Regex Recipes

V 1.0

By Steven R. Brandt

Cooking up code with regular expressions

Regular Expressions in Java
http://www.javaregex.com

Published in the United States of America

Table of Contents

Introduction

You may have noticed that this book has no ISBN number. That is because, at least for now, it is very much a work in progress. For that reason it will be versioned, like software, and will be updated as new sections are added.

The current version will always be available at http://www.javaregex.com. Both electronic and print-on-demand versions will be available from that site.

The cover is a picture of my son and me, playing in the kitchen. Remember, programming is supposed to be fun. So play with your regular expressions – there's even a regular expression game to help you do that. It is discussed at the end of this book.

My son's still a little too young to understand regular expressions right now. That adventure is still in his future.

Tutorial

This tutorial makes heavy use of example code snippets. These code blocks will generally all compile and run if they are just inserted into the main block of a java class, and the packages java.util, and java.util.regex are included.

It is my hope that you will run some of these code snippets yourself, and modify them. This is the best way to learn what they really do. [The best way to learn what these code snippets really do is to run some yourself and modify them.]

Basic Text Matching

Regular expressions are a valuable tool by which one can process text. The most basic regular expression is a literal text string (all the alphabetic and numeric characters that are not preceded by a “\” will be interpreted literally). We can find the word "shells" in a String, as well as where in the string the pattern was, as follows:

```
// create a Pattern object
Pattern p = Pattern.compile("shells");

// search for a match within a string
String txt = "She sells sea shells by the sea shore.";
Matcher m = p.matcher(txt);
boolean found = m.find();

System.out.println(found);
// Prints "true" – the find() method returns a boolean that
// tells us whether the match succeeded.

System.out.println(m.group());
// Prints "14" – the starting position of the match

System.out.println(txt.substring(0,m.start()));
// Prints "She sells sea " -- the part of the string before
// the match.

System.out.println(m.end());
// Prints "20" – the ending position of the match

System.out.println(txt.substring(m.end()));
// Prints “ by the sea shore.”  The text following the match.
```

Ignoring Case

The above bit of code does not match if we encounter the substring "SHELLS" rather than "shells".

```
Pattern p = Pattern.compile("shells");

String txt = "SHE SELLS SEA SHELLS BY THE SEA SHORE.";
Matcher m = p.matcher(txt);
boolean found = m.find();

System.out.println(found);
// Prints "false"
```

We can fix this by changing our pattern.

```
Pattern p = Pattern.compile("(?i)shells");

String txt = "SHE SELLS SEA SHELLS BY THE SEA SHORE.";
Matcher m = p.matcher(txt);
boolean found = m.find();

System.out.println(found);
// Prints "true"
```

The "(?i)" tells the pattern to ignore the case of all letters.

What if you don't want to ignore the case of all parts of your pattern? The ignore text flag does not take effect until after it appears in the pattern. Also, you can turn off the ignore case flag with “(?-i)”.

```
Pattern p = Pattern.compile("apple(?i)bug(?-i)apple");
String txt = "APPLEBUGAPPLE appleBUGAPPLE appleBUGapple";
Matcher m = p.matcher(txt);
m.find();
System.out.println(m.group()); // prints "appleBUGapple"
```

The pattern matcher will only ignore case for the word “bug”. The case of the word “apple”, both the instance before and after “bug” must be matched. Therefore the pattern matcher must skip “APPLEBUGAPPLE” and “appleBUGAPPLE” and match on “appleBUGapple”. That is the only place where the case matches everywhere but the word “bug”.

But there is some funny business here in the ignore case world – the full set of unicode cases is not matched using “(?i)”. If you want that to happen, you have to include the “u” flag. The ability to leave the “u” flag off is for optimization and compatibility with other regular expression compilers.

```
Pattern p = Pattern.compile("(?i)\u00e0"); // This is an à
String txt = "\u00c0"; // this is an À
Matcher m = p.matcher(txt);
System.out.println(m.find()); // prints false

Pattern p2 = Pattern.compile("(?iu)\u00e0");
Matcher m2 = p2.matcher(txt);
System.out.println(m2.find()); // prints true
```

In this example I've chosen a simple letter “À” (note the accent). The “u” flag is required here to get the case insensitive matching to work. Incidentally, I could have written “(?ui)” or “(?i)(?u)” or even “(?u)(?i)” to turn on both the “u” and “i” flags inside the regular expression.

Character Classes
Fun with Square Brackets

Let's get back to the original example. What if you just want to ignore case on one letter, the first one? “(?i)s(?-i)hells” seems like a lot of typing.

```
Pattern p = Pattern.compile("[Ss]hells");

String txt = "SHELLS Shells shells";
Matcher m = p.matcher(txt);
boolean found = m.find();

System.out.println(m.group());
// Prints "Shells"
```

When the regular expression engine sees square brackets, it understands that you want to match one of the characters inside them. Thus, "[Ss]" matches either "S" or "s". This type of pattern, however, has more uses than simply matching two cases of a letter. You can, for example, use it to match a digit. The pattern "[0123456789]" does this.

Note also that the pattern matcher matches the first thing it can. If our pattern were still “(?i)shells” then it would have matched on “SHELLS”. If our pattern had been simply “shells” it would have matched on the last “shells” in the string.

```
Pattern p = Pattern.compile("[012345678]");

String txt = "How old are you? I'm 35.";
Matcher m = p.matcher(txt);
boolean found = m.find();

System.out.println(m.group());
// Prints "3"
```

Ranges of Characters

It may have occurred to you that typing out the sequence of digits "[012345789]" is a little awkward. Imagine if we wanted to match all the letters of the alphabet, we would have to type a rather long string indeed. Fortunately, there is a shorter way to write this. We can specify ranges of letters and numbers. Thus "[0-9]" matches any digit; it matches all the characters in the range from 0 to 9. We can use "[a-z]" to match any lower case letter. We can use "[A-Z]" to match any upper case letter, or we can use "[A-Za-z0-9]" to match a character that is either an upper case letter, a lower case letter, or a digit.

```
Pattern p =
   Pattern.compile("[A-Z][a-z][a-z][a-z]");
// Matches an upper case letter,
// followed by three lower case letters

String txt =
  "What is your name?  My name is Fred.";

Matcher m = p.matcher(txt);
boolean found = m.find();
System.out.println(m.group());
// Prints "What"
```

Hmm. I was really hoping to match "Fred" not "What". So, I will just rewrite my pattern.

```
Pattern p =
  Pattern.compile("[A-VX-Z][a-z][a-z][a-z]");
// Matches an upper case letter (excluding W),
// followed by three lower case letters.

String txt =
  "What is your name?  My name is Fred.";

Matcher m = p.matcher(txt);
boolean found = m.find();
System.out.println(m.group());
// Prints "Fred"
```

Negation

We could also have matched using a negated character class:

```
Pattern p =
  Pattern.compile("[A-VX-Z][^ ][^ ][^ ]");
// Matches an upper case letter (excluding W),
// followed three non-space characters.

String txt =
  "What is your name?  My name is Fred.";

Matcher m = p.matcher(txt);
boolean found = m.find();
System.out.println(m.group());
// Prints "Fred"
```

When a "^" appears as the first character inside []'s it negates the pattern. Thus "[^]" matches any character other than a space (" "), and "[^0-9]" matches any character that is not a digit.

There are some special character classes, shortcuts if you will, to typing out long sequences of ranges and literals inside the square brackets. For numbers we have these:

Regular Expression Text	***Meaning***
\p{Digit}	Anything recognized as a digit by Unicode.
\d	[0-9]
\D	[^0-9]

With this little bit of knowledge we could re-write our previous example like this:

```
Pattern p = Pattern.compile("\\d");
// could have used "\\p{Digit}" – it would have
// been the same.

String txt = "How old are you? I'm 35.";
Matcher m = p.matcher(txt);
boolean found = m.find();

System.out.println(m.group());
// Prints "3"
```

Escapes

Notice that I had to put the backslash in twice. The reason for this is that the regular expression needs to see one backslash, but the java compiler will turn “\\” into “\”. Worse (or maybe it's “better”), it won't even compile if you had just typed: “\d”.

You can see this if you run the following program which gets the patten by prompting the user rather than from a hard-coded string:

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

public class ReadPatternFromInput {

  public static void main(String[] args)
       throws Exception {

    // Prompt the user for input. When you
    // see it, type "\d" and hit carriage return.
    System.out.println("Enter pattern:");

    // Get a buffered reader
    InputStreamReader isr =
      new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);

    // Read and compile the pattern
    Pattern p = Pattern.compile(br.readLine());

    String txt = "How old are you? I'm 35.";
    Matcher m = p.matcher(txt);
    boolean found = m.find();

    System.out.println(m.group());
    // Prints "3" if you chose "\d"
  }
}
```

The “\” is sometimes called an “escape.” And it can be useful in a variety of ways inside a regular expression. It can be used, for example, to make characters like “[“ match literally.

```
Pattern p = Pattern.compile("\\[[a-z]\\]");

String txt = "What's in the box? [q]";
Matcher m = p.matcher(txt);
boolean found = m.find();

System.out.println(m.group());
// Prints "[q]"
```

But what if you want to match on an actual backslash? Then, of course, you must escape the backslash. However, you must actually do it four times. Why? Each pair of backslashes gets turned into a single backslash by the java compiler – and in the end you need to have a pair of backslashes.

```
Pattern p = Pattern.compile("-\\\\-");
// This string actually contains two backslashes

String txt = "Where's the backslash? -\\-";
// This string actually contains one backslash

Matcher m = p.matcher(txt);
boolean found = m.find();

System.out.println(m.group());
// Prints "-\-"
```

So, again, to match a single backslash you need four backslashes.

Note that if a backslash precedes an alphanumeric character it has a special meaning. If it precedes any other kind of character it merely makes it literal. Thus “\\[” literally matches an open square bracket, while “\\Q” does not literally match an Q. In fact, what it does is signify the beginning of a literal match.

If you want a whole sequence of literal characters, one way to do it is to begin the sequence with “\\Q” and end it with “\\E”.

