OOP TA Session 12

String Processing

*Based on www.cis.upenn.edu/~matuszek/cit591-2002/.../**java-regex.ppt***

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

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 - you can easily find out which part

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 - 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

Some simple Patterns

abc	exactly this sequence of three letters	“abc” ”ab”
[abc]	any *one* of the letters a, b, or c	“a” “ab”
[^abc]	any *one* 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 ^)	“z” “b” “za”
[a-z]	any *one* character from a through z, inclusive	“t” “9”
[a-zA-Z0-9]	any *one* letter or digit	“P” “r” “8” “_“ “(“ “#”

Some simple Quantifiers

*	Matches zero or more occurrences	**[abc]*** **"abba"** **"abta"**
+	Matches one or more occurrences	**abc+** **"abcc"** **"" "aabc"**
?	Matches zero or one occurrences (optinal)	**[abc]?** **"" "a"** **"bcbt"**

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

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 abc or one of x,y,z at least one time.

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]

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

Example – **match entire string**

- For the regular expression:

\d+.?|abc|[xyz][abc]\s*

“90”

“abc”

“a”

“xa ”

“0”

“”

“90_”

“xab”

“90abc”

“xp”

Example – **match entire string**

- For the regular expression:

\d+.?|abc|[xyz][abc]\s*

“90”

“abc”

“a”

“xa ”

“0”

“”

“90_”

“xab”

“90abc”

“xp”

Doing it in Java, I

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("abc|[xyz]");
        Matcher m = p.matcher("abc");
    }
}
```

Doing it in Java, I

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("abc|[xyz]");
        Matcher m = p.matcher("abc");
    }
}
```

Import regex package

Doing it in Java, I

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("abc|[xyz]");
        Matcher m = p.matcher("abc");
    }
}
```

Import regex package

Compile the regex pattern

Doing it in Java, I

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("abc|[xyz]");
        Matcher m = p.matcher("abc");
    }
}
```

Import regex package

Compile the regex pattern

Create a matcher for a specific piece of text

Doing it in Java, II

- Now that we have a matcher m,

  1) m.matches() returns true if the pattern matches the entire text string, and false otherwise

Doing it in Java, II - example

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("[a-z]+");
        Matcher m = p.matcher("Now is the time");
        System.out.println(m.matches());
    }
}
```

Doing it in Java, II - example

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("[a-z]+");
        Matcher m = p.matcher("Now is the time");
        System.out.println(m.matches());
    }
}
```

False

Doing it in Java, II - example

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("[a-zA-Z]+");
        Matcher m = p.matcher("Now is the time");
        System.out.println(m.matches());
    }
}
```

Doing it in Java, II - example

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("[a-zA-Z]+");
        Matcher m = p.matcher("Now is the time");
        System.out.println(m.matches());
    }
}
```

False

Doing it in Java, II - example

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("[a-zA-Z ]+");
        Matcher m = p.matcher("Now is the time");
        System.out.println(m.matches());
    }
}
```

Doing it in Java, II - example

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("[a-zA-Z ]+");
        Matcher m = p.matcher("Now is the time");
        System.out.println(m.matches());
    }
}
```

True

Doing it in Java, II

2) m.lookingAt() returns true if the pattern matches at the beginning of the text string, and false otherwise

Doing it in Java, II - example

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("[A-Z\\d]+");
        Matcher m = p.matcher("Now is the time");
        System.out.println(m.lookingAt());
    }
}
```

Doing it in Java, II - example

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("[A-Z\\d]+");
        Matcher m = p.matcher("Now is the time");
        System.out.println(m.lookingAt());
    }
}
```

True

Doing it in Java, III

3) 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)

Doing it in Java, III

- *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

Doing it in Java, III - example

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

Doing it in Java, III - example

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

```
ow
is
the
time
```

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!
  - **"\b[a-z]+\b"** is a string with backspace characters in it
  - Solution: add another backslash: **"\\b[a-z]+\\b"**

- Note: if you *read* a backslash into a String from some stream, this does not apply

Escaping Metacharacters

- A lot of special characters are used in defining regular expressions; these are called *metacharacters*
  - parentheses (“(“,”)”,“[“,”]”,”{“,”}”)
  - stars (“*”)
  - plus signs (“+”)
  - etc.

- 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). This is a *compilation error*
  - "a*" *does* work; it’s the three-character string a, \, *

Double Backslashes - example

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("^\\w\\d\\d");
        Matcher m = p.matcher("a11");
        System.out.println(m.find());
    }
}
```

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

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
  - For example: find a list of Integers i.e., 7,6,4,3,2,8,10:

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
  - For example: find a list of Integers i.e., 7,6,4,3,2,8,10:

(\d+,)+(\d+)

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
  - For example: find a list of Integers i.e., 7,6,4,3,2,8,10:

(\d+,)+(\d+)

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
  - For example: find a list of Integers i.e., 7,6,4,3,2,8,10:

(\d+,)+(\d+)

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
  - For example: find a list of Integers i.e., 7,6,4,3,2,8,10:

(\d+,)+(\d+)

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
  - For example: find a list of Integers i.e., 7,6,4,3,2,8,10:

(\d+,)+(\d+)

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
  - For example: find a list of Integers i.e., 7,6,4,3,2,8,10:

(\d+,)+(\d+)

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
  - For example: find a list of Integers i.e., 7,6,4,3,2,8,10:

(\d+,)+(\d+)

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
  - For example: find a list of Integers i.e., 7,6,4,3,2,8,10:

(\d+,)+(\d+)

  - 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

- In regular expressions, ***parentheses*** are used for grouping, but they also capture (keep for later use) anything matched by that part of the pattern
  - For example: find a list of Integers i.e., 7,6,4,3,2,8,10:

(\d+,)+(\d+)

  - 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

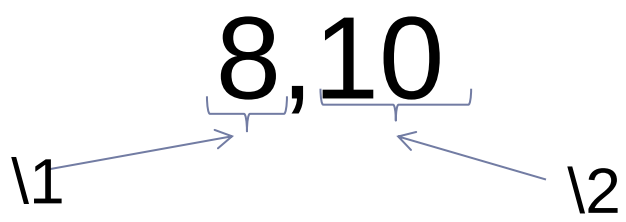
8,10

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
  - For example: find a list of Integers i.e., 7,6,4,3,2,8,10:

(\d+,)+(\d+)

  - 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 in Java

- If *m* is a Matcher that has just performed a successful match
  - ***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 also be an empty string

Capturing groups - example

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("(\\d+,)+(\\d+)");
        Matcher m = p.matcher("1,10");
        System.out.println(m.matches());
        System.out.println(m.group(1));
        System.out.println(m.group(2));
    }
}
```

