Appendix 1 Regular Expressions

Regular Expressions

We will step through our own tutorial, but if you'd like a more thorough introduction, you can follow through this link:

http://docs.oracle.com/javase/tutorial/essential/regex/

and test out things as they are introduced there instead of this portion of our lab dedicated to regular expressions.

Regular expressions are an extremely powerful mechanism for more nuanced searching through strings than a simple search for specific substrings. A regular expression defines a pattern that we can search for within a given string. We might ask whether a particular string entirely and exactly matches a pattern, or we might instead ask whether a pattern can be found to match some portion (substring) of the string. We might also use a regular expression to extract substrings from a string, or to replace portions of a string with some replacement, whenever the pattern matches in that string. In short, whenever we find ourselves wanting to perform string manipulations, chances are that regular expressions are available as a way to express what we want to occur.

Regular expressions show up all over the place in programming and computer science – not just in Java. One common use of regular expressions is to search for a file on your computer (e.g., using the <code>egrep UNIX</code> command).

A regular expression is simply a way to represent structure within a string of symbols—for instance, identifying what makes a valid phone number, identifying if a particular word is included, et cetera.

A regular expression defines a set of strings. That set of strings is all the strings that comply with the regular expression; all strings which do not comply are not in the set.

We can match things directly by simply having them there. The regular expression <u>Cat</u> defines a set of strings with exactly one element, the string "Cat" (notice the sensitivity to case—it does not match "cat"). If we want to match other characteristics, we start developing a set of symbols that imply things like "at least this many of those", "any one of these", "anything not of this group", and so on. Let's define some of those now.

This does not cover all of regular expressions in Java – it is just an introduction.

pattern/symbol	meaning
any non-special char	matches itself
	matches any single char (except newline in some circumstances)
*	repetition: matches zero or more of the preceding thing
+	repetition: matches one or more of the preceding thing
?	repetition: matches zero or one of the preceding thing
{7}	repetition: matches exactly 7 of the preceding thing
{3,6}	repetition: matches from 3 to 6 (inclusive) of the preceding thing
{7,}	repetition: matches 7 or more of the preceding thing
(pattern)	parentheses group a pattern.
pattern1 pattern2	selection: matches either pattern1 or pattern2
[aeiou]	character class: matches any single character listed in []'s
[a-z]	matches any single character in ASCII range a to z.
[^stuff]	matches any single char not in stuff
[main[nested]]	matches any single char in the main or nested character class
[main&&[nested]]	matches any single char that is in both main and nested
^	anchor: 'matches' beginning of input
\$	anchor: 'matches' end of input
\b \B	matches a word boundary (\B matches a non-word boundary)
\d \w \s	matches a single char of a: digit, word, whitespace.
\D \W \S	matches a single char that is NOT a digit, word, whitespace.

For testing cases, repeatedly modify the value of regex in the following code:

```
public class TestLabRegex {
```

Your Turn!

- For each of the following regular expressions, think of some Strings that do match it, and Strings that do not match it. Test it with the code that calls the matches method on a String.
 - (way-)+back machine
 - $\circ \underline{x?y+z^*}$
 - <u>(a|b)*</u>
 - o (something) to do
 - <u>abc+</u>
 - <u>(abc)+</u>
 - (do|re|mi|fa|sol|la|ti|do)+

When designing your own regular expression, you should always ask yourself two questions: (1) Does it accept enough Strings? (2) Does it accept too many Strings?

Your Turn!

- Create regular expressions that exactly match the set of Strings described in each example.
 - any number of letters, followed by seventy exclamation points
 - one of the Pac-Man ghosts: inky, binky, pinky, and clyde (try to be concise)
 - zero to twenty a's, followed by an h.
 - write a regular expression that matches one of the following smileys: :) :(:?) 8-P
 - valid Java identifiers
 - a valid Java int representation (just allow base 10 representations, so no leading 0's)
 - any valid int[] initializer, such as {1,4,2,5} or {100,101,-4}

Let's review some of those with more descriptive examples. We will underline a regular expression to help mark the boundaries without resorting to the specific String representations, which have a couple of twists related to escape characters.