```
Pattern p = Pattern.compile("\\Q[not a class]\\E");
String txt = "==> [not a class] <==";
Matcher m = p.matcher(txt);
m.find();
System.out.println(m.group()); // "[not a class]"
```

Shortcuts

Here are some other examples of character classes / shorthand notations. I have restricted it to the set I find most useful:

Regular Expression Text	***Meaning***
\w	A shorthand for [a-zA-Z0-9_]

Regular Expression Text	***Meaning***
\W	A shorthand for [^a-zA-Z0-9_]
\d	A shorthand for [0-9]
\D	A shorthand for [^0-9]
\s	A shorthand for [\t\n\x0B\f\r]
\S	A shorthand for [^ \t\n\x0B\f\r]
\p{Lower}	Any lower case character.
\p{Upper}	Any upper case character.
\p{ASCII}	Any character with a numeric value between \0x00 and \0x7F.
\p{Alpha}	The union of \p{Lower} and \p{Upper}
\p{Alnum}	The union of \p{Alpha} and \p{Digit}
\p{Punct}	Any of these characters: !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
\p{Print}	The union of \p{Alnum} and \p{Punct}
\p{Blank}	A space or tab
\p{Space}	Shorthand for: [\t\n\x0B\f\r]

You can read more about these in the documentation page for java.util.regex.Pattern. Actually, that doc page is a good brief reference to the entire pattern matching syntax. As of this writing you can find it at:

http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html

The Dot

```
Pattern p1=
  Pattern.compile(".");
// matches anything except a "line terminator."
// Effectively a shortcut for "[^\n\r\u0085\u2028\u2029]".
// In most cases, this serves the purpose
// of matching any character.
// The pattern ".*" is a popular way
// to match arbitrary regions of text.
```

What's a “line terminator?” It can be one of several things – any of the set of characters \n (line feed, the unix line terminator), \r (carriage return), \u0085 (next line), \u2028 (line separator), \u2029 (paragraph separator), or the sequence \r\n. However, when considering the “.” pattern, the sequence “\r\n” for end of line is not really relevant. It would match the same way whether we include it in our end-of-line definition or not.

So you should just think of a “.” as being the same as the pattern: `"[^\n\r\u0085\u2028\u2029]"` – except that we can use flags to change the meaning of “.”

Many times you just want a character that just matches anything. Since "." doesn't match anything, what does? Well, "." can match anything if the “s” flag is enabled. To enable the “s” flag, include the string "(?s)" in the front of your pattern.

```
Pattern p2=
  Pattern.compile("(?s).");
// will match any character
Pattern p3=
  Pattern.compile("(?s)foo:.");
// matches on the string "foo:"
// followed by any character.
```

You can also use the “d” flag to signify a “Unix line terminator.” In other words, if you set the “d” flag then “.” will be the same as “[^\n]”. You might want to prepend “(?d)” to your pattern if you are using a pattern developed in Perl to Java.

Excessively fancy stuff: Unions and Intersections

It is possible to combine the various character classes to make a more accurate specification of the character class you are most interested in. Here is an example of how to combine two character classes, “\\d” and “\\D” to make a character class that matches anything.

```
Pattern p4=
  Pattern.compile("[\\d\\D]");
// will a digit character or non-digit
// character – in short, it will
// match any character!
```

You could also match a region of punctuation and space characters

```
Pattern p5=
  Pattern.compile("[\\p{Punct}\\p{Space}]");
// matches on any punctuation or space character
```

These combinations of character classes are unions. That is, the enclosing “[]” combines the adjacent character classes that it contains, matching on a larger set of characters.

It is also possible to get an intersection, and match on a smaller range of characters.

Here is a complete program that you can use to help you experiment with matching on a range of characters and to get a better feel for unions and intersections.

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

public class Combine {
      public static void main(String[] args) throws Exception {
            // Insert your own pattern here
            String pstr = "[[a-n]&&[d-z]]";

            Pattern p = Pattern.compile(pstr);
            for(int i=0;i<256;i++) {
                  String s = Character.toString((char)i);
                  Matcher m = p.matcher(s);
                  if(m.find()) {
                        System.out.print(m.group());
                        System.out.print(' ');
                  }
            }
            System.out.println();
      }
}
```

If you run this program without modifications, it will produce the following output:

“`d e f g h i j k l m n`”. Only the characters that are in the character class “`[a-n]`” and the character class “`[d-z]`” will print. This will be the same as the character class “`[d-n]`”.

To try other patterns, just edit the value of `pstr` and run.

Quantifiers

Greedy Quantifiers

In some of our examples above we re-typed character classes in order to get repeated matches. That is, we used “ `[A-VX-Z][^ ][^ ][^ ]`” to match an upper case letter (excluding W) followed by 3 characters which were not a space. We also used “ `[0123456789][0123456789]`” to match two digits. As you doubtless suspected, there's a better way.

```
Pattern p = Pattern.compile("\\d{1,4}");

String txt = "How old are you? I'm only 8.";

Matcher m = p.matcher(txt);
boolean found = m.find();
System.out.println(m.group());
// Prints "8"

m = p.matcher("How old are you?  I'm 35");
found = m.find();
System.out.println(m.group());
// Prints "35"

m = p.matcher(" When were you born?  In 1963");
found = m.find();
System.out.println(m.group());
// Prints "1963"
```

The “{1,4}” means match at least one and at most four of the preceding pattern element. “\\d” is, as we remember, a shorthand for “[0-9]” or a digit. Thus “\\d{1,4}” matches from one to four digits.

It is important to notice that "{1,4}" is *greedy*. That is, it matches as many times as it can.

It may be that we don't want to specify a maximum number of characters to match. Perhaps we just want to match one or more digits. We can do this by not supplying the second digit to the “{}” pattern element.

```
Pattern p = Pattern.compile("\\d{1,}");

String txt = "The SN is 18547-2993576";

Matcher m = p.matcher(txt);
boolean found = m.find();
System.out.println(m.group());
// Prints "18547"
```

Notice that the pattern matcher matches on the first thing it can. In this case it is the string 18547. Even though the string “2993576” is longer, it matches on “18547” because it is first.

There are some popular shorthands that apply to quantifiers.

Regular Expression Text	***Meaning***
*	A shorthand for {0,}
+	A shorthand for {1,}
?	A shorthand for {0,1}

Generally I advise you to avoid “*” and use “+” if possible. A more (but not overly) restrictive pattern is more likely to be faster (since the pattern matcher will generally need to check fewer possible matches) and more likely to give you what you actually want (since your search was more specific). Consider this:

```
Pattern p = Pattern.compile("\\d*");

String txt = "The SNX is 18547-2993576-99-8";

Matcher m = p.matcher(txt);
boolean found = m.find();
System.out.println(m.group());
// Prints nothing
```

What happened? Why did it match on a blank? Isn't the pattern supposed to be greedy and match the longest thing it can?

The problem is that the pattern matching engine will match on the first thing it can, and the quantifier will only be greedy within that context.

The search was “\\d*”, which is the same as “\\d{0,}”. There is a zero length string of digits right at the start of the string. If I had searched for “\\d*-” I would found “18547-” because the pattern matcher would be forced to find a string where the sequence of digits was followed by a “-”. This would have taken it all the way to the numeric sequence before it tried to match with the “\\d*” and then it would have hungrily matched the 5 digit string.

Reluctant Quantifiers

Does matching always have to be greedy? The answer is no. By following the pattern with a “?” we can make the matcher match on the shortest thing it can. If we use the pattern “\\d*?-” in our last example, what do we get?

```
Pattern p = Pattern.compile("\\d*?-");

String txt = "The SNX is 18547-2993576-99-8";

Matcher m = p.matcher(txt);
boolean found = m.find();
System.out.println(m.group());
// Prints "18547-”
```

So what happened? Why did it match on exactly the same thing? Shouldn't it have matched on just “7-”? Again, the reason is that the pattern matcher takes the first valid match first, the desires of the quantifier come second. To see the difference, try this:

```
Pattern p = Pattern.compile("-[\\d-]*-");

String txt = "The SNX is 18547-2993576-99-8";

Matcher m = p.matcher(txt);
boolean found = m.find();
System.out.println(m.group());
// Prints "-2993576-99-"

p = Pattern.compile("-[\\d-]*?-");
m = p.matcher(txt);
found = m.find();
System.out.println(m.group());
// Prints "-2993576-"
```

The pattern element “[\\d-]” matches on a digit or the “-”. Normally, the “-” has a special meaning inside square brackets, it delimits two ends of a range of values like “[0-9]”. However, in this case, when the “-” is just before the “]” it is clearly not part of a range and the pattern matcher interprets it as a literal. But that's a digression.

So “-[\\d-]*-” matches the most things it can. In this case, it will match everything from the first “-” to the last “-”. That means it gets the whole string “-2993576-99-”. With the “*?” quantifier, however, it is less greedy. It will match the shortest thing it can, and this means that it can leave off the “-99-” and match just “-2993576-”.

So we have seen greedy and non-greedy quantifiers. There is yet a third category of quantifiers in the java pattern matcher. The difference here is rather subtle.

