Introduction to Java

**Preface**

That’s right, this is the introduction to the introduction. I have so much to write about concerning Java that, frankly, I’m not even sure where to start. Note that this document will very likely be the first of three on the Java programming language. We will begin with theory and basic syntax, then we will look into data structures and efficiency, and finally, application programming in Java. This is simply the way my college course was structured and is therefor the easiest way for me to organize my thoughts. If you are new to my documents, good! This is exactly the right place to start. I have written many notes on many subjects and programming languages, but I have been avoiding Java simply because of how daunting the undertaking is. My other notes on programming languages gloss over most of the theory because you are expected to learn the basics from this document. Repeating myself over and over would not be helpful, nor practical. If you are completely unfamiliar with programming, or know a minute amount, this guide will attempt to ease you into it, but will gradually get more and more complex, as is only natural with any resource for knowledge. Anyways, I could ramble, but you came here to learn how to program, so let’s begin!

**Intro**

What exactly is Java, you may be wondering (or maybe not, I’m not sure)? Perhaps you have seen it boasted on older software that it has been installed on over 3 billion devices! Although, I’m pretty sure they realised how silly that brag was and stopped advertising it... Java, is a programming language among hundreds, if not thousands, if not tens of thousands of programming languages (if you’re counting all of the oddities). It’s popularity mostly began in the 1990’s and became especially popular with the rise of android and the mobile market. Why do we need so many programming languages? It’s because it is impossible to create an all-encompassing programming language. Programming languages are created to tackle a specific problem, or for a specific purpose. Java happens to be used primarily in the mobile market, and perhaps a bit of web development. Sure, you *could* use Java for writing your own operating system, or for data analysis, but there are already languages that specialize in those areas and that are much more useful. One thing that makes Java a bit unique is that it is platform independent de novo (out of the box). Certain computers are incapable of reading certain languages on their own. For example. I could not run an app designed for the apple iPhone on my Windows PC. This is because the iPhone uses a proprietary language called Swift, that Windows was not designed to understand. Java is able to get around this by running in it’s own enclosed environment. Most computers nowadays come pre-installed with software called the Java Virtual Machine (JVM). This is a virtual computer with limited capabilities that runs within your computer. Like computer inception! All programming languages require more software than just that to run though. They require compilers, interpreters, linkers, and runtime environments. This is where the Java Development Kit (JDK) comes into play. The JDK is a collection of all these programs that work together to make Java work. Java, the JVM, and the JDK are all owned by Oracle. Since the JVM is likely already on your computer, we need only install the latest version of Java, and of the JDK, which you can do from Oracle’s website.

**Terms**

I’m going to define some terms and abbreviations for you (there are so many abbreviations in computer science so don’t worry if you forget one, you can always refer back to here).

SDK: SDK stands for software development kit. This is a loose term that isn’t used too, too often, used to describe a collection of software that co-exist, and co-operate as one functional body. Sort of like how your organs work together to achieve one purpose (keeping you alive)!

JDK: We already looked at the Java Development Kit but I’ll re-iterate. The JDK is a form of SDK. It is a collection of software that co-exist and co-operate to make Java programs run. It contains the Java compiler called Javac, the Java Class Library (JCL), Debuggers, and most notably the Java Runtime Environment. We will look at the JRE in a second, but just remember that the JDK = JRE + Development tools.

JRE: The Java Runtime Environment is sort of a strange but important piece of software used to run Java. If you only want to *run* a Java program, then all you need is the JRE, but you will not be able to *write* any Java code without the JDK. The JRE is sort of the program that prepares the code for the JVM. If you just wrote some Java code and expected the JVM to read it directly, it wouldn’t be able to. Instead, the JRE must take some initial steps before calling the JVM. I know that that’s very vague but I don’t feel the need to confuse you with details at the moment.

JVM: The Java Virtual Machine, as I mentioned earlier, is a piece of software that thinks it itself is an actual computer. This program takes the bytecode created by the JRE and turns it into code that the OS can understand. It uses a special compiler called the Just-in-Time compiler that converts Java source code into machine language (readable by all computers).

JCL: The Java Class Library is a collection/library of pre-written code that we the programmers can use. If you thought that programmers wrote all their code from scratch, you’d be sorely mistaken. Common code used for input and output, math functions, and data structures are all available to use through the JCL. The JCL is what we call a **dynamic** class library, which means that the pre-written code I’m talking about it loaded at runtime, when the JRE is doing it’s thing. I may explain dynamic vs static libraries at another time, although not likely in this document.

