Java Programming Tutorial

Regular Expression (Regexe) in Java

Introduction

Regular Expression (regexe) is extremely useful in programming, especially in processing text files.

The Sun's online Java Tutorial trail on "Regular Expressions" is excellently written. Please read if you are new to regexe.

`java.util.regex.Pattern` & `java.util.regex.Matcher` (JDK 1.4)

Regular expression was introduced in Java 1.4 in package `java.util.regex`. This package contains only two classes: `Pattern` and `Matcher`.

1. The `Pattern` class represents a compiled regular expression. You get a `Pattern` object via `static` method `Pattern.compile(String regexe)`.
2. The resulting `Pattern` object is used to obtain a `Matcher` instance, which is used to parse the input source.

```
String regexe = "......";
String input  = "......";
Pattern pattern = Pattern.compile(regexe);
Matcher matcher = pattern.matcher(input);
```

Regexe by Examples

Example 1: Find Text

For example, given the input "This is an apple. These are 33 (thirty-three) apples", you wish to find all occurrences of pattern "Th" (case-sensitive or case insensitive).

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

public class RegexeFindText {
   public static void main(String[] args) {

      // Input for matching the regexe pattern
      String input = "This is an apple. These are 33 (thirty-three) apples";
      // Regexe to be matched
      String regexe = "Th";

      // Step 1: Allocate a Pattern object to compile a regexe
      Pattern pattern = Pattern.compile(regexe);
      //Pattern pattern = Pattern.compile(regexe, Pattern.CASE_INSENSITIVE);  // case-insensitive matching

      // Step 2: Allocate a Matcher object from the compiled regexe pattern,
      //         and provide the input to the Matcher
      Matcher matcher = pattern.matcher(input);

      // Step 3: Perform the matching and process the matching result

      // Use method find()
      while (matcher.find()) {     // find the next match
         System.out.println("find() found the pattern \"" + matcher.group()
               + "\" starting at index " + matcher.start()
               + " and ending at index " + matcher.end());
      }

      // Use method matches()
      if (matcher.matches()) {
         System.out.println("matches() found the pattern \"" + matcher.group()
               + "\" starting at index " + matcher.start()
               + " and ending at index " + matcher.end());
      } else {
         System.out.println("matches() found nothing");
      }

      // Use method lookingAt()
      if (matcher.lookingAt()) {
         System.out.println("lookingAt() found the pattern \"" + matcher.group()
               + "\" starting at index " + matcher.start()
               + " and ending at index " + matcher.end());
      } else {
         System.out.println("lookingAt() found nothing");
      }
   }
}
```

```
find() found the pattern "Th" starting at index 0 and ending at index 2
find() found the pattern "Th" starting at index 18 and ending at index 20
matches() found nothing
lookingAt() found the pattern "Th" starting at index 0 and ending at index 2
```

Explanation:

- Java's regexe classes are kept in package `java.util.regex.Pattern`. There are only two classes in this package: `Pattern` and `Matcher`. You should browse the Javadoc for `Pattern` class, followed by `Matcher` class.
- Three steps are required to perform regexe matching:
  - Allocate a `Pattern` object. There is no constructor for the `Pattern` class. Instead, you invoke the `static` method `Pattern.compile(`*`regexeString`*`)` to compile the *`regexeString`*, which returns a `Pattern` instance.
  - Allocate a `Matcher` object. Again, there is no constructor for the `Matcher` class. Instead, you invoke the `matcher(`*`inputString`*`)` method from the `Pattern` instance (created in Step 1). You also bind the input sequence to this `Matcher`.
  - Use the `Matcher` instance (created in Step 2) to perform the matching and process the matching result. The `Matcher` class provides a few `boolean` methods for performing the matches:
    - `boolean find()`: scans the input sequence to look for the *next* subsequence that matches the pattern. If match is found, you can use the `group()`, `start()` and `end()` to retrieve the matched subsequence and its starting and ending indices, as shown in the above example.
    - `boolean matches()`: try to match the entire input sequence against the regexe pattern. It returns true if the entire input sequence matches the pattern.
    - `boolean lookingAt()`: try to match the input sequence, starting from the beginning, against the regexe pattern. It returns true if a *prefix* of the input sequence matches the pattern.
- To perform case-insensitive matching, use `Pattern.compile(`*`regexeString`*`, Pattern.CASE_INSENSITIVE)` to create the `Pattern` instance (as commented out in the above example).

Example 2: Find Pattern (Expressed in Regular Expression)

The above example to find a particular piece of text from an input sequence is rather trivial. The power of regexe is that you can use it to specify a pattern, e.g., `(\w)+` matches any word (delimited by space), `\b[1-9][0-9]*\b` matches any number with a non-zero leading digit, separated by spaces from other words..

Try changing the regexe pattern of the above example to the followings and observe the outputs. Take not that you need to use a escape sequence `'\'` for special characters such as `'\'` inside a Java's string.