Possessive Quantifiers

This third quantifier seeks to match as many characters as it can, so it is like the greedy quantifier, but if it can't match as much everything it fails. What does this mean? It means, for example, that the pattern “\\d++\\d” will always fail to match. The “\\d++” will match an entire group of integers, but the next “\\d” asks to match for one more and the possessive will not part with it. The greedy will.

```
Pattern p = Pattern.compile("(\\d++)\\d");

String txt = "The SNX is 18547-2993576-99-8";

Matcher m = p.matcher(txt);
boolean found = m.find();
if(found)
    System.out.println(m.group());
else
    System.out.println("not found");
// Prints "not found"

p = Pattern.compile("(\\d+)\\d");
m = p.matcher(txt);
found = m.find();
if(found)
    System.out.println(m.group());
else
    System.out.println("not found");
// Prints "18547"
```

One key to understanding the possessive quantifier is to see that the piece of text it will match on is context independent.

What do I mean by that?

I mean that you can tell which piece of text it will match on without considering what the rest of the pattern will do. In the example above, in the pattern “(\\d+)\\d”, the “(\\d+)” had to limit its greediness in order to let the finally “\\d” match on something. The set of digits it matched on depended on its context, on the pattern element that followed.

When considering what the possessive quantifier does it is not necessary to look at this context. “\\d++” matches on a whole sequence of digits, or the whole pattern fails.

Grouping

Capture Groups

What if you want to match on one of a set of words?

```
Pattern p = Pattern.compile("(?i)(apple|orange|banana)");

String txt = "List some fruits: apple, orange, banana";

Matcher m = p.matcher(txt);
boolean found = m.find();
System.out.println(m.group());
// Prints "apple"

txt = "What's left? Orange and banana";
m = p.matcher(txt);
found = m.find();
System.out.println(m.group());
// Prints "Orange"
```

What's going on here? You may remember the “(?i)” pattern element which turns on the ignore case flag. We do this because we wish to match fruit names regardless of their capitalization.

When I want to match on one of a set of alternatives I can use parenthesis, and separate the alternatives with the “|” character. In this case I make a list of all fruits (well, okay, just three) so that I can match on any one of them.

Actually, the group was not strictly necessary here. If I had used `"(?i)apple|orange|banana"` as my pattern I would have gotten the same result above. The “|” pattern doesn't need to be in a group.

One of the other cool things about groups, however, is that you can find out what part of the sub-pattern the group matched on.

```
Pattern p = Pattern.compile("(\\w)+");

String txt = "List some fruits: apple, orange, banana";

Matcher m = p.matcher(txt);
boolean found = m.find();
System.out.println(m.group());
// Prints "List"

System.out.println(m.group(1));
// Prints "t"

p = Pattern.compile("(\\w+)");

m = p.matcher(txt);
found = m.find();
System.out.println(m.group());
// Prints "List"

System.out.println(m.group(1));
// Prints "List"
```

Notice that the difference in the patterns above. In the first pattern, the group is on a single letter, and the group can match 1 or more times. When we go to print out what the group matched we see the last thing it matched, the “t”. Going from left to right, the pattern matched first on the “L”, then the “a”, then the “s”, and finally the “t”, and that's the substring we get back from `m.group(1)`.

In the second case, the quantifier is inside the group and so all the characters matched get returned by `m.group(1).`

Now let's combine what we have learned with quantifiers. We are going to match a list of fruits.

```
Pattern p = Pattern.compile(
    "(?i)((apple|orange|banana)[\\s,]*)+");

String txt = "List some fruits: apple, orange, banana";

Matcher m = p.matcher(txt);
boolean found = m.find();
System.out.println(m.group());
// Prints "apple, orange, banana"

System.out.println(m.group(1));
// Prints "banana"

System.out.println(m.group(2));
// Prints "banana"
```

So what happened here? We used two layers of groups, and we used them for different reasons. The inner grouping (group 2) is just our list of fruits. The outer grouping (group 1) contains the pattern “(apple|orange|banana)[\\s,]*”, which matches on a fruit followed by spaces and commas. The final quantifier “+” applies to the whole thing, allowing the whole inner pattern to repeat.

Groups are numbered according to the order their left parentheses appear in the pattern. That's why `m.group(1)` above is the outer group, and `m.group(2)` is the inner one.

The pattern inside the outer group matches three times. Going from left to right, the inner group first takes on the value “apple”, the outer group takes on the value “apple” plus the following comma and spaces. The inner value next takes on the value of “orange”, and the outer group takes on the value of “orange” plus the spaces and comma. Finally, both inner and outer groupings take on the value of “banana”. At this point the quantifier's greediness is satisfied.

Backreferences

There is a neat trick you can do if you want to match a captured group, a backreference, to another part of the string.

```
String s = "Hello, you big beautiful world!";
Pattern p = Pattern.compile("([aeiou]).*\\1");
Matcher m = p.matcher(s);
m.find();
System.out.println(m.group());
// prints "ello, you big be"

s = "Lots of people like regex!";
m = p.matcher(s);
m.find();
System.out.println(m.group());
// prints "ots of peo"
```

In this case our pattern finds the first vowel, then matches the entire string up to and including the next occurrence of the same vowel. In the first string above the vowel that is used is “e”, in the second it is “o”.

The idea is that the “\\1” refers to what you would get from “m.group(1)”. You could also use “\\2” to refer to what you'd get from “m.group(2)” and so on.

Non Capture Groups

What if you don't want to capture text? Why wouldn't you?

Suppose I had a pattern like

```
Pattern p = Pattern.compile(
    "([abc])[xy]*([def])");
String txt = "bxxyyd, axyyyf, cxyyxyye";
Matcher m = p.matcher(txt);
if(m.find()) {
    System.out.println(m.group(1)); // prints “b”
    System.out.println(m.group(2)); // prints “d”
}
```

And I wanted to modify it, changing the “[xy]*” part into something more complicated. I want to match on repeated sequences of “xyy”. In doing this I cause the capture group for “([def])” to change from being group 2 to group 3.

```
Pattern p = Pattern.compile(
    "([abc])(xyy)*([def])");
String txt = “bxxyyd, axyyyf, cxyyxyye”;
Matcher m = p.matcher(txt);
if(m.find()) {
    System.out.println(m.group(1)); // prints “c”
    System.out.println(m.group(2)); // prints “xyy”
}
```

If I use a non-capture group I can avoid shifting the group count.

```
Pattern p = Pattern.compile(
    "([abc])(?:xyy)*([def])");
String txt = “bxxyyd, axyyyf, cxyyxyye”;
Matcher m = p.matcher(txt);
if(m.find()) {
    System.out.println(m.group(1)); // prints “c”
    System.out.println(m.group(2)); // prints “e”
}
```

As you can see, a non capture group looks like this: “(?: ...)” where a normal capture group looks like this: “(...)”.

Lookaheads

The “(?” sequence was invented as a way to make all kinds of new extensions to the pattern list of regular expressions.

The lookahead pattern “(?= ...)” is a zero width assertion about what follows. What does that mean?

```
Pattern p = Pattern.compile(
      "foo(?=bar)([a-z])");
String txt = "foocat foodog foobar";
Matcher m = p.matcher(txt);
if(m.find()) {
      System.out.println(m.group()); // prints "foob"
      System.out.println(m.group(1)); // prints "b"
}
```

So in this case the pattern is saying that “foo” must be followed by “bar”, but “bar” is not part of the match. That is to say, “(?=bar)” does not match a sequence of characters, it only places a restriction on what must match from this point forward.

Logically, then, capture group 1 comes right after “foo”, so it will always return a “b” if the match succeeds.

Alternatively, you can use a negative lookahead.

```
Pattern p = Pattern.compile("foo(?!bar)");
String txt = "foobar";
Matcher m = p.matcher(txt);
System.out.println(m.find()); // prints "false"
```

Lookbehinds

And if you can look ahead, you can also look behind.

```
// a lookbehind
Pattern p = Pattern.compile("(?<=(a+))foo(.)");
String txt = "aaaafoobar foocat";
Matcher m = p.matcher(txt);
if(m.find()) {
    System.out.println(m.group(1)); // prints "a"
    System.out.println(m.group(2)); // prints "b"
}

// and a negative lookbehind
p = Pattern.compile("(?<!a+)foo(.)");
m = p.matcher(txt);
if(m.find()) {
    System.out.println(m.group(1)); // prints "c"
}
```

Notice that the lookbehind for “a+” was not greedy. The quantifier is only greedy in the forward direction. So when the lookbehind operator asked how to match starting from one character before the “foo” it greedily gobbled up the one “a” that was in front of it. In short, greediness does not work as you might expect in a lookbehind.

The next thing to notice is that I don't put capture groups inside negative lookbehinds or negative lookaheads in my examples. You might want to try it as an experiment. It gives you a rather strange result that you should not rely on.

Independent Groups

Remember possessive quantifiers? You could understand how they would match without considering the rest of the pattern.

The way in which a greedy quantifier matched depended on its context within a pattern. Thus when you apply the pattern “[abc]+[a-d]” to the text “abc” the pattern element “[abc]+”

will match “ab” and the final pattern element, “[a-d]” will match the “c”. If the second pattern element “[a-d]” had not been present, if “[abc]+” had been the whole pattern, then it would have been allowed to greedily match the whole sequence “abc”. But because of its context in the whole pattern it could not be as greedy as that.

The possessive quantifier was context independent, however. It would greedily match as many characters as it could, independent of the context. If it could not match as much as it wanted, it would allow the whole match to fail. Thus, the pattern “[abc]++[a-d]” when applied to the text “abc” would fail. The “[abc]++” would match the whole sequence, there would be nothing left for “[a-d]” to match on.

The independent group is like the possessive quantifier, it ignores its context. In fact, the pattern “(?>[abc]+)” does exactly the same thing as “[abc]++”.

