The Java Programming Structure
IFT 194: Lab 1

Brandon Doyle
bdoyle5@asu.edu
1215232174

Dr. Usha Jagannathan
Usha.Jagannathan@asu.edu

July 3, 2018

Part A

In this activity I didn't learn many new things, but this is primarily because I've already taken an introductory course in Java. I was not aware, however, of the history of Java's versioning (numbering) system, so I appreciated that background. The objective of this activity is to get students up and running with an environment tailored to writing Java. The activity also walks us through the installation process of the Java Developement Kit (JDK) and Eclipse, an Integrated Development Environment (IDE) for Java.

I've installed the JDK on my laptop, which is running Ubuntu 16.04 LTS. The process is quite simple – all we need to do is download the appropriate JDK file and add the included `bin/` subdirectory (wherever it may be) to our path. The `bin/` subdirectory contains all the executables for running our code. It's actually quite convenient, because a lot of languages (like Python) require compilation of some sort. Also, decompressing compressed `tar` archives can be even simpler for Linux distributions than remembering all of the appropriate flags with `dtrx`, short for "do the right extraction." The package is written in Python and you can view it on GitHub.

From here, I can verify the installation as follows in a shell.

```
brandon@ideapad:~/Desktop/IFT_194/labs$ java --version
java 10.0.1 2018-04-17
Java(TM) SE Runtime Environment 18.3 (build 10.0.1+10)
Java HotSpot(TM) 64-Bit Server VM 18.3 (build 10.0.1+10, mixed mode)
```

According to Oracle's downloads page, this is the latest version (as of July 2, 2018). Moreover, looking over my `~/.bashrc`, it appears at some point in the past I added the package to my path.

```
export PATH=/home/brandon/bin:/home/brandon/.local/bin:/home/brandon/.cabal/bin:/home/
brandon/anaconda3/bin:/home/brandon/bin:/home/brandon/.local/bin:/usr/local/sbin:/usr/
local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/usr/lib/
jvm/java-10-oracle/bin:/usr/lib/jvm/java-10-oracle/db/bin:/usr/local/go/bin:/home/
brandon/go/bin/gdrive
```

I've also re-created `Welcome.java` (cf. Figure 3) in my eclipse-workspace, and I'm able to run it as follows from my terminal.

```
brandon@ideapad:~/eclipse-workspace/ift_194_labs/src/lab_1$ javac Welcome.java
brandon@ideapad:~/eclipse-workspace/ift_194_labs/src/lab_1$ java Welcome
Welcome to the Introduction to Java Technologies
================================================
```

Installing Eclipse is almost as easy in a Debian-based Linux OS, but I had to add an appropriate `eclipse.desktop` file under `/usr/share/applications/` to lock the program's icon to my Launcher. I've done this many times in the past for packages that do not install a `*.desktop` file or aren't handled by the default Debian package manager `dpkg`. See Figure 1 for an image of my IDE.

I was also not aware that Java SE 10 had been released. Features of modern Java I hope to learn more about include its functional capabilities, such as those introduced in Java SE8. I'm also aware of various projects, such as functional java library, which extend these capabilities. I'm also a fan of local-variable type inference, which I think increase code readability (e.g. see Figure 6).

You may view the source of this document on GitHub.

