A Quick Introduction to Regular Expressions in Java

Lecture 10a

RegEx

Readings

- SUN regexps tutorial

http://java.sun.com/docs/books/tutorial/extra/regex/index.html

- Java.util.regex API

http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/package-summary.html

Regular Expressions

- Regular expressions (regex's) are sets of symbols and syntactic elements used to match patterns of text.

Motivating Example

- "I want to find all book titles that contain the word JDBC"

  - Go through all strings and search for "JDBC"
  - In file system we would do: **`ls *JDBC*`**

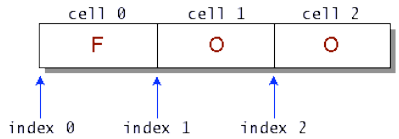
Basic Syntax

Char	Usage	Example
.	Matches any single character	.at = cat, bat, rat, 1at…
*	Matches zero or more occurrences of the single preceding character	.*at = everything that ends with at 0*123 = 123, 0123, 00123…
[...] [^...]	Matches any single character of the ones contained Matches any single character except for the ones contained	[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
...	…	

Matches

- Input string consumed from left to right
- Match ranges: inclusive of the beginning index and exclusive of the end index

- Example:



Current REGEX is: foo
Current INPUT is: foofoofoo
I found the text "foo" starting at index 0 and ending at index 3.
I found the text "foo" starting at index 3 and ending at index 6.
I found the text "foo" starting at index 6 and ending at index 9.

Character Classes

[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z, or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z] (subtraction)

Predefined Character Classes

.	Any character (may or may not match line terminators)
\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]

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 reads in 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. *(opposite of greedy)*
- **Possessive**: try to match only once on the whole input stream.

Example

- **Greedy**:

Current REGEX is: .*foo
Current INPUT is: xfooxxxxxxfoo
I found the text "xfooxxxxxxfoo" starting at index 0 and ending at index 13.

Greedy	Reluctant	Possessive	Meaning
X?	X??	X?+	X, once or not at all
X*	X*?	X*+	X, zero or more times

- **Reluctant**:

Current REGEX is: .*?foo
Current INPUT is: xfooxxxxxxfoo
I found the text "xfoo" starting at index 0 and ending at index 4.
I found the text "xxxxxxfoo" starting at index 4 and ending at index 13.

- **Possessive**

Current REGEX is: .*+foo
Current INPUT is: xfooxxxxxxfoo
No match found.

Groups

- With parentheses, we can create groups to apply quantifiers to several characters: "(abc)+"
  - Treat multiple characters as a unit
- Also useful for parsing results (see last slide)
- Groups are numbered by counting their opening parentheses from left to right
- Example: groups in "((A)(B(C)))"
  1. ((A)(B(C)))
  2. (A)
  3. (B(C))
  4. (C)

Boundary matchers

Search at particular location in the string

^	The beginning of a line
$	The end of a line
\b	A word boundary
\B	A non-word boundary
\A	The beginning of the input
\G	The end of the previous match
\Z	The end of the input but for the final terminator, if any
\z	The end of the input

Examples

```
Current REGEX is: ^dog$ // beginning line, end line
Current INPUT is: dog
I found the text "dog" starting at index 0 and ending at
index 3.

Current REGEX is: ^dog$
Current INPUT is:       dog
No match found.

Current REGEX is: \s*dog$   // white spaces
Current INPUT is:             dog
I found the text "            dog" starting at index 0 and
ending at index 15.

Current REGEX is: ^dog\w*  // word char.
Current INPUT is: dogblahblah
I found the text "dogblahblah" starting at index 0 and
ending at index 11.
```

RegExps in Java

- Two important classes:
  - **java.util.regex.Pattern** -- a compiled representation of a regular expression
  - **java.util.regex.Matcher** -- an engine that performs match operations by interpreting a Pattern
- Example

```
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches();
```

- ! To produce a slash in a Java String: "//"

Simple Example

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

public final class MatcherTest {

    private static final String REGEX = "\\bdog\\b";
    private static final String INPUT = "dog dog dog doggie dogg";

    public static void main(String[] argv) {
       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());
       }
    }
}
```

More complex Example

```
import java.util.regex.*;
public class RegEx{
  public static void main( String args[] ){
    String amounts = "$1.57 $316.15 $19.30 $0.30 $0.00 $41.10 $5.1 $.5";
      Pattern strMatch = Pattern.compile( "\\$(\\d+)\\.(\\d\\d)" );
      Matcher m = strMatch.matcher( amounts );
      while ( m.find() ){
       System.out.println( "$" + ( Integer.parseInt( m.group(1) ) + 5 )
                   + "." + m.group(2) );
      }
   }
}
```

=> Adds 5$ to every amount except the last two


```
//Checks for email addresses starting with inappropriate symbols like dots or @ signs.
Pattern p = Pattern.compile("^\\.|^\\@");
Matcher m = p.matcher(input);
if (m.find())
    System.err.println("Email addresses don't start" + " with dots or @ signs.");
//Checks for email addresses that start with www. and prints a message if it does.
p = Pattern.compile("^www\\.");
m = p.matcher(input);
if (m.find()) {
   System.out.println("Email addresses don't start" + " with \"www.\", only web pages do.");
}
p = Pattern.compile("[^A-Za-z0-9\\.\\@_\\-~#]+");
m = p.matcher(input);
StringBuffer sb = new StringBuffer();
boolean result = m.find();
boolean deletedIllegalChars = false;

while(result) {
   deletedIllegalChars = true;
   m.appendReplacement(sb, "");
   result = m.find();
}

// Add the last segment of input to the new String
m.appendTail(sb);

input = sb.toString();

if (deletedIllegalChars) {
 System.out.println("It contained incorrect characters" + " , such as spaces or commas.");
}
```

Summary

- Regular expressions are a powerful way to search for characters in strings
- Can be used in several different programming languages (e.g. Perl)
- Generally applicable