Perhaps “independent quantifier” would have been a better name than “possessive” quantifier for this pattern element, so that the connection to independent group would have been more clear.

```
// greedy quantifier is context dependent
Pattern p = Pattern.compile("[abc]+[a-d]");
String txt = "abc";
Matcher m = p.matcher(txt);
System.out.println(m.find()); // prints "true"

// possessive quantifier is context independent
p = Pattern.compile("[abc]++[a-d]");
m = p.matcher(txt);
System.out.println(m.find()); // prints "false"

// this pattern does exactly the same thing as the
// preceeding one.
p = Pattern.compile("(?>[abc]+)[a-d]");
m = p.matcher(txt);
System.out.println(m.find()); // prints "false"
```

Note also that the independent group is a “non-capturing” group. That is to say, we can't extract its contents using m.group(1).

The non-capturing group allows you to do more than write possessive quantifiers in a different way.

```
Pattern p = Pattern.compile("(?>ab|a|c)b");
String txt = "ab abb";
Matcher m = p.matcher(txt);
m.find();
System.out.println(m.group()); // prints "abb"

p = Pattern.compile("(?:ab|a|c)b");
m = p.matcher(txt);
m.find();
System.out.println(m.group()); // prints "ab"
```

In the first match above, we have an independent group. When it encounters the substring “ab”, we get the first match in the group. That means there's nothing for the “b” following the group to match on. It then continues looking through the string for other matches. Eventually, it finds “abb” and is satisfied.

In the second match we don't use an independent group, so the matcher tries to use the “ab” first, fails to get a match, then tries the second option in the group. This means the b can match.

Let's try one more example. Suppose you are looking through your source code for an integer declaration that follows a comment. You might be tempted to use “/\\*.*?\\*/” with the “s” flag to match the comment. Unfortunately, it may not work.

```
String code =
        "/*\n"+
        " * comment 1\n"+
        " */\n"+
        "double d = 1.8;\n"+
        "/*\n"+"" +
        " * comment 2\n"+
        " */\n"+
        "int foo = 3;";
Pattern p =
    Pattern.compile("(?s)/\\*.*?\\*/\\s+int\\s+(\\w+)");
Matcher m = p.matcher(code);
m.find();
System.out.println(m.group());
```

Try compiling and running it. It actually will print out *both* comments in a single match. The reason is that the “.*?” does not really match the minimum, but the minimum for the pattern as a whole to succeed. So if it has to reach ahead through the next comment to do it, it will. An independent group can come to your rescue here. The following matches only comment 2.

```
String code =
        "/*\n"+
        " * comment 1\n"+
        " */\n"+
        "double d = 1.8;\n"+
        "/*\n"+"" +
        " * comment 2\n"+
        " */\n"+
        "int foo = 3;";
Pattern p = Pattern.compile("(?s)(?>/\\*.*?\\*/)\\s+int\\s+(\\w+)
");
Matcher m = p.matcher(code);
m.find();
System.out.println(m.group());
System.out.println(m.group());
```

Flags and Groups

We've touched on flags in several places in this tutorial. Flags change the way the pattern elements work or the way the pattern is parsed. For example, the “i” flag tells us to ignore case. We saw before that we can turn it on with “(?i)”, or off with “(?-i)”.

But you can also use a non-capturing group to turn flags on or off.

```
Pattern p = Pattern.compile("apple(?i)bug(?-i)apple");
String txt = "APPLEBUG appleBUGAPPLE appleBUGapple";
Matcher m = p.matcher(txt);
m.find();
System.out.println(m.group()); // prints "appleBUGapple"

p = Pattern.compile("apple(?i:bug)apple");
m = p.matcher(txt);
m.find();
System.out.println(m.group()); // same as above

p = Pattern.compile("apple(?:(?i)bug)apple");
m = p.matcher(txt);
m.find();
System.out.println(m.group()); // same as above

p = Pattern.compile("(?i)apple(?-i:bug)apple");
txt = "APPLEBUGAPPLE APPLEbugAPPLE";
m = p.matcher(txt);
m.find();
System.out.println(m.group()); // prints "APPLEbugAPPLE"
```

In both of the first three patterns, only the pattern “bug” matches regardless of case, and the two occurrences of apple match only lower case. In fact, the first three patterns are identical.

The second pattern creates a non-capture group that *turns off* the ignore case flag, and this only applies to the pattern “bug”. The second uses the fact that when I change a flag's state, that change only remains in effect as long as I'm in the current group.

So, if you want a capturing group that ignores case you could just write “((?i) ...)”.

Boundary Matches

The End of a Line or String

If you are looking for a word at the end of a string or a line you can use a boundary matching pattern. The “$” matches a boundary at the end of a string.

```
Pattern p = Pattern.compile("(?i)be$");
String txt = "To BE or not to be";
Matcher m = p.matcher(txt);
m.find();
System.out.println(m.group()); // prints "be"

txt = "To BE or not to be\n"+
      "That is the question";
m = p.matcher(txt);
System.out.println(m.find()); // prints "false"

p = Pattern.compile("(?im)be$");
m = p.matcher(txt);
m.find();
System.out.println(m.group()); // prints "be"
```

I know the text looks odd, but I used to different capitalizations of “be” so that we could more easily see which one we matched.

In the first match attempt we succeed, because “be” occurs right at the end of the string and “$” matches end of string. In the second attempt we fail. Even though “be” is at the end of the line (it occurs just before “\n”), “$” wants to match only on the end of the string.

In the last pattern match attempt the “$” matches the end of a line. Why? Because the “m” flag was turned on by the pattern “(?im)”. When the “m” flag is turned on the meaning of a the “$” changes. It will then match on either end of string or end of line. Note that instead of “(?im)” we could have written “(?i)(?m)” or “(?mi)” and it would all have been the same.

The end of a line is one of the characters “\n” (Linux/Unix line ending), “\r” (Macintosh line ending), “\u0085” (next line), “\u2028” (line separator), “\u2029” (paragraph separator), or the character sequence “\r\n” (Windows line ending).

If you just want the end of line to just be the Unix line ending, you can get this behavior using the “d” flag. Why did Sun choose to single out the Unix mode ending? Why not a “w” for a windows ending?

The reason is probably to make things more compatible with Perl, which understands only “\n” as the line ending when the “m” flag is enabled.

There are two other ways to match the end of the string, “\\z” and “\\Z”. These patterns do not change their behavior in response to the “m” flag. They always match end of input. The “\\Z”, however, will not match the final line terminator.

The Start of a Line or String

The start of a string is matched by “^”. As with the “$” pattern, the “m” flag can be used to make it match the start of a line instead.

```
Pattern p = Pattern.compile("(?i)^that");
String txt = "To BE or not to be\n"+
             "That is the question";
Matcher m = p.matcher(txt);
System.out.println(m.find()); // prints "false"

p = Pattern.compile("(?mi)^that");
m = p.matcher(txt);
m.find();
System.out.println(m.group()); // prints "That"
```

The “\\A” matches just like “^” with the “m” flag turned off.

Word Boundaries

Another sort of boundary you can match on is a word boundary – the beginning or end of a word. A word boundary can be either the beginning/end of a line, string, or the place where a word character (includes upper and lower case letters, digits, and the “_” character) and a non word character meet.

We can match on a word boundary by using the pattern “\\b”.

```
Pattern p = Pattern.compile("(?i)be");
String txt = "Beethoven may be the greatest";
Matcher m = p.matcher(txt);
m.find();
System.out.println(m.group()); // prints "Be"

p = Pattern.compile("(?i)\\bbe\\b");
m = p.matcher(txt);
m.find();

System.out.println(m.group()); // prints "be"
```

The first attempt matches on the start of the name “Beethoven”. Because the first pattern does not specify word boundaries, there is no reason to exclude the match.

The second attempt requires a boundary both at the beginning and the end of the word.

So the second attempt matches on “be” not the “Be” in “Beethoven”.

You can also match on the absence of a word boundary using “\\B”.

```
Pattern p = Pattern.compile("(?i)be");
String txt = "Beware of the bumblebee";
Matcher m = p.matcher(txt);
m.find();
System.out.println(m.group()); // prints "Be"

p = Pattern.compile("(?i)\\bbe\\b");
m = p.matcher(txt);
System.out.println(m.find()); // prints "false"

p = Pattern.compile("(?i)\\Bbe\\B");
m = p.matcher(txt);
m.find();
System.out.println(m.group()); // prints "be"
```

In this case, the final match finds the “be” in the middle of “bumblebee”.

Repeated Matching

So far I've just looked at what happens if you want to match just one time. It may be that you want to find all the matches of a pattern in a string. You can do that by simply calling “find()” repeatedly.

```
Pattern p = Pattern.compile("(?i)a");
String txt = "How many A's are there in this string?";
Matcher m = p.matcher(txt);
int count = 0;
while(m.find()) {
    System.out.println(
         "an \"A\" was found at position: "+m.start());
         // finds 5, 9, 13
    count++;
}
System.out.println("count = "+count); // prints "3"
```

There is a special pattern element, a kind of boundary match, that applies to repeated matching. It is the “\\G” pattern which matches on the end of the previous match. If this the first call to “find()” then “\\G” matches on the beginning of the string.

```
Pattern p = Pattern.compile("(?i)\\Ga");
String txt = "Aaa, quadruple: Aaaa";
Matcher m = p.matcher(txt);
int count = 0;
while(m.find()) {
    System.out.println(
        "an \"A\" was found at position: "+m.start());
        // finds 0, 1, 2
    count++;
}
System.out.println("count = "+count); // prints "3"
```

We only find 3 occurrences of the letter “a”, not 8 (as we would without the “\\G”), because the “\\G” requires the next match to begin where the previous one left off.

Flags

You have encountered a number of flags as you have read through this text. Here is a brief summary.