JAR: You have likely stumbled across a .jar file once before and wondered what it was. JAR stands for Java archive. You are probably more familiar with a .zip file, and they share some similarities. The purpose of both the .zip and .jar file is to compress the contents of multiple files to save space. The difference between them is that .jar files are executable, meaning a java program can not only store all it’s files into one .jar file, but you can also run the program using that .jar file.

IDE: The final term that I think is necessary to know the IDE, or Integrated Development Environment. An IDE is a concept that is not exclusive to Java. In fact, most languages have multiple IDE’s. An IDE is a third-party application created for developers to code their code in. Usually, an IDE contains a source code editor (like a text editor that we can actually type our code in), automation tools, and a debugger. It can also contain compilers or interpreters, or both. The best Java IDE’s are Jetbrains, Eclipse, and Netbeans. I would start with Eclipse and move to Jetbrains as quickly as possible (and don’t touch Netbeans).

**Getting Started**

Unfortunately, I do not think I am going to explain the extensive process for getting everything setup to run Java code. There are more than one ways to do this, and each method varies depending on the operating system that you’re running. The basic rundown that I can give you is to install the latest version of Java, the latest version of the JDK, and optionally install one of the IDEs I mentioned above. You do not *need* an IDE, and some people don’t even recommend them to beginners, but this is your journey, so read some guides or forums if you want and decide how you want to go about it.

Theory and Syntax

We now begin learning the central concepts of Java, and the syntax of the language. Think of syntax as grammar. There are rules that we must follow and conventions that we must follow when writing our code so that the compiler can understand what we are saying.

**Datatypes**

Programming languages can either be **statically typed**, or **dynamically typed**. Java is a statically typed language, meaning that it contains datatypes. Data types are a way of categorizing data. For example, we refer to text as “strings”, characters as “char”, integers as int, and decimals as float. Some programming languages like Python, PHP, and JavaScript are dynamically typed, meaning that the computer is held responsible to interpret data. In other words, statically typed means the datatype is explicitly described, whereas dynamically typed means that the datatype is implicitly described, and the computer may not always interpret it in the manner that you intended, but when it does, it is more convenient. When a language is statically typed, data can only change types under certain circumstances, whereas dynamically typed languages can change a piece of data’s type at will. I prefer statically typed languages believe it or not, because it leaves less room for error and discrepancies. As a programmer, my person philosophy has been that it should be the programmer’s duty to instruct the computer on what to do and how to do it, not the reverse where the computer is responsible for doing all the dirty work. In other words, things tend to run more efficiently if you stoop down to the computers limited understanding, rather than expecting it to decode abstract code.

**Primitive Datatypes**

All statically typed languages which have data types will at least have the primitive datatypes. There are others which we will go over, but they are based off of the primitive types as you will see.

byte: An 8-bit signed two’s compliment integer. A bit is a singular binary placeholder digit. It is capable of representing 2 states: 1 or 0. A byte is 8 bits, and since each bit can represent 2 states, we get = 256 possible decimal values. Given that one of those 256 numbers is 0, the highest number in binary that we can have is 255. This assumes an unsigned byte, however, a Byte in Java is signed, meaning that it uses the Most Significant Bit (MBR for short or the left-most bit) to represent negative or positive values (1=negative number, 0=positive number). This essentially cuts our possible values in half since we lose 1 bit and every bit multiplies the amount of possible values by 2. The formula is +1 = 128 possible values. Since 0 once again occupies one of those values, and since we have both negative and positive values, we get a range of -128 --> 127. Finally, two’s compliment refers to the format that the computer uses for adding or subtracting negative numbers and representing their sum/difference. Essentially, what it boils down to is that One’s compliment has two representations of 0, whereas two’s compliment only has one.

short: A short (short form for “short int”) is a 16-bit signed two’s compliment integer. It is simply double the length of a byte, meaning 16 bits to represent one number. Since it is signed, the range for positive numbers is halved to add a negative range. The minimum value is -32,768, and the max is 32,767. Shorts are not too common, but they can come in handy every now and again.

int: Int or integer is probably the most common datatype you’ll see aside from maybe string. An int is a 32-bit signed two’s compliment integer. It has a minimum value of and max of -1 which I won’t calculate because it is around 2 billion. This enormous range makes int a good candidate for most things without taking up more memory than your computer can handle.

long: In the case that int was not large enough, the largest datatype for integers is a long (short form for long int). Longs are 64-bit **unsigned** two’s complement integer. I have no definitive answer for why long is unsigned by default, but I can only presume it was done under the assumption that the user would only be using long if they required a very, very large number, so they wanted to maximize the possible values. There are ways to convert a long to be signed, but we wont get into those here.

