Regular Expressions

Advanced Programming

Regular expression

- Regular expressions are a way to describe a set of strings based on common characteristics shared by each string in the set
- They can be used to search, edit, or manipulate text and data.

Source: Oracle documentation

What we need

- One must learn a specific syntax to create regular expressions in addition to the Java syntax
- Then one must use the **Pattern** and **Match** classes of **regex** package to store and manipulate the expressions

What is a regular expression?

Examples:

Date in format yyyy-MM-dd

(19|20)\d\d([- /.])(0[1-9]|1[012])\2(0[1-9]|[12][0-9]|3[01])

Roman Number Regexp

^(?i:(?=[MDCLXVI])((M{0,3})((C[DM])|(D?C{0,3}))?((X[LC])|(L?XX{0,2})|L)?((I[VX])|(V?(II{0,2}))|V)?))$

more examples: http://myregexp.com/examples.html

Exercise

- Apply the reg expression for the date YYYY-MM-DD

(19|20)\d\d([- /.])(0[1-9]|1[012])\2(0[1-9]|[12][0-9]|3[01])

- to the text

1900-01-01 2007/08/13 1900.01.01 1900 01 01
1900-01.01 1900 13 01 1900 02 31

What is a regular expression?

- A regular expression is a string of characters that describes a character sequence
- It is comprised of
  - characters
  - sets of characters
  - wildcards
  - quantifiers

full list at http://www.cheatography.com/davechild/cheat-sheets/regular-expressions/

or at http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

Characters

- Characters are matched **as-is**. A pattern consisting in “xy” applied to a text “xy” returns xy or a pattern “Java” applied to text “Javaj” returns “Java”
- Characters like carriage return, new line are specified with the backslash escape sequence
  - \n matches new line
  - \b matches word start
  - …

Set of characters

- **Identified by square brackets []** – matches a single character inside the square

  [ecl] matches “c” and “l” in the text “cloud”

  - note that the match is on a single character of the word per time when a match is found, it restarts on the next character of the text

  [123] matches “3” and “1” in the text “ab31”
- Using in addition ^ the expression matches the characters except the one specified:

  [^ecl] matches “o”, “u”, and “d” in “cloud”

  [1-9] identifies a range of digits [a-z] a range of letters and [A-Z] range of capitalized letters

wildcard “.”

- it matches any character including spaces

Number of matches

- + 1 or more – after the character

x+ matches in text “xxx” x, xx, and xxx

- * 0 or more

x* matches in text “xxx” none, “x”, “xx”, and “xxx”

- ? 0 or 1

x? matches in text “xxx” none and “x”

{n,m} matches between n and m (greedy = the longest)

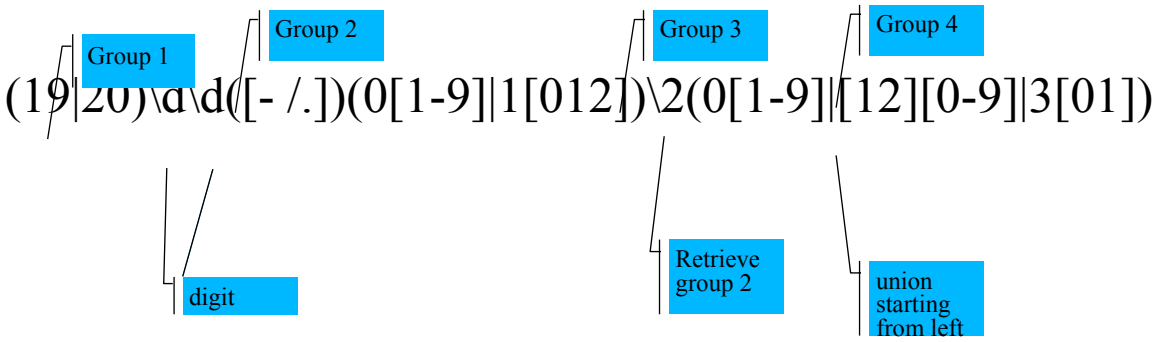
Groups

- Capturing groups are numbered by counting their opening parentheses from left to right.
- the expression ((A)(B(C))) have four groups:
- 1 ((A)(B(C)))
- 2 (A)
- 3 (B(C))
- 4 (C)