Flag	*Meaning*
i	Ignore case mode: do not take case into account when matching.
u	Unicode: combine with “i” to take unicode case mappings into account.
d	Unix mode: sets “\n” as the line ending
m	Multi line mode: allow “$” (“^”) to match on line ending (beginning)
s	Dot all mode: allow the “.” to match any character.

There are, however, a few other flags, and a few other ways of setting them.

The “comments” flag, represented by the “x”, allows one to embed white space and comments within a regular expression.

Thus:

```
Pattern p = Pattern.compile("(?x)a  b # white space in
pattern\n"+
        "# as well as anything from # to end of line\n"+
        "# does not count as part of the pattern.\n"+
        "c  d # so this should match 'abcd'");
String txt = "abcd";
Matcher m = p.matcher(txt);
System.out.println(m.find()); // prints "true"

p = Pattern.compile("(?x)a b\\ c d # the escaped white space does
count as part of the pattern");
m = p.matcher(txt);
System.out.println(m.find()); // prints "false" -- txt contains
no white space character

m = p.matcher("ab cd");
System.out.println(m.find());
// prints "true"
// We match on the white space character correctly
```

There is an additional flag “CANON_EQ”. This flag cannot be turned on and off within a pattern. It is enabled, instead, using a second method argument to “Pattern.compile()”.

```
Pattern p = Pattern.compile("mary",Pattern.CASE_INSENSITIVE);
Matcher m = p.matcher("My name is Mary");
System.out.println(m.find()); // prints "true"

p = Pattern.compile("n\u0303",Pattern.CANON_EQ);
m = p.matcher("\u00f1"); // this is the "ñ"
System.out.println(m.find()); // prints "true"

p = Pattern.compile("\u00f1",Pattern.CANON_EQ);
m = p.matcher("n\u0303");
System.out.println(m.find()); // prints "true"

p = Pattern.compile("\u00f1",Pattern.CASE_INSENSITIVE);
m = p.matcher("a\u0303");
System.out.println(m.find()); // prints "false"
```

You can see here what “CANON_EQ” does. It recognizes that “`n\u0303`” and “`ñ`” mean the same thing according to unicode – that they are canonically equivalent.

Splitting

One of the handy uses of regular expressions is the ability to split a String.

```
String s = "Hello, you big beautiful world!";
String patternStr = "[\\p{Punct}\\p{Space}]+";
System.out.println(Arrays.asList(s.split(patternStr)));
// Prints "[Hello, you, big, beautiful, world]"
```

Here we have used a pattern matching one or more space or punctuation characters to split up the string. In other words, we give the regular expression for the pieces we do *not* want in the string array. When split is called in this way, you can't get blank strings at the end of the array.

The array was converted to a List using the “Arrays.asList()” function in order gain access to the print formatting ability of the ArrayList object.

Optionally you can give split a second argument, the “limit,” which is the maximum size of the array that split will return. Here are some examples

```
String s = "Hello, you big beautiful world!";
String patternStr = "[\\p{Punct}\\p{Space}]+";

System.out.println(Arrays.asList(s.split(patternStr,0)));
// Prints "[Hello, you, big, beautiful, world]"

System.out.println(Arrays.asList(s.split(patternStr,3)));
// Prints "[Hello, you, big beautiful world!]"

System.out.println(Arrays.asList(s.split(patternStr,200)));
// Prints "[Hello, you, big, beautiful, world, ]"

System.out.println(Arrays.asList(s.split(patternStr,-1)));
// Prints "[Hello, you, big, beautiful, world, ]"
```

For a limit of zero, we get the same result as we had originally – apply the pattern as many times as possible in order to split up the string, and discard trailing empty strings.

For a limit of 3, we get back an array with a maximum size of 3. This means that our pattern can be applied at most 2 times within the string, and that is the size of the array we get back.

For a limit of 200, we get back an array of a maximum size of 200. This means that our pattern can be applied at most 199 times. However, the array we actually get back is of size 6.

For a limit of -1, the pattern is applied as many times as possible and trailing blank strings are kept. This works out to be the same as our choice of 200 – except that we didn't have to guess what our biggest possible array would be.

If you wanted to, you could easily write your own split function.

```
public static List mySplit(String patternStr, String s) {
    Pattern p = Pattern.compile(patternStr);
    Matcher m = p.matcher(s);
    List list = new ArrayList();
    int lastEnd = 0;
    while(m.find()) {
        list.add(s.substring(lastEnd,m.start()));
        lastEnd = m.end();
    }
    list.add(s.substring(lastEnd,s.length()));
    return list;
}
```

This list function is just the same as “String.split(String str,int limit)” where a “-1” was supplied for the limit. It also differs in that it returns a List rather than a String array.

Substituting Strings

Now that you know how to build patterns it's time to use them to make substitutions in strings. There are two ways to do this, the quick and simple way, and the more complex and flexible way. The quick and simple way first:

```
String s = "Hello, world!";

System.out.println(s.replaceFirst("[aeiou]","[V]"));
// prints "H[V]llo, world!"

System.out.println(s.replaceAll("[aeiou]","[V]"));
// prints "H[V]ll[V], w[V]rld!"
```

The “String.replaceFirst()” and “String.replaceAll()” methods allow us to do just what their names imply – replace either the first occurrence or all occurrences of a pattern with a given string. In this example we replaced individual vowels with the string “[V]”.

But what if we wanted to do something more complex? What if we wanted to replace lower case vowels with upper?

```
String s = "Hello, world!";
Pattern p = Pattern.compile("[aeiou]");
Matcher m = p.matcher(s);

StringBuffer sb = new StringBuffer();
while(m.find()) {
    m.appendReplacement(sb,m.group().toUpperCase());
}
m.appendTail(sb);

System.out.println(sb.toString());
// prints "HEllO, wOrld!"
```

So the only thing we need to learn in order to make more complex substitutions is the two methods -- “Matcher.appendReplacement()” and “Matcher.appendTail()”.

So far, these two methods don't seem to be doing much at all. In fact, you could easily do without them if you wanted to...

```
String s = "Hello, world!";
Pattern p = Pattern.compile("[aeiou]");
Matcher m = p.matcher(s);

StringBuffer sb = new StringBuffer();
int lastEnd = 0;
while(m.find()) {
    sb.append(s.substring(lastEnd,m.start()));
    sb.append(m.group().toUpperCase());
    lastEnd = m.end();
}
sb.append(s.substring(lastEnd,s.length()));

System.out.println(sb.toString());
// prints "HEllO, wOrld!"
```

It cost us just two extra lines of code. But there is more trickiness here.

```
Pattern p = Pattern.compile("([aeiou])");
Matcher m = p.matcher(s);

StringBuffer sb = new StringBuffer();
while(m.find()) {
    m.appendReplacement(sb,"<$1>");
}
m.appendTail(sb);

System.out.println(sb.toString());
// H<e>ll<o>, w<o>rld!
```

When the replacer on the matcher is used, “$n” gets replaced by the contents of “m.group(n)”. This means that if you want to actually get a “$” to appear in your replacement text you have to precede it with a backslash. This, in turn, means that if you want to see a backslash in your replacement text you have to have *two* backslashes – except that the compiler turns “\\” into a single backslash, so you need to use “\\\\” to get a single backslash in your replacement.

While this trickiness may be useful, you may also find it confusing and wish to use the “roll your own replacement” I provided in the last example.

Matching on a Byte Array

One of the cool things about the pattern matcher in java is that it is not restricted to matching on character arrays. You can also use it to match on a StringBuffer, a java.nio.CharBuffer, or a class of your own invention. The reason is that, deep down, the pattern matcher works on a CharSequence. This is an interface which String merely implements.

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

/** Basic implementation of a CharSequence */
public class ByteString implements CharSequence {
    byte[] bytes;
    public ByteString(byte[] sourceBytes) {
        this(sourceBytes,0,sourceBytes.length);
    }
    public ByteString(byte[] sourceBytes,int start,int end) {
        this.bytes = new byte[end-start];
        int n = 0;
        for(int i=start;i<end;i++) {
            this.bytes[n++] = sourceBytes[i];
        }
    }

    /** Required by CharSequence interface */
    public int length() {
        return bytes.length;
    }
    /** Required by CharSequence interface */
    public char charAt(int index) {
        return (char)bytes[index];
    }
    /** Required by CharSequence interface */
    public CharSequence subSequence(int start, int end) {
        return new ByteString(bytes,start,end);
    }

    /** for testing */
    public static void main(String[] args) throws Exception {
        ByteString bs =
          new ByteString("I like jello".getBytes("US-ASCII"));
        Pattern p =
          Pattern.compile("JELLO",Pattern.CASE_INSENSITIVE);
        Matcher m = p.matcher(bs);
        if(m.find()) {
            System.out.println(m.group());
        }
    }
}
```

Matching on a File

The fact that a CharBuffer object is also a CharSequence makes this task relatively straightforward. This little recipe shows you how it can be done. You'll need to import some stuff from java.io, and java.nio to make it work, however.

```
// Get a ByteBuffer from a file
FileInputStream fin = new FileInputStream("/etc/passwd");
FileChannel fc = fin.getChannel();
MappedByteBuffer mbb =
     fc.map(FileChannel.MapMode.READ_ONLY,0,fc.size());

// Get a CharBuffer from a ByteBuffer using an encoding
Charset cs = Charset.forName("UTF-8");
CharBuffer cb = cs.decode(mbb);

Pattern p = Pattern.compile("root.*");
Matcher m = p.matcher(cb);
if(m.find()) {
     System.out.println(m.group());
     // prints "root:x:0:0:root:/root:/bin/bash"
     // on my linux box
}
```

One of the key things to notice here is the use of the Charset object to decode your file. Unless your file is written in ASCII, its physical length will not necessarily be the same as the number of characters it contains. Since the CharSequence interface on which the matcher is based needs to know the character length, use of Charsets is very important.

Matching on a Stream

The problem with matching on a stream is that you don't know its physical length. This means that you can't simply match on it – you have to break it into chunks and match on the chunks. The must convenient chunk you can use for your input is probably the line.

