**Java Compilation Process**

The compilation of java programs involves a two-step execution process, the first step is through an OS independent compiler, and the second step through a virtual machine, or JVM.

First the compilation, the source “.java” file is first passed on to a compiler, which will encode the source code into Bytecode, which is a machine independent encoding. The classes that are inside of the source file are stored in a separate “.class” file.

The compilation process follows some steps. First is **Parse**, this reads a set of source code from the “.java” file and maps it to an Abstract Syntax Tree – Node. Then there is **Enter** which inputs symbols into the symbol table for definitions. **Process annotation** is for processing annotations that are found. **Attribute** is to attribute the Syntax trees and steps include: Name resolution, type checking and constant folding. Next, **Flow** is where the attributed Syntax trees’ dataflow are analyzed and steps include, checks for assignments and reachability. **Desugar** rewrites the Syntax tree and translates some syntactic sugar. Finally, we have **Generate** which generates the “.class” files.

After the generation of the class files, it is ready to be run in any machine or OS, as the class files generated do not depend on the machine or the OS. The main file is then passed to the JVM and goes through 3 stages before it will be done executing.

The first stage is called **Class Loader**. While the main class is passed through a JVM, all the other classes are loaded through the class loader, which is an object that creates a flat name space of class bodies that are referenced by a string name. There are two types of class loaders, primordial and non-primordial. A primordial class loader is the default class loader for all JVM, while a non-primordial class loader is a user defined, or customized, class loader. A non-primordial class loader is coded to handle class-loading process in a different way. If a non-primordial class loader is available, it will be preferred over the default primordial class loader.

In the second stage, we have **Bytecode Verifier**. This step is where the bytecode is inspected after being loaded by the class loader in the previous stage. This stage ensures that there are no damaging actions in the code. The checks that are followed in these stage are, check if variables are initialized before they are used, check if method calls are matched with the types of object references, check if rules of accessing private data and methods are not violated, check if local variable accesses fall within the runtime stack and finally, check if the runtime stack does not overflow. If any of those checks failed, the class is going to be stopped from loading by the verifier.

Finally, the final step, **Just-In-Time Compiler**. The last step ensures that the bytecode is converted into machine code. We don’t need to use a JVM to interpret the same sequence of code repeatedly because a JIT compiler can execute the native code, leading to faster execution speed, unless some methods are loaded less frequently.