Regular Expressions in Java

Regular Expressions

- **Regular expressions are an extremely useful tool for manipulating text, heavily used**
  - in the automatic generation of Web pages,
  - in the specification of programming languages,
  - in text search.
- **generalized to patterns that can be applied to text (or strings) for string matching.**
- **A pattern can either match the text (or part of the text), or fail to match**
  - If matching, you can easily find out which part.
  - For complex regular expression, you can 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

Perl and Java

- **Perl is the most famous programming language in which r**egular expressions are built into syntax.
- **since jdk 1.4, Java has a regular expression package:** java.util.regex
  - almost identical to those of Perl
  - 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 to use in Perl compared to java.**

Classes In Regex

Pattern Class:

A Pattern object is a compiled representation of a regular expression.

The Pattern class provides no public constructors.

To create a pattern, you must first invoke one of its public static compile methods, which will then return a Pattern object.

These methods

accept a regular expression as the first argument.

Matcher Class:

A Matcher object is the engine that interprets the pattern and performs

match operations against an input string.

Like the Pattern class, Matcher defines no public constructors.

You obtain a Matcher object by invoking the matcher method on a Pattern

object.

PatternSyntaxException:

A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

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 the target text is “The game is over”.**
- **Then patterns can be applied in three ways:**
  - ▯ **To the *entire string*:**
  - ▯ **=> fails to match since 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 he.**
  - ▯ **=> If applied repeatedly, it will find game, then is, then over, then fail.**

Pattern match in Java

- First, you must *compile* the pattern

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

- Next, create a *matcher* for a target text by sending a message to your pattern

```
Matcher m = p.matcher(“The game is over");
```

- Notes:
  - Neither Pattern nor Matcher has a public constructor;
    - use static Pattern.compile(String regExpr) for creating pattern instances
    - using Pattern.matcher(String text) for creating instances of matchers.
  - The matcher contains information about *both* the pattern *and* the target text.

Pattern match in Java (continued)

After getting a matcher m,

- use m.match() to check if there is a match.
  - returns true if the pattern matches the entire text string, and false otherwise.
- use m.lookingAt() to check if the pattern matches a prefix of the target text.
- m.find() returns
  - true iff the pattern matches any part of the text string,
  - 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
  - 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 (a RuntimeException).
- **Example:**
  - “The game is over".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 = “The game is over";
        Pattern p = Pattern.compile(pattern);
        Matcher m = p.matcher(text);
        while (m.find()) {
            System.out.print(text.substring(m.start(),
m.end()) + "*");
        }
    }
}
```

Output: he*is*over*

Capturing Groups:

- Capturing groups are a way to treat multiple characters as a single unit.
- They are created by placing the characters to be grouped inside a set of parentheses.

For example, the regular expression dog creates a single group containing the letters "d", "o", and "g".

- Capturing groups are numbered by counting their opening parentheses from left to right.

In the expression (AB(C)), for example, there are four such groups:

(AB(C))

A

B(C)

C

groupCount() method

- It returns no of groups are present in the expression with matcher object.
- There is also a special group, group 0, which always represents the entire expression.
- This group is not included in the total reported by groupCount.

<u>find a digit string from the given alphanumeric string</u>

```
 String line = "This order was placed for QT3000! OK?";
String pattern = "(.*)(\\d+)(.*)";
Pattern r = Pattern.compile(pattern);
   // Now create matcher object.
   Matcher m = r.matcher(line);
   if (m.find( )) {
      System.out.println("Found value: " + m.group(0) );
      System.out.println("Found value: " + m.group(1) );
      System.out.println("Found value: " + m.group(2) );
   }
```

Result

