Portfolio Project

Christopher Rodriguez

Colorado State University Global

WC21-CSC200-2: Computer Science Fundamentals

Prof. Chintan Thakkar

2022-03-13

Portfolio Project

In software development, algorithm design is paramount to solving even the simplest of problems. Indeed, algorithms are the verbs of computer science, and a well-designed algorithm is a crucial prerequisite to truly repeatable computation (**Ewing2017**). It is not an over-exaggeration to say that problem solving begins with algorithm design. Algorithmic problem solving is the cornerstone of computational thinking (Levitin, 2018, p. 5), upon which all other facets thereof are built.

It is said that great scientists always have a feeling about the direction of their research, whereas merely good scientists may charge forward blindly (Lai, 2017, p. 2). It is therefore important, when beginning a new software project, to have a clear and well-defined idea of what, exactly, the project will be. What problem does this project solve, and how does it solve that problem? This paper walks through a common four-part process to develop a simple algorithm, and examines the results.

Problem Examination

The problem addressed by this project is a common issue: parsing and ``sanitizing" a string for use in a file name.

There are a lot of characters one might use in a filename that could cause issues on different systems: from common characters like spaces, colons, ampersands to more obscure and unprintable Unicode characters.

This problem can be broken into a few smaller pieces: accepting user input, substituting disallowed characters within the string, appending a file-type suffix, and returning the filename for general use (possibly to be passed to another function for output collection of some sort.

In the end, the program will read a string from standard input, and reply with a string to standard output. This choice was made as it is easily redirectable and implementable as a minimal working example.

A couple of constraints: For brevity, this implementation will only concern itself with the Latin alphabet symbols, and produce a final string in ASCII. For simplicity, immutable data structures will be used whenever possible. And the entire project will be packaged as a `jar` file and usable from the shell.

Task Layout

The tasks, in order, are fairly self-evident.

First, receive a string from the user. This can be further broken down into prompting the user for input, and then in storing that input in a variable.

Second, feed that original string through a chain of sanitization functions, each meant to deal with one of the following problems: whitespace removal, symbol substitution, and conversion to ASCII text. This new value will need to be stored in a new variable.

Third, append an appropriate file extension to this file name. This is another section where input will be needed from the user, and the result stored in a new variable. Note that there should be a default (no extension) and that the actual input from the user will not necessarily be appended to the filename: The input should be used to append a known file type from a list. Another function needs to execute that lookup, which will be stored in a new variable.

Fourth, the final filename should be concatenated and then returned to the user. This needn't be stored, as the program is over at the point of return, but the program will do so anyway, in a final variable.

Module Development

Module Development took a top-down approach: Starting with the `main` method, a scanner was declared and two functions were called: `getUserInput` and `filenameOutput`. Each of these functions then needed to be defined; `getUserInput` called functions from the scanner passed to it and conversed with the user, and `filenameOutput` branched out into `sanitizeString` and `fileExtension`. While `fileExtension` passes through a chain of `if-else` statements to de-

termine an appropriate suffix, `sanitizeString` calls a series of *sanitize-* functions that clean the string up according to the rules outlined above.

At this point, a few updates were made to the specification: Latin-1 (ISO-8859-1) is just as easy to use in Java as ASCII, and provides more useful characters, so the project switched to it. A few of the intermediary variables were removed; The only variables living through the entirety of `main` are the user input, and the rest are never stored anywhere. And the `PortfolioProject` class was put under the `csc200` package.

At this time, comments would normally be inserted into the code in order to document it for developers as well as actual documentation for users of the library would be composed. However, as this is a school project and not an actual software project, this step was omitted in order to keep the examples as minimal as possible.

Update

As this is the only time this project is likely to receive an update, the code was examined immediately.

The algorithm for `sanitizeLatin1` was needlessly creating a secondary string as a middle-man; This was removed to optimize performance. A bug in the logic for `getUserInput` prevented the exception handling from ever being triggered (calling wrong exception name). This was fixed. And the conversion to and from Latin1 was (silently in `bash`, but visible in `eshell`) appending a C-style NULL character `\0` to the end of the filename string; This was trimmed away as it could cause issues down the road.

At this point, unit and regression testing would begin on the algorithm, and it would be integrated into a wider ecosystem of libraries, likely revealing other improvements that should be prioritized. However, as this is an exercise for a school assignment, it will likely end here.

End Result

At the end, we have an executable jar file compiled with the following function:

```
jar -cvfe 2022-03-13.Portfolio-Project.jar csc200.PortfolioProject \
csc200/PortfolioProject.class
```

Upon run, it will ask the user for a filename to process as well as what type of file it is. The program currently supports Text files and Java files. Everything else does not get an extension. And it prints the complete and sanitized filename to the console before exiting.