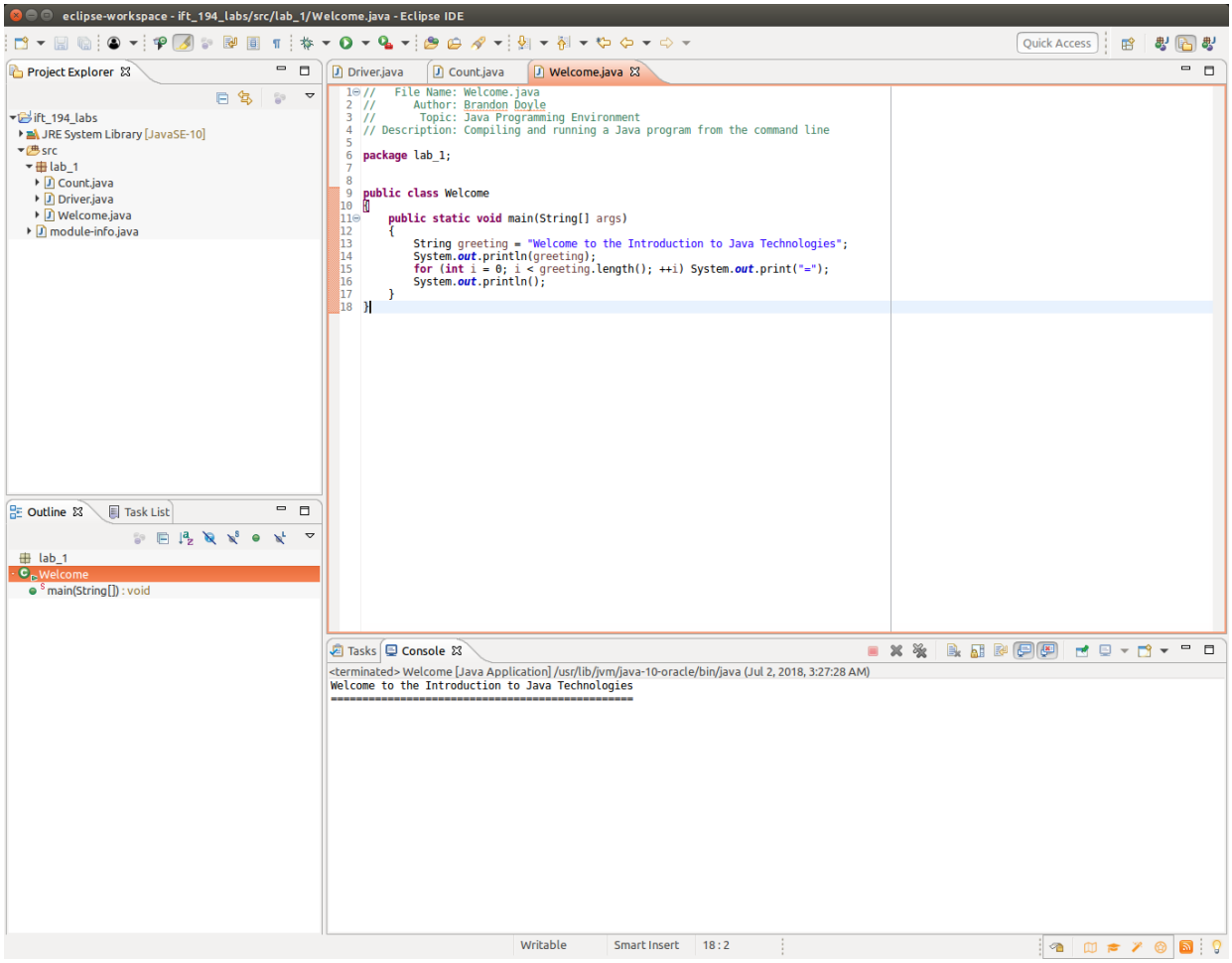

Figure 1: Eclipse Photon.

Part B

1 Poem

In this section we are tasked with writing a simple program in Java that prints the following lines of text.

```
Roses are red
Violets are blue
Sugar is sweet
And so are you!
```

See Figure 4 for my short code example. It's also possible to accomplish this task with a single string and method invocation using the newline character '`\n`'.

2 Comments

1) In this activity we are tasked with writing comments in our programs, which either start with '`//`' for single line comments, or enclose text in '`/*...*/`' for multiline comments. Multiline comments aren't, unfortunately, a feature of every language (for instance, Python, which isn't well-known), so they are a convenience for designating blocks of text the lexer should ignore completely. See Figure 5 for an example of using single-line comments in code.

