# SoftEng 701 Assignment 02

Michael Lo
mlo450
5530588

From hardware to punchcards, to Assembly and C, to modern high-level languages, the history of programming has always shown a strong trend towards simplicity, readability, and ease of use over low-level control and micromanagement. Hardware performance has accelerated to the point where developers no longer need concern themselves with squeezing every last millisecond out of their processor, and the human's time is more valuable than the machine's. Additionally, today's compilers are very intelligent and capable of optimising even the most complex code. With this in mind, the over-engineered and verbose standard I/O libraries are more of a hindrance than a help. I propose an alternative, in the one-line style of languages like Python, for opening data input streams and files for both reading and writing.

My additional feature simplifies Java's overly complex I/O systems into a much easier to use set of reserved keywords. Java was originally designed back when the common languages were much more low-level, like C, and shares many design aspects and concepts with them **[1]**. However, modernised languagues like Python and GoLang are specifically designed to be much more user-friendly, while still offering the same flexibility and power **[2]**. I believe that today's programmers should work at a high level, and not need to concern themselves with many underlying workings of the language in which they choose to work. As Java's I/O libraries are almost exclusively used in the same way (BufferedReader/Writer wrapping some form of Input or OutputStream), I see no reason for developers to have to manually write ~10 additional lines of code just to open a file. This includes variable declarations, try/catch blocks, and exception handling – all of which is boilerplate code that can be easily added automatically. Many programmers, myself included, have expressed frustration at the difficulty of handling simple I/O procedures in Java, especially when compared to the single-line syntax found in languages like Python **[1][3]**.

My additions to the .jj file consist of the tokens "open", "reader", and "writer". The syntax is open [reader, writer] [path to file (as string literal), stream (as name of existing variable)] { block of code }. A BufferedReader or Writer is provided (the variable b_reader or b_writer, respectively), which exists within the scope of the BlockStatement (the curly braces). Use of the reader or writer is contained within a try/catch block with the relevant exceptions (FileNotFound and IO) printing their stack traces when caught. My additional AST node is similar to a BlockStatement, but also contains boolean isFile and isReader values, as well as the stream variable name/path to file.

My visitors currently handle a small subset of Java functionality, but one which covers a large percentage of actual use cases – particularly those which are likely to be encountered in a project consisting of only a single file, as is the scope of this project. My visitors, in order:

1) Populate the symbol table with the scopes present in the file.
2) Populate the symbol table with the types (classes) defined in the file.
3) Populate each scope with the variables (and methods) present in them.
4) Check that those variables are used correctly. This includes ensuring that variables are not used out of scope, that they are only assigned values of their static typing, and that there are no duplicate definitions of a variable name within any one scope.
5) Ensure that the additional feature has been used correctly – throw an A2SemanticsException if b_reader/b_writer is never used within that block, as well as if a duplicate variable of that name is declared within a corresponding open {} block. After the final use of the b_reader or b_writer variable, it is closed to flush the buffer and free up memory.
6) Print the verified source code, as well as the extra boilerplate I/O code, to a .java file.

My symbol table is an extension of the one described in class, and used in the

CitySemantics example. As well as GlobalScope and LocalScope, I have added class, interface, enum and method scopes. GlobalScope's constructor sets up definitions for the built-in types, as well as things like my b_reader and b_writer variables and their corresponding BufferedReader and BufferedWriter types. In addition to the resolve() method, my Scopes also implement a resolveType() method which performs a check to ensure that the symbol matching that name is a Type. This makes variable declaration and use checking much simpler, and avoids duplicate code (no more type coercion outside the Scope).

My alterations to the Java grammar free programmers from the tedium of writing large quantities of boilerplate code just to perform a simple task. The syntax is simple and easy to remember, but still offers a lot of flexibility. Those who require more fine control over their code, or who need slightly different I/O functionality, are still free to write regular code manually. However, I believe that for the majority of cases this new method should suffice. It vastly decreases the scope for human error – the single most problematic issue when writing code.

References:


**[1]**
https://pythonconquerstheuniverse.wordpress.com/2009/10/03/python-java-a-side-by-side-comparison/

**[2]**
http://www.infoworld.com/article/2625481/development-tools/google-executive-frustrated-by-java--c---complexity.html

**[3]**
http://www.javalobby.org/java/forums/t17056.html

**[4]**
https://www.python.org/doc/essays/comparisons/

**[5]**
http://www.quora.com/Should-I-learn-Python-or-Java

**[6]**
http://en.wikipedia.org/wiki/Python_%28programming_language%29#Features_and_philosophy

**[7]**
http://en.wikipedia.org/wiki/Java_(programming_language)#Principles

**[8]**
*Java in a Nutshell*, David Flanagan, 2005 (Pages 252-276)

**[9]**
https://golang.org/doc/faq#What_is_the_purpose_of_the_project

**[10]**
http://python-history.blogspot.co.nz/2009/01/pythons-design-philosophy.html