the variable. If a == true, then !a==false.

**If Statements**

An if statement if created through the following general formula:

Text

Description automatically generated

Under the what is being considered brackets, there will generally be a Boolean expression to get a true or false condition.

else if is used if there is another condition to be considered before arriving at the else statement, which can be seen as the default answer if all other conditions are not met.

**Scope for If Statements**

Text

Description automatically generated

In the above code sample I made, we can see that a and b are both of type int and are declared prior to the if statement. Due to this a and b can be accessed through out the Class file (**\*\*QUICK NOTE ALL JAVA PROGRAMS ARE MADE WITH JAVA CLASSES**). Within the first if statement we can see that the int c is created and given the value 60. Here the int c cannot be accessed outside of the if statements, and if we wanted to do a System.out.println(c) outside the if statement, we would not be able to access the variable c or the value it holds.

**Nested If Statements**

Text

Description automatically generated

Above we can see the use of nested if statements where there are multiple conditions that must be passed in order to achieve and output or return. Nested if’s make code easier to follow and less messy than having many conditions within the initial if statement.

**Switch Statements**

Switch statements literally do the same thing as if statements, however they are much neater. (had to switch to Sublime here, swapped to my laptop)

Text

Description automatically generated

To learn Switch statements in Module 5 we compared the use of if, else statements with switch statements. Both the if code block and the switch code block do the exact same thing. Instead of having if and else cases, we have cases.

For switch you start with a switch statement which looks like switch(the variable to watch). In the above code, the JVM monitors the int month, and when the user input is a 1 then we can see that friendlyMonth will change to the String, “January”, for 2 it will go to “February” and so on after a case is stated, there must be a break statement to tell the JVM to continue to the rest of the code outside of the switch block. A switch code block also includes a default, where if the variable being considered does not match a case, the program doesn’t crash but had a default action to make. In this case it is to set friendlyMonth to “I don’t know!”.

REMEMBER SAME AS IF, ELSE IF, ELSE STATEMENTS, JUST MUCH NEATER.

**Shortcuts For If Statements**

Text

Description automatically generated

Similar to what was taken up during Module 5, to take it one step further. We can see the myVar gets checked whether it is true or false, in this example myVar would be equal to true. Since it is true after the “?” (the “?” shows the JVM what to do, if myVar is true then it would assign what is to the left of the colon, in this scenario 55, if it was false it would assign what is to the right of the colon, which would be 10. In this example if we did System.out.println(myVar); we would get 55. To confirm this theory we input the same code into an online compiler including the println statement and get:

Text

Description automatically generated