```
Found value: This order was placed for QT3000! OK?
Found value: This order was placed for QT300
Found value: 0
```

Additional methods

If m is a matcher, then

- m.replaceFirst(*newText*)
  - returns a new String where the first substring matched by the pattern has been replaced by *newText*
- m.replaceAll(*newText*)
  - returns a new String where every substring matched by the pattern has been replaced by *newText*
- 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.

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**

[ab^c]

- **a, b, ^ or c.**
- **(immediately within [, ^ mean “not,” but anywhere else mean the character ^)**

[a-z]

- **any *one* character from a through z, inclusive**

[a-zA-Z0-9]

- **any *one* letter or digit**

```
Pattern pattern = Pattern.compile("[0-9]");
    //("[a-zA-Z]");
    //("[^abc]");
    // Pattern.compile(" ");
    // Pattern.compile("[a-z]");
    // Pattern.compile("abc");
Matcher matcher = pattern.matcher(var);

while (matcher.find()) {
System.out.println( matcher.group());
```

Sequences and alternatives

- If one pattern is followed by another, the two patterns must match consecutively
  - Ex: [A-Za-z]+ [0-9] will match one or more letters immediately followed by one digit

- The vertical bar, |, is used to separate alternatives
  - Ex: the pattern abc|xyz will match either abc or xyz

Metacharacters

```
 /a.g/              # matches aag,abg,a1g etc
 /a[pmt]g/          # matches apg,amg or atg
/a[^pmt]g/          #matches aag,abg but not apg or amg or
 atg
 [0-9]              # match any single non-digit
[^aeiouAEIOU] # match any single non-vowel
[^\^]               # match single character except a caret.
[a-z]               # match single character from a to z
[^a-z]              #  match character other than lower case
                          letters
[a-zA-Z]            # match single character upper or lower.
```

Maximal Quantifiers

*	Zero or more occurrences of preceding character
+	One or more occurrences of preceding character
?	Zero or one occurrences of preceding character
{count}	match exactly "count" times
{min, }	match at least "min" times
{min,max}	match at least "min" and at most "max"

```
String s="we are learning java7 and 8";
Pattern pattern=Pattern.compile("we");
//("[a-z]?");
//("[a-z]+");
//("[a-z]*");
//("[0-9]");
```

Maximal Quantifiers

/de+f/	will match def,deef,deef…
/de*f/	matches df, def, deef…
/de{1,3}f/	matches def,deef,deeef
/de{3}f/	matches deeef
/de{3,}f/	matches deeef,deeeef…..
/de{0,3}f/	matches df,def,deef,deeef
/adg*/	ad followed by zero or more g characters
/.*/	any character, any number of times (except a newline)
/[a-z]+/	Any non-zero sequence of lower case letters
/jelly|cream/	Either jelly or cream
/(eg|pe)gs/	Either eggs or pegs
/(da)+/	Either da or dada or dadada or...
/[a-z]+|[0-9]+/	matches one or more lowercase letters or one or more digits.

```
//String var="deeeef";

Pattern pattern = Pattern.compile("de{1,3}");

//("de{1,}f");  min 1 e

//("de{3}f"); exact 3 e
//("de*f");
//("de?f");
//("de+f");
```

Pattern repetition

- **Assume *X* represents some pattern**

***X*?**	**optional, *X* occurs zero or one time**
***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.

ha* matches e.g. "haaaaaaaa"

ha{3} matches only "haaa"

(ha)* matches e.g. "hahahahaha"

(ha){3} matches only "hahaha"

Can be used with replace and replace all

```
 System.out.println(
        "xxxxx".replaceAll("x{2,3}", "[x]")
   );
```

(Minimal) Quantifiers

- **Placing a "?" after a quantifier disables greedyness, making them "non-greedy", "thrifty",or "minimal" quantifiers.**

*?	match zero or more times
+?	match one or more times
{min,}?	match at least min times
{min,max}?	match at least **min** times but no more than max times.

Some predefined character classes

.	any one character except a line terminator (Note: . denotes itself inside […]).
\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]

Notice the space.
Spaces are significant
in regular expressions!

Metacharacters

- Metacharacters do not get interpreted as literal characters. Instead they tell perl to interpret the metacharacter (and sometimes the characters around metacharacter) in a different way.
- The following are metacharacters in perl regular expression patterns:

  \ | ( ) [ ] { } ^ $ * + ? .

.	match any single character (usually not "\n")
[]	define a character class, match any single character in class
\	Quote the next metacharacter
()	Grouping class
|	alternation: (patt1 | patt2) means (patt1 OR patt2)

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