Capturing groups - example

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("(\\d+,)+(\\d+)");
        Matcher m = p.matcher("1,10");
        System.out.println(m.matches());
        System.out.println(m.group(1));
        System.out.println(m.group(2));
    }
}
```

```
1,
10
```

Capturing groups - example

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("(\\d+,)+(\\d+)");
        Matcher m = p.matcher("1,3,5,6,10");
        if(m.matches()){
            System.out.println(m.group(1));
            System.out.println(m.group(2));
        }
    }
}
```

Capturing groups - example

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("(\\d+,)+(\\d+)");
        Matcher m = p.matcher("1,3,5,6,10");
        if(m.matches()){
            System.out.println(m.group(1));
            System.out.println(m.group(2));
        }
    }
}
```

```
6,
10
```

Capturing groups - backreference

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("(\\d\\d)\\1");
        Matcher m = p.matcher("1212");
        if(m.matches()){
            System.out.println(m.group(1));
        }
    }
}
```

Capturing groups - backreference

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("(\\d\\d)\\1");
        Matcher m = p.matcher("1212");
        if(m.matches()){
            System.out.println(m.group(1));
        }
    }
}
```

Try to find again what \1 captured

Capturing groups - backreference

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("(\\d\\d)\\1");
        Matcher m = p.matcher("1212");
        if(m.matches()){
            System.out.println(m.group(1));
        }
    }
}
```

Try to find again what \1 captured

```
12
```

Capturing groups - backreference

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("(\\d\\d)\\1");
        Matcher m = p.matcher("1212");
        if(m.matches()){
            System.out.println(m.group(1));
        }
    }
}
```

Without \1, matches would return false

Try to find again what \1 captured

12

Capturing groups - backreference

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("(\\d\\d)\\1");
        Matcher m = p.matcher("1212");
        if(m.matches()){
            System.out.println(m.group(1));
        }
    }
}
```

Without \1, matches would return false

Try to find again what \1 captured

What group(2) will return?

12

Capturing groups - backreference

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("(\\d\\d)\\1");
        Matcher m = p.matcher("1212");
        if(m.matches()){
            System.out.println(m.group(1));
        }
    }
}
```

Without \1, matches would return false

Try to find again what \1 captured

```
true
12
```

What group(2) will return?
Exception!

Example Use of Capturing Groups

- Say *word* holds a word in English
- We want to move all the consonants at the beginning of word (if any) to the end of the word (so **cat** becomes **atc**)

Example Use of Capturing Groups

- Say *word* holds a word in English

- We want to move all the consonants at the beginning of word (if any) to the end of the word (so **cat** becomes **atc**)

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("([^aeiou]*)([a-z]*)");
        Matcher m = p.matcher("clone");
        if(m.matches())
            System.out.println(m.group(2) + m.group(1));
    }
}
```

Example Use of Capturing Groups

- Say *word* holds a word in English

- We want to move all the consonants at the beginning of word (if any) to the end of the word (so **cat** becomes **atc**)

```
class ExampleClass{
    public static void main(String args[])  {
        Pattern p = Pattern.compile("([^aeiou]*)([a-z]*)");
        Matcher m = p.matcher("clone");
        if(m.matches())
            System.out.println(m.group(2) + m.group(1));
    }
}
```

```
onecl
```

Additional Methods

- If *m* is a Matcher, *p* is a Pattern then
  - m.replaceFirst(String *replacement*) returns a new String where the first substring matched by *p* has been replaced by *replacement*
    - *Note: replacement is a regular string, not a pattern string.*
  - m.replaceAll(String *replacement*) returns a new String where every substring matched by *p* has been replaced by *replacement*
  - m.reset() resets this matcher (i.e. it will start searching from the beginning of the text)
  - m.reset(*newText*) resets this matcher and gives it a new text to examine (which may be a String, StringBuffer, or CharBuffer)

So Far…

- **Regular expressions** are a very powerful tool to analyze and manipulate text
  - Basic expressions: ., *, +, [a-z], a{n,m}, w?, …
  - Recursive: *expr*, *expr1*|*expr2*, *expr1 expr2*
  - Capturing (exp1)((exp2)|(exp3))\1\2

- Patterns in java

  Pattern patt = Pattern.compile("[a-z]+");

  Matcher matcher = patt.matcher("Now is the time");

  matcher.matches(), matcher.find(), matcher.group(i)
  - white spaces, \\, …