- **Character Class:** To say that we want any of a batch of characters, but just one of them, we surround all of them with square brackets [], with no space in between. The regular expression [aeiou]t matches "at", "et", "it", "ot", and "ut". We always choose exactly one of the options in the brackets: not zero, not many.
- To say that we want to match any character *except* what's listed in a character class, we use the ^ sign at the front of them all. The regular expression [^aeiou]t matches all consonant-followed-by-t strings: "bt", "ct", "dt", "ft"; but it also matches non-characters; "5t", "(t", and more would also match.
- Since typing things like <a>[456][456][456][456] can get tedious, we can specify how many of a particular pattern we want, by following the brackets with curly braces indicating how many. That could be reduced to <a>[456][3], indicating we want exactly three digits (all either a 4, 5, or 6). If we want *at least* n repetitions but no more than m repetitions, we represent this as in <a>[456][n,m]. A specific example would be <a>[456][1,3], which matches either a one, two, or three digit number (each digit being a 4, 5, or 6). Thus, the regular expression <a>[45][1,2] matches "4", "5", "44", "45", "54", and "55". If we have a minimum bound but no upper bound, we can leave that part out—<a>[45][1,] matches numbers with at least one digit, of only 4's and 5's. If we wanted up to some number of matches, we use a zero for the lower bound: e.g., <a>[456][0,4].
- We can match ranges of values by a dash; [0-9] matches any single digit; [a-z] matches any single lowercase letter; [A-Za-z] matches any letter regardless of case, and so on. Notice this only works inside brackets: 0-9 matches the string "0-9". Also, [1-10] matches "1" or "0", it does not match "1","2","3",...,"8","9","10".
- If we want to allow a pattern to repeat any amount of times, we place an asterisk after it: <u>a*t</u> matches "t", "at", "aat", "aaat", etc. This represents zero or more a's followed by exactly one t. A close cousin to the star is the plus symbol +, used to represent one or more repetitions of the pattern. <u>a+t</u> matches "at", "aat", etc., but does *not* match "t".
- To choose between one sub-pattern and another, we use the vertical bar <u>|</u> to represent union of the sets created by each of them. So <u>(a|parthe)non</u> matches "anon" and "parthenon".
- If we want a range of values minus a few particular values, we can use <u>&&</u> to intersect the two expressions: <u>[a-z&&[^abc]]</u> is all lowercase letters except a, b, c. Be sure that you use the inner []'s! If ^ doesn't appear as the very first character of a character class, it is not a special character and just matches itself.
- When we want to group parts of a pattern together, we can use parentheses to separate them. For instance, if we want to match as many repetitions of the prefix "sub" at the front of the word

"saharan", we could use parentheses as follows: (sub)*saharan matches "saharan", "subsaharan", subsaharan", etc.

- We can match *any* character at all with a period (.), except a newline character (\n). So <u>.t</u> matches "at", "bt", ..., "At", ..., "1t", ..., "\$t", "_t", Oftentimes we mean a more restricted set like [a-zA-z].
- Spaces are matched by leaving in the space directly; spaces *matter* in a pattern. a lot does not match "alot"; it only matches "a lot".

Your Turn!

- write a character class that matches a single one of the following letters: ceiknop (pick one)...
- write a character class that matches a single upper-case non-vowel in the second half of the alphabet.
- write a character class that matches any single digit character(this is just \d being defined manually).
- write a regular expression using a character class to represent a phone number in the format of 555-5555.
 - bonus: disallow any String that starts with 911.
- write a regular expression for length-3 Strings that don't match your initials in any position. (if you have more or fewer initials, adapt it to yourself of course!)

Escaping Special Characters

What if we wanted to match a square bracket, or a parenthesis, or a period? All characters which get 'used up' by controlling how we match characters can still be included, by placing a back-slash in front of them to indicate that we want the actual character, and not the function it usually provides in a regular expression. So $\langle [[0-9] \rangle]$ matches [0], [1], [2], ... [9]. This also works for others: $\langle * \rangle + \rangle . \rangle$, matches the string "*+.,". This can be done with all the special characters.

Representing Regular Expressions in Java

One last thing to note, which can really throw a wrench in the gears: Java itself needs backslashes to represent a quote sign; so Strings are already using the backslash and the quote sign ". In order to keep the two separate, it gets a little messy: to represent the regular expression $"[\d]"$ as a String (notice the pattern is matching some quote characters), it is written:

String str = "\"[\\d]\"";

What happened?! We had to place a backslash in front of the first quote of the regular expression to indicate to Java that it was part of the String and not the end of the String; we had to put an extra backslash in front of the backslash (which was itself in front of d) to indicate that it was not for Java to figure out what control character was being used, but rather that it should leave the slash in as part of the regular expression; the second quote of the regular expression is like the first. The short version of handling this is that all backslashes and quotes get a backslash added in front of them when we go from a regular expression on paper to a regular expression stored as a String in-between the quotes when we construct it.