```
String var = "we are learning  java and  perl  we";

Pattern pattern = Pattern.compile("(we){2}");

     //("we$") found at end of line
      //("^we"); found at beginning of line
      //("[\\W]");
    // [^\\w]  or [\\W]  -- non word char
    //[a-zA-Z_0-9] or [\w]  -- word char
    // "[^\\s]" or  "[\\S]"
    // "[\\s]" -- white space
      // "[^0-9]"   or [\\D]
    //("[0-9]"); or ("[\\d]");
    //("java|we");
    //("[a-z]+[0-9]");
```

A string that begins with a capital letter and ends with a period, a question mark, or an exclamation point.

```
String pattern = "^[A-Z].*[\\.?!]$";
String s = "Java is fun!";
s.matches(pattern);
```

s.matches(".*\\bJava\\b.*");

// True if s contains the word "Java" anywhere

// The b specifies a word boundary

import java.util.regex.*;

```
Pattern javaword = Pattern.compile("\\bJava(\\w*)",
Pattern.CASE_INSENSITIVE);
Matcher m = javaword.matcher(sentence);
boolean match = m.matches();
```

// True if text matches pattern exactly

counts the number of times the word "cat" in given string

```
private static final String REGEX = "\\bcat\\b";
private static final String INPUT =
                                "cat cat cat cattie cat";
public static void main( String args[] ){
   Pattern p = Pattern.compile(REGEX);
   Matcher m = p.matcher(INPUT); // get a matcher object
   int count = 0;

   while(m.find()) {
     count++;
     System.out.println("Match number "+count);
     System.out.println("start(): "+m.start());
     System.out.println("end(): "+m.end());
  }
```

Email validation

```
static final Pattern PATTERN = Pattern.compile("[a-zA-Z0-9_]+(\\.[a-zA-Z0-9_]+)*@[a-zA-Z0-9_]+(\\.[a-zA-Z0-9_]+)+");

public static void main(String[] args) {
Scanner sc = new Scanner(System.in);

int N = sc.nextInt();
sc.nextLine();
String text = readText(sc, N);

SortedSet<String> emails = new TreeSet<String>();
Matcher matcher = PATTERN.matcher(text);
while (matcher.find()) {
String email = matcher.group();
emails.add(email);
}
System.out.println(String.join(";", emails.stream().collect(Collectors.toList())));
}
```

Split number

```
static final Pattern PATTERN = Pattern.compile("(\\d{1,3})[- ](\\d{1,3})[- ](\\d{4,10})");

public static void main(String[] args) {
Scanner sc = new Scanner(System.in);

int N = sc.nextInt();
sc.nextLine();
for (int tc = 0; tc < N; tc++) {
String line = sc.nextLine();

Matcher matcher = PATTERN.matcher(line);
matcher.find();
String countryCode = matcher.group(1);
String localAreaCode = matcher.group(2);
String number = matcher.group(3);

System.out.println(
String.format("CountryCode=%s,LocalAreaCode=%s,Number=%s", countryCode,
localAreaCode, number));
}
```

Detect domain name

```
static final Pattern PATTERN = Pattern
.compile("https?://(www.|ww2.)?([a-zA-Z0-9-]+(\\.[a-zA-Z0-9-]+)+)");
Scanner sc = new Scanner(System.in);

int N = sc.nextInt();
sc.nextLine();
String html = readHtml(sc, N);