```
// Get a stream
URL url = new URL("http://javaregex.com");
InputStream in = url.openStream();

// Create a reader that decodes the stream
Charset cs = Charset.forName("UTF-8");
InputStreamReader isr = new InputStreamReader(in,cs);

// Read the stream and match line by line
Pattern p = Pattern.compile("web\\w+");
BufferedReader br = new BufferedReader(isr);
for(String chunk=br.readLine();
        chunk!=null;chunk=br.readLine()) {
    Matcher m = p.matcher(chunk);
    if(m.find()) {
        System.out.println(m.group());
        // prints "website" as of today
    }
}
```

The Recipes

From this point on we assume that you have a fair idea of how to do regular expressions, that you have reasonably digested the material in the tutorial. However, you may not yet feel confident with regular expressions. To get to this point you need to do a little practice in more practical ways.

Matching a Quoted String

There are certain pitfalls you can run into quite easily. One place to see them is when attempting to match on a quoted string.

When I say “quoted string” I am thinking of the sort you might find in java source code. That is, you might find a quote character, preceded by a backslash, embedded in the string. You might also find two strings on the same line.

You might think, naively, that it's as simple as putting a “.*” pattern between two quote characters. Maybe you can see right away why this doesn't work.

```
Pattern p = Pattern.compile("\".*\"");
Matcher m1 = p.matcher("This string has a \"quote\"");
m1.find();
System.out.println(m1.group());
  // prints: "quote"

Matcher m2 = p.matcher("\"This string has a \\\"quote\\\"\"");
m2.find();
System.out.println(m2.group());
  // prints: "This string has a \"quote\""

Matcher m3 = p.matcher("This \"string\" has two \"quotes\".");
m3.find();
System.out.println(m3.group());
  // prints: "string" has two "quotes"
```

The problem comes when we have two strings. We wanted to get just one of the quoted strings at a time. The most obvious thing to try next is to use “.*?”, and this does get the third matcher to work correctly – but at the expense of making the second one fail.

The second matcher gets confused by the embedded quote character.

```
Pattern p = Pattern.compile("\".*?\"");
Matcher m1 = p.matcher("This string has a \"quote\"");
m1.find();
System.out.println(m1.group());
  // prints: "quote"

Matcher m2 = p.matcher("\"This string has a \\\"quote\\\"\"");
m2.find();
System.out.println(m2.group());
  // prints: "This string has a \"

Matcher m3 = p.matcher("This \"string\" has two \"quotes\".");
m3.find();
System.out.println(m3.group());
  // prints: "string"
```

The thing that's actually going to work is a bit more complex.

```
Pattern p = Pattern.compile("\"(?:\\\\.|[^\"\\\\])*\"");
Matcher m1 = p.matcher("This string has a \"quote\"");
m1.find();
System.out.println(m1.group());
  // prints: "quote"

Matcher m2 = p.matcher("\"This string has a \\\"quote\\\"\"");
m2.find();
System.out.println(m2.group());
  // prints: "This string has a \"quote\""

Matcher m3 = p.matcher("This \"string\" has two \"quotes\".");
m3.find();
System.out.println(m3.group());
  // prints: "string"
```

What is this magic?

Remember that four backslashes in a row match a single literal backslash. So “\\\\.” matches a backslash followed by a character that is not used for a line terminator.

This pattern is inside a non-capture group “(?: ...)” along with the pattern “[^\”\\\\]”, which matches a non-quote non-backslash character.

Thus, the group matches either a pair of characters the first of which is a backslash, or a single character that isn't a quote or backslash.

So this pattern matches a quote, followed by any number of escaped characters or characters that are not a backslash or quote, and ends with another quote.

But there's another wrinkle to the story. Very often there's more than one way to write the pattern that you want, and this last way is just a bit more concise:

```
Pattern p = Pattern.compile("\"(?>\\\\.|.)*?\"");
Matcher m1 = p.matcher("This string has a \"quote\"");
m1.find();
System.out.println(m1.group());
  // prints: "quote"

Matcher m2 = p.matcher("\"This string has a \\\"quote\\\"\"");
m2.find();
System.out.println(m2.group());
  // prints: "This string has a \"quote\""

Matcher m3 = p.matcher("This \"string\" has two \"quotes\".");
m3.find();
System.out.println(m3.group());
  // prints: "string"
```

In this pattern we use a capture group and the minimal match to allow us to use “.” instead of “[^\”\\\\]”. The capture group assures us that if we can match a two character element that starts with a backslash we will.

Matching a Floating Point Number

When matching a floating point number I am thinking about the sort of thing you might encounter in java source code. What you want to match is something that either has a decimal point or an exponent. If you have an exponent you may have digits leading it or trailing it or both.

What follows is our first attempt. A simple grouping of four basic patterns that could describe matches of floating point numbers in a language like java. Of course, as you've learned by now, our first tries never work :)

```
String ptxt =
        "(?i)(?:"+
        "\\d+\\.\\d*(?:e[+-]?\\d+|)[fF]?|"+ // ex: 34., 3.E2
        "\\d*\\.\\d+(?:e[+-]?\\d+|)[fF]?|"+ // ex: .2, .994e+4
        "\\d+e[+-]?\\d+[fF]?|"+ // ex: 1e-15
        "\\d+[fF]"+ // ex: 22f
        ")[fF]?";
Pattern p = Pattern.compile(ptxt);

double a9e = 1.;
// n8 is a class, e2 is a static member variable
double d = 34.+3.E2+.2+.994e+4+1e-15+a9e+15.+n8.e2+1f;
String txt =
  "double d = 34.+3.E2+.2+.994e+4f+1e-15+a9e+15.+n8.e2+1f;";

Matcher m = p.matcher(txt);
while(m.find()) {
    System.out.print(" ["+m.group()+"]");
}
System.out.println();
  // prints " [34.] [3.E2] [.2] [.994e+4f] [1e-15] [9e+15] [8.e2]
[1f]"
```

Just in front of the text string I intend to use for matching I have placed an actual computation using double precision numbers. This is to show that it is a valid example.

The result that you see is a bit mixed up. For example, “9e+15” is a valid double precision number, however “a9e” is a variable name and should not be counted as part of the number.

A similar problem occurs a little later -- “8.e2” is a validly formatted number, but in this case “n8” is a class name and “e2” is a static floating point variable name. These two problems are readily fixable if you just add some “\\b”'s. Our “\\d+” patterns should never be preceded by letters.

```
String ptxt =
        "(?i)(?:"+
        "\\b\\d+\\.\\d*(?:e[+-]?\\d+|)[fF]?|"+// ex: 34., 3.E2
        "\\d*\\.\\d+(?:e[+-]?\\d+|)[fF]?|"+ // ex: .2, .994e+4
        "\\b\\d+e[+-]?\\d+[fF]?|"+ // ex: 1e-15
        "\\b\\d+[fF]"+ // ex: 22f
        ")[fF]?";
Pattern p = Pattern.compile(ptxt);

double a9e = 1.;
double d = 34.+3.E2+.2+.994e+4+1e-15+a9e+15.+n8.e2+1f;
String txt = "double d = 34.+3.E2+.2+.994e+4f+1e-
15+a9e+15.+n8.e2+1f;";

Matcher m = p.matcher(txt);
while(m.find()) {
    System.out.print(" ["+m.group()+"]");
}
System.out.println();
  // prints " [34.] [3.E2] [.2] [.994e+4f] [1e-15] [15.] [1f]"
```

Parsing XML

The thing to remember when using a regex to parse a large document is that “|” is your friend. It can be used to separate different kinds of elements fairly easily.

At each point in the parsed string our parser asks, can I find a complete directive here? Can I find a complete comment here? Can I find a complete cdata section? Our parser really consists of a set of patterns for each of these joined by “|”.

Our parser will be an event based parser, simliar to the sax parser. In other words, as we parse we will call methods when we encounter certain events: i.e. “startElement()” when we encounter the start of an xml element. To use an event based parser, simply over-ride the event methods.

We are not claiming that this is the best or most efficient way of doing xml parsing, however it does illustrate the convenience and simplicity of regular expressions for parsing.

```
import java.io.File;
import java.io.FileInputStream;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.FileChannel;
import java.nio.charset.Charset;
import java.util.HashMap;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * @author Steven R Brandt
 */
public class XmlParse {
    static String mainPatternSrc = "(?s)\\G(?:" + // dot any flag
            "<!-.*?-->|" + // comment
            "<\\?(.*?)\\?>|" + // directive
            "<!\\[CDATA\\[(.*?)\\]\\]>|" + // cdata
            "<(/|)([\\w\\.\\-]+)([^>]*?)(/|)>|" + // element
            "([^<>]+))"; // text
    static Pattern mainPattern = Pattern.compile(mainPatternSrc);

    private CharSequence seq;
    private Matcher m;

    public XmlParse(CharSequence seq) {
        this.seq = seq;
    }
    ...
```

Our parser is initialized, now lets look at the main body of our code. Note that because we have used capture groups in the various sections we can use them to test which pattern sub element we've matched. If group 1 is non-null, we have a directive, if group 2 is non-null we have a cdata section and so on. Everything falls out rather naturally.