How to deal with this issue: first, just write out your regular expression, not worrying about Java. Perhaps in a comment if you want to record it in your code. Then, character-for-character, represent them in a Java String. Given the bizarre regex abc'' + bsheB + e. we can represent it character for character:

- <u>a</u> is just "a"; same for <u>b</u> and <u>c</u>.
- Then, the very next character to consider is <u>\</u>. Don't worry that it is logically part of <u>\</u>*, because we just attack one character at a time. <u>\</u> becomes "\\" when we put it in a String.
- Next, <u>**</u> is just "**"
- \ is "\\"
- bshe is "bshe"
- \ is "\\"
- B is "B"
- \perp is "\\", again the second \perp is "\\"
- <u>++</u> becomes "++"
- <u>"</u> becomes "\""
- and lastly <u>.</u> is just "."

Let's put it all together:

<u>abc"**\bshe\B\\++".</u> \rightarrow "abc\"**\\bshe\\B\\\\++\"."

Your Turn!

• With the testing code above, explore uses of all the rest of the patterns and symbols shown above. If you work with a Scanner, reading input from the user, though, the notion of "end of input" won't be the same as "end of line", and \$ won't behave quite the way we'd like for simple explorations. (This would require "embedded flags", but that is advanced enough that we won't cover them in this introductory lab).

We next want to get practice with the basic pattern matching capabilities that were added to the String class. We now have two new methods, with the following signatures:

```
public boolean matches(String regex) {...}
public String replaceAll(String regex, String replacement){...}
```

matches

We will learn how to use the first method, matches. First, we need to create a String to use with this method. In practice, this is often part of a document or user-entered information – something to be checked for resemblance to some known pattern.

String str = "kitty";

Next, we can check if it matches some regular expression, calling the matches method on it with a single parameter, the regular expression:

```
String regex = "ki[t]{2}y";
boolean b = str.matches(regex);
```

b should be true after this runs. What if we want to find out if a String simply *contains* some pattern in it, and we don't care where in the String that might be? Consider this:

```
String str = "cat on a hot tin roof";
boolean b = str.matches(".*hot.*");
```

This will return true as long as there are zero or more characters before h, o, t, and then zero or more characters. Test this with different values than what is in str above. Try it at the start of a word, at the end of a word, and with things in between the h, o, and t.

Consider trying to match phone numbers with this pattern:

 $[0-9] \{3\} - [0-9] \{3\} - [0-9] \{4\}$

Test a few phone numbers. Suppose, though, that you were asking someone to enter in their phone number and you didn't want to be too strict on the input format; let's assume we also want to allow phone numbers like (555)-334-2190, (555)334-2190, (555) 334 2190, or any other patterns you can think of. How can we handle this?

We have the parentheses to group patterns, and the vertical bar | to say "choose exactly one side of what's on either side of me". So we can make the patterns, 'or' them all together, and get a very flexible regular expression for phone numbers.

Your Turn!

• Try testing your program with two different kinds of phone number patterns first, and then adding others.

replaceAll

Remember the signature from above, which takes a regular expression String, a replacement string, and returns the String created by replacing all matches:

public String replaceAll(String regex, String replacement){...}

Suppose we want to not just find out if something matches, but we want to replace the occurrence with something else; perhaps you wrote a form letter, and need to replace all instances of "John" with "Jack". Consider the following:

We can get this letter for Jack by just getting the String returned by replacing all "John"s with "Jack"s:

```
String jacksLetter = str.replaceAll("\\bJohn\\b", "Jack");
```

A few things to notice: str's value was not changed; jacksLetter operates on a *copy* of str. Also, "John" is a pretty tame pattern to match—that's where we could match things like our telephone number example from above, replacing all of them with just the local area phone numbers, for instance. Why do we have the \b word boundaries at the edge? What would happen if we left them out, and tried to replace every occurrence of "a" with "an"? Try this:

And that's why we have the word boundary delimiter: we care about the surrounding environment of the "a" matches that should be replaced; only the ones surrounded by word boundaries should be replaced.

