The Java Programming Structure
IFT 194: HW 1

Brandon Doyle
bdoyle5@asu.edu
1215232174

Dr. Usha Jagannathan
Usha.Jagannathan@asu.edu

July 5, 2018

Reading Questions

1. According to the reading, the original name of the Java programming language was "Oak."
2. The author of the reading, Michael O'Connel, made the claim that "... the solution preceeded the problem" regarding the creation of Java. In this statement, the "problem" arose in mid-1994 when the Internet's popularity began growing. As Gosling says, there was a need for a "really cool browser," which is an application that should be architecture-neutral, real-time, and secure. These are all qualities that had been built into Java since 1991.
3. The creator of Java is James Gosling. The author also mentions Patrick Naughton, who was the "project lead on Sun's OpenWindows user environment before joining the secret Green team," which was the team name for a consumer electronics effort at Sun Microsystems in the 1990s. Gosling's contributions to the team were usually to solve "tooling" problems that required extensive programming. Gosling had already, by that time, produced a number if interesting projects, such as the first implementation of the Emacs text editor in pure C (I actually have GNU's Emacs installed on my system now).
4. The creators' motivations in writing Java were to make the language platform-neutral for consumers. Today there are a large number of different processors and architectures, even more-so than there were in the '90s. For example, the text also states that the goal was to "build a system that would let us do a large, distributed, heterogeneous network of consumer electronic devices all talking to each other."
5. Java was spun into the web when Jonathan Payne wrote a web browser called WebRunner in Java. This allowed developers to write applications in the browser.
6. The original target market for Java was consumer electronics. Java was created to solve the development issues programmers were having with creating systems running on different platforms (operating systems and architectures) that could communicate with each other. Languages (like Scala) that run on the JVM are platform independent.

Textbook Exercises

1.14

According to the text and the Java SE10 language specification, there are two types of comments, including

1. end-of-line comments, denoted by '`//`', and
2. traditional comments, denoted by '`/*...*/`'.

End-of-line comments delimit text on a single line that is to be ignored by the compiler, and traditional comments, or multiline comments, can mark several lines of text that are to be ignored.

View the source of this document on GitHub.

1.15

We are tasked with determining which of the following identifiers are valid and invalid.

1. `Factorial` – This is a valid identifier. However, it should be used as an identifier for classes or types, since it begins with a capital letter.
2. `anExtremelyLongIdentifierIfYouAskMe` – This is a valid identifier. Again, referencing the Java SE10 specification, identifiers have an unlimited length limit. This has been a feature since at least Java 6.
3. `2ndLevel` – This is an invalid identifier because it starts with a number.
4. `level2` – This is a valid identifier.
5. `MAX_SIZE` – This is also a valid identifier.
6. `highest$` – This is again a valid identifier, because '`$`' is a valid special character in identifiers.
7. `hook&ladder` – This is invalid, since '`$`' is not a valid character in an identifier, according to the language specification.

Actually, interestingly enough, the specification states the following, which I thought was pretty interesting.

> The "Java letters" include uppercase and lowercase ASCII Latin letters A-Z (\u0041-\u005a), and a-z (\u0061-\u007a), and, for historical reasons, the ASCII dollar sign ($, or \u0024) and underscore (_, or \u005f). The dollar sign should be used only in mechanically generated source code or, rarely, to access pre-existing names on legacy systems. The underscore may be used in identifiers formed of two or more characters, but it cannot be used as a one-character identifier due to being a keyword.

1.16

We are tasked with determining, based on the text, which of the following are good identifiers.

1. `q` – This is a poor identifier unless the letter '`q`' may have some significance in the context (e.g. how `x` and `y` may be used to denote coordinates in an equation).
2. `totVal` – This is, in my humble opinion, a mediocre identifier; for readability's sake, you could probably just write the words out as `totalValue`.
3. `theNextValueInTheList` – This is an unnecessarily long identifier. I think a better identifier would be something like "`nextValue`."

1.17

If a language is case-sensitive it means that identifiers with capital letters will be treated as distinct from those that may have the same letters, but lower-case. For example, `nextValue` and `nextvalue` are distinct identifiers in Java.

There are languages, like SQL, that are *not* case-sensitive, so statements like

```
mysql root@(none):(none)> show databases
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
4 rows in set
Time: 0.016s
```

are the same as

```
mysql root@(none):(none)> SHOW DATABASES
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
4 rows in set
Time: 0.017s
```

1.18

The Java Virtual Machine (JVM) is an abstract computing machine. The JVM is, according to the text, an interpreter for Java bytecode. Many people like to divide languages into those that are interpreted and compiled. As it turns out, Java is a hybrid of both, being initially compiled to bytecode, which is then interpreted by the JVM.

