Overview 1:

The JVM, Variables, and Casting

Please note that this review document (or any other document posted to T-Square) is meant only to supplement the instructions that you receive in lecture and recitation. I strongly advise you to attend your scheduled classes because these overviews are not a comprehensive set of explanations for every topic covered in CS 1331. Also, you should not expect to see an overview for most recitation topics.

The Java Virtual Machine:

A programming language is an abstraction, a sort of tool that we use because humans and machines don't speak the same language. We use high-level languages as a way of formatting our commands so that they can be easily translated into commands that a computer can recognize. Generally speaking, we can divide programming languages into two types:

1. **Interpreted Languages:** These are languages that are fed through an interpreter program, which translates each bit of code line by line into commands that can be understood by a computer's CPU. Python and Perl are both interpreted languages.
2. **Compiled Languages:** These are languages that are run through a program called a compiler, which translates the entire program before the computer can receive any instructions. Java, C, and C++ are a few examples of compiled languages.

Compiled languages are capable of running significantly faster than interpreted languages, because all of the translation into machine code happens just once before the code is ever run (as opposed to interpreted languages, which have to be interpreted anytime a program is run as it is running). An issue that you run into with most compiled languages, however, is that the machine code that you generate using the compiler is specific to the hardware it was compiled for. To solve this problem, Java takes a dual approach:

1. Source code written in Java is compiled into **bytecode**, an intermediate language that is halfway between high-level and lower level code. This code can be found in a file with the same name as the compiled Java file but with the ".class" extension.
2. This code is then fed into a program called the **Java Virtual Machine (JVM)**, which interprets the bytecode, translating it line by line into **machine code**. This process is

aided by the **Just-In-Time (JIT)** compiler, which speeds up the translation of the bytecode by taking the code for the most-called methods in the program and compiling them directly into machine code which is stored in temporary memory. Any time the interpreter encounters a call to one of these methods, it invokes the compiled version of the method to save itself the trouble of translating the same method multiple times. In this sense, you can think of the JIT compiler as the author of a bytecode-to-machine-code dictionary that allows the interpreter to quickly translate parts of the program that are used many times.

The advantage to this approach is that compiled Java files are **portable**, meaning they can be compiled on one machine and then run on any other machine which also has the JVM, regardless of what hardware that machine is using.

Variables:

Just like in algebra, **variables** are names that we assign to values. These values may change over time. Each variable has three parts: a type, a name, and a value. The last two parts are pretty straightforward: the **name** is just a set of characters that we use to reference some piece of data when we're programming (e.g. if x = 1, then when I say x + 5, I'm actually saying 1 + 5) and the **value** of a variable is the piece of data that we are associating with that name (in the case of the last example, the value is the 1 that the x variable represents). In addition to this, each variable is also of a particular **type**. This refers to the kind of data that we are using the variable to represent. Since Java is a **strongly-typed language**, we must declare the type of a variable before we are able to assign a value to it. Once a variable has been given a type, you cannot change the type of that variable. In Java there are several primitive types used to represent numbers:

byte (8 bits) < **short** (16 bits) < **int** (32 bits) < **long** (64 bits) < **float** (32 bits) < **double** (64 bits)

In a way, variables are like specially designed habitats at a zoo. You can't house an elephant in a habitat designed for housing monkeys. Similarly, you can't store a double in a variable that has been allocated for an int. So, the size of a primitive type determines how large of a number you can store in a variable of that type. You need not know the actual amount of memory each of these types require, but you should know their relative sizes. It's worth mentioning that even though a float has fewer bits than a long, it is still a larger data type because it must account for decimal values. Furthermore, the only two number types which can store non-integer numbers (such as 3.24 or 5.6766) are float and double.

In addition to these types, there are also primitive types called **char** (designed to represent single characters, such as 'h') and **boolean** (which can be true or false). Note that the boolean type does not have numerical value (i.e. you can't add or subtract two boolean values). You can however assign an integer 0-127 to a char variable and the computer will convert it to the symbol corresponding to that integer if it were an ASCII value.

Casting:

Often times, we want to be able to use operations between two different types of data. In order to support interaction between pieces of data that are not of the same type, we use a process called **casting**. Casting is a way of changing the way that the computer looks at a variable; specifically, it involves telling the computer to treat the data stored in a variable as if it were of a different type. It is important to note that casting does *not* change the actual type of the variable, nor does it change the type of the piece of data that the variable refers to. Casting can be either implicit or explicit:

1. **Implicit Casting:** This is casting that the computer does for us in certain situations where casting can be performed without loss of precision. For example, in the line `double d =1;` the computer will automatically cast (or **promote**) the integer 1 to the double 1.0. Since the computer now views the piece of data as a double, it is able to store an equivalent piece of data in a variable designed for storing doubles. Similar implicit casting can occur in any situation where you are attempting to store something of a smaller primitive type in a variable of a larger variable type. Implicit casting also occurs when using linear operators (+, -, /, *) on two primitives of different types. For example, in the expression `double d = 3 +1.5` the computer will promote the integer 3 to a double 3.0 so that the addition operator can be used without losing precision. In this case, the expression would evaluate as a double 4.5. Implicit casting does not occur

when trying to store a piece of data with a larger data type into a variable meant for a smaller data type. The line `int i = 1.5 * 2;` will not compile because the computer will not implicitly cast 1.5 down to an integer and risk the loss of precision. In order to make something like this work, the computer requires the user to cast 1.5 explicitly.

2. **Explicit Casting:** This is the process of manually telling the computer that we are willing to accept any possible loss of precision that may take place as a result of casting from one data type to another. To explicitly cast a piece of data, we simply write the data type that we want to convert the data to inside a pair of parentheses to the left of the data itself. For example, `(int) 1.0` is an explicit casting of the double 1.0 to the integer 1.

   The "loss of precision" part comes in when we explicitly cast a double or float that is not a whole number to a data type that only supports whole numbers. When this happens, the computer will truncate the decimal value (e.g. if we explicitly cast a double valued at 4.9, this action will yield an integer of value 4). Explicit casting has a relatively high precedence level, which means that when an expression is evaluated, any explicit cast in that expression will occur before any linear (+, -, /, *), relational (<, >, <=, >=), or equality (==, !=) operators. Sometimes, however, we can work around the loss of precision incurred by an explicit cast. Returning to the example from the implicit casting section, just lumping an explicit cast in front of the expression `int i = (int) 1.5 * 2;` would cast 1.5 down to 1, causing the resulting value of the expression to be 2, which is obviously not what we hope for when multiplying 1.5 by 2. Alternatively, we can place parentheses around the entire expression and then explicitly cast the more precise result of the calculation. In this example, `int i = (int) (1.5 * 2);` would allow the multiplication to evaluate first and then cast 3.0 down to 3, which, at least in this case, seems to be a more desirable outcome. Of course this method does not help you avoid loss of precision if the expression in the parentheses evaluates to something that isn't a whole number.