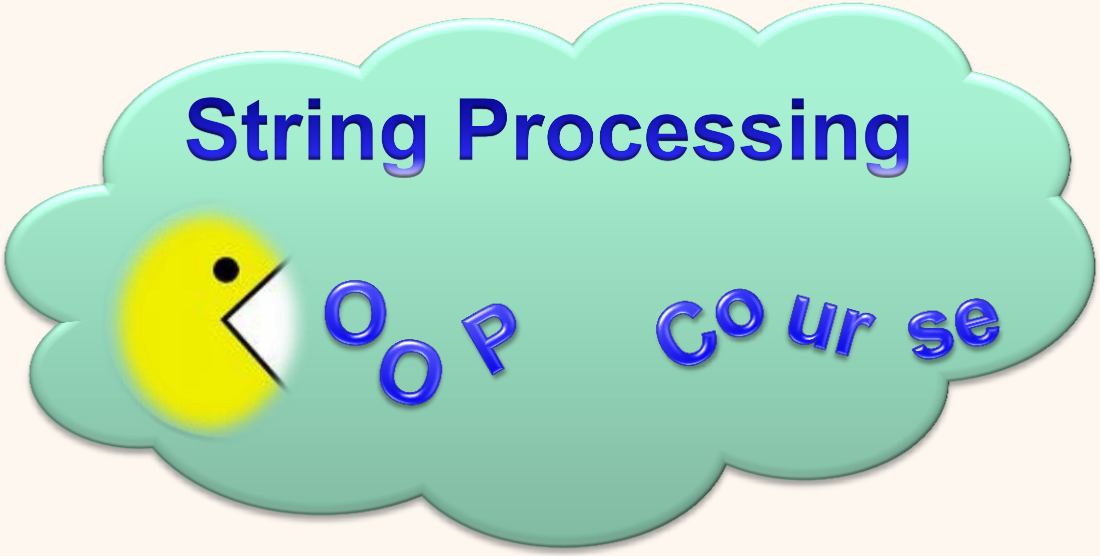
Introduction to Object Oriented Programming

(Hebrew University, CS 67125 / Spring 2014)

Lecture 11



String Processing

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

Question

```
/**
 * A method that return true iff the given string is a legal
 * date of the format dd/MM/yyyy
 */
boolean isDate(String s) {
    ???
}
```

- 01/05/2012
- 15/15/1214
- Hello
- 1a/02/2011
- 111/01/2012

Regular Expressions

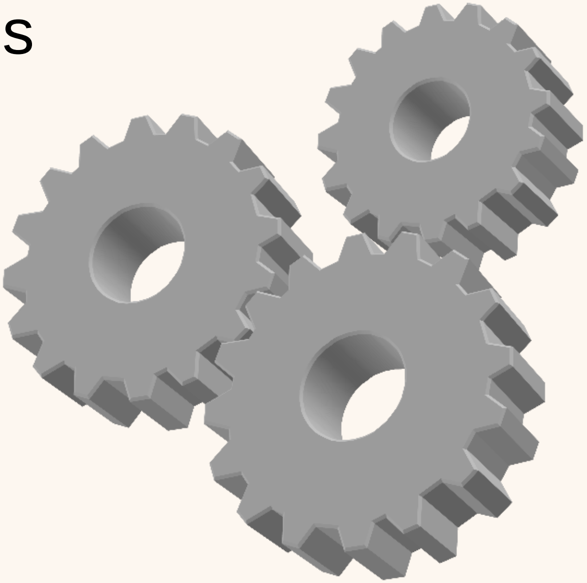
- A regular expression is a kind of **pattern** that can be applied to text (*Strings*, in Java)

- A regular expression either **matches the text** (or part of the text), or it **fails** to match it
  - **Which part** of the text matches?
  - **Which parts** of the regular expression match **which parts** of the matched text?
  - Can we **perform substitutions** on the text?

- Regular expressions are an extremely useful tool for manipulating text

Regular Expressions Uses

- Text search
- Filtering/mining/tagging text data (books/forums/blogs)
- Automatic generation of Web pages
- Processing database queries
- Spelling
- Summarization
- ….



How Do They Look Like?

[A-Za-z]+ (or)? .*$

^[^A]+a+[^a]+

(.*) .+ (.*)

^[ABC]+\d*$

Cats? and rats?