Group number

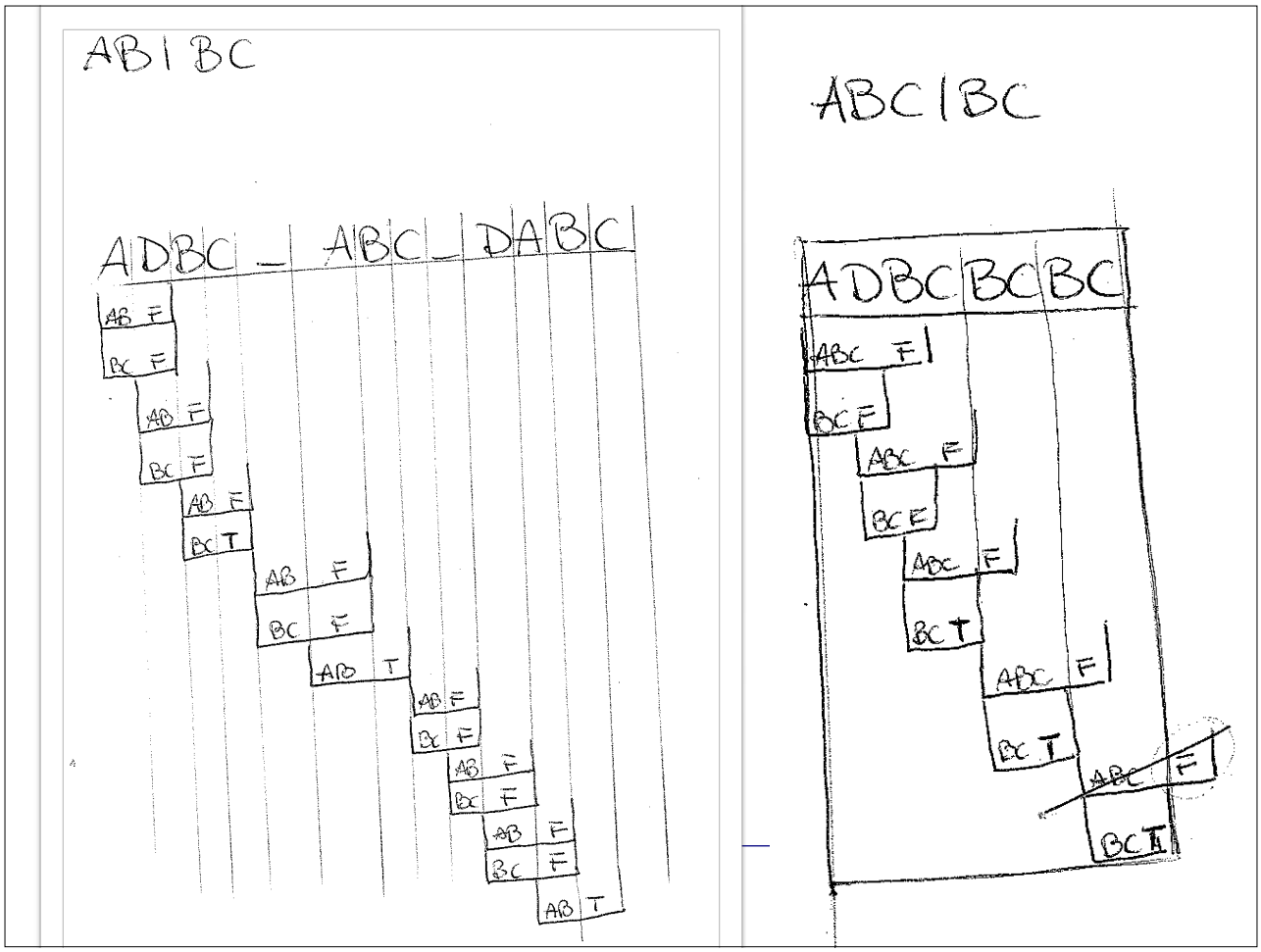
- Group zero always stands for the entire expression.
- During a match, each subsequence of the input sequence that matches such a group is saved.
- The captured subsequence may be used later in the expression, via a back reference (\i), and may also be retrieved from the matcher once the match operation is complete.