SortedSet<String> domainNames = new TreeSet<String>();
Matcher matcher = PATTERN.matcher(html);
while (matcher.find()) {
String domainName = matcher.group(2);
domainNames.add(domainName);
}
System.out.println(String.join(";",
domainNames.stream().collect(Collectors.toList())));
```

Duplicate word

```
String pattern = "(?<!\\w)(\\w+)( \\1)*(?!\\w)";
Pattern r = Pattern.compile(pattern, Pattern.CASE_INSENSITIVE);

@SuppressWarnings("resource")
Scanner in = new Scanner(System.in);
int testCases = Integer.parseInt(in.nextLine());
while (testCases > 0) {
        String input = in.nextLine();
        Matcher m = r.matcher(input);
        boolean findMatch = true;
        while (m.find()) {
                input =
input.replaceAll(String.format("(?<!\\w)%s(?!\\w)", m.group()), m.group(1));
                findMatch = false;
        }
        System.out.println(input);
        testCases--;
}
```

Find Digit

```
int T = in.nextInt();
for (int tc = 0; tc < T; tc++) {
long N = in.nextLong();
System.out.println((N + "").chars()
.filter(digit -> digit != '0' && N % (digit - '0') == 0)
          .count());
                    }
```

Ip Address validation

```
String IPV4_REGEX = "^((\\d|\\d\\d|1\\d\\d|2[0-4]\\d|25[0-5])\\.){3}(\\d|\\d\\d|1\\d\\d|2[0-4]\\d|25[0-
5])$";
            String IPV6_REGEX = "^([0-9a-f]{1,4}:){7}[0-9a-f]{1,4}$";

                    Scanner sc = new Scanner(System.in);

                    int N = sc.nextInt();
                    sc.nextLine();
                    for (int tc = 0; tc < N; tc++) {
                            String line = sc.nextLine();

                            if (isIPv4(line)) {
                                    System.out.println("IPv4");
                            } else if (isIPv6(line)) {
                                    System.out.println("IPv6");
                            } else {
                                    System.out.println("Neither");
                            }
                    }
```

Matching digit and non dogit charcter

```
String Regex_Pattern=“\\d\\d\\D\\d\\d\\D\\d\\d\\d\\d";

            Scanner Input = new Scanner(System.in);
            String Test_String = Input.nextLine();
            Pattern p =
Pattern.compile(Regex_Pattern);
            Matcher m = p.matcher(Test_String);
            System.out.println(m.find());
```

matches vs find

- **matches** tries to match the expression against the entire string and implicitly add a ^ at the start and $ at the end of pattern.
- It will not look for a substring.
- **Find** It tries to find a substring that matches the pattern.
- It consider the sub-string against the regular expression
- But matches() will consider complete expression.
- find() will returns true only if the sub-string of the expression matches the pattern.

Example

matches() only 'sees' a123b which is not the same as 123 and thus outputs false.

```
public static void main(String[] args) {
        Pattern p = Pattern.compile("\\d");
        String candidate = "Java123";
        Matcher m = p.matcher(candidate);

        if (m != null){
            System.out.println(m.find());//true

System.out.println(m.matches());//false
        }
    }
```

```
Pattern p = Pattern.compile("\\d\\d\\d"); //  or 123
 Matcher m = p.matcher("a123b");
 System.out.println(m.find());
 System.out.println(m.matches());
p = Pattern.compile("^\\d\\d\\d$");
m = p.matcher("123");
System.out.println(m.find());
System.out.println(m.matches());
```

Output

```
true
false
true
true
```

---

123 is a substring of a123b so the find() method outputs true.

The matches and lookingAt Methods:

- The matches and lookingAt methods both attempt to match an input sequence against a pattern.
- The difference, however, is that matches requires the entire input sequence to be matched, while lookingAt does not.
- Both methods always start at the beginning of the input string.

Example