float: So far we’ve only seen integer types with the exception of byte which is binary. In computer science, a decimal number is called a float. Floats can either be single precision or double precision. The precision refers to the amount of bits used for converting a decimal number from base-10 (our counting system) to binary. Whenever you convert a base-10 decimal number to a float, you will always lose accuracy. The technical name for a float is an IEEE 754 decimal floating-point number. The Institute of Electrical and Electronics Engineers (IEEE) is one of a few large-scale organizations who coin the terminology/vocabulary and naming conventions for concepts or hardware in the field of computer science. They happened to name the arithmetic algorithm for converting from base-10 to binary “754” (yes, a pretty generic name). I don’t think we’ll get into that arithmetic operation at the moment, but perhaps later or in another document.

double: Speaking of single and double precision, we have a data-type specifically for double precision floating-point numbers called double. Typically, it is best to use double in Java. This is because float is actually pretty imprecise and can lead to a lot of errors. While double takes up twice as much memory as floats, we generally have that sort of memory to sacrifice for better accuracy. If you are note performing any arithmetic, and urgently need space, float is probably the way to go, but in most cases double should suit you well.

boolean: A boolean is a single bit used to represent one of two states: 1=TRUE 0=FALSE. There is something in computer science that really confuses noobies, and understandably so, and that is the concept of “literals”. A literal is essentially a value that Java takes to be valid for a given datatype. To illustrate this, in more primitive languages like C programming, boolean is not a primitive type ie. it does not exist as a valid datatype by default. In C, you would define the keyword TRUE to be 1 and false to be 0 because that’s what booleans actually are. Even though booleans are in fact 1 or 0 with TRUE and FALSE being aliases, Java was only created to comprehend TRUE and FALSE, therefor they keywords TRUE and FALSE are boolean **literals**. These are very useful for comparison operations.

char: The final primitive datatype that we come to is char. Char is short for character. Computers only have electric signals that can be sent from the keyboard interface, through the I/O handler, and interpreted by firmware/software. That means that the computer will receive a stream of bits/bytes and have to interpret those bits/bytes as letters. A popular standard for this conversion is ascii. Think of ascii as a lookup table. When we get a binary number, say 65 for example, this is interpreted by the computer to be the letter A. A character is not always alphabetic, it can also be numeric. That being said, Java will always read char to be a literal character ie. you cannot perform standard arithmetic on a char. It should be noted that another popular standard for representing characters aside from ascii is unicode. Unicode was designed with the intention of being a global standard so that anyone could use any character from any language. It is one massive table, and based on the region code you provide, it will map certain keycodes to certain characters. For example, in North America, our standard is UTF-8, which gives us access to the regular charset on a US keyboard + extras.

**Datatypes vs Variables**

One point of confusion as a new programmer might be the difference between a datatype and a variable. Every variable has a datatype. In fact you could view a variable as an alias for some piece of data. Think about algebra for a moment. In algebra, you have variables such as a,b,c,x,y,z,f,g,n,p,q,etc. These always represent a number of some sort, but they are called variables because their value may vary depending on the equation. In essence, variables are just names that we provide some sort of data with, not caring what the data actually is. The datatype is the type of data that the variable must contain (since Java is statically typed). The syntax for defining a variable is: <type> <name> = <data>;. For example: int number = 5; Note that a semi colon terminates the line. Semi colons are common to see in standard programming languages (less so in scripting languages like Bash, CMD, Python, etc.) This helps the compiler parse the code easier. The term for what we’ve just done is **initializing** the variable “number”. When you **declare** a variable, you declare it’s existence to the program, but do not assign it data. When you initialize a variable, you assign it with a value. A variable must be initialized before any operations can be performed on it, but there are instances where we only need to declare a variable.