Example



Exercises

- match AB|BC in text “ADBC ABC DABC”
- match ABC|BC in the text “ADBCBCBC”



Special characters in Java

- A character preceded by a backslash (\) is an **escape sequence** and has special meaning to the Java compiler
- Thus to match the word (Hello) we need to use
- \\(Hello)\\ and not \(Hello)\ as \(and \) have been already reserved for the Java syntax

java.util.regex package

- Most relevant classes **Pattern** and **Matcher**
- Pattern accepts the regular expression
- Matcher matches the regular expression on a text

Pattern

- A compiled representation of a regular expression
- It is final
- It implements the interface Serializable
- It has only static fields
- It has the method compile() that transforms a regular expression into a pattern that can be used by the Matcher. It returns a Pattern object

How to match – three steps

- First way:
- create an object of type Pattern: compile() from Pattern
- create an object of type Matcher: matcher() from Pattern
- match the pattern on the text: matches() from Matcher

compile()

public static Pattern compile(String regex)

- PatternSyntaxException - If the expression's syntax is invalid

- Compile the regular expression: e.g., verifies that the regular expression can be read in Java

Example

- Regular expression that finds a sequence of char with the longest sequence of “a” and “b” in “aaaaaab”
- Create a pattern:

```
Pattern p = Pattern.compile("a*b");
```


matcher()

public Matcher matcher(CharSequence input)

- Creates a **matcher** that matches the sequence of characters “input”
- CharSequence is an interface implemented for example by String, StringBuffer, StringBuilder
  - we can pass a string

Summarising

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


Other two ways to match

- Directly from Pattern or
- By invoking the method matches() of String on a string

Directly from Pattern

```
public static boolean matches(String regex,
CharSequence input);

boolean b = Pattern.matches("a*b", "aaaaab"););
```

- This method automatically compiles the pattern and matches the expression with the input string
- It does **not allow** the compiled **pattern to be reused** thus is not efficient in repeated matching

matches() method of String

boolean **matches**(String pattern)

- If the invoking String object matches the regular expression in the String pattern it returns TRUE.
- It does not create a Pattern or a Matcher object though. Thus it is not that efficient in repeated matching

```
String myString = "aaaaab";
boolean b = myString.matches("a*b")
```

split()

- It splits a text by tokens according to a regular expression


```
Pattern pat = Pattern.compile("[ ,.!]");
String[] str = pat.split("one two, alpha9 12!done.");
for (int i=0; i<str.length;i++)
    System.out.println("Next token: " + str[i]);
}
```

Output

```
Next token: one
Next token: two
Next token:
Next token: alpha9
Next token: 12
Next token: done
```

Matcher

- The class has no constructor (i.e., it has only the default one in which we cannot pass any parameter) and it is created with the matcher() method of Pattern

Matcher

- The resulting pattern from compile() is used to create a Matcher object that can match arbitrary character sequences against the regular expression
- All the services to perform a match resides in the matcher object,
  - so many matchers can share the same pattern
  - matches can be repeated with different strings

matches()

- it matches

Output

aaaaab

Pattern p = Pattern.compile("a*b");

Matcher m = p.matcher("aaaaab");

boolean b = m.matches();


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

boolean b = m.matches();
```

find()

- To determine a match of a subsequence of an input string
- It can be used to search the input sequence for repeated occurrences of the same pattern
- Use the start() method to retrieve the occurrence index of the match


```
public static void main(String[] args) {
    Pattern pat = Pattern.compile("Java");
    Matcher matmat = pat.matcher("Java 8");
    // create the first matcher
    System.out.println("Looking for Java in Java 8");
    if(mat.find()) System.out.println("subsequence found");
    else System.out.println("no match");
    Matcher mat2=pat.matcher("Java 1 2 3 Java");
    System.out.println("Looking for Java in Java 1 2 3 Java");
    while(mat2.find()) System.out.println("subsequence found at index"+mat2.start());

}
```

Output

```
Looking for Java in Java 8
subsequence found

Looking for Java in Java 1 2 3 Java
subsequence found at index0
subsequence found at index11
```

Wildcards and quantifiers

```
// one or more character
Pattern pat = Pattern.compile("W+");
Matcher mat = pat.matcher("W WW WWW");
while(mat.find()) System.out.println("Match: "+mat.group());
//the greedy syntax {n,m} : from n to m including extremes
pat = Pattern.compile("W{2,4}");
mat = pat.matcher("W WW WWW");
while(mat.find()) System.out.println("Match: "+mat.group());
```

Output

