Contents

- Overview of Regular Expressions
- Regex Syntax
- Java Regex API
- Regex Operations
- String Class and Regex
- Scanner
What are regular expressions?
- powerful string pattern-matching tools
- commonly used for extracting structure from text
- describe (or express) a pattern of characters that may or may not be contained within a target character sequence

Why use regex?
- eliminates complicated brute-force string parsing code
  - often done otherwise with literal tests and C-like character array traversal
- can handle greater variety of cases without branching
- simplifies code
- improves clarity, once you get used to regex meta-syntax
Regex Syntax

- Structure of a regular expression
  - zero or more branches
  - each branch has 1 or more pieces
  - each piece has an atom with optional quantifier

matches:
123-AB
9876123
non-matches:
2468ABZ
12-BAC
321-Z2
201-sm

Atoms

- Describe one or more characters
  - character literal – a abc (a | b | c)
  - meta-characters – . \ ? * + | { } ( ) [ ]
    - have special meaning in a regular expression
    - must be escaped with ‘\’ to be treated as literal, e.g. \\
  - character classes
    - define a class of multiple characters
  - predefined character classes
    - define common character classes
Character Classes

- Character class expression
  - specifies a single choice among set of characters
  - expression is enclosed by square brackets \[expr\]
  - represents exactly one character of possible choices
  - may include escaped meta-characters
  - use \- to specify range (boundaries inclusive)
  - use \^ to negate expression
  - examples
    - \[a-zA-Z0-9\] matches a, N, 0
    - \[-0-9\] matches -, 1, 2
    - \[\./\d\] matches ., 1, 2
    - \[^\s\] matches non-whitespace character
  - example regex expression using character classes
    - \[_a-zA-Z][_a-zA-Z0-9]*\] matches a Java identifier
Character Classes

- Character class subtraction
  - a range subset may be subtracted from a character class
    - subtraction must be itself a character class
      - \[a-z&[^aeiou]]\] matches lowercase consonants

- Predefined character classes
  - more convenient to use
  - may be used within a character class you define
    - [\d] from previous example
  - common ones
    - . (dot)– any character except carriage return & newline
    - \d – decimal digit (or \D for non-decimal digit)
      - equivalent character class: [0–9]
    - \s – whitespace (or \S for non-whitespace)
    - \w – word character (or \W for non-word character)
      - equivalent character class: [a-zA-Z0-9]
Boundary Matchers

- A special class of match specifiers
  - most common
    - ^ – beginning of line
    - $ – end of line
  - others
    - \b – word boundary
    - \B – non-word boundary
    - \A – beginning of input
    - \G – end of previous match
    - \z – end of input
    - \Z – end of input before final terminator
Quantifiers

Specify how often an atom appears in a matching string
- applies to preceding character or class

[none] exactly once
? zero or one times
* zero or more times
+ one or more times
{n} exactly n times
{n,} n or more times
{n,m} n to m times

examples
(a|b)c ac,bc (ab)?c abc, c (ab)*c abc, ababc, c
(ab)+c abc, ababc (ab){2}c ababc
(ab){2,}c ababc, abababababababc
(ab){2,4}c ababc, abababc, ababababc

use parentheses to quantify complex atoms
Capturing Groups

- Capturing groups can be used to capture matching substrings
  - denoted by enclosing sub-expressions in parentheses
  - may be sequenced and/or nested
  - ordered from left to right
    - numbering starts with 1 (0 denotes the entire expression)
  - example: \(( (A) (B (C) ) )\)
    - group 0: \(( (A) (B (C) ) )\)
    - group 1: \(( (A) (B (C) ) )\)
    - group 2: \((A)\)
    - group 3: \((B (C) )\)
    - group 4: \((C)\)
- matching engine will maintain back references to captured groups
  - more on this later
Non-Capturing Groups

- Groups that do not capture (save) matched text nor count towards group total
  - matching engine does not maintain back references
- Frequently used to group sub-expressions for quantification
  - such as matching frequency of occurrence with *, ?, +, etc
- Denoted as with capturing groups but with ?:
  - after opening parenthesis
    - capturing group: (regex)
    - non-capturing group: (? : regex)
Non-Capturing Groups

- In example below, we don't need to save first group
  - only used to test existence of package name
  - included trailing dot character to discard

- Capturing
  
  ```
  ((.*)\.)?([^\.]*)
  group 1:  ((.*)\.)
group 2:  (.*)
group 3:  ([^\.]*)
  ```