2) Spaces between lines of code can sometimes increase readability. If it's one thing that's been discussed among developers since the dawn of programming languages, it's likely code readability. After all, code isn't for computers – it's for humans, like those who wrote it. In this case, I think the spaces increase readability, however redundant they may be.

3) Removing one of the comment's backslashes introduces a syntax error into our program, which is probably because Java's lexer (see e.g. `javac`'s compilation overview) cannot tokenize the string for the parser to create an AST. In other words, it doesn't know what to do with something like '`/ some text`' in the source file because the operation isn't defined in the language's grammar.

4) Nesting single line comments is not an issue in Java. However, Java does *not* support nested block comments. This behavior is similar to that of pure C.

5) In such a basic program, I do not believe it's necessary to explain what the program is supposed to do. For instance, writing comments like "print 1 to 5" are practically useless to anyone who is familiar with Java (or, for that matter, *any* language). Moreover, there is a lot someone can do to reduce the amount of comments they have to make; for instance, reducing the length of functions (better compartmentalization, break the problem into more steps).

Another thing that is missing is method documentation. I think it's a good idea to stick to formatting used by Javadoc.

3 Program Names

We are tasked with indicating which of the following would be valid class or variable identifiers in Java. See also comments in Figure 6.

1. `simple` – This is a valid identifier. However, for class names, it's good practice to use uppercase letters so it's easy to recognize a type (class) from a local variable or field.
2. `SimpleProgram` – Again, this is a valid identifier that uses CamelCase.
3. `1 Simple` – This is an invalid identifier because it contains a space *and* it starts with a number.
4. `_Simple_` – This is again a valid identifier. I typically use a leading underscore in classes to indicate that an identifier represents a private field variable.
5. `*Simple*` – This is invalid because it contains special characters.
6. `$123_45` – This is a valid identifier, but using the '`$`' character is generally discouraged.
7. `Simple!` – This is again invalid because it contains the special character '`!`'.

4 Recognizing Syntax Errors

1) In this section we introduce typical errors into a program `Hello.java` (cf. Figure 7). Running this program of course creates a '`Hello, World!`' message in the terminal.

2) If we remove a single '`l`' from the class name, I receive the following error.

```
Error: Could not find or load main class lab_1.Hello in module ift_194_labs
```

3) Removing any character from a string does not produce an error. Java does not, to the extent of my knowledge, work on the contents of `String` objects unless they contain an escape sequence. However, we may run into font errors while attempting to print to the terminal, or in certain situations involving '`char`' arrays and supplementary characters. Unicode support is a tricky subject for a lot of programming languages. Because most of the data I've worked with has involved numerical data, it hasn't been necessary for me to look into it further; however, I hope that changes throughout the duration of this course.

4) When one removes the ending quotation of a string literal in Java, compilation produces the following error.

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
    String literal is not properly closed by a double-quote
    at ift_194_labs/lab_1.Hello.main(Hello.java:13)
```

In words, Java is telling us we didn't close the string literal.

5) Removing the starting quote around our string produces the following error.

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
    Hello cannot be resolved to a variable
    Syntax error on token ",", delete this token
    Syntax error, insert ")" to complete MethodInvocation
    Syntax error, insert ";" to complete BlockStatements

    at ift_194_labs/lab_1.Hello.main(Hello.java:13)
```

In this case, the tokenizer first tells us we've made a syntax error as early as the comma, which again is not a valid character in an identifier. Because we're missing an opening quote, the first quote the lexer sees is prior to '`);`', which are valid string characters. However, because the program is now missing a closing '`)`' character on the method call '`System.out.println(`', the tokenizer is unable to complete the invocation. Lastly, as part of Java's C-like syntax, we're missing a semicolon at the end of the statement (since the lexer now considers it part of a string literal). Therefore, Java's parser cannot proceed to build an AST.

