

Regular Expressions in Java



*

Regular Expressions

- A regular expression is a kind of 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
 - If a regular expression matches a part of the text, then you can easily find out which part
 - If a regular expression is complex, then you can easily find out which parts of the regular expression match which parts of the text
 - With this information, you can readily extract parts of the text, or do substitutions in the text
- Regular expressions are an extremely useful tool for manipulating text
 - Regular expressions are heavily used in the automatic generation of Web pages

Perl and Java

- The Perl programming language is heavily used in server-side programming, because
 - Much server-side programming is text manipulation
 - Regular expressions are built into the syntax of Perl
- Beginning with Java 1.4, Java has a regular expression package, java.util.regex
 - Java's regular expressions are almost identical to those of Perl
 - This new capability greatly enhances Java 1.4's text handling
- Regular expressions in Java 1.4 are just a normal package, with no new syntax to support them
 - Java's regular expressions are just as powerful as Perl's, but
 - Regular expressions are easier and more convenient in Perl

A first example

- The regular expression "[a-z]+" will match a sequence of one or more lowercase letters
 - [a-z] means any character from a through z, inclusive
 - + means "one or more"
- Suppose we apply this pattern to the String "Now is the time"
 - There are *three ways* we can apply this pattern:
 - 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

Doing it in Java, I

• First, you must *compile* the pattern

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

Next, you must create a *matcher* for a specific piece of text by sending a message to your pattern

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

- Points to notice:
 - Pattern and Matcher are both in java.util.regex
 - Neither Pattern nor Matcher has a public constructor; you create these by using methods in the Pattern class
 - The matcher contains information about *both* the pattern to use *and* the text to which it will be applied

Doing it in Java, II

- Now that we have a matcher m,
 - m.matches() returns true if the pattern matches the entire text string, and false otherwise
 - m.lookingAt() returns true if the pattern matches at the beginning of the text string, and false otherwise
 - m.find() returns true if the pattern matches any part of the text string, and false otherwise
 - If called again, m.find() will start searching from where the last match was found
 - m.find() will return true for as many matches as there are in the string; after that, it will return false
 - When m.find() returns false, matcher m will be *reset* to the beginning of the text string (and may be used again)



Finding what was matched

- After a successful match, m.start() will return the index of the first character matched
- After a successful match, m.end() will return the index of the last character matched, plus one
- If no match was attempted, or if the match was unsuccessful,
 m.start() and m.end() will throw an IllegalStateException
 - This is a RuntimeException, so you don't have to catch it
- It may seem strange that m.end() returns the index of the last character matched plus one, but this is just what most String methods require
 - For example, "Now is the time".substring(m.start(), m.end()) will return exactly the matched substring

A complete example

```
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*

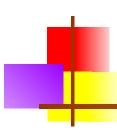


Additional methods

- If m is a matcher, then
 - m.replaceFirst(*replacement*) returns a new String where the first substring matched by the pattern has been replaced by *replacement*
 - m.replaceAll(replacement) returns a new String where every substring matched by the pattern has been replaced by replacement
 - m.find(startIndex) looks for the next pattern match, starting at the specified index
 - m.reset() resets this matcher
 - m.reset(newText) resets this matcher and gives it new text to examine (which may be a String, StringBuffer, or CharBuffer)

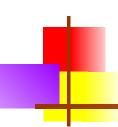
Some simple patterns

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 through **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
 - For example, [A-Za-z]+[0-9] will match one or more letters immediately followed by one digit
- The vertical bar, |, is used to separate alternatives
 - For example, the pattern abc | xyz will match either abc or xyz



Some predefined character classes

any one character except a line terminator

Notice the space. Spaces are **significant** in regular expressions!

```
\d a digit: [0-9]
```

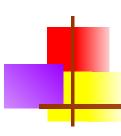
```
\D a non-digit: [^0-9]
```

\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

- These patterns match the *empty string* if at the specified position:
 - ^ 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

Greedy quantifiers

```
(The term "greedy" will be explained later) Assume X represents some pattern
```