(\d+)_(\d+).(\d+)

([a-z]{1,4}?[A-Z1-9]

Basic Syntax

Characters

Char	Usage	Example
a,b,c,...	Regular text	**abc** *matches* abc
.	Matches any single character	**.at** *matches* cat, bat, rat, 1at…
[…]	Matches any single character of the ones contained	**[cbr]at** *matches* cat, bat, rat.
[^…]	Matches any single character except for the ones contained	- **[^bc]at** *matches* rat, sat…, but *does **not** match* bat, cat.
[a-z]	Matches any character in the range a-z Also works for A-Z, 0-9, and in the negative form (with ^)	- **[f-l]aaa** *mathces* faaa, gaaa,…,laaa - **[^a-f]aaa** *mathces* gaaa, 5aaa, &aaa, but *does **not** match* aaaa, caaa, ….
...	…	

Basic Syntax

Quantifiers

Char	Usage	Example
*****	Matches zero or more occurrences of the single preceding character	- **.*at** *matches* everything that ends with *at:* at, hat, 123_$&treat… - **<[^>]*>** *matches* <…anything…>
+	Matches one or more occurrences of the single preceding character	**0+123** *matches* 0123, 00123, 000123…

How to catch a rabbit?

Goal: detect in the text all the instances of the word “rabbit”, where each letter may be duplicated many times

Example: rrrrrrabbit, rrabbit, rabbbit, rraabbiitt, rabbbit

Regexp:

r+a+bb+i+t+

Search Types

- Consider the regular expression "[a-z]+"

- This expression 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 it on the String "Now is the time"

Search Types

"[a-z]+" , "Now is the time"

- There are *two ways* we can apply this pattern:
  - Apply it on the ***entire string***:
    - **Failure:** The string contains characters other than **lowercase letters**
  - To search the string ***sequentially*** (i.e. search for a substring which matches the given pattern):
    - **Success:** match ow
    - If applied repeatedly, it will find is, then the, then time, then **fail**

How to catch a rabbit?

Goal: detect in the text all the instances of words ending with “rabbit”, where each letter except the last may be capitalized

Example: MyRabbit, HISRABBIt, RabBit, gOOdrABBIt

Regexp:

[A-Za-z]*[Rr][Aa][Bb][Bb][Ii]t

Recursive Structure

- The structure of regular expressions is **recursive**
  - The vertical bar, |, is used to separate alternatives
    - For example, the pattern abc|xyz will match either abc or xyz
  - 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 ([A-Za-z]+) immediately followed by one digit ([0-9])
  - Both these operators can be used recursively
- Parentheses may be used in building the expressions
  - For example, (a|b)c will search either a or b, followed by c

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]

Notice the space. Spaces are significant in regular expressions!

How to catch a rabbit?

Goal: detect in the text all the instances of “1/one rabbit” or “X rabbits” where X is a number

Example: one rabbit, 256 rabbits

Regexp:

```
(1|one) rabbit|\d+ rabbits
```

Boundary Matchers

^	The beginning of a line
$	The end of a line
\b	A word boundary
\B	A non-word boundary

Example:

REGEX is: \brabbit\b

INPUT is: Some rabbits plays in the yard.

No match found.

Quantifiers

- Assume *X* represents some pattern
  - *X* may be a **single character**, a **range of characters** or an **expression wrapped by parentheses**

  - *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
  - *X*? *X* is optional (it occurs once or not at all) [⇔ X{0,1}]
  - *X** *X* occurs zero or more times [⇔ X{0,}]
  - *X*+ *X* occurs one or more times [⇔ X{1,}]

- These are *postfix* operators (comeing ***after*** the operand)

How to catch a rabbit?

Goal: detect in the text all the instances of the word rabbit, where each letter is duplicated at most one time and there is a possible single “s” at the end

Example: rrabbbbit, rabbbits

Regexp:

r{1,2}a{1,2}b{2,4}i{1,2}t{1,2}s?

Types of Quantifiers

- A **greedy quantifier** will match **as much as it can**, and **back off** if it needs to

  X? *X** *X*+ *X*{*n*} *X*{*n*,} *X*{*n*,*m*}

- 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*}+

Examples
Greedy Quantifier

will match as much as it can, and back off if it needs to

- Current INPUT is: xfooxxxxxxfoo
  - Current REGEX is: .*foo
    - Searching for .*
    - Searching for foo ☹
    - Going back one letter
    - Searching for foo ☹
    - Going back another letter
    - ….
    - Found foo! ☺

xfooxxxxxxfoo

- <span>"**xfooxxxxxxfoo**"</span> found starting at index [0,13]