- Non-capturing
  
  ```
  (?: (.*\.)?([^\.]*)
  group 1:  (.*
  group 2:  ([^\.]*)
  ```

In example below, we don't need to save first group
- only used to test existence of package name
- included trailing dot character to discard
Examples

- match leading/trailing whitespace
  \s*.*\s*\$

- match enclosing parentheses
  ^\(\[^\(\)\]*\)\$  

- match quoted string, capture string
  ^"(.*)"$  

- match Java identifier
  [\w&&[^\d]] [\w]*

- match Zip+4 code
  [\d]{5}-[\d]{4}

- match phone number: (xxx) xxx-xxxx or xxx-xxx-xxxx
  (?:(?:\([^\d]{3}\)\s?)|(?:\[[\d]{3}\-)) [\d]{3}-[\d]{4}
A More Complex Example

- Regex to match SQL type definitions
  - e.g. Char, Varchar(6), Number(8,2)
    
    `([^\(\]+)((\(\d+\),((\d+))?)?)`

  - group 1: `([^\(\]+)`
    - matches type
  - group 2: `((\(\d+\),((\d+))?)?)`
    - tests existence of type qualifier
  - group 3: `((\d+))`
    - matches first qualifier arg (length digits)
  - group 4: `((\d+))`
    - tests existence of 2\(^{nd}\) qualifier arg (precision digits)
  - group 5:
    - matches second qualifier arg
  - with non-capturing groups
    - `(?::([^\(\]+))\(?::((\(\d+\) (?::,((\d+))?)?)??`
Java Regex API

- Introduced with J2SE 1.4
  - for J2SE 1.3 and earlier, (incompatible) third party APIs are available
- Based on Perl regular expressions
- Defined by two classes and one exception in, representing the abstraction of pattern matching
  - in package: java.util.regex
  - Pattern encapsulates a compiled regular expression
  - Matcher is a matching engine that operates by interpreting regex patterns on character sequences
  - PatternSyntaxException for syntax errors in regex patterns
Java Regex API

- Adds support for basic *regex* operations to `java.lang.String`
  - pattern matching, replacement, and splitting strings
- Also utilizes new `java.lang.CharSequence` interface for abstracting readable strings
- The javadocs for `java.util.Pattern` provide details for support of regular expression syntax
Special Java Considerations

- **Double escaping regex escapes**
  - regex expression string literals have to be escaped to compile
    - `\s*` to `\\s*`, `\` to `\\\`, etc.
  - *RegexTester Pro* Eclipse plugin does this for you
    - was free, but still cheap at €5.00 (via PayPal)

- **Escaping back-references in replacement text**
  - i.e. `\` and `$` in replacement text treated as back references
  - solved by J2SE 5 `Matcher.quoteReplacement()` method

- **Use unit tests for testing regular expressions**
  - create test cases to validate regular expression
  - when regex operation fails for input expected to match
    - create a new test to expose failure
    - change regex to support input
    - execute test suite to validate old and new input cases
Regex Operations

- Matching and Capturing
  - test a string against some pattern, possibly capturing a substring
  - result is true/false, or a captured substring

- Replacement
  - test a string against some pattern
  - replace matches with some other string
  - or keep matched sub-string(s) and discard the rest
    - use capturing groups

- Splitting
  - find a recurring pattern in a string and split the string into tokens
  - matched substrings are delimiter and discarded

- Translation (complex replacement)
  - i.e. Perl: \$string =~ tr/originalexample/newtext/;
Pattern Class

- Represents a compiled regular expression
  - Serializable so expressions can be persisted
- Javadopts explain regex support
- Factory methods
  - create the compiled Pattern instance
  - create matching engine
    - for matching strings against the compiled regex
- Class method highlights

```java
static Pattern compile(String regex)
Matcher matcher(CharSequence input)
static boolean matches(String regex, CharSequence input)
String[] split(CharSequence input)
```
Matcher Class

- The regular expression matching engine
  - performs operations on input strings using a regex pattern
  - created with the `Pattern.matcher(CharSequence)` method

- Class method highlights
  - matching
    - `boolean matches()` – attempts to match entire sequence
    - `boolean find()` – attempts to match next subsequence
    - `boolean lookingAt()` – attempts to match sequence from beginning
  - capturing
    - `String group(int group)` – returns matched capturing group
    - `int groupCount()` – returns number of capturing groups in pattern
More Matcher