Your Turn!

- replace every occurrence of the word "he" with "she", and also "him" with "her" from some given input. Be sure to not accidentally match portions of other words such as there becoming tshere.
- replace every digit with an X. (Pretend you're making a safe version of a bank receipt where you don't want to print the account number).

Capture Groups

As we write our regular expressions, we can use parentheses to group things. If we have a match, wouldn't it be nice to extract the portions of our String that matched each specific parenthesized group? We can do this, but we first need to understand how to use the Pattern class (for "compiling" a String that we want to use as a regular expression into some actual internal representation of a regular expression), and how to use the Matcher class (for using a Pattern on specific Strings and seeing if matches are possible).

More Efficient Pattern Matching

Although we are capable of writing virtually all the pattern matching we'll ever want with this, it can get a bit tedious, creating Strings, calling matches on the regular expression and the string again and again; also, though it is definitely not a focus in this class, it takes significant time to figure out what a regular expression means before it can be applied, and so if we were to apply the same pattern numerous times (say, in a loop), it would be wasteful to re-figure out what that regular expression is every single usage. So we can create something to handle a particular regular expression and keep it around, just asking it if a String matches its pattern.

Enter the classes Pattern and Matcher. We can make an object of the class Pattern to store the understanding of a regular expression:

```
// need to import java.util.regex.*;
Pattern p = Pattern.compile( regex ) ;
```

(Notice that we didn't call the constructor, we called a static method of the Pattern class). Then, we can make an object of the class Matcher, which has an associated String with it:

```
Matcher m = p.matcher ( str ) ;
```

Then, when we want to know if str matches regex, we call the matches method on m (notice there's no parameter now):

```
boolean b = m.matches();
```

In the grand scheme of things (such as in a loop), this would be prudent as follows: instead of using the matches method with a String,

```
while (...) {
    String str =...;
    boolean b = str.matches ( "bar");
    ...
}
```

We can instead pull the compilation of the regular expression out of the loop:

```
Pattern p = Pattern.compile (regex);
while (...) {
    String str = ...;
    Matcher m = p.matcher(str);
    boolean b = m.matches();
}
```

Even though it looks like more code, it turns out that when we call matches on a String, it's just going to create a Pattern, create a Matcher, and call the matches method of the Matcher object and then throw the two objects away and just yield the result; even though we see more code, there's still less work going on.

Finally, we can discuss our capture groups! If we have a Matcher object and we call matches, and we successfully find a match, then we can next call the group method and retrieve any specific capture group by number, starting with 1 for the leftmost open parenthesis, and numbering upwards for each found open parenthesis:

public String group (int groupNum)

Here is an example:

If you call group(0), you get the entire match. You can also call group(), an overloaded version that is equivalent to group(0).

Your Turn!

- Match a phone number in some format, but do so using the Pattern and Matcher class. Now, remove the first three numbers (the area code), and print back just the rest.
- If the nth capture group was part of some repetition, only the last of the matches is remembered (and returned from the corresponding call to group (n)). Write a regular expression that matches any number of "yes" or "no" strings, followed by "final answer." Print back what the final answer was (the last yes or no).

Adventures in One-Liner Land

Though we like to keep lines of code "short" for ease of reading, sometimes fewer lines is also considered easy to read. Whenever we put in a string literal, like "puppy", what actually happens is that Java makes a new string object with that contents and then uses it right there. So we could match a literal string to a regular expression in a really-hardwired way, declaring no variables to hold the string or Pattern:

```
boolean b = ("puppy").matches("p[aeiou][p]{2}y");
```

Similarly, and perhaps more likely, if we were getting str from some computation or method call, we could put that in the first set of parentheses:

```
boolean b = (sc.nextLine()).matches("p[aeiou][p]{2,5}y");
```

Beyond the Chapter

Now you're a pattern-matching pro. Sort of. But there is so much more than we've done here! This was just the simplest of introductions. There are more complex patterns, strategies for how greedy or hesitant a pattern is to consume as much or as little as possible when finding a match, and many, many more pre-defined character classes. See here

(<u>http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html</u>) for more character classes, and see here (<u>http://docs.oracle.com/javase/tutorial/essential/regex/</u>) for a much more in-depth tutorial on regular expressions.