```
String regexe = "\\w+";        // Escape needed for \
String regexe = "\\b[1-9][0-9]+\\b";
```

Read Javadoc for the class `java.util.regex.Pattern` for the list of regular expression constructs supported by Java.

Read Sun's online Java Tutorial trail on "Regular Expressions" on how to use regular expression.

Example 3: Find and Replace Text

Finding a pattern and replace it with something else is probably one of the most frequent tasks in text processing. Regexe allows you to express the pattern liberally, and also the replacement text/pattern. This is extremely useful in batch processing a huge text document or many text files. For example, searching for stock prices from many online HTML files, rename many files in a directory with a certain pattern, etc.

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

public class RegexeFindReplace {
   public static void main(String[] args) {
      String input = "This is an apple. These are 33 (Thirty-three) apples";
      String regexe = "apple";        // pattern to be matched
```

```
        String replacement = "orange";  // replacement pattern

        // Step 1: Allocate a Pattern object to compile a regexe
        Pattern pattern = Pattern.compile(regexe, Pattern.CASE_INSENSITIVE);

        // Step 2: Allocate a Matcher object from the pattern, and provide the input
        Matcher matcher = pattern.matcher(input);

        // Step 3: Perform the matching and process the matching result
        String output = matcher.replaceAll(replacement);     // all matches
        //String output = matcher.replaceFirst(replacement); // first match only
        System.out.println(output);
    }
}
```

Explanation:

- First, create a `Pattern` object to compile a regexe pattern. Next, create a `Matcher` object from the `Pattern` and specify the input.
- The `Matcher` class provides a `replaceAll(`*`replacement`*`)` to replace all the matched subsequence with the *`replacement`*; or `replaceFirst(`*`replacement`*`)` to replace the first match only.

Example 4: Find and Replace with Back References

Given the input "One:two:three:four", the following program produces "four-three-two-One" by matching the 4 words separated by colons, and uses the so-called back-references (`$1`, `$2`, `$3` and `$4`) in the replacement pattern.

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

public class RegexeReplaceBackReference {
   public static void main(String[] args) {
      String input = "One:two:three:four";
      String regexe = "(.+):(.+):(.+):(.+)"; // pattern to be matched
      String replacement = "$4-$3-$2-$1";    // replacement pattern with back references

      // Step 1: Allocate a Pattern object to compile a regexe
      Pattern pattern = Pattern.compile(regexe, Pattern.CASE_INSENSITIVE);

      // Step 2: Allocate a Matcher object from the Pattern, and provide the input
      Matcher matcher = pattern.matcher(input);

      // Step 3: Perform the matching and process the matching result
      String output = matcher.replaceAll(replacement);     // all matches
      //String output = matcher.replaceFirst(replacement); // first match only
      System.out.println(output);
   }
}
```

Parentheses `()` have two usages in regular expressions:

1. To resolve ambiguity: For example `xyz+` matches a '`x`', a '`y`', followed by one or more '`z`'. But `(xyz)+` matches one or more groups of '`xyz`', e.g., '`xyzxyzxyz`'.
2. Provide references to the matched subsequences. The matched subsequence of the first pair of parentheses can be referred to as `$1`, second pair of patentee as `$2`, and so on. In the above example, there are 4 pairs of parentheses, which were referenced in the replacement pattern as `$1`, `$2`, `$3`, and `$4`. You can use `groupCount()` (of the `Matcher`) to get the number of groups captured, and `group(groupNumber)`, `start(groupNumber)`, `end(groupNumber)` to retrieve the matched subsequence and their indices. In Java, `$0` denotes the *entire* regular expression.

Example 5: Rename Files of a Given Directory

The following program rename all the files ending with "`.class`" to "`.out`" of the directory specified.

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

public class RegexeRenameFiles {
   public static void main(String[] args) {
      String regexe = ".class$";   // ending with ".class"
      String replacement = ".out"; // replace with ".out"

      // Allocate a Pattern object to compile a regexe
      Pattern pattern = Pattern.compile(regexe, Pattern.CASE_INSENSITIVE);
      Matcher matcher;
      File dir = new File("d:\\temp");  // directory to be processed
      int count = 0;
      File[] files = dir.listFiles();   // list all files and dirs
      for (File file : files) {
         if (file.isFile()) {   // file only, not directory
            String inFilename = file.getName();    // get filename, exclude path
            matcher = pattern.matcher(inFilename); // allocate Matches with input
            if (matcher.find()) {
               ++count;
               String outFilename = matcher.replaceAll(replacement);
               System.out.print(inFilename + " -> " + outFilename);

               if (file.renameTo(new File(dir + "\\" + outFilename))) {  // execute rename
                  System.out.println(" SUCCESS");
               } else {
                  System.out.println(" FAIL");
               }
            }
         }
      }
      System.out.println(count + " files processed");
   }
}
```

You can use regexe to specify the pattern, and back references in the replacement, as in the previous example.

The `String.split()` Method

The `String` class contains a method `split()`, which takes a regular expression and splits this `String` object into an array of `String`s.

```
// In String class
public String[] split(String regex)
```

For example,

```
public class StringSplitTest {
   public static void main(String[] args) {
      String source = "There are thirty-three big-apple";
      String[] tokens = source.split("\\s+|-");  // whitespace(s) or -
      for (String token : tokens) {
         System.out.println(token);
      }
   }
}
```

```
There
are
thirty
three
big
apple
```

The `Scanner` & `useDelimiter()`

The `Scanner` class, by default, uses *whitespace* as the delimiter in parsing input tokens. You can set the delimiter to a regexe via use `delimiter()` methods:

```
public Scanner useDelimiter(Pattern pattern)
public Scanner useDelimiter(String pattern)
```

For example,

```
import java.util.Scanner;
public class ScannerUseDelimiterTest {
   public static void main(String[] args) {
      String source = "There are thirty-three big-apple";
      Scanner in = new Scanner(source);
      in.useDelimiter("\\s+|-");  // whitespace(s) or -
      while (in.hasNext()) {
         System.out.println(in.next());
      }
   }
}
```

REFERENCES & RESOURCES

- Sun's online Java tutorial trail on "Regular Expressions"
- Javadoc for package `java.util.regex`.

Latest version tested: JDK 1.6
Last modified: January 18, 2010