5 Correcting Syntax Errors

In this section we're tasked with correcting errors in the program listed in Figure 8. I've reproduced in Figure 2 below my solution, albeit with a different file (and hence class) name.

```java
//********************************************
// Problems.java
//
// Provide lots of syntax errors for the user to correct.
//
//********************************************

package lab_1;  // unnecessary for a standalone file

public class Solution   // modified class name to avoid overwriting figure 8
{
    public static void main(String[] args)
    {
        System.out.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
        System.out.println("This program used to have lots of problems,");
        System.out.println("but if it prints this, you fixed them all.");
        System.out.println(" *** Hurray! ***");
        System.out.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
    }
}
```

Figure 2: Solution.java. This program is my solution to fixing the code presented in Figure 8. Although it is not necessary, I've also removed the whitespace between the opening parenthesis on each method invocation.

Conclusion

I spent approximately 8-9 hours completing this lab. The quickest portion was setting up my environment as I already had the JDK installed on my Linux machine and Eclipse. I of course also found several topics that I wasn't extremely familiar with, such as modifying my `PATH` variable. Although I've done this before, I wanted to understand exactly what commands like `export` do.

Challenges I faced in writing this lab report were primarily around formatting. Because I've chosen LaTeXto present my code and findings, I had a number of issues to figure out with this language.

```java
package lab_1;

public class Welcome
{
    public static void main(String[] args)
    {
        var greeting = "Welcome to the Introduction to Java Technologies";
        System.out.println(greeting);
        for (var i = 0; i < greeting.length(); ++i) System.out.print("=");
        System.out.println();
    }
}
```

Figure 3: Welcome.java

```java
//   File Name: Poem.java
//      Author: Brandon Doyle
//       Topic: Java Programming Environment
// Description: Print a message to the console.

package lab_1;

public class Poem
{
    public static void main(String[] args)
    {
        System.out.println("Roses are red");
        System.out.println("Violets are blue");
        System.out.println("Sugar is sweet");
        System.out.println("And so are you!");
    }
}
```

Figure 4: Poem.java

```java
//*****************************************************************************
//   File Name: Count.java
//      Author: Brandon Doyle
//       Topic: Java Programming Environment
// Description: Print to the console and try single line comments in Java.
//*****************************************************************************

package lab_1;

public class Count
{
    public static void main(String[] args)
    {
        // English
        System.out.println("one two three four five");

        // French
        System.out.println("un deux trois quatre cinq");

        // Spanish
        System.out.println("uno dos tres cuatro cinco");
    }
}
```

Figure 5: Count.java

```java
//   File Name: Simple.java
//      Author: Brandon Doyle
//       Topic: Java Programming Environment
// Description: Using Eclipse, determine which identifiers are valid.

package lab_1;

public class Simple
{
    public static void main(String[] args)
    {
        var simple = 1;         // valid
        var SimpleProgram = 1;  // valid
        var 1 Simple = 1;       // invalid
        var _Simple_ = 1;       // valid
        var *Simple* = 1;       // invalid
        var $123_45 = 1;        // valid
        var Simple! = 1;        // invalid
    }
}
```

Figure 6: Simple.java. Here I've also used local-variable type inference through the `var` keyword.

```java
//   File Name: Hello.java
//      Author: Brandon Doyle
//       Topic: Java Programming Environment
// Description: Used for understanding common syntax errors we may introduce into
//              our programs.

package lab_1;

public class Hello
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

Figure 7: Hello.java

```java
// ****************************************
// Problems.java
//
// Provide lots of syntax errors for the user to correct.
//
****************************************
public class problems
{
    public Static main (string[] args)
    {
        System.out.println ("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
        System.out.println ("This program used to have lots of problems,");
        System.out.println ("but if it prints this, you fixed them all.")
        System.out.println (" *** Hurray! ***");
        System.out.println ("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
    }
}
```

Figure 8: Problems.java. This program has many errors.