```
    ...
    public boolean parseNext() {
        if (m == null) {
            m = mainPattern.matcher(seq);
            documentStart();
            return true;
        } else if (m.find()) {
            if (m.group(1) != null) { // directive
                directive(m.group());
            } else if (m.group(2) != null) { // cdata
                text(m.group(2));
            } else if (m.group(4) != null) { // element
                if ("/".equals(m.group(3))) { // closing element
                    endElement(m.group(4));
                } else {
                    startElement(m.group(4));
                    if ("/".equals(m.group(6))) { // empty element
                        endElement(m.group(4));
                    }
                }
            } else if(m.group(7) != null) { // text
                text(translate(m.group(7)));
            }
            return true;
        } else {
            documentEnd();
            return false;
        }
    }
    ...
```

Our next task is to process entity definitions. We will use this for text areas, and for the values of attributes as we shall see in a little bit.

```
    ...
    static Map entityDefs = new HashMap();
    static {
        entityDefs.put("lt","<");
        entityDefs.put("gt",">");
        entityDefs.put("amp","&");
        entityDefs.put("quot","\"");
        entityDefs.put("apos","'");
    }

    static String entityPatternSrc = "&#(\\d+);|&(\\w+);";
    static Pattern entityPattern = Pattern.compile(entityPatternSrc);

    private String translate(String s) {
        Matcher m = entityPattern.matcher(s);
        StringBuffer sb = null;
        while(m.find()) {
            if(sb == null) sb = new StringBuffer();
            if(m.group(1) != null) { // numeric entity
                int i = Integer.parseInt(m.group(1));
                m.appendReplacement(sb,
                        Character.toString((char) i));
            } else { // named entity
                String entityStr = m.group(2);
                String entityVal = (String) entityDefs.get(entityStr);
                if(entityVal != null) {
                    m.appendReplacement(sb,entityVal);
                } else {
                    throw new Error("Unkown entity: "+entityStr);
                }
            }
        }
        if(sb != null) {
            m.appendTail(sb);
            return sb.toString();
        } else {
            return s;
        }
    }
    ...
```

Here we distinguish whether we have a named or numeric entity and return the appropriate character in either case. We use a hash map to help us look up the named entities. We use the check for null group values trick again.

We have also included a small optimization. We don't allocate the string buffer unless we find a match.

```
...
static String attributePatternSrc =
    "\\b(\\w+)=(?:'([^']+)'|\"([^\"]+)\")";
static Pattern attributePattern = Pattern.compile(attributePatternSrc);

/**
 * Call this from within startElement if you want to parse out
 * the attributes of an element.
 */
public Map getAttributes() {
    Map map = new HashMap();
    String attrString = m.group(5);
    Matcher mattr = attributePattern.matcher(attrString);
    while(mattr.find()) {
        String attrVal =
            mattr.group(2) == null ? mattr.group(3) : mattr.group(2);
        map.put(mattr.group(1),
                translate(attrVal));
    }
    return map;
}
...
```

This section simply parses out attribute value pairs on demand. You must call this method from within “startElement()” or the value of “m.group(5)” will not be available. This section is actually a bit oversimplified. It is possible to have comments, etc. inside the element tag itself.

Our final task is to provide basic methods to be over-ridden by the parsing program. In this example we're just going to print most of this stuff.

```
    ...
    // Events
    public void endElement(String s) {
        System.out.println("end element: "+s);
    }

    public void startElement(String s) {
        System.out.println("start element: "+s+" "+getAttributes());
    }

    public void text(String s) {}

    private void directive(String s) { System.out.println("directive: "+s); }

    public void documentEnd() { System.out.println("end document"); }

    public void documentStart() { System.out.println("start document"); }

    // Test method
    public static void main(String[] args) throws Exception {
        System.out.println(mainPatternSrc);
        File f = new File(args[0]); // file from command line
        FileInputStream fis = new FileInputStream(f);
        ByteBuffer bb =
            fis.getChannel().map(FileChannel.MapMode.READ_ONLY,0,f.length());
        CharBuffer cb = Charset.forName("UTF-8").decode(bb);
        XmlParse xp = new XmlParse(cb);
        while(xp.parseNext()) {
            ;
        }
    }
}
```

Notice that to process a file we need to repeatedly call parse next. Calling parse next will cause one (or possibly two) events to be delivered.

This parser has certain limitations – it cannot detect errors. It won't notice if the encoding doesn't match the xml directive. It won't notice if we have more than one top level node, or if our document is not well-formed. It does not understand namespaces. However, the purpose of this recipe was to show you how much you could do with just a few little regular expressions, not to provide an enterprise-grade parser.

Parsing a Java Source File

One of the interesting things you can do with regular expressions is make a simple parser of java source files. This parser will easily separate tokens like comments, strings, and variable names from one another.

We will use the same sort of tricks that we used in the xml parser section, but we will also use our recipes for floating point number and string matching.

```
import java.io.*;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.charset.Charset;
import java.util.regex.*;

public class ParseCode {

    public static void main(String[] args) throws Exception {
        String floatStr = "(?:" +
                "\\b\\d+\\.\\d*(?:e[+-]?\\d+|)[fF]?|"+//ex: 34., 3.E2
                "\\d*\\.\\d+(?:e[+-]?\\d+|)[fF]?|" +  // ex: .2, .994e+4
                "\\b\\d+e[+-]?\\d+[fF]?|" +           // ex: 1e-15
                "\\b\\d+[fF]" +                       // ex: 22f
                ")[fF]?";
        String codeStr = "(?:" +
                "((?>/\\*[\\d\\D]*?\\*/))|" + /* a comment */
                "(//.*+)|" +                 // a comment
                "(\"(?:\\\\.|[^\"\\\\])*\")|" +  // "quoted string"
                "('(?:\\\\.|[^'\\\\])*\')|" +    // 'quoted char'
                "("+floatStr + ")|" + // floating point number
                ")";
        Pattern codePattern = Pattern.compile(codeStr);

        File f = new File(args[0]);
        FileInputStream fin = new FileInputStream(f);
        FileChannel fc = fin.getChannel();
        ByteBuffer bb =
                fc.map(FileChannel.MapMode.READ_ONLY,0,f.length());
        Charset cs = Charset.forName("UTF-8");
        Matcher mat = codePattern.matcher(cs.decode(bb));
        while (mat.find()) {
            if(mat.group(1) != null) {
                System.out.println("comment1<"+mat.group(1)+">");
            } else if(mat.group(2) != null) {
                System.out.println("comment2<"+mat.group(2)+">");
            } else if(mat.group(3) != null) {
                System.out.println("dquote<"+mat.group(3)+">");
            } else if(mat.group(4) != null) {
                System.out.println("squote<"+mat.group(4)+">");
            } else if(mat.group(5) != null) {
                System.out.println("float<"+mat.group(5)+">");
            }
        }
    }
}
```

Notice we've also used our independent capture group on the comment (not really necessary in this case). Try running this program on some of your code and watch as it finds comments, strings, and floating point numbers. Perhaps you can re-write it to list your methods and fields – simpilar to javap but operating on source code.

A Simple Calculator

This will not be the most sophisticated calculator in the world, but it will, like a good student, show all its work.

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

public class Calc {
    // (), *, /
    static String calcStr1 = (
            "\\(num\\)|"+
            "num\\*num|"+
            "num/num").replaceAll("num","(\\\\d+(?:\\\\.\\\\d+|))");
    static Pattern calcPattern1 = Pattern.compile(calcStr1);

    // +, -
    static String calcStr2 = (
            "num\\+num|"+
            "num-num").replaceAll("num","(\\\\d+(?:\\\\.\\\\d+|))");
    static Pattern calcPattern2 = Pattern.compile(calcStr2);

    static NumberFormat nf = NumberFormat.getInstance();
    static { nf.setMaximumFractionDigits(5); }

    private static double d(Matcher m,int group) {
        return Double.parseDouble(m.group(group));
    }
    ...
```

Here is the intialization of our calculator. We've used a little shorthand to initialize our pattern. We've used String's “replaceAll()” to insert what we mean by “num” into our pattern string. Building the pattern string in this way, I think, adds a little clarity to what we want to say. I think “num\\+num” is a bit easier to understand than “(\\d+(?:\\.\\d+))\\+(\\d+(?:\\.\\d+)) ”.

Notice we've gotten a number formatter and set it to precision of 5. Maybe you want more precision – but for this example I think 20 looks ugly.

```
    ...
    public static double eval(String s) {
        s = s.replaceAll("\\s+",""); // strip out all white space
        while(true) {
            System.out.println(" => "+s); // show our work
            Matcher m1 = calcPattern1.matcher(s);
            if(m1.find()) { // (), *, /
                StringBuffer sb = new StringBuffer();
                if(m1.group(1) != null) {
                    m1.appendReplacement(sb,m1.group(1));
                } else if(m1.group(2) != null) {
                    m1.appendReplacement(sb,nf.format(d(m1,2)*d(m1,3)));
                } else if(m1.group(4) != null) {
                    m1.appendReplacement(sb,nf.format(d(m1,4)/d(m1,5)));
                }
                m1.appendTail(sb);
                s = sb.toString();
                continue;
            } else {
                Matcher m2 = calcPattern2.matcher(s);
                if(m2.find()) { // +, -
                    StringBuffer sb = new StringBuffer();
                    if(m2.group(1) != null) {
                        m2.appendReplacement(sb,nf.format(d(m2,1)+d(m2,2)));
                    } else if(m2.group(3) != null) {
                        m2.appendReplacement(sb,nf.format(d(m2,3)-d(m2,4)));
                    }
                    m2.appendTail(sb);
                    s = sb.toString();
                    continue;
                }
            }
            return Double.parseDouble(s);
        }
    }
    public static void main(String[] args) throws Exception {
        System.out.println(eval("3.2*(1+9)-2.345/1.97"));
    }
}
```

We first do replacement for the operators * and /, and for parenthesis, because that will give us the proper order of operations. “1+2*3” should be “7” not “9”.

Parsing a CSV File

One of the standard things you might want to do in life is parse a CSV file. You might naively want to just call String.split(“,”) -- but this would not do what you want if the fields of the CSV contain commas, new lines, etc.