**Arithmetic Operations** Fix so that equality operators are separate

I would now like to go over the many operations we can use in Java. Operations are actions that we can perform on variables/data. These operations include arithmetic operations (math), comparative operations (comparing equality), and logical operations (comparing multiple statements using logic gates). Obviously, we want to be able to do basic arithmetic on the numeric datatypes such as short, int, long, float, and double (char is technically a numeric type as well but we don’t perform arithmetic on chars without converting them to one of the other numeric types). The basic operations include addition (+), subtraction (-), multiplication (\*), and division (/). In addition to these we have the modulus operator (%) which divides an int or float and returns the remainder of the division. Since short, int, and long are not floating point, when we divide them, they omit the remainder. For example, if I set int integer = 3/2; then integer would carry the value 1 because 3/2 = 1.5 and the remainder (5) is omitted. The appropriate datatype to use in this instance then, would be a double. Similarly, if we set int integer = 3%2; then integer would be set to 5 since the remainder is what is returned by the modulus operation. If we want to set a variable equal to the sum, difference, product, quotient, or remainder of itself and another number, we use the appropriate operator in conjunction with the equal’s symbol. For example, let say I have a variable int five = 5; and I want to add 3 to it. I would then write the line: five += 3; This will take the value contained in the variable five (which is 5), then add 3, and then set the variable five equal to the sum of the addition (8). This applies to all arithmetic operations: +=, -=, \*=, /=, %=. Finally, we can take a look at a special operation called the increment and/or decrement operator. These are functionally the exact same as +=1 or -=1 but they are represented as ++ and -- respectively. So if I wanted to increment the variable int count = 99; then I could say count++; which would increment it to 100. The most difficult part to grasp with the increment and decrement operator is that we can change the order of precedence of this operation depending on the placement of ++ or --. If they are placed before the variable eg. --subOne; then the variable is incremented/decremented before any other operations can affect it ie. the increment or decrement operation takes precedence over all other operations that will act on that variable. If the increment or decrement operator is placed after the variable like in the first example with count, then the value is only incremented or decremented after that line is finished executing.

**Comparative Operations**

Comparative operations, as I mentioned, are operations which compare two or more variables/values and which return true or false. Since the = sign is used to assign values to variables, we use == to imply that we are comparing equality between two things. For example: 9 == 4; returns false. If we wanted, we could assign a boolean with this result. Eg. boolean bool = (9 == 4); bool would now be set to false. If you are familiar with logic gates, you will be familiar with the NOT gate, which inverts the input. This can be represented with != which can be read as “not equal”. For example, ‘A’ != ‘B’ would return true, because it is true that the char A does not equal the char B. The final four comparison operations that we have are greater than (>), less than (<), greater than or equal to (>=), and less than or equal to (<=). These are pretty self-explanatory hopefully. These will return true or false based on whether or not the value on the left meets the criteria of the operator when compared to the value on the right. Note that in ascii, the alphabetic characters A-Z are represented as the numeric values 65-90. This means that boolean isSmaller = (‘A’ < ‘B’); will be true because the character ‘A’ is numerically smaller than the character ‘B’.

**Logical Operations**

Logical operations are the final operation that we will look at, at the moment (bitwise operations also exist but we probably won’t need those for this crash course). Logical operators refer to the three basic logic gates: AND, OR, and NOT. They are used, not to compare two pieces of data, but to compare the results of two or more comparative and/or quality operations. That sounds confusing, but intuitively, it’s not. Essentially, we are just checking if one statement is true/false, AND, OR another statement is true. The NOT gate only works on one statement unlike AND and OR. These are represented as &&, ||, and ! respectively. For example, boolean bothTrue = ((4 != 5) && (‘G’ >= ‘G’)); will be set to true because 4 != 5 is true (four does not equal 5), the letter G is equal to G which is covered in the >= operation, and since statement 1 AND statement 2 are both true, the final result is true. Here’s another example: boolean oneIsTrue = (!(7 < 45) || !(‘B’ == ‘F’)); This will also return true. 7 < 45 will return true, although we invert this result so that it is false. ‘B’ == ‘F’ is false, but we invert it to be true. Since the output will be true if statement 1 is true, OR statement 2 is true, the output is true. OR operator only returns false if all statements are false, otherwise it returns true, and the AND operator only returns true if all statements are true, otherwise it returns false.

**Behaviors (methods) and Attributes (Instance Variables)**

Java is what we called an Object-Oriented Programming (OOP) language, as opposed to a procedural programming language. This implies that the entire language is oriented around the concept of “objects. Many argue that centering programming around objects that mimic reality help maintain clean and efficient code, while making things easier for programmers to comprehend. Every object is treated as a unique **datatype**. We create variables which contain object data. Objects differ in two areas: their attributes, and their behaviors. These are the abstract words we use to describe the object’s **instance variables** and **methods**. Behaviours = methods, and attributes = instance variables. Think of attributes as the data that might belong to a specific object. For instance, a book might have an attribute for number of pages which would be represented by a variable called pageNum. Because this variable applies to the book object, we call it an instance variable, because we **instantiate** every book object with it’s own pageNum variable. Other attributes for a book might be a boolean called hardCover to specify whether or not the book is a hard cover book, or maybe a String called author (we haven’t looked into String yet, but it’s basically just a word or sentence). These are all things that describe the book object ie. attributes. Objects can also have behaviors (although I won’t use the book example again because they don’t do much). If the object needs to interact with other objects in some way or performs actions, we create methods to describe an objects behavior. The term “method” is exclusive to OOP languages. It is interchangeable with the term “function”, as “function” is the procedural equivalent to “method”. A method is a block of code which, in general, uses the objects attribute data to do something.