Could this code be improved? Definitely. All of the code is public in the class, it relies on hard-coded values to determine file extension, and it is interactive only: UNIX pipes will not work with it as is. However, that would be the *next* cycle of development for this project. As a first iteration, this project is complete.

Conclusion

There are myriad methodologies that might be employed to better design algorithms, and all will likely do things differently than presented here (Levitin, 2018, p. 5). However, the important point of this walk-through was showing the basic process: Take a Problem, break it down into small pieces, implement each piece and ensure they work together as expected. This is the crucial and fundamental goal of algorithm design: solve a problem in the most efficient way possible, by examining the parts of that problem and improving *them* where possible.

Creativity is just as important as reason when it comes to problem solving (Lai, 2017, p. 2). It is important to experiment and try new ideas, see what might be improved. Having a solid framework to begin with, then, allows for greater experimentation and an more cohesive creative process than sitting down at a blinking cursor having never considered the problem before. Hickey (2010) famously said to step away from the computer and think about something for at least an hour before beginning to write code. This methodology helps to enforce that kind of approach.

Java Implementation

```java
package csc200;                          // Declare our section as the package.
import java.util.Scanner;                // For parsing user input.
```

```java
import java.nio.charset.StandardCharsets; // For handling UTF and LATIN-1


public class PortfolioProject {
    public static void main(String[] args) {
        // Handle I/O and passing down to other functions.
        Scanner input = new Scanner(System.in);
        String userInputString =
            getUserInput(input, "Please Input a Filename: ");
        String userInputType =
            getUserInput(input, "What type of file is this? ");
        System.out.println("\n" +
                           filenameOutput(userInputString,
                                          userInputType));
    }
    public static String getUserInput(Scanner scanner, String prompt) {
        // Handle user input in a standard way, checking for no input.
        String result = "";
        Boolean success=false;
        String noInputMessage= "No input detected.\n";
        while(!success) {
            System.out.println(prompt);
            try {
                success=true;
                result = new String(scanner.nextLine());
                if(result.equals("")) {
                    throw new IllegalArgumentException(noInputMessage);
                }
```

```
        }
        catch(IllegalArgumentException e) {
            success=false;
            System.out.println(noInputMessage);
        }
    }
    return result;
}
public static String filenameOutput(String inputName, String inputType) {
    // Sanitize input, append correct file extension.
    return (new StringBuilder())
        .append(sanitizeString(inputName))
        .append(fileExtension(inputType))
        .toString();
}
public static String sanitizeString(String x) {
    // Delegate to dedicated sanitize- functions.
    return (new StringBuilder())
        .append(sanitizeLatin1(sanitizeSymbols(sanitizeWhitespace(x))))
        .toString()
        .toLowerCase();
}
public static String sanitizeWsSymbols(String string) {
    // Handle case where symbol is followed by space, to avoid '_-'.
    return string
        .replace(": ", ":")
        .replace("? ", "?")
```

```
        .replace("& ", "&");

}
public static String sanitizeWhitespace(String string) {
    // Remove extra whitespace at start/end, and turn spaces into dashes.
    return string
        .trim()
        .replace(' ', '-');
}

public static String sanitizeSymbols(String string) {
    // Replace inappropriate characters with an underscore.
    return string
        .replace(':', '_')
        .replace('?', '_')
        .replace('&', '_')
        .replace('$', '_');
}
public static String sanitizeLatin1(String string) {
    // Convert the string to Latin1 and back to remove UTF-8 chars.
    return (new String(StandardCharsets.ISO_8859_1
                       .encode(string)
                       .array(),
                       StandardCharsets.UTF_8)
            .trim()); // Needed to remove trailing \0.
    }
public static String fileExtension(String x) {
```

```
        // Hardcoded guesses at user input to look up file ext. Default to none.
        if (x.equals("txt") ||
            x.equals("TXT") ||
            x.equals("text") ||
            x.equals("TEXT") ||
            x.equals("t"))
            return ".txt";
        else if (x.equals("java") ||
                 x.equals("JAVA") ||
                 x.equals("j"))
            return ".java";
        else
            return "";
    }
}
```

References

Hickey, R. (2010, October 23). *Step away from the computer: Hammock driven development* (M. Courtney, Ed.). ClojureTV. Retrieved January 30, 2022, from https://www.youtube.com/watch?v=f84n5oFoZBc Live Recording Published on Youtube on 2012-12-26

Lai, A. (2017). Ada Lovelace: `poetical scientist'. *OR/MS Today*, *44*(1). Retrieved January 23, 2022, from https://www.informs.org/ORMS-Today/Public-Articles/February-Volume-44-Number-1/Ada-Lovelace-poetical-scientist

Levitin, A. (2018). On two algorithm design techniques (J. Meinke, Ed.). *Journal of Computing Sciences in Colleges*, *33*(3), 5–11. https://doi.org/10.5555/3144687.3144688