Chapter 4
Regular Expressions

Written by Dr. Mark Snyder
[minor edits for this semester by Dr. Kinga Dobolyi]

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 grep UNIX 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 Cat 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.

<table>
<thead>
<tr>
<th>pattern/symbol</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>any non-special char</td>
<td>matches itself</td>
</tr>
<tr>
<td>.</td>
<td>matches any single char (except newline in some circumstances)</td>
</tr>
<tr>
<td>*</td>
<td>repetition: matches zero or more of the preceding thing</td>
</tr>
<tr>
<td>+</td>
<td>repetition: matches one or more of the preceding thing</td>
</tr>
<tr>
<td>?</td>
<td>repetition: matches zero or one of the preceding thing</td>
</tr>
<tr>
<td>(pattern)</td>
<td>parentheses group a pattern</td>
</tr>
<tr>
<td>pattern1</td>
<td>selection: matches either pattern1 or pattern2</td>
</tr>
<tr>
<td>[aeiou]t</td>
<td>character class: matches any single character listed in [ ]’s</td>
</tr>
<tr>
<td>[a-z]</td>
<td>matches any single character in ASCII range a to z</td>
</tr>
<tr>
<td>^</td>
<td>anchor: 'matches' beginning of input</td>
</tr>
<tr>
<td>$</td>
<td>anchor: 'matches' end of input</td>
</tr>
<tr>
<td>\d \w \s</td>
<td>matches a single char of: digit, word, whitespace</td>
</tr>
<tr>
<td>\D \W \S</td>
<td>matches a single char that is NOT a digit, word, whitespace</td>
</tr>
</tbody>
</table>

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.
- 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 (including none), we place an asterisk after it: a*t 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. a+t matches "at", "aat", etc., but does not match "t".
- To choose between one sub-pattern and another, we use the vertical bar | to represent union of the sets created by each of them. So `(a|parthe)non` matches only "anon" and "parthenon".
- 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 \t matches "at", "bt", ..., "At", ..., "1t", ..., "$t", "_t", ... .
- 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".
For testing cases, repeatedly modify the value of regex in the following code (or use the link in project 2 to find an online regular expression tester):

```java
public class TestLabRegex {
    // basic matching example
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        String str = "";
        String regex = "k.t+y";
        System.out.println("\nPattern: "+regex+"\n");
        while (!str.equals("quit")) {
            System.out.println("next input please: ");
            str = sc.nextLine();
            System.out.println(
                "\nstring: "+str+"\nPattern: "
            +regex+"\nmatches: "+str.matches(regex)+"\n";
        }
    }
}
```

**Your Turn!**

1. 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. Write at least three strings that do match and one string that doesn't on your homework (does not have to be code).
   a. (way-)+back machine
   b. x?y+z*
   c. (ab)*
   d. (something|nothing|anything) to do
   e. abc+
   f. (abc)+
   g. ktty
   h. a*
   i. b+c*
   j. go+al!
   k. ba(na)*
   l. b(a|e|i|o|u)t
   m. (un)?sure?
   n. d(o|i)*t
   o. ((c|l|r|d)(o|a))+
   p. b[aeiou]t
   q. [a-zA-Z]
   r. [0-9]+

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!
2. Create regular expressions that exactly match the set of Strings described in each example. You can write just the regular expression for your homework; you don't need to write code.
   a. any number of letters, followed by seventy exclamation points
   b. one of the Pac-Man ghosts: inky, binky, pinky, and clyde (try to be concise)
   c. zero to three a's, followed by an h.
   d. write a regular expression that matches one of the following smileys: :) :(? 8-P
   e. a valid Java int representation (just allow base 10 representations, so no leading 0's)
   f. any valid int[] initializer, such as {1,4,2,5} or {100,101,-4}
   g. first and last name of you or one of your family members
   h. "tweedledee" or "tweedledum"
   i. "I like you", "I like like you", "I like like like you", etc.
   j. an even number

Your Turn!
3. Write a character class that matches a single one of the following letters: ceiknop (pick one)...
4. Write a character class that matches a single upper-case non-vowel in the second half of the alphabet.
5. Write a character class that matches any single digit character (this is just \d being defined manually).

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 \\[[0-9]\]] matches [0], [1], [2], ... [9]. This also works for others: \\*\\+\\-\\., 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:

```java
String str = "\\[\d]\";`
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"\**\bshe\B\++"`, we can represent it character for character:

- `a` is just "a"; same for `b` and `c`.
- Then, the very next character to consider is ", which becomes "\" when we put it in a String
- \ is "\"
- Next, ** is just "**
- \ is "\"
- `b` is "b"
- `she` is "she"
- \ is "\"
- `B` is "B"
- ++ becomes "++"
- _ becomes "\"
- and lastly . is just "."

Let's put it all together:

```
abc"\**\bshe\B\++". → "abc\"\**\bshe\B\\\++\".
```

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:

```java
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][t]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.
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 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:

```java
// 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:

```java
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):

```java
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,

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

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

```java
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:

```java
public String group (int groupNum)
```

Here is an example:

```java
// group numbers:            1 2         3
Pattern p = Pattern.compile("(c(alu)r)tai(n|l)\)"};
Matcher m = p.matcher("curtail");
if (m.matches()){
    String first  = m.group(1);
    String second = m.group(2);
    String third  = m.group(3);
    System.out.println("saw "+second+" inside of " + first
                      +", all in front of "+third+".");
}
else {
    System.out.println("no match. If called, group(#) would throw "
                      +"an IllegalStateException."");
}
```

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!

6. Match an address in some format, but do so using the `Pattern` and `Matcher` class. Now, remove the last five numbers (the zip code), and print back just the rest. Use groups (parentheses) to solve this.

Greedy pattern matching

Imagine you have the regular expression `(a*)(a*)`, and the string “aaaaaaaaaa”. Obviously, the regular expression matches the string. However, if we were to print out group 1 and group 2, how many a-s would each group contain? Try it out. The point here is that there are rules that govern how greedy a pattern is; there is no need to memorize them, but you should be aware that a regular expression may match a string differently than you might expect, if there are multiple valid ways to match a string to a pattern.

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