```
Match: W
Match: WW
Match: WWW

Match: WW
Match: WWW
```


replaceAll()

- to replace sequences

```
String str = "Jon Jonathan Frank Ken Todd";
Pattern pat = Pattern.compile("Jon.*? ");
// "." matches any character
Matcher mat = pat.matcher(str);
System.out.println("Original sequence: "+str);
while(mat.find()) System.out.println("Match: "+ mat.group());

str =mat.replaceAll("Eric ");
System.out.println("Replaced sequence: "+str);
```

Output

```
Original sequence: Jon Jonathan Frank Ken Todd
Match: Jon
Match: Jonathan

Replaced sequence: Eric Eric Frank Ken Todd
```

Greedy and lazy matching

- The three quantifiers (*, + and ?) taken individually are greedy:
  - They match as many characters as possible
- By appending "?" they may be made lazy or minimal by matching as few characters as possible:
  - ".*?"
  - We matched "Jon " and "Jonathan "; Jon plus zero characters and a space or Jon plus the minimal number of characters and a space

Date format

```
Pattern pat = Pattern.compile("^((19|20)\\d\\d)([-/.])(0[1-9]|
1[012])([-/.])(0[1-9]|[12][0-9]|3[01])$");
    Matcher mat = pat.matcher("2011-01-01");
    boolean b = mat.matches();
    if(b)System.out.println("Match");else System.out.println("no
match");


if(Pattern.matches("^((19|20)\\d\\d)-(0?[1-9]|1[012])-(0?[1-9]|[12]
[0-9]|3[01])$", "2011-1-1")) System.out.println("Match"); else
System.out.println("no match");
```

Find an email in a text

- email
- "[^@]+@[^\.]+\..+"
- See code examples
- Let's change it:
- "[^@]+{8}@[^\.]+\..+"
- "[^@]{2,4}@[^\.]+\..+"

Advanced Programming

THE SCANNER CLASS

Scanner

- public final class Scanner
- A simple text scanner which can parse primitive types and strings using regular expressions.
- A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace.
- The resulting tokens may then be converted into values of different types using the various next methods.

Example:

- Read a number from keyboard

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

Delimiters

- Use delimiters other than whitespace.

```
String input = "1 fish 2 fish red fish blue fish";
// “\\s” white space
Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

Output:

```
1
2
red
blue
```

With regular expressions

- to parse all four tokens at once:

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input);
s.findInLine("(\\d+) fish (\\d+) fish (\\w+) fish (\\w+)");
MatchResult result = s.match();
for (int i=1; i<=result.groupCount(); i++)
   System.out.println(result.group(i));
s.close();
```

- The default whitespace delimiter used by a scanner is as recognised by Character.isWhitespace
- The reset() method resets the value of the scanner's delimiter to the default whitespace delimiter regardless of whether it was previously changed

Same output

1

2

red

blue

Block waiting for input

- A scanning operation may block waiting for input
- The next() and hasNext() methods and their primitive-type companion methods (such as nextInt() and hasNextInt()) first skip any input that matches the delimiter pattern, and then attempt to return the next token.
- Both hasNext and next methods may block waiting for further input

---

- The findInLine(java.lang.String), findWithinHorizon(java.lang.String, int), and skip(java.util.regex.Pattern) methods operate independently of the delimiter pattern
- These methods attempt to match the specified pattern with no regard to delimiters in the input and thus can be used in special circumstances where delimiters are not relevant
- These methods may block waiting for more input

- When a scanner throws an InputMismatchException, the scanner will not pass the token that caused the exception, so that it may be retrieved or skipped via some other method.

- Depending upon the type of delimiting pattern, empty tokens may be returned:
- For example, the pattern "\\s+" will return no empty tokens since it matches multiple instances of the delimiter
- The delimiting pattern "\\s" could return empty tokens since it only passes one space at a time

- A scanner can read text from any object which implements the Readable interface.
- If an invocation of the underlying readable's Readable.read(java.nio.CharBuffer) method throws an IOException then the scanner assumes that the end of the input has been reached

- When a Scanner is closed, it will close its input source if the source implements the Closeable interface.
- It implements Closeable so it can be used with try-with resources