Examples

Reluctant Quantifier

will match ***as little as possible****, then take more if it needs to*

- Current INPUT is: xfooxxxxxxfoo
  - Current REGEX is: .*?foo
    - Searching for .*
    - Searching for foo ☹
    - Grabbing another character
    - Searching for foo ☺
    - Searching for .* again
    - Searching for foo again ☹
    - Grabbing another character
    - ….
    - Found foo! ☺

xfooxxxxxxfoo

↑

- **"xfoo"** found starting at index [0,4]
- **"xxxxxxfoo"** found starting at index [4,13]

Examples

Possessive Quantifier

*will match as much as it can, and **never let go***

- Current INPUT is: xfooxxxxxxfoo
  - Current REGEX is: .*+foo
    - Searching for .*
    - Searching for foo ☹

xfooxxxxxxfoo

- No match found

Backreferences

- Say we want to look for a letter that appears twice in a row
  - aa, bb, CC, ...

- How about [a-zA-Z]{2}?
  - This pattern looks for two letters, **not necessarily the same**
  - ab, BC, Aa, …

- To look for the **same letter** twice, we need to use capturing groups

Capturing Groups

- Parentheses ***capture*** anything matched by that part of the pattern
  - Example: **(**[a-zA-Z]***)(**[0-9]***)**
  - If match succeeds, **\1** holds the matched **letters** and **\2** holds the matched **digits**
  - We can access these parts later in the expression

- ([a-zA-Z])**\1** will match a double letter, such as le**tt**er

How to catch a rabbit?

Goal: Detect a ‘rabbit’ with the same number before and after him

Example: 1rabbit1,123rabbit123

Regexp:

(\d+)rabbit\1

Capturing Groups

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

$(_1\ (_2\ A\ )\ (_3\ B\ (_4\ C\ )\ )\ )$

\1 = ((A)(B(C))),   \2 = (A),   \3 = (B(C)),   \4 = (C)

Spaces

- There is only one thing to be said about white 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, it means you want to match a **space** in the text string

- It’s a *really bad idea* to put spaces in an expression just to make it look better

More Than One Way To Do It

- There is more than one way to write (almost) all regular expressions
  - Especially complex ones

- This applies to mere syntax changes
  - [a-c]+ ⇔ [abc]+ ⇔ [abc][abc]* ⇔ …

- But also to equivalence between **different** expressions
  - [ab]{2} ⇔ aa|ab|ba|bb

Writing Good Regexps

- Writing simple expressions
  - [ab]{1,2} is better than a|b|aa|ab|ba|bb

- Search for the minimal expression you need
  - If you are only looking for numbers, use [0-9]* (or \d*), not .*

- Anchors are good!
  - When looking for text at the beginning/end of the line, use ^/$

- Use possessive quantifiers whenever it is possible

- More in *http://www.tideway.com/confluence//display/Configipedia/Writing+Efficient+Regex*

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
- They form a miniature programming language
  - It’s a different kind of programming language than Java, and requires you to learn new thought patterns
- Despite all this, regular expressions bring much a lot of **power** and **convenience** to String manipulation

Doing It in Java – Preparation

- First, you must ***compile*** the pattern

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

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

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

Pattern and Matcher

- Comments:
  - 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
  - A Matcher object contains information about *both* the **pattern** to use *and* the **text** to which it will be applied
  - One compiled Pattern can be used for many Matchers

Doing It in Java – Matching

- Now that we have a Pattern *p* and a Matcher *m*
  - m.matches() returns **true** ⇔ *p* matches the **entire** text string
  - m.find() returns **true** ⇔ *p* matches **any part** of the text string
    - If **called again**, m.find() will start searching from where the last match was found
    - m.find() returns **true** for as many matches as there are in the string; after that, it returns **false**

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*
  - These values correspond to what most String methods require
    - For example, str.substring(matcher.start(), matcher.end()) returns the exact matched substring

- 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

A Complete Example

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

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

Output: ow_is_the_time_

Java Shortcuts for Regexps for String Class

Return value	*Method name and description*
boolean	*matches(String regex)* Tells whether or not this string matches the given regular expression
String []	*split(String regex)* splits this string around matches of the given regular expression
String	*replaceAll(String regex, String replacement)* Replaces each substring of this string that matches the given regular expression with the given replacement

+ Convenient

- Inefficient & inflexible

Question

```
/**
 * A method that return true iff the given string is a legal
 * date of the format dd/MM/yyyy
 */
boolean isDate(String s) {
    ???
}
```

Question

```
/**
 * A method that return true iff the given string is a legal
 * date of the format dd/MM/yyyy
 */
boolean isDate(String s) {
    return s.matches(“\\d{2}/\\d{2}/\\d{4}”);
}
```

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, expr1expr2*
  - 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()
```