Victoria University of Wellington
School of Engineering and Computer Science

SWEN221: Software Development

Assignment 1

Due: Monday 16th March @ Mid-night

1 Regular Expressions

A *regular expression* is used to determine whether or not a particular line of text contains some pattern we're interested in. For example, we might want to check whether the text contains a particular word.

The syntax of our regular expressions is fairly straightforward: any character will match itself, except for the following *special characters*:

Regex	Operation
`.`	Match any character
`*`	Match any sequence of zero of more occurrences of the preceding character
`^`	Match start of input string
`$`	Match end of input string

The following examples illustrates regular expressions with matching and non-matching input strings:

Regex	Inputs which match	Inputs which don't match
abc	"daabc", "daveabcasd", etc	"adbc", "da", "ab", "ac", etc
`.b`	"ab", "bab", "davebee", etc	"b", "bd", "dd", etc
`bc*`	"daveb", "davebc", "dbccaa", etc	"da", "cc", "ccd", "cd", etc
`^c`	"cdavid", "chello", etc	"dcavid", "dc", "", etc
`c$`	"davidc", "helklkoc", etc	"cd", "d", "", etc
`.c*`	"d", "dc", "dcc", "dccd", etc	""
`^c*$`	"", "c", "cc", "ccc", etc	"dc", "cd", "cdc", etc
`^..`	"aa", "bc", "cdc", "cccd", etc	"", "d", "a", etc

Regular expressions are used frequently in software development. For example, Java has a library for matching regular expressions, whilst Perl provides constructs for manipulating them directly. For more information about regular expressions, see the Wikipedia page:

`http://en.wikipedia.org/wiki/Regular_expression`.

Finally, the regular expressions we're considering in this assignment are a simplified form of those commonly found elsewhere. We do not expect you to use any special operators other than those discussed in this document.

2 Part 1 — Debugging and Testing (worth 50% of mark)

The aim here is to debug and test an existing implementation for matching regular expressions. You will need to fix some existing bugs, and write new test cases. The following steps should get you started on the assignment.

1. Download the file `regex.jar` available on the SWEN221 lecture schedule.
2. Create a new project and import `regex.jar`.
3. Run the JUnit tests provided (called `BasicMatcherTests`); you should find that a number of the tests fail.
4. Debug the file `BasicMatcher.java` so that all the JUnit tests pass.
5. Finally, you should add more JUnit tests to help you find (and correct) any other bugs in `BasicMatcher`.

Note: a method named `matchWithLib()` is provided as a reference: it accepts the same input arguments as `match()`, but always gives the right answer! However, you may only use this method (or the Java regex library) for testing, but not as a way to fix bugs in `BasicMatcher`.

3 Part 2 — Extending the Code (worth 40% of mark)

The aim here is to extend the implementation for matching regular expressions to support the "+" special operator:

Regex	Operation
`+`	Match any sequence of one of more occurrences of the preceding character

Observe that this is very similar to the "*", except that it requires at least one occurrence of the character in question. The following examples illustrates some regular expressions and matching and non-matching input strings:

Regex	Inputs which match	Inputs which don't match
`bc+`	"davebc", "dbccaa", "bc", etc	"daveb", "b", "bac", "cc", "ccb", etc

You are strongly advised to begin by writing as many JUnit tests as you can think of. Only once you've done this, should you begin extending the system. When your implementation is complete you should ensure that all your JUnit tests, including those from before, pass.

4 Submission

Your program code should be submitted electronically via the *online submission system*, linked from the course homepage. You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. See the export-to-jar tutorial linked from the course homepage for more on how to do this. *Note, the jar file does not need to be executable.*

2. **The names of all classes, methods and packages remain unchanged**. That is, you may add new classes and/or new methods and you may modify the body of existing methods. However, you may not change the name of any existing class, method or package. *This is to ensure the automatic marking script can test your code.*

3. **All JUnit test files supplied for the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. This does not prohibit you from adding new tests, as you can still create additional JUnit test files. *This is to ensure the automatic marking script can test your code.*

4. **You have removed any debugging code that produces output, or otherwise affects the computation.** *This ensures the output seen by the automatic marking script does not include spurious information.*

Note: Failure to meet these requirements could result in you getting zero marks for the assignment.

5 Assessment

This assignment will be marked as a letter grade (A+ ... E), based primarily on the following criteria:

- **Correctness of Part 1 (50%)** — does submission adhere to specification given for Part 1.
- **Correctness of Part 2 (40%)** — does submission adhere to specification given for Part 2.
- **Style (10%)** — does the submitted code follow the style guide and have appropriate comments (inc. Javadoc)

5.1 Style

As indicated above, part of the assessment for the coding assignments in SWEN221 involves a qualitative mark for style, given by a tutor. Whilst this is worth only a small percentage of your final grade, it is worth considering that good programmers have good style. The qualitative marks for style are given for the following points:

- **Division of Concepts into Classes**. This refers to how *coherent* your classes are. That is, whether a given class is responsible for single specific task (coherent), or for many unrelated tasks (incoherent). In particular, big classes with lots of functionality should be avoided.

- **Division of Work into Methods**. This refers to how well a given task is split across methods. That is, whether a given task is broken down into many small methods (good) or implemented as one large method (bad). The approach of dividing a task into multiple small methods is commonly referred to as *divide-and-conquer*.

- **Use of Naming**. This refers to the choice of names for the classes, fields, methods and variables in your program. Firstly, naming should be consistent and follow the recommended Java Coding Standards (see `http://g.oswego.edu/dl/html/javaCodingStd.html`). Secondly, names of items should be descriptive and reflect their purpose in the program.

- **JavaDoc Comments**. This refers to the use of JavaDoc comments on classes, fields and methods. We certainly expect all `public` and `protected` items to be properly documented. For example, when documenting a method, an appropriate description should be given, as well as for its parameters and return value. Good style also dictates that `private` items are documented as well.

- **Other Comments**. This refers to the use of commenting within a given method. Generally speaking, comments should be used to explain what is happening, rather than simply repeating what is evident from the source code.

Finally, in addition to a mark, you should expect some written feedback highlighting the good and bad points of your solution.