- Highlights (cont’d)
  - replacement
    String replaceFirst(String replacement) – replaces first matched subsequence with replacement
    String replaceAll(String replacement) – replaces all matched subsequences with replacement
  - advanced replacement (used together in a loop with find())
    appendReplacement(StringBuffer sb, String replacement)
    appendTail(StringBuffer sb)

- Numerous other methods
  - for more complex matching operations
  - see the javadocs
Matching

- The simplest regex operation

```java
String input = ...
String regex = ...
Pattern p = Pattern.compile(regex);
Matcher m = p.matcher(input);
boolean result = m.matches();
```

or

```java
result = Pattern.matches(regex, input);
```
Capturing Groups

- Captured groups are extracted using a Matcher method
  - `String group([int group])`
    - returns `null` if match successful, but specified group isn't
    - `IllegalStateException` if no match has been attempted
    - `IndexOutOfBoundsException` if group is not specified in pattern
Capturing Group Example

Extract package and class names from qualified class name

```java
public String getTypenameComponent(String classname, int group) {
    // regex is: (?:(.*)\.)?([^\.]*)
    Pattern p = Pattern.compile("(?:(.*)\.)?([^\.]*)");
    Matcher m = p.matcher(classname);
    return m.matches() ? m.group(group) : null;
}

//...
String typeName = "com.ociweb.regex.CapturingExample";
String packageName = getTypenameComponent(typeName, 1);
String className = getTypenameComponent(typeName, 2);
// packageName is "com.ociweb.regex",
// className is "CapturingExample"
```
String sqlType = "NUMBER(10,2)";
String type = getColumnDatatypeComponent(sqlType, 1);
String length = getColumnDatatypeComponent(sqlType, 2);
String precision = getColumnDatatypeComponent(sqlType, 3);

String getColumnDatatypeComponent(String dataType, int group) {
    // (?:\[^\(]+)(?:\((\d+)(?:,(\d+))?)?
    final String regex = "(?:\[^\(]+)(?:\((\d+)(?:,(\d+))?)")?";
    return getCapturedGroup(dataType.replaceAll("\s*",""),
        regex, group);
}

String getCapturedGroup(String value, String pattern, int group) {
    Matcher m = Pattern.compile(pattern).matcher(value);
    if (m.matches() && (group >= 0) && (group <= m.groupCount())) {
        return m.group(group);
    } else return null;
}
Replacement

Pattern p = Pattern.compile("cat");
Matcher m = p.matcher("one cat two cats in the yard");

StringBuffer sb = new StringBuffer();
while (m.find()) {
    m.appendReplacement(sb, "dog");
}
m.appendTail(sb);
System.out.println(sb.toString());
// one dog two dogs in the yard

// easier
String result = m.replaceAll("dog");
// easiest, a one liner
// see String class topic...
Splitting Strings

- Use the split method of Pattern
  
  \[
  \text{String}[] \text{ split(CharSequence input[, int limit])}
  \]
  
  - splits the input string around a regex pattern
  - limit is a result threshold
    - use \(-1\) (any int < 0) for no limit
    - use \(0\) for no limit, but to discard trailing empty strings (default)
  - preserves empty strings
  - example

  ```java
  String record = "Wette , Dean, ,OCI,Engineer";
  Pattern p = Pattern.compile("\s*,\s*");
  String[] fields = p.split(record);
  // { "Wette", "Dean", ",", "OCI", "Engineer" } 
  ```
The `String` class adds regex features

- `boolean matches(String regex)`
  - equivalent to
    ```java
    boolean Pattern.matches(String regex, CharSequence input)
    ```

- example

```java
String person = "George W. Bush, President of the United States";
boolean isPrez = person.matches("(\S+\s+)+President(\s+\S+)+");
// isPrez is true
```
String Class (cont’d)

- String replaceFirst(String regex,  
  String replacement)

  equivalent to

  ```java
  String str = ...
  Pattern.compile(regex).matcher(str)
      .replaceFirst(replacement)
  ```

- example

  ```java
  String removeEnclosing(String str, String enclosing) {
      String regex = "^" + enclosing + 
          "(.*)" + enclosing + "$";
      return str.replaceFirst(regex, "$1";
  }
  ```