If the first character in a CSV cell is a quote, then we have a quoted cell. Quoted cells start with a quote and end with a quote, and may contain pairs of quotes in the middle. This pair of quotes in the middle is an encoding for a single quote. Confusing?

```
Pattern p = Pattern.compile("(?m)\\G\r*(\n)?(?:^|,)(?:" +
    "\"((?:\"\"|[^\"])*)\"([^,\n]*)|" + // quoted cell
    "([^,\n]*)" + // unquoted cell
    ")");

// Match on whole file at once -- newlines can be in a quoted cell
Matcher m = p.matcher(
    "a,b, \"c\",,\"d,e\",\" >\"\"< \"\n" +
    "x,y,\"z\nw\",p,\"u\" frog");

int colNum=1;
while(m.find()) {
    if(m.group(1) != null) colNum = 1; // new row detected
    if (m.group(4) != null) { // unquoted cell
        System.out.println((colNum++)+"] "+m.group(4));
    } else { // quoted cell, replace "" with ""
        String inQuote = m.group(2).replaceAll("\"\"","\"");
        String afterQuote = m.group(3);
        System.out.println((colNum++)+"] "+inQuote+afterQuote);
    }
}
// 1] a
// 2] b
// 3]  "c"
// 4]
// 5] d,e
// 6]  >"<
// 1]  x
// 2] y
// 3] z
//    w
// 4] p
// 5] u frog
```

Column 1 of row 1 is just “a”, and column 2 is just “b”, these are plain old unquoted cells. Column 3 is also considered an unquoted cell because its first character is not a quote. Thus, the quote marks are part of the text of the cell. Column 4 is an unquoted empty cell. Column 5 is a quoted cell, and actually contains a comma. Column 6 is a quoted cell and uses the double quote to encode a single quote.

Then we have a new row. Note that in column 3 of row 2 we have a line feed inside the quoted cell.

Named Groups

Ever get tired of counting your parenthesis to figure out what group number you are in? Ever get annoyed at the fact that inserting a new group into your pattern throws off your pattern count? Then maybe you want “named groups.” Unfortunately, regular expressions don't offer you such a thing off the shelf, but it's amazingly easy to add that feature.

You do it by parsing regular expression text with a simple regular expression.

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;
import java.util.Map;
import java.util.Hashtable;
import java.util.HashMap;

public class NamedGroup {
    static Pattern p = Pattern.compile("(?<!\\\\)\\(\\?'([^']+)'(?!\\?)");

    /** Give names to group numbers in the pattern string --
     * (?'a'foo) gets translated into (foo) and the nameMap
     * will map "a" to 1.
     */
    public static Pattern compile(String patternStr,Map nameMap) {
        Matcher m = p.matcher(patternStr);
        StringBuffer sb = new StringBuffer(patternStr.length());
        int groupNum = 1;
        while(m.find()) {
            nameMap.put(m.group(1),new Integer(groupNum++));
            m.appendReplacement(sb,"(");
        }
        m.appendTail(sb);
        System.out.println(patternStr+" -> "+sb);
        return Pattern.compile(sb.toString());
    }

    /** A quick example/test */
    public static void main(String[] args) throws Exception {
        Map m = new HashMap();
        Pattern p = NamedGroup.compile("(?'a'[a-z]+(?'b'[0-9]+))",m);
        String txt = "look at this: abc123";
        Matcher c = p.matcher(txt);
        // look up group number by name
        int a = ((Integer)m.get("a")).intValue();
        int b = ((Integer)m.get("b")).intValue();
        while(c.find()) {
            System.out.println("a="+c.group(a));
            System.out.println("b="+c.group(b));
        }
    }
}
```

The syntax of the named group follows the convention, established in perl, of extending the pattern matching syntax through the use of the character sequence “(?”. We will extend it to “(?'name'” this will identify a capture group named “name”.

Notice that in the pattern we used to identify this we were careful to exclude parenthesis that were preceded by a backslash, and we also excluded the possibility of naming any other kind of grouping with the negative lookahead “(?!\\?)”. We only want capture groups here.

Because we cannot override Pattern or Matcher, we can't make the use of this class as transparent as we might like. However, it is still fairly straightforward. We use the compile method on NamedGroup rather than Pattern. It is the same, except that it understands our new pattern syntax and takes a Map as a second argument. The map will hold a mapping of name to group number.

Later, when we want to access a capture group we first look up the group number for that name using our map.

A Regular Expression Game

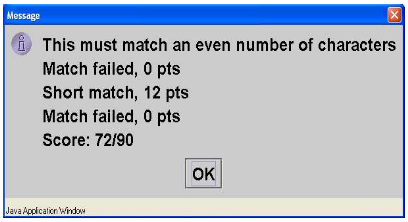
Finally, I've cooked up a Java WebStart application to help you practice your regular expressions. You can find it at http://javaregex.com/game. This game works by showing you three pieces of text, and how a regular expression matched against it.

When the game starts you see how the pattern matched. Against the text “that's life” the pattern string matched the “a” in “that's”. In other words, “a” is what was returned by “m.group()”.

But that's not the whole story of the match. There was a group in the pattern, group 1, and it returned the second “t” in “that's”.

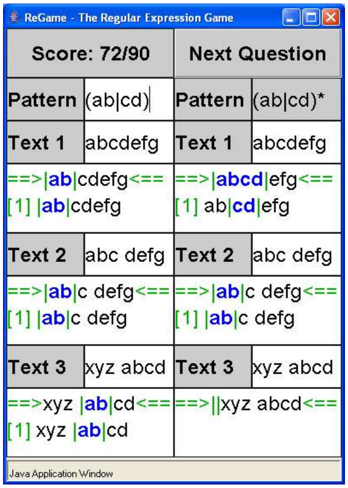
You type the pattern you think will match this text in the blank at the top left. When you hit enter, you'll see how your pattern matched – you'll also see your answer and your score so far.

If your pattern matches the same as the secret answer you'll get 10 points per text element it worked against. If your pattern is shorter than the answer, you get 12 points for the match. If your pattern works, but is longer than the answer you get 5 points per match.



This must match an even number of characters
Match failed, 0 pts
Short match, 12 pts
Match failed, 0 pts
Score: 72/90

After you have seen the answer, the “Next Question” button becomes enabled. No further questions, answers, or scoring will occur until you press it. The game will still allow you to experiment



ReGame - The Regular Expression Game

<table>
<tr><td colspan="2">Score: 72/90</td><td colspan="2">Next Question</td></tr>
<tr><td>Pattern</td><td>(ab|cd)</td><td>Pattern</td><td>(ab|cd)*</td></tr>
<tr><td>Text 1</td><td>abcdefg</td><td>Text 1</td><td>abcdefg</td></tr>
<tr><td colspan="2">==>|ab|cdefg<==
[1] |ab|cdefg</td><td colspan="2">==>|abcd|efg<==
[1] ab|cd|efg</td></tr>
<tr><td>Text 2</td><td>abc defg</td><td>Text 2</td><td>abc defg</td></tr>
<tr><td colspan="2">==>|ab|c defg<==
[1] |ab|c defg</td><td colspan="2">==>|ab|c defg<==
[1] |ab|c defg</td></tr>
<tr><td>Text 3</td><td>xyz abcd</td><td>Text 3</td><td>xyz abcd</td></tr>
<tr><td colspan="2">==>xyz |ab|cd<==
[1] xyz |ab|cd</td><td colspan="2">==>||xyz abcd<==</td></tr>
</table>

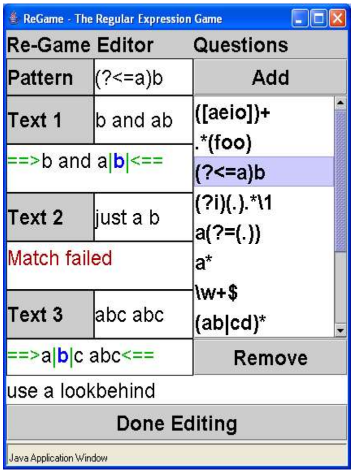
Java Application Window

and enter new patterns and apply them to the text, however, until you do so.

But there are more things you can do with the regular expression game – you can also design your own questions, as well as save and load question files. Of course you can also cheat by using the game editor to look at all the questions.

The editor allows you to select a pattern by name. You can provide each of the three text samples, and a description that will be shown along with the score. Buttons for add and delete are supplied.

When you are done editing, press the done button and you will be taken back to the main menu. From there you can save or load your game file. The file is stored in xml format as generated by the bean encoding mechanism.



ReGame - The Regular Expression Game

<table>
<tr><td colspan="2">Re-Game Editor</td><td>Questions</td></tr>
<tr><td>Pattern</td><td>(?<=a)b</td><td>Add</td></tr>
<tr><td>Text 1</td><td>b and ab</td><td rowspan="6">([aeio])+
.*(foo)
(?<=a)b
(?i)(.).*\1
a(?=(.))
a*
\w+$
(ab|cd)*</td></tr>
<tr><td colspan="2">==>b and a|b|<==</td></tr>
<tr><td>Text 2</td><td>just a b</td></tr>
<tr><td colspan="2">Match failed</td></tr>
<tr><td>Text 3</td><td>abc abc</td></tr>
<tr><td colspan="2">==>a|b|c abc<==</td></tr>
<tr><td colspan="2"></td><td>Remove</td></tr>
<tr><td colspan="3">use a lookbehind</td></tr>
<tr><td colspan="3">Done Editing</td></tr>
</table>

Java Application Window

Conclusion

I hope you have benefited from the tutorial and the recipes, and I hope you have enjoyed the game. Please send new game questions and other regular expression questions/ideas to me through the email page on my website: http://javaregex.com/support.html

Happy programming!