Programs in Java reside in `class` files, which are created when `javac` compiles code files. There are also a lot of interesting properties of these bytecode files; for instance, just as image formats typically start with a file signature, or a sequence of signature bytes (e.g. see Figure 1), `class` files always start with the sequence of bytes `CA FE BA BE`. We can confirm this by opening one of our files in `~/eclipse-workspace/ift_194_labs/bin/lab_1` (cf. Figure 2).

1.20

We are tasked with labeling each of the following situations as a compile-time error, run-time error, or logical error.

1. *Multiplying two numbers, when we meant to add them* – This is a logical error, because our program will likely compile correctly, but it will produce an inaccurate result.

2. *Dividing by zero* – This is a run-time error, because the program will again compile correctly, but there's no way for the compiler to know that we will be dividing by zero; hence, the program will terminate abnormally.

3. *Forgetting a semicolon at the end of a programming statement* – This is a compile-time error, because forgetting a semicolon is at odds with the language's grammar.

4. *Spelling a word incorrectly in the output* – If I'm not mistaken, this isn't a Java-related error at all, because we may put many different characters in `String` objects (that is to say, Java doesn't *care* if we spell a word wrong, as long as it's not an identifier). I believe the closest applicable category, then, is a logical-error.

5. *Producing inaccurate results* – likely a logical error.

6. *Typing a { when you should have typed a (* – This is likely a compile-time error, depending on where the character was mistyped. For example, if the character is residing within a `String`, then it won't produce an error.

```
/* Checks whether the supplied bytes match the PNG signature.  We allow
 * checking less than the full 8-byte signature so that those apps that
 * already read the first few bytes of a file to determine the file type
 * can simply check the remaining bytes for extra assurance.  Returns
 * an integer less than, equal to, or greater than zero if sig is found,
 * respectively, to be less than, to match, or be greater than the correct
 * PNG signature (this is the same behavior as strcmp, memcmp, etc).
 */
int PNGAPI
png_sig_cmp(png_const_bytep sig, png_size_t start, png_size_t num_to_check)
{
   png_byte png_signature[8] = {137, 80, 78, 71, 13, 10, 26, 10};

   if (num_to_check > 8)
     num_to_check = 8;

   else if (num_to_check < 1)
     return (-1);

   if (start > 7)
     return (-1);

   if (start + num_to_check > 8)
     num_to_check = 8 - start;

   return ((int)(memcmp(&sig[start], &png_signature[start], num_to_check)));
}
```

Figure 1: Lines in `png.c` that define the signature bytes of the PNG image specification. In hexadecimal format, the sequence is `89 50 4E 47 0D 0A 1A 0A`.

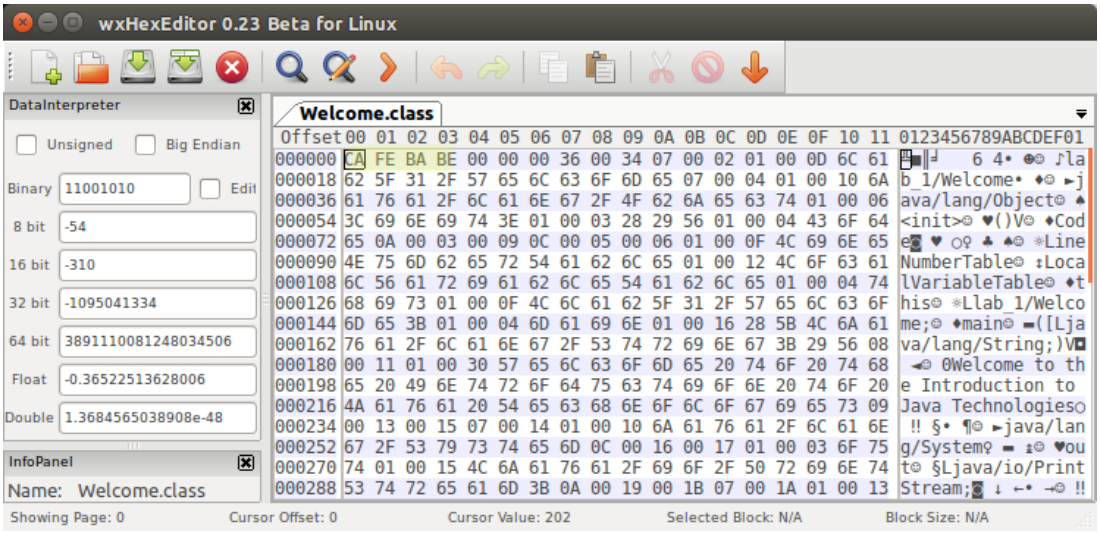

Figure 2: First four signature bytes of my `Welcome.class` file, generated for Lab 1.