```
 private static final String REGEX = "foo";
   private static final String INPUT = "fooooooooooooooooo";
   private static Pattern pattern;
   private static Matcher matcher;
public static void main( String args[] ){
      pattern = Pattern.compile(REGEX);
      matcher = pattern.matcher(INPUT);
      System.out.println("Current REGEX is: "+REGEX);
      System.out.println("Current INPUT is: "+INPUT);
      System.out.println("lookingAt(): "+matcher.lookingAt());
      System.out.println("matches(): "+matcher.matches());
  }
}
```

Types of quantifiers

- A greedy quantifier [longest match first] (default) will match as much as it can , and back off if it needs to
  - An example given later.
- A reluctant quantifier [shortest match first] 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 [longest match and never backtrack] will match as much as it can, and never back off
  - You make a quantifier possessive by appending a +:
    X?+ *X**+ *X*++ *X*{*n*}+ *X*{*n*,}+ *X*{*n*,*m*}+

Quantifier examples

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
- 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
- Using the pattern au c*+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.

Capturing groups

- In RegExpr, parentheses (…) are used
  - for grouping, and also
  - for *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,
  - \2 holds the matched digits and
  - \0 holds everything matched by the entire pattern

Reference to matched parts

- **Capturing groups are numbered by counting their *left parentheses* from left to right:**

  -
    ```
    ( ( A ) ( B ( C ) ) )
    1 2     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**
- Note: Use of \1, \2, etc. in fact makes patterns more expressive than ordinary regular expression (and even context free grammar).
  - Ex: ([01]*)\1 represents the set { w w | w ∈ {0,1}* }, which is not context free.

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
  - = null if the pattern matched but this particular group didn't match anything.
  - Ex: If pattern a (b | (d)) c is applied to "abc".
  - then \1 = b and \2 = null.
  - m.group() = m.group(0) returns the String matched by the entire pattern.
- 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.
- goal: move all the consonants at the beginning of word (if any) to the end of the word
  - Ex: string → ingstr

  -
    ```
    Pattern p = Pattern.compile( "([^aeiou]*)(.*)" );
    Matcher m = p.matcher(word);
    if (m.matches()) {
      System.out.println(m.group(2) + m.group(1));
    }
    ```
- Notes
  - there are only five vowels a,e,i,o,u which are not consonants.
  - the use of (.*) to indicate “all the rest of the characters”

Double backslashes

- Backslashes(\) have a special meaning in both java and regular expressions.
  - \b means a word boundary in regular expression
  - \b means the backspace character in java
- The precedence : Java syntax rules apply first!
  - If you write “\b[a-z]+\b"
  - you try to get a string with two backspace characters in it!
  - you should use double backslash(\\)in java string literal to represent a backslash in a pattern, so
  - if you write "\\b[a-z]+\\b" you try to find a word.

Escaping metacharacters

- metacharacters : special characters used in defining regular expressions.
  - ex: (,), [,], {, }, *, +, ?, etc.
  - dual roles: Metacharqcters are also ordinary characters.
- Problem: search for the char sequence “a+” (an a followed by a +)
  - "a+ “ (x) it means “one or more as”
  - "a\+"; (x) compile error since ‘+’ could not be escaped in a ava string literal.
  - "a\\+" (0); it means a \ + in java, and means two ordinary chars a + in reg expr.

Spaces

- One importtant thing to remamber about spaces (blanks) in regular expressions:
  - *Spaces are significant!*
  - I.e., A space is an ordinary char and stands for itself, a *space*
  - So It’s a *bad idea* to put spaces in a regular expression just to make it look better.
- Ex:
  - Pattern.compile("a b+").matcher("abb"). matches()
  - return false.

Conclusions

- Regular expressions are *not* easy to use at first
  - It's a bunch of punctuation, not words
  - it takes practice to learn to put them together correctly.
- Regular expressions form a sublanguage
  - It has a different syntax than Java.
  - It requires new thought patterns
  - can't *use* regular expressions directly in java; you have to create Patterns and Matchers first.
- Regular expressions is powerful and convenient to use for string manipulation
  - It is worth learning !!