```
X? optional, X occurs once or not at all
X* X occurs zero or more times
X+ X occurs one or more times
X{n} X occurs exactly n times
X{n,} X occurs n or more times
X{n,m} X occurs at least n but not more than m times
```

Note that these are all *postfix* operators, that is, they come *after* the operand

Types of quantifiers

- A greedy quantifier will match as much as it can, and back off if it needs to
 - We'll do examples in a moment
- A reluctant quantifier will match as little as possible, then take more if it needs to
 - You make a quantifier reluctant by appending a ?: X?: X*? X*? X{n}? X{n,}? X{n,m}?
- A possessive quantifier will match as much as it can, and never let go
 - You make a quantifier possessive by appending a +: $X?+ X*+ X++ X\{n\}+ X\{n,\}+ X\{n,m\}+$

Ç

Quantifier examples

- Suppose your text is aardvark
 - Using the pattern a*ardvark (a* is greedy):
 - The a* will first match aa, but then ardvark won't match
 - The a* then "backs off" and matches only a single a, allowing the rest of the pattern (ardvark) to succeed
 - Using the pattern a*?ardvark (a*? is reluctant):
 - The a*? will first match zero characters (the null string), but then ardvark won't match
 - The a*? then extends and matches the first a, allowing the rest of the pattern (ardvark) to succeed
 - Using the pattern a*+ardvark (a*+ is possessive):
 - The a*+ will match the aa, and will not back off, so ardvark never matches and the pattern match fails

C

Capturing groups

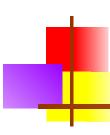
- In regular expressions, parentheses are used for grouping, but they also capture (keep for later use) anything matched by that part of the pattern
 - Example: ([a-zA-Z]*)([0-9]*) matches any number of letters followed by any number of digits
 - If the match succeeds, \1 holds the matched letters and \2 holds the matched digits
 - In addition, **\0** holds everything matched by the entire pattern
- Capturing groups are numbered by counting their opening parentheses from left to right:

```
• ((A)(B(C)))

12 3 4

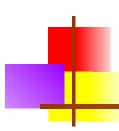
(0 = 1 = ((A)(B(C))), (2 = (A), (3 = (B(C)), (4 = (C)))
```

Example: ([a-zA-Z])\1 will match a double letter, such as letter



Capturing groups in Java

- If m is a matcher that has just performed a successful match, then
 - m.group(n) returns the String matched by capturing group n
 - This could be an empty string
 - This will be **null** if the pattern as a whole matched but this particular group didn't match anything
 - m.group() returns the String matched by the entire pattern (same as m.group(0))
 - This could be an empty string
- If m didn't match (or wasn't tried), then these methods will throw an IllegalStateException



Example use of capturing groups

- Suppose word holds a word in English
- Also suppose we want to move all the consonants at the beginning of word (if any) to the end of the word (so string becomes ingstr)

```
Pattern p = Pattern.compile("([^aeiou]*)(.*)");
Matcher m = p.matcher(word);
if (m.matches()) {
   System.out.println(m.group(2) + m.group(1));
}
```

 Note the use of (.*) to indicate "all the rest of the characters"

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

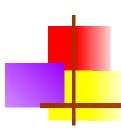


Escaping metacharacters

- 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 as"
 - "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, \, *
- Just to make things even more difficult, it's *illegal* to escape a *non*-metacharacter in a regular expression

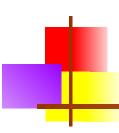
Spaces

- There is only one thing to be said about spaces (blanks) in regular expressions, but it's important:
 - 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
- It's a *really bad idea* to put spaces in a regular expression just to make it look better



Additions to the String class

- All of the following are public:
 - public boolean matches(String regex)
 - public String replaceFirst(String regex, String replacement)
 - public String replaceAll(String regex, String replacement)
 - public String[] split(String regex)
 - public String[] split(String regex, int limit)
 - If the limit n is greater than zero then the pattern will be applied at most n
 1 times, the array's length will be no greater than n, and the array's last
 - entry will contain all input beyond the last matched delimiter.
 - If n is non-positive then the pattern will be applied as many times as possible



Thinking in regular expressions

- Regular expressions are *not* easy to use at first
 - It's a bunch of punctuation, not words
 - The individual pieces are not hard, but it takes practice to learn to put them together correctly
 - Regular expressions form a miniature programming language
 - It's a different kind of programming language than Java, and requires you to learn new thought patterns
 - In Java you can't just *use* a regular expression; you have to first create Patterns and Matchers
 - Java's syntax for String constants doesn't help, either
- Despite all this, regular expressions bring so much power and convenience to String manipulation that they are well worth the effort of learning

