Regular Expressions in Java

Regular Expressions

- A regular expression is a pattern that can be applied to text (Strings, in Java)

- A regular expression
  - either matches the text,
  - or part of the text,
  - or it fails to match

Getting started …

- The regular expression "[a-z]+" will match a sequence of lowercase letters
  - [a-z] means any character from a to z, inclusive
  - + means “one or more”
- Apply this pattern to "Now is the time"
  - To the *entire string:* it fails to match because the string contains characters other than lowercase letters
  - To the *beginning of the string:* it fails to match because the string does not begin with a lowercase letter
  - To *search the string:* it will succeed and match ow
    - If applied repeatedly, it will find is, then the, then time, then fail

Getting Started

- First, you must *compile* the pattern

```
import java.util.regex.*;
Pattern p = Pattern.compile("[a-z]+");
```

- Next, you must create a *matcher*

```
Matcher m = p.matcher("Now is the time");
```

Some useful methods

- m.matches() returns true if the pattern matches the entire text string
  - m.start() will return the index of the first character matched
  - m.end() will return the index of the last character matched, plus one

- m.lookingAt() returns true if the pattern matches at the beginning of the string

- m.find() returns true if the pattern matches any part of the text string; If called again, m.find() will start searching from where the last match was found

```
import java.util.regex.*;

public class RegexTest {
    public static void main(String args[]) {
        String pattern = "[a-z]+";
        String text = "Now is the time";
        Pattern p = Pattern.compile(pattern);
        Matcher m = p.matcher(text);
        while (m.find()) {
            System.out.print(text.substring(m.start(), m.end()) + "*");
        }
    }
}
```

Output: ow*is*the*time*

Some simple patterns

abc	exactly this sequence of three letters
[abc]	any *one* of the letters a, b, or c
[^abc]	any character *except* one of the letters a, b, or c (immediately within an open bracket, ^ means “not,” but anywhere else it just means the character ^)
[a-z]	any *one* character from a to z, inclusive
[a-zA-Z0-9]	any *one* letter or digit

Sequences and alternatives

- If one pattern is followed by another, the two patterns must match consecutively
  - `[A-Za-z]+[0-9]` will match one or more letters immediately followed by one digit
- The vertical bar, `|`, is used to separate alternatives
  - `abc|xyz` will match either `abc` or `xyz`

Basic Syntax

Char	Usage	Example
.	Matches any single character	.at = cat, bat, rat, 1at…
*****	Matches zero or more occurrences of the preceding character	.*at = everything that ends with at 0*123 = 123, 0123, 00123…
[…] **[^…]**	Matches any single character of the contained character Matches any single character except for the contained characters	[cbr]at = cat, bat, rat. [^bc]at = rat, sat…, *but not* bat, cat. <[^>]*> = <…anything…>
^	Beginning of line	^a = line starts with a
$	End of line	^$ = blank line (starts with the end of line)
****	Escapes following special character: . \ / & [ ] * + -> \. \\ \/ \& \[\] * \+	[cbr]at\. = matches cat., bat. and rat. only
…	…	

Some predefined character classes

`.`	any one character except a line terminator
`\d`	a digit: `[0-9]`
`\D`	a non-digit: `[^0-9]`
`\s`	a whitespace character: `[ \t\n\x0B\f\r]`
`\S`	a non-whitespace character: `[^\s]`
`\w`	a word character: `[a-zA-Z_0-9]`
`\W`	a non-word character: `[^\w]`

Boundary matchers

^	the beginning of a line
$	the end of a line
\b	a word boundary
\B	not a word boundary
\A	the beginning of the input (can be multiple lines)
\Z	the end of the input except for the final terminator, if any
\z	the end of the input
\G	the end of the previous match

Quantifiers

Greedy	Reluctant	Possessive	Meaning
X?	X??	X?+	X, once or not at all
X*	X*?	X*+	X, zero or more times
X+	X+?	X++	X, one or more times
X{n}	X{n}?	X{n}+	X, exactly n times
X{n,}	X{n,}?	X{n,}+	X, at least n times
X{n,m}	X{n,m}?	X{n,m}+	X, at least n but not more than m times

Quantifier Types

- **Greedy**: first, the quantified portion of the expression eats the whole input string and tries for a match.
  - If it fails, the matcher backs off the input string by one character and tries again, until a match is found.
- **Reluctant**: starts to match at the beginning of the input string.
  - Then, iteratively eats another character until the whole input string is eaten.
- **Possessive**: try to match only once on the whole input stream.

Suppose your text is succeed

Using the pattern suc*ce{2}d (c* is greedy):

- The c* will first match cc, but then ce{2}d won't match
- The c* then "backs off" and matches only a single c, allowing the rest of the pattern (ce{2}d) to succeed

Suppose your text is succeed

Using the pattern suc*?ce{2}d (c*? is reluctant):

- The c*? will first match zero characters (the null string), but then ce{2}d won't match
- The c*? then extends and matches the first c, allowing the rest of the pattern (ce{2}d) to succeed

Suppose your text is succeed

Using the pattern suc*+ce{2}d (c*+ is possessive):

– The c*+ will match the cc, and will not back off, so ce{2}d never matches and the pattern match fails.

Double backslashes

- Backslashes have a special meaning in regular expressions; for example, \b means a word boundary
- Backslashes have a special meaning in Java; for example, \b means the backspace character
- Java syntax rules apply first!
    - If you write "\b[a-z]+\b" you get a string with backspace characters in it--this is *not* what you want!
    - Remember, you can quote a backslash with another backslash, so "\\b[a-z]+\\b" gives the correct string
- Note: if you *read in* a String from somewhere, this does not apply--you get whatever characters are actually there

Escaping metacharacters

- A lot of special characters--parentheses, brackets, braces, stars, plus signs, etc.--are used in defining regular expressions; these are called metacharacters

- Suppose you want to search for the character sequence `a*` (an `a` followed by a star)
  - `"a*"`; doesn't work; that means "zero or more `a`s"
  - `"a\*"`; doesn't work; since a star doesn't *need* to be escaped (in Java String constants), Java just ignores the `\`
  - `"a\\*"` *does* work; it's the three-character string `a`, `\`, `*`

Spaces

- There is only one thing to be said about spaces (blanks) in regular expressions

  – *Spaces are significant!*

- A space stands for a *space*--when you put a space in a pattern, that means to match a space in the text string