  String str = "(example of parenthesized text)"
  str = removeEnclosing(str, "[\[\(\)]");
  // example of parenthesized text
String replaceAll(String regex, 
                String replacement)

  equivalent to

        String str = ...
        Pattern.compile(regex).matcher(str) 
            .replaceAll(replacement)

  example

    public static String removeWhitespace(String str) {
        return str.replaceAll("\\s", "");
    }

    String str = "   Dean \t \t \n \t Wette ";
    str = removeWhitespace(str);
    // DeanWette
Replacement Gotcha

- Applies to Matcher and String replacement methods
- Treats 2 characters in replacement string specially
  - '$' treated as a back reference to a captured group
    - i.e. see String.replaceFirst() prior example
  - '\ ' treated as an escape in replacement string
- Need to be escaped for literal treatment
- Solution in J2SE 5
  - use Matcher quoteReplacement method on replacement first
    - creates a result string where '\ ' and '$' have no special meaning
  static String quoteReplacement(String s)
- In J2SE 1.4...
Replacement Gotcha

This is similar to quoteReplacement implementation

```java
String translateToEscapedRegex(String str) {
    StringBuffer buf = new StringBuffer(str.length());

    for (int i = 0, len = str.length(); i < len; ++i) {
        char c = str.charAt(i);
        switch (c) {
            case '\':
            case '$':
                buf.append("\\");
                break;
            default:
                buf.append(c);
        }
    }
    return buf.toString();
}
```
String Class (cont’d)

- `String[] split(String regex[, int limit])`
  - equivalent to

    `String[] Pattern.split(CharSequence input[, int limit])`

- example

```java
String str = "This is a sentence. This is another one.";

String[] sentences = str.split("\s*\.\s*");
// { "This is a sentence", "This is another one" }

String[] words = str.split("(\s+)|(\s*\.\s*)");
// { "This", "is", "a", "sentence", "This", "is", "another", "one" }

String[] characters = str.split(""");
// { ",", "T", "h", "i", "s", ",", ",", ",", ",", ",", "," }
StringTokenizer vs. Split

- `java.util.StringTokenizer`
  - consumes empty tokens, i.e. an empty string is discarded
  - delimiter is a `String` object
  - must use iteration to extract tokens

- `split()` methods in `Pattern` and `String`
  - retain empty strings
  - delimiter is any regular expression
  - provide array for direct access to tokens
StringTokenizer vs. Split

String example = "one, two, three,, five,";

List tokens = new ArrayList();
StringTokenizer st = new StringTokenizer(example, ",");
while (st.hasMoreTokens()) {
    tokens.add(st.nextToken().trim());
}
String[] tokenArray =
    (String[])tokens.toArray(new String[0]);
// { "one", "two", "three", "five" }

tokenArray = example.split("\s*,\s*", -1);
// { "one", "two", "three", ",", "five", "," }
Scanner

- New in J2SE 5
- Text scanner that parses primitives and strings using regex
- Constructors overloaded for
  - File, InputStream, Reader, NIO, etc
- Numerous methods for
  - extracting primitives from stream
  - matching and capturing subsequences
- Can simplify parsing vs. using Java regex package directly
Formatter Example

static String dataInput =
"0.020000 -0.001234 -.5931E-03 0.014454 -4.00200 -2.23541 0.045117 \n" +
"222.962 0.600000 30000.0 1.00000 4.82400 0.000 -0.657461 \n" +
"-1.27151 -0.326195 0.787285 -0.451386 -0.486815 -1.27151 \n" +
"-0.326195 -0.163894 0.286443 1.85980 -0.170646 0.000 \n" +
"0.554936 0.505573 -2.31165 -0.170646 0.000 \n" +
"0.505573 -2.31165 -0.414285 -2.53640 4.54728 2.01358 \n" +
"4.85477 20.0000 20.0000 20.0000 20.0000 \n";

public static void main(String[] args) {
    final String regex = ".{10}";
    String str = null;
    Scanner s = new Scanner(dataInput);
    for (int i = 1; (str = s.findInLine(regex)) != null; ++i) {
        if (!"".equals(str.trim())) {
            System.out.printf("%1$2d ", i);
            System.out.print(" "+str+" \\
" = ");
            System.out.println(Double.valueOf(str));
        }
    }
    s.close();
}
Thank You for Attending

Java Regular Expressions – Q & A

Dean Wette
Principal Software Engineer, Instructor
Object Computing